

Model*Sim* EE/SE

User's Manual

Version 5.3

The Model*Sim* Elite and Special Editions
for VHDL, Verilog, and Mixed-HDL Simulation

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

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

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

Copyright (c) 1990 -1999, Model Technology Incorporated.
All rights reserved. Confidential. Online documentation may be printed by licensed customers of Model Technology Incorporated for internal business purposes only.

Software Version: 5.3a

Published: September 1999

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

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

EE User's Manual - Part # M16535

US\$50

Software License Agreement

This is a legal agreement between you, the end user, and Model Technology Incorporated (MTI). By opening the sealed package you are agreeing to be bound by the terms of this agreement. If you do not agree to the terms of this agreement, promptly return the unopened package and all accompanying items to the place you obtained them for a full refund.

Model Technology Software License

1. LICENSE. MTI grants to you the **nontransferable, nonexclusive** right to use one copy of the enclosed software program (the "SOFTWARE") for each license you have purchased. The SOFTWARE must be used on the computer hardware server equipment that you identified in writing by make, model, and workstation or host identification number and the equipment served, in machine-readable form only, as allowed by the authorization code provided to you by MTI or its agents. All authorized systems must be used within the country for which the systems were sold. Model*Sim* licenses must be located at a single site, i.e. within a one-kilometer radius identified in writing to MTI. This restriction does not apply to single ModelSim PE licenses locked by a hardware security key, and such Model*Sim* PE products may be relocated within the country for which sold.

2. COPYRIGHT. The SOFTWARE is owned by MTI (or its licensors) and is protected by United States copyright laws and international treaty provisions. Therefore you must treat the SOFTWARE like any other copyrighted material, except that you may either (a) make one copy of the SOFTWARE solely for backup or archival purposes, or (b) transfer the SOFTWARE to a single hard disk provided you keep the original solely for backup or archival purposes. You may not copy the written materials accompanying the SOFTWARE.

3. USE OF SOFTWARE. The SOFTWARE is licensed to you for internal use only. You shall not conduct benchmarks or other evaluations of the SOFTWARE without the advance written consent of an authorized representative of MTI. You shall not sub-license, assign or otherwise transfer the license granted or the rights under it without the prior written consent of MTI or its applicable licensor. You shall keep the SOFTWARE in a restricted and secured area and shall grant access only to authorized persons. You shall not make software available in any form to any person other than your employees whose job performance requires access and who are specified in writing to MTI. MTI may enter your business premises during normal business hours to inspect the SOFTWARE, subject to your normal security.

4. PERMISSION TO COPY LICENSED SOFTWARE. You may copy the SOFTWARE only as reasonably necessary to support an authorized use. Except as permitted by Section 2, you may not make copies, in whole or in part, of the SOFTWARE or other material provided by MTI without the prior written consent of MTI. For such permitted copies, you will include all notices and legends embedded in the SOFTWARE and affixed to its medium and container as received

from MTI. All copies of the SOFTWARE, whether provided by MTI or made by you, shall remain the property of MTI or its licensors.

You will maintain a record of the number and location of all copies of the SOFTWARE made, including copies that have been merged with other software, and will make those records available to MTI or its applicable licensor upon request.

5. **TRADE SECRET.** The source code of the SOFTWARE is trade secret or confidential information of MTI or its licensors. You shall take appropriate action to protect the confidentiality of the SOFTWARE and to ensure that any user permitted access to the SOFTWARE does not provide it to others. You shall take appropriate action to protect the confidentiality of the source code of the SOFTWARE. You shall not reverse-assemble, reverse-compile or otherwise reverse-engineer the SOFTWARE in whole or in part. The provisions of this section shall survive the termination of this Agreement.

6. **TITLE.** Title to the SOFTWARE licensed to you or copies thereof are retained by MTI or third parties from whom MTI has obtained a licensing right.

7. **OTHER RESTRICTIONS.** You may not rent or lease the SOFTWARE. You shall not mortgage, pledge or encumber the SOFTWARE in any way. You shall ensure that all support service is performed by MTI or its designated agents. You shall notify MTI of any loss of the SOFTWARE.

8. **TERMINATION.** MTI may terminate this Agreement, or any license granted under it, in the event of breach or default by you. In the event of such termination, all applicable SOFTWARE shall be returned to MTI or destroyed.

9. **EXPORT.** You agree not to allow the MTI SOFTWARE to be sent or used in any other country except in compliance with this license and applicable U.S. laws and regulations. If you need advice on export laws and regulations, you should contact the U.S. Department of Commerce, Export Division, Washington, DC 20230, USA for clarification.

Important Notice

Any provision of Model Technology Incorporated SOFTWARE to the U.S. Government is with "Restricted Rights" as follows: Use, duplication, or disclosure by the Government is subject to restrictions as set forth in subparagraphs (a) through (d) of the Commercial Computer-Restricted Rights clause at FAR 2.227-19 when applicable, or in subparagraph (c)(1)(ii) of the Rights in Technical Data and Computer Software clauses in the NASA FAR Supplement. Any provision of Model Technology documentation to the U.S. Government is with Limited Rights. Contractor/manufacturer is Model Technology Incorporated, 10450 SW Nimbus Avenue / Bldg. R-B, Portland, Oregon 97223 USA.

Limited Warranty

LIMITED WARRANTY. MTI warrants that the SOFTWARE will perform substantially in accordance with the accompanying written materials for a period of 30 days from the date of receipt. Any implied warranties on the SOFTWARE are limited to 30 days. Some states do not allow limitations on duration of an implied warranty, so the above limitation may not apply to you.

CUSTOMER REMEDIES. MTI's entire liability and your exclusive remedy shall be, at MTI's option, either (a) return of the price paid or (b) repair or replacement of the SOFTWARE that does not meet MTI's Limited Warranty and which is returned to MTI. This Limited Warranty is void if failure of the SOFTWARE has resulted from accident, abuse or misapplication. Any replacement SOFTWARE will be warranted for the remainder of the original warranty period or 30 days, whichever is longer.

NO OTHER WARRANTIES. MTI disclaims all other warranties, either express or implied, including but not limited to implied warranties of merchantability and fitness for a particular purpose, with respect to the SOFTWARE and the accompanying written materials. This limited warranty gives you specific legal rights. You may have others, which vary from state to state.

NO LIABILITY FOR CONSEQUENTIAL DAMAGES. In no event shall MTI or its suppliers be liable for any damages whatsoever (including, without limitation, damages for loss of business profits, business interruption, loss of business information, or other pecuniary loss) arising out of the use of or inability to use these MTI products, even if MTI has been advised of the possibility of such damages. Because some states do not allow the exclusion or limitation of liability for consequential or incidental damages, the above limitation may not apply to you.

Table of Contents

Software License Agreement	3
Model Technology Software License	3
Important Notice	4
Limited Warranty	5

1 - Introduction (21)

ModelSim software versions documented in this manual	21
Standards supported	22
Assumptions	23
Sections in this document	24
Command reference	26
Text conventions	26
What is an "HDL item"	26
Where to find our documentation	27
Download a free PDF reader with Search	28
Online References - www.model.com	29
News	29
Partners	29
Products	29
Resources	29
Sales	29
Support	29
Comments	30

2 - Design Libraries (31)

Design library contents	32
Design unit information	32
Design library types	32
Library management commands	33
Working with design libraries	33
Creating a library	33
Viewing and deleting library contents	35
Assigning a logical name to a design library	37

Moving a library 39
Specifying the resource libraries 40
VHDL resource libraries 40
Predefined libraries 40
Alternate IEEE libraries supplied 41
Rebuilding supplied libraries 41
Regenerating your design libraries 41
Verilog resource libraries 42

3 - Projects and system initialization (43)

What is a project? 44
A new file extension 44
INI and MPF file comparison 44
The [Project] section in the .mpf file 45
Project operations 45
Creating a Project 46
Working with a Project 49
Open a project 49
Compile a project 49
Simulating a project 49
Modifying a project 49
The project command 50

4 - VHDL Simulation (51)

Compiling VHDL designs 52
Creating a design library 52
Invoking the VHDL compiler 52
Dependency checking 53
Simulating VHDL designs 53
Invoking the simulator from the Main window 53
Invoking Code Coverage with vsim 54
Using the TextIO package 55
Syntax for file declaration 55
Using STD_INPUT and STD_OUTPUT within ModelSim 56
TextIO implementation issues 56

Writing strings and aggregates 56
Reading and writing hexadecimal numbers 57
Dangling pointers 57
The ENDLINE function 58
The ENDFILE function 58
Using alternative input/output files 58
Providing stimulus 58
Obtaining the VITAL specification and source code 59
VITAL packages 59
ModelSim VITAL compliance 59
VITAL compliance checking 60
Compiling and Simulating with accelerated VITAL packages 61

5 - Verilog Simulation (63)

ModelSim variables 64
Compilation 65
Incremental compilation 66
Library usage 68
Verilog-XL compatible compiler options 70
Verilog-XL ‘uselib compiler directive 72
Simulation 74
Invoking the simulator 75
Simulation resolution limit 75
Event order issues 76
Verilog-XL compatible simulator options 78
Cell Libraries 80
SDF timing annotation 81
Delay modes 81
System Tasks 82
IEEE Std 1364-1995 system tasks 82
Verilog-XL compatible system tasks 85
Compiler Directives 87
IEEE Std 1364-1995 compiler directives 88
Verilog-XL compatible compiler directives 89
Using the Verilog PLI 90
Registering PLI applications 90
Compiling and linking PLI applications 92

The callback reason argument 97
The sizetf callback function 99
Object handles 99
Third party PLI applications 99
Support for VHDL objects 100
IEEE Std 1364 ACC routines 101
IEEE Std 1364 TF routines 103
Verilog-XL compatible routines 104
PLI tracing 105

6 - Mixed VHDL and Verilog Designs (107)

Separate compilers, common libraries 108
Mapping data types 108
VHDL generics 108
Verilog parameters 109
VHDL and Verilog ports 109
Verilog states 110
VHDL instantiation of Verilog design units 112
Verilog instantiation criteria 112
Component declaration 112
vgencomp component declaration 114
Verilog instantiation of VHDL design units 116
VHDL instantiation criteria 116
SDF annotation 117

7 - ModelSim SE Performance Analyzer (119)

Introducing Performance Analysis 120
A Statistical Sampling Profiler 120
Getting Started 121
Interpreting the data 122
Viewing Performance Analyzer Results 123
Interpreting the Name Field 125
Interpreting the %Under and %In Fields 125
Differences in the Ranked and Hierarchical Views 126
Ranked/Hierarchical Window Features 128
The report option 129

Setting preferences with Tcl variables	130
Performance Analyzer commands	131

8 - ModelSim SE Code Coverage (133)

Enabling Code Coverage	134
The coverage_summary window	136
Coverage Summary window preferences	137
Code Coverage commands	138

9 - Multiple logfiles, datasets and virtuals (139)

Multiple logfiles and datasets	139
Opening and viewing datasets	140
Using datasets with ModelSim commands	142
Restricting the dataset prefix display	143
Virtual Objects (User-defined buses, and more)	144
Virtual signals	145
Virtual functions	146
Virtual regions	146
Virtual types	147
Logfile and virtual commands reference table	147

10 - ModelSim EE Graphic Interface (149)

Window overview	150
Window features	151
Quick access toolbars	152
Drag and Drop	152
Command history	153
Automatic window updating	153
Finding names, searching for values, and locating cursors	154
Sorting HDL items	154
Multiple window copies	154
Menu tear off	154
Customizing menus and buttons	155
Combine signals into a user-defined bus	155
Tree window hierarchical view	155

Main window	158
The Main window menu bar	159
The Main window tool bar	165
The Main window status bar	167
Editing the command line, the current source file, and notepads	167
Dataflow window	170
The Dataflow window menu bar	171
Tracing HDL items with the Dataflow window	172
Saving the Dataflow window as a Postscript file	173
List window	174
HDL items you can view	174
The List window menu bar	175
Setting List window display properties	177
Adding HDL items to the List window	180
Editing and formatting HDL items in the List window	181
Examining simulation results with the List window	183
Finding items by name in the List window	184
Searching for item values in the List window	184
Setting time markers in the List window	186
List window keyboard shortcuts	187
Saving List window data to a file	188
Process window	189
The Process window menu bar	190
Signals window	192
The Signals window menu bar	193
Selecting HDL item types to view	195
Forcing signal and net values	195
Adding HDL items to the Wave and List windows or a logfile	197
Finding HDL items in the Signals window	198
Defining clock signals	198
Source window	200
The Source window menu bar	201
The Source window tool bar	203
Editing the source file in the Source window	204
Checking HDL item values and descriptions	204
Setting Source window options	205
Structure window	206
The Structure window menu bar	207
Variables window	209

The Variables window menu bar	210
Wave window	212
Wave window panes	215
HDL items you can view	216
The Wave window menu bar	217
Wave window tool bar	220
Adding HDL items in the Wave window	223
Combining and grouping items in the Wave window	225
Editing and formatting HDL items in the Wave window	226
Setting Wave window display properties	230
Sorting a group of HDL items	231
Finding items by name or value in the Wave window	231
Searching for item values in the Wave window	231
Using time cursors in the Wave window	233
Zooming - changing the waveform display range	235
Wave window keyboard shortcuts	237
Printing and saving waveforms	238
Compiling with the graphic interface	244
Locating source errors during compilation	245
Setting default compile options	245
Simulating with the graphic interface	251
Design selection page	252
VHDL settings page	254
Verilog settings page	256
SDF settings page	258
Setting default simulation options	260
ModelSim tools	263
Quick Start	264
The Button Adder	265
The Macro Helper	266
The Tcl Debugger	267
The GUI Expression Builder	272
The FPGA Library Manager	274
Graphic interface commands	277
Customizing the interface	279

11 - Standard Delay Format (SDF) Timing Annotation (281)

Specifying SDF files for simulation	282
---	-----

Instance specification	282
SDF specification with the GUI	283
Errors and warnings	283
VHDL VITAL SDF	284
SDF to VHDL generic matching	284
Resolving errors	285
Verilog SDF	286
The \$sdf_annotate system task	286
SDF to Verilog construct matching	287
Optional edge specifications	291
Optional conditions	292
Rounded timing values	292
SDF for Mixed VHDL and Verilog Designs	293
Interconnect delays	293
Troubleshooting	294
Specifying the wrong instance	294
Mistaking a component or module name for an instance label	295
Forgetting to specify the instance	295
Obtaining the SDF specification	296

12 - VHDL Foreign Language Interface (297)

Using the VHDL FLI	298
Using the VHDL FLI with foreign architectures	298
Using the VHDL FLI with foreign subprograms	300
Using checkpoint/restore with the FLI	305
Support for Verilog instances	308
Support for Windows platforms	308
FLI function descriptions	309
Mapping to VHDL data types	328
VHDL FLI examples	330
Compiling and linking FLI applications	331
Windows NT/95/98 platforms	331
Solaris platform	331
SunOS 4 platform	332
HP700 platform	332
IBM RISC/6000 platform	333
FLI tracing	334

The purpose of tracing files	334
Invoking a trace	334

13 - Value Change Dump (VCD) Files (337)

ModelSim VCD commands and VCD tasks	338
Resimulating a VHDL design from a VCD file	338
Extracting the proper stimulus for bidirectional ports	338
Specifying a filename and state mappings	339
Creating the VCD file	339
.	340
A VCD file from source to output	340
VHDL source code	340
VCD simulator commands	341
VCD output	341
Capturing port driver data with -dumpports	344
Supported TSSI states	344
Strength values	345
Port identifier code	345
Example VCD output from -dumpports	346

14 - Logic Modeling SmartModels (347)

VHDL SmartModel interface	348
Creating foreign architectures with sm_entity	349
Vector ports	352
Command channel	354
SmartModel Windows	354
Memory arrays	356
Verilog SmartModel interface	357
LMTV usage documentation	357
Linking the LMTV interface to the simulator	357
Compiling Verilog shells	357
.	357

15 - Logic Modeling Hardware Models (359)

VHDL Hardware Model interface	360
---	-----

Creating foreign architectures with hm_entity	361
Vector ports	364
Hardware model commands	366

16 - Tcl and ModelSim (367)

Tcl features within ModelSim	368
Tcl References	368
Tcl print references	368
Tcl online references	368
Tcl tutorial	368
Tcl commands	369
Tcl command syntax	370
if command syntax	373
set command syntax	374
Command substitution	374
Command separator	375
Multiple-line commands	375
Evaluation order	375
Tcl relational expression evaluation	375
Variable substitution	376
System commands	377
List processing	377
VSIM Tcl commands	378
ModelSim Tcl time commands	378
Conversions	378
Relations	379
Arithmetic	379
Tcl examples - UNIX only	380

A - Technical Support, Updates, and Licensing (383)

ModelSim worldwide contacts	383
Technical support - by telephone	384
Mentor Graphics customers In North America	384
Mentor Graphics customers outside North America	384
Model Technology customers worldwide	384
Technical support - electronic support services	384

Mentor Graphics customers	384
Model Technology customers	385
Technical support - other channels	386
Updates	386
Mentor customers: getting the latest version via FTP	386
Model Technology customers: getting the latest version via FTP	386
FLEXIm Licenses	386
Where to obtain your license	386
If you have trouble with licensing	387
All customers: ModelSim licensing	387
All customers: maintenance renewals and licenses	387
All customers: license transfers and server changes	387
Additional licensing details	387

B - ModelSim Variables (389)

Variable settings report	390
Personal preferences	390
Returning to the original ModelSim defaults	390
Environment variables	391
ModelSim Environment Variables	391
Setting environment variables in Windows	392
Referencing environment variables within ModelSim	393
Removing temp files (VSOUT)	394
Preference variables located in INI and MPF files	394
[Library] library path variables	395
[vcom] VHDL compiler control variables	395
[vlog] Verilog compiler control variables	397
[vsim] simulator control variables	398
[lmc] Logic Modeling variables	401
Setting variables in INI / MPF files	402
[Project] project file section (MPF files only)	402
Reading variable values from the INI file	403
Variable functions	403
More preferences	406
Preference variables located in TCL files	406
User-defined variables	407
Viewing the default preference file (pref.tcl)	407

Preference variable arrays	407
Main window preference variables	411
The Main window uses preference variables similar to other ModelSim window to control colors and fonts. The variables below control some additional functions.	411
Individual preference variables	411
The addons variable	412
Setting Tcl preference variables	412
More preferences	419
Preference variable loading order	419
Simulator state variables	420
Referencing simulator state variables	421

C - ModelSim Shortcuts (423)

Wave window keyboard shortcuts	423
.	424
List window keyboard shortcuts	424
Command shortcuts	425
Command history shortcuts	425
.	425
Editing the command line, the current source file, and notepads	426
Right mouse button	427

D - Using the FLEXlm License Manager (429)

Starting the license server daemon	430
Locating the license file	430
Controlling the license file search	430
Manual start	430
Automatic start at boot time	431
What to do if another application uses FLEXlm	431
Format of the license file	432
Format of the daemon options file	432
License administration tools	434
lmstat	434
lmdown	435
lmremove	435
lmreread	436

E - Tips and Techniques (437)

How to use checkpoint/restore	438
The difference between checkpoint/restore and restarting	439
Using macros with restart and checkpoint/restore	439
Running command-line and batch-mode simulations	440
Command-line mode	441
Batch mode	441
Passing parameters to macros	442
Source code security and -nodebug	443
Saving and viewing waveforms	445
Setting up libraries for group use	445
Bus contention checking	446
Bus float checking	447
Design stability checking	447
Toggle checking	447
Detecting infinite zero-delay loops	448
Referencing source files with location maps	449
Using location mapping	449
Pathname syntax	450
How location mapping works	450
Mapping with Tcl variables	450
Accelerate simulation by locking memory under HP-UX 10.2	451
Modeling memory in VHDL	452
Setting up a List trigger with Expression Builder	456

F - What's new in ModelSim 5.3 (459)

New features	459
GUI changes and new features	461
New file types	461
Command and variable changes	462
Documentation changes	463
Find 5.2 features in the 5.3 interface	465
Main window toolbar buttons and menu selections	466

Main window menu changes	467
File menu selections	467
View menu selections	469
Library menu selections	470
Run menu selections	471
Options menu selections	472
Window menu selections	473
Help menu selections	474
List window menu changes	475
Prop menu selections	475
Process window menu changes	476
File menu selections	476
Signals window menu changes	477
Edit menu selections	477
Structure window menu changes	478
File menu selections	478
Edit menu selections	479
Variables window menu changes	480
Edit menu selections	480
Wave window menu changes	481
File menu selections	481
Edit menu selections	482
Prop menu selections	483
What's new for ModelSim 5.3a	484

Index (485)

1 - Introduction

Chapter contents

Standards supported 22
Assumptions 23
Sections in this document 24
Text conventions 26
What is an "HDL item" 26
Online References - www.model.com 29

Model Technology's *ModelSim* EE/PLUS simulation system provides a full VHDL, Verilog and mixed-HDL simulation environment for UNIX, Microsoft Windows NT 4.0, and Windows 95/98. *ModelSim* EE/PLUS's single-kernel simulator allows you to efficiently simulate mixed VHDL and Verilog designs within one consistent interface.

ModelSim software versions documented in this manual

This documentation was written to support *ModelSim* EE/PLUS 5.3 for UNIX, Microsoft Windows NT 4.0, and Windows 95/98. If the *ModelSim* software you are using is a later release, check the README file that accompanied the software. Any supplemental information will be there. The online documentation included with *ModelSim* may be more current than the printed version as well.

Although this document covers both VHDL and Verilog simulation, you will find it a useful reference even if your design work is limited to a single HDL.

Performance tools included with *ModelSim* SE

ModelSim Special Edition (SE) features are also documented in this manual. *ModelSim* SE delivers *ModelSim* EE\LNL language flexibility plus the following performance tools. (*ModelSim* LNL supports VHDL **or** Verilog simulation.)

- [ModelSim SE Performance Analyzer](#) (7-119)
Easily identify areas in your simulation where performance can be improved.
- [ModelSim SE Code Coverage](#) (8-133)
Gives you graphical and report file feedback on how the source code is being executed.

ModelSim's graphic interface

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

- SPARCstation with OpenWindows or OSF/Motif
- IBM RISC System/6000 with OSF/Motif
- Hewlett-Packard HP 9000 Series 700 with HP VUE or OSF/Motif
- Microsoft Windows NT and Windows 95/98

Because ModelSim's graphic interface is based on Tcl/Tk, you also have the tools to build your own simulation environment. Easily accessible ["Preference variables located in INI and MPF files"](#) (B-394), and ["Graphic interface commands"](#) (10-277) give you control over the use and placement of windows, menus, menu options and buttons.

See ["Tcl and ModelSim"](#) (16-367) for more information on Tcl.

For an in-depth look at ModelSim's graphic interface see, *Chapter 10 - ModelSim EE Graphic Interface*.

Standards supported

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

ModelSim Verilog is based on the IEEE Std 1364-1995 *Standard Hardware Description Language Based on the Verilog Hardware Description Language*. 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 EE users.

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

Assumptions

We assume that you are familiar with the use of your operating system. You should be familiar with the window management functions of your graphic interface: either OpenWindows, OSF/Motif, or Microsoft Windows NT/95/98.

We also assume that you have a working knowledge of VHDL and Verilog. Although *ModelSim* is an excellent tool to use while learning HDL concepts and practices, this document is not written to support that goal. If you need more information about HDLs, check out our [Online References - www.model.com](http://www.model.com) (1-29).

Finally, we make the assumption that you have worked the appropriate lessons in the *ModelSim Tutorial* or the *ModelSim Quick Start* and are therefore familiar with the basic functionality of *ModelSim*.

The *ModelSim Tutorial* and *Quick Start* are both available from the *ModelSim Help* menu. The *ModelSim Tutorial* is also available from the Support page of our web site: www.model.com.

For installation instructions please refer to the *Start Here for ModelSim* guide that was shipped with the *ModelSim* CD.

Start Here may also be downloaded from our website: www.model.com.

Sections in this document

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

2 - Design Libraries (2-31)

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.

3 - Projects and system initialization (3-43)

This chapter provides a definition of a *ModelSim* "project" and discusses the use of a new file extension for project files.

4 - VHDL Simulation (4-51)

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

5 - Verilog Simulation (5-63)

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

6 - Mixed VHDL and Verilog Designs (6-107)

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

7 - ModelSim SE Performance Analyzer (7-119)

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

8 - ModelSim SE Code Coverage (8-133)

This chapter describes the Code Coverage feature of *ModelSim* EE Special Edition. Code Coverage gives you graphical and report file feedback on how the source code is being executed.

9 - Multiple logfiles, datasets and virtuals (9-139)

This chapter describes logfiles, datasets, and virtuals - new methods for viewing and organizing simulation data in *ModelSim*.

10 - ModelSim EE Graphic Interface (10-149)

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

-
- 11 - [Standard Delay Format \(SDF\) Timing Annotation](#) (11-281)
This chapter discusses ModelSim's implementation of SDF (Standard Delay Format) timing annotation. Included are sections on VITAL SDF and Verilog SDF, plus troubleshooting.
 - 12 - [VHDL Foreign Language Interface](#) (12-297)
This chapter covers ModelSim's VHDL FLI (Foreign Language Interface).
 - 13 - [Value Change Dump \(VCD\) Files](#) (13-337)
This chapter explains Model Technology's Verilog VCD implementation for ModelSim. The VCD usage is extended to include VHDL designs.
 - 14 - [Logic Modeling SmartModels](#) (14-347)
This chapter describes the use of the SmartModel Library, SmartModel Windows with ModelSim.
 - 15 - [Logic Modeling Hardware Models](#) (15-359)
This chapter describes the use the Logic Modeling Hardware Modeler with ModelSim.
 - 16 - [Tcl and ModelSim](#) (16-367)
This chapter provides an overview of Tcl (tool command language) as used with ModelSim. Additional Tcl and Tk (Tcl's toolkit) can be found through several [Tcl online references](#) (16-368).
 - A - [Technical Support, Updates, and Licensing](#) (A-383)
How and where to get tech support, updates and licensing for ModelSim. Links to the Model Technology web site; a collection of references to books, organizations, and companies involved in EDA and simulation.
 - B - [ModelSim Variables](#) (B-389)
This appendix environment, system and preference variables used in ModelSim.
 - C - [ModelSim Shortcuts](#) (C-423)
A collection of ModelSim keyboard and mouse shortcuts.
 - D - [Using the FLEXlm License Manager](#) (D-429)
This appendix covers Model Technology's application of FLEXlm for ModelSim licensing.
 - E - [Tips and Techniques](#) (E-437)
An extended collection of ModelSim usage examples taken from our manuals, and tech support solutions.
 - F - [What's new in ModelSim 5.3](#) (F-459)
A listing of new features and changes in the current version of ModelSim.

Command reference

The complete command reference for all ModelSim commands is located in the [ModelSim EE/SE Command Reference](#). Command Reference cross reference page numbers are prefixed with "CR", i.e., "[ModelSim Commands](#)" (CR-9).

Text conventions

Text conventions used in this manual include:

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

What is an "HDL item"

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

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

Where to find our documentation

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

Document	Format	How to get it
<i>Start Here for ModelSim EE</i> (installation & support reference)	paper	shipped with ModelSim
	PDF online	from the ModelSim Help menu (select EE Documentation > Licensing and Support), or find <i>ee_start.pdf</i> in the \modeltech\docs directory; also available from the Support page of our web site: www.model.com
<i>ModelSim EE Quick Guide</i> (command and feature quick-reference)	paper	shipped with ModelSim
	PDF online	from the ModelSim Help menu (select EE Documentation > EE Quick Guide), or find <i>ee_guide.pdf</i> in the \modeltech\docs directory; also available from the Support page of our web site: www.model.com
<i>ModelSim EE Tutorial</i>	PDF online	from the ModelSim Help menu (select EE Documentation > EE Tutorial), or find <i>ee_tutor.pdf</i> in the /modeltech/docs directory on the CD-ROM, or hard drive after installation, also available from the Support page of our web site: www.model.com
	paper	first two copies free with request cards in <i>Start Here</i> document; additional copies at \$50 each (for customers with current maintenance)
<i>ModelSim EE User's Manual</i>	PDF online	from the ModelSim Help menu (select EE Documentation > EE User's Manual), or find <i>ee_man.pdf</i> in the /modeltech/docs directory on the CD-ROM, or hard drive after installation
	paper	first two copies free with request cards in <i>Start Here</i> document; additional copies at \$50 each (for customers with current maintenance)

Document	Format	How to get it
<i>ModelSim EE Command Reference</i>	PDF online	from the ModelSim Help menu (select EE Documentation > EE Command Reference), or find <i>ee_cmds.pdf</i> in the <i>/modeltech/docs</i> directory on the CD-ROM, or hard drive after installation
	paper	first two copies free with request cards in <i>Start Here</i> document; additional copies at \$50 each (for customers with current maintenance)
Tcl man pages (Tcl manual)	HTML	use the Main window menu selection: Help > Tcl Man Pages , or find <i>contents.html</i> in <i>\modeltech\docs\html</i>
tech notes	ASCII	from the ModelSim Help menu, or located in the <i>\modeltech\docs\technotes</i> directory after installation

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 <http://www.adobe.com>. Be sure to download the Acrobat Reader with Search to take advantage of the index file supplied with our documentation; the index makes searching for key words much faster.

Online References - www.model.com

The Model Technology web site includes links to support, software downloads, and many EDA information sources. Check the links below for the most current information.

News

Current news of Model Technology within the EDA industry.

model.com/news/index.html

Partners

Model Technology's value added partners, OEM partners, FPGA partners, ASIC partners, and training partners.

model.com/partners/index.html

Products

A complete collection of Model Technology product information.

model.com/products/index.html

Resources

Books, Tcl/Tk links, technical notes, and training information.

model.com/resources/index.html

Sales

Locate ModelSim sales contacts anywhere in the world.

model.com/sales/index.html

Support

Model Technology email support and software downloads.

model.com/support/index.html

Comments

Comments and questions about this manual and *ModelSim* software are welcome.
Call, write, fax or email:

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

phone: 503-641-1340

fax: 503-526-5410

email: manuals@model.com

home page: <http://www.model.com>

2 - Design Libraries

Chapter contents

Design library contents 32
Design unit information. 32
Design library types 32
Library management commands. 33
Working with design libraries 33
Creating a library 33
Viewing and deleting library contents 35
Assigning a logical name to a design library 37
Moving a library 39
Specifying the resource libraries 40
VHDL resource libraries 40
Predefined libraries 40
Alternate IEEE libraries supplied 41
Rebuilding supplied libraries 41
Regenerating your design libraries 41
Verilog resource libraries 42

VHDL has a concept of a *library*, which is an object that contains compiled design units; libraries are given names so they may be referenced. Verilog designs simulated within *ModelSim* are compiled into libraries as well. 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. For additional information on *ModelSim* "[Library management commands](#)" (2-33) introduced in this chapter see "[ModelSim Commands](#)" (CR-9).

Design library contents

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

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

Design unit information

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

- retargetable, executable code
- debugging information
- dependency information

Design library types

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

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

Library management commands

These library management commands are available from the UNIX/DOS prompt, or the Transcript command line, or from the ModelSim graphic interface. Only brief descriptions are provided here; for more information and command syntax see "ModelSim Commands" (CR-9).

Command	Description
vdel (CR-175)	deletes a design unit from a specified library
vlib (CR-198)	selectively lists the contents of a library.
vlib (CR-198)	creates a design library
vmake (CR-205)	prints a makefile for a library to the standard output
vmap (CR-207)	defines or displays a mapping between a logical library name and a directory by modifying the <i>modelsim.ini</i> file; may also be invoked from the Main window command line

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

Before you run the compiler, you need to create a working design library. This can be done from either the UNIX/DOS command line or from the ModelSim graphic interface.

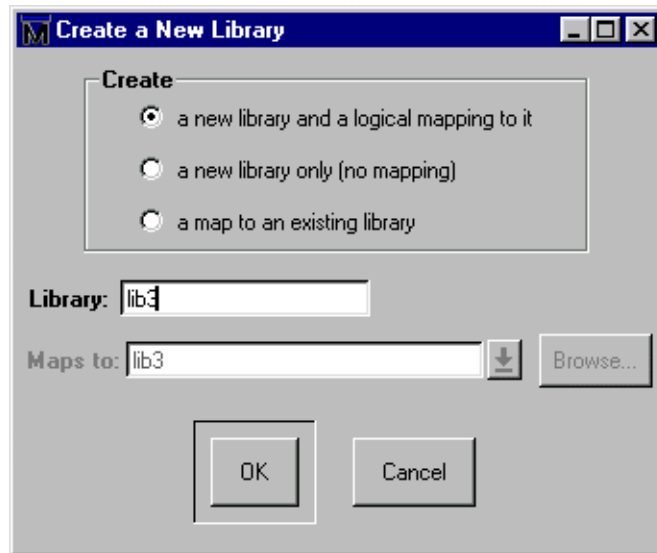
Creating a working library from the command line

From either the UNIX/DOS prompt, or the ModelSim prompt, use this [vlib](#) command (CR-198):

```
vlib <directory_pathname>
```

Creating a working library with the graphic interface

To create a new library with the ModelSim graphic interface, use the Main window menu selection: **Design > Create a New Library**. This brings up a dialog box that allows you to specify the library name along with several mapping options.



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

Create

- **a new library and a logical mapping to it**
Type the new library name into the **Library** field. This creates a library sub-directory in your current working directory, initially mapped to itself. Once created, the mapped library is easily remapped to a different library.
- **a new library only (no mapping)**
Type the new library name into the **Library** field. This creates a library sub-directory in your current working directory.
- **a map to an existing library**
Type the new library name into the **Library** field, then type into the **Maps to** field or **Browse** to select a library name for the mapping.

and

- **Library**
Type the new library name into this field.

- **Maps to**

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

When you click **OK**, ModelSim creates the specified library directory and writes a specially-formatted file named `_info` into that directory. The `_info` file must remain in the directory to distinguish it as a ModelSim library.

If a mapping option is selected, a map entry is written to the `modelsim.ini` file in the [Library] section. See "[Library] library path variables" (B-395) for more information.

Note: It is important to 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-198). Do not create libraries using UNIX or Windows.

Viewing and deleting library contents

The contents of a design library can be viewed or deleted using either the command line or graphic interface.

Viewing and deleting library contents from the command line

Use the **vlib** command (CR-198) to view the contents of a specified library (the contents of the **work** library are shown if no library is specified). Its syntax is:

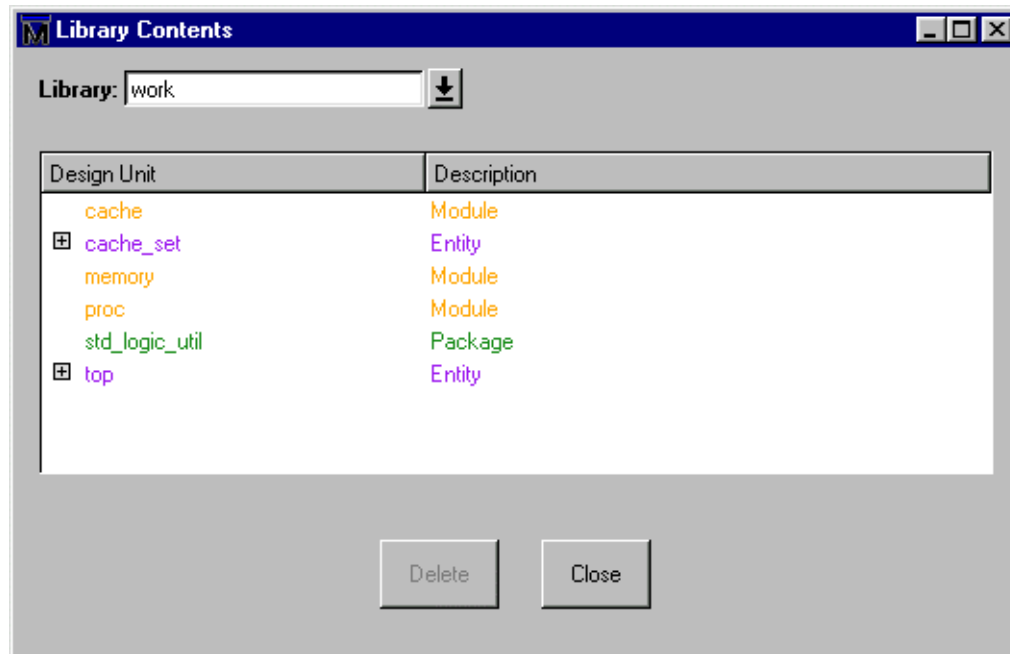
```
vdir -lib <library_name>
```

Use the **vdel** command (CR-175) to delete an entire library or a design unit from a specified library (the design unit is deleted from the **work** library if no library name is specified). Its syntax is:

```
vdel -lib <library_name> <design_unit>
```

Viewing and deleting library contents with the graphic interface

Selecting **Design > View Library Contents...** allows you to view the design units (configurations, modules, packages, entities, and architectures) in the current library and delete selected design units.



The **Library Contents** dialog box includes these options:

- **Library**
Select the library you wish to view from the drop-down list.
- **DesignUnit/Description list**
Entity/architecture pairs are indicated by a box prefix; select a plus (+) box to view the associated architecture, or select a minus (–) box to hide the architecture.

You can delete a package, configuration, or entity by selecting it and clicking **Delete**. This will remove the design unit from the library. If you delete an entity that has one or more architectures, the entity and all its associated architectures will be deleted.

You can also delete an architecture without deleting its associated entity. Just select the desired architecture name and click **Delete**. You are prompted for confirmation before any design unit is actually deleted.

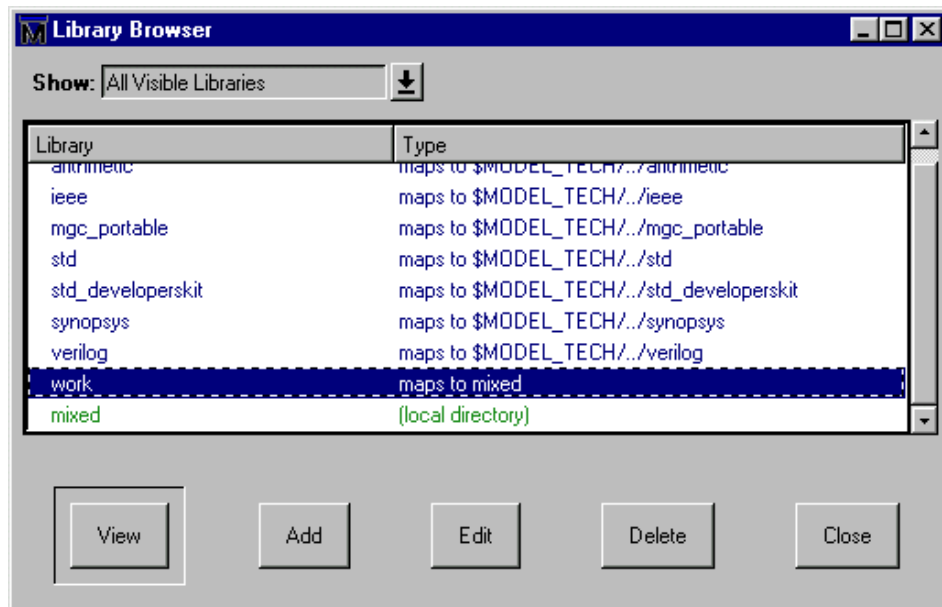
Assigning a logical name to a design library

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 graphic interface, a command or the project file to assign a logical name to a design library.

Library mappings with the GUI

To associate a logical name with a library, you select the **Design > Browse Libraries** command. This brings up a dialog box that allows you to view, add, edit, and delete mappings, as shown below:



The **Library Browser** dialog box includes these options:

- **Show**
Choose the mapping and library scope to view from the drop-down list.

- **Library/Type list**

To view the contents of a library

Select the library, then click the **View** button. This brings up the **Library Contents** (2-35) dialog box. From there you can also delete design units from the library.

To create a new library mapping

Click the **Add** button. This brings up **Create a New Library** (2-33) dialog box that allows you to enter a new logical library name and the pathname to which it is to be mapped.

It is possible to enter the name of a non-existent directory, but the specified directory must exist as a ModelSim library before you can compile design units into it. When you complete all your mapping changes and click the **OK** button in the Library Browser dialog box, ModelSim will issue a warning if any mappings are unresolved.

To edit an existing library mapping

Select the desired mapping entry, then click the **Edit** button. This brings up a dialog box that allows you to modify the logical library name and the pathname to which it is mapped. Selecting **Delete** removes an existing library mapping, but it does not delete the library. The library can be deleted with this **vdel** command (CR-175):

```
vdel -lib <library_name> -all
```

Library mapping from the command line

You can issue a ModelSim/PLUS command to set the mapping between a logical library name and a directory; its form is:

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

This command may be invoked from either a UNIX/DOS prompt or from the command line within ModelSim.

When you use **vmap** (CR-207) this way you are modifying the *modelsim.ini* file. You can also modify *modelsim.ini* manually by adding a mapping line. To do this, edit the *modelsim.ini* file using any 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-207) 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.

See also

See "[ModelSim Commands](#)" (CR-9) for more information about the library management commands, "[ModelSim EE Graphic Interface](#)" (10-149) for more information about the graphical user interface, and "[Projects and system initialization](#)" (3-43) for more information about the *modelsim.ini* file.

Moving a library

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

Specifying the resource libraries

VHDL resource libraries

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

Note that the **library** clause is not used to specify the working library into which the design unit is placed after compilation; the **vdel** command (CR-175) 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 **vdel -work** and specify the name of the desired target library.

Predefined libraries

Certain resource libraries are predefined in standard VHDL. The library named **std** contains the packages **standard** and **textio**, which should not be modified. The contents of these packages and other aspects of the predefined language environment are documented in the *IEEE Standard VHDL Language Reference Manual, Std 1076-1987* and *ANSI/IEEE Std 1076-1993*. See also, "[Using the TextIO package](#)" (4-55).

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

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

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

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

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

Alternate IEEE libraries supplied

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

- *ieeepure*
Contains only IEEE approved std_logic_1164 packages (accelerated for VSIM).
- *ieee*
Contains precompiled Synopsys and IEEE arithmetic packages for the std_logic base type, which have been accelerated by Model Technology.

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

Rebuilding supplied libraries

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

```
do rebldlibs.do
```

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

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

Note: Because accelerated subprograms require attributes that are available only under the 1993 standard, many of the libraries are built using **vdcl** (CR-175) with the **-93** option.

Regenerating your design libraries

Depending on your current ModelSim version, you may need to regenerate your design libraries before running a simulation. Check the installation README file to see if your libraries require an update. You can easily regenerate your design libraries with **-refresh**. You must use **vcom** (CR-169) with the **-refresh** option to update the VHDL design units in a library, and **vlog** (CR-199) with the **-refresh** option to update Verilog design units. By default, the work library is updated; use

-work <library> to update a different library. For example, if you have a library named **mylib** that contains both VHDL and Verilog design units:

```
vcom -work mylib -refresh  
vlog -work mylib -refresh
```

An important feature of **-refresh** is that it rebuilds the library image without using source code. This means that models delivered as compiled libraries without source code can be rebuilt for a specific release of *ModelSim* (4.6 and later only). In general, this works for moving forwards or backwards on a release. Moving backwards on a release may not work if the models used compiler switches or directives (Verilog only) that do not exist in the older release.

Note: As in the example above, you will need to use **vcom** for VHDL and **vlog** for Verilog design units. Also, 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.

Verilog resource libraries

ModelSim supports and encourages separate compilation of distinct portions of a Verilog design. The **vlog** (CR-199) compiler is used to compile one or more source files into a specified library. The library thus contains pre-compiled modules and UDPs (and, perhaps, VHDL design units) that are drawn from by the simulator as it loads the design. See "[Library usage](#)" (5-68).

3 - Projects and system initialization

Chapter contents

What is a project?	. 44
A new file extension.	. 44
INI and MPF file comparison	. 44
The [Project] section in the .mpf file	. 45
Project operations	. 45
Creating a Project	. 46
Working with a Project	. 49
Open a project	. 49
Compile a project	. 49
Simulating a project.	. 49
Modifying a project.	. 49
The project command	. 50

This chapter provides a definition of a ModelSim "project" and discusses the use of a new file extension for project files.

With the 5.3 release, ModelSim incorporates the file extension *.mpf* to denote project files. In past releases the *modelsim.ini* file (the system initialization file) was used as the project file.

Note: If you are looking for information on project file variables, please see *Appendix B - ModelSim Variables*.

What is a project?

A project is a collection entity for an HDL design under specification or test. At a minimum, it has a root directory, a work library and session state which is stored in a .mpf file located in the project's root directory. A project may also consist of:

- HDL source files
- subdirectories
- Local libraries
- References to global libraries

A new file extension

Why create a new file extension instead of using the .ini extension?

Project files with the new .mpf file extension contain some information that is specific to a given design and project directory. For this reason, usage of a .mpf file is associated with current working directories residing in the project's directory tree.

On Windows systems the .ini file extension is used extensively to represent configuration files for many applications, including the OS. By changing the project file extension to .mpf, a new file type can be defined for Windows systems that won't be confused with other configuration files. Please note that old .ini files are still supported.

INI and MPF file comparison

What is the difference between old project (.ini) files and new project (.mpf) files?

- A .ini file specifies initial tool settings and is fully supported outside of a project.
- By convention the new project files will have a .mpf extension.
- New features of the project file are most useful when used in conjunction with the ModelSim graphical user interface (GUI).
- A .mpf project file is specific to a given work session and may include the settings from a .ini file.
- A .mpf project file is located in the project working directory. This ensures that the path to a .mpf file will be <some_dir_path>/<project_name>/<project_name>.mpf

- A .mpf project file may be updated with current tool settings, whereas a .ini file is used as initial tool defaults. A .mpf project file is kept up to date with changes to project settings.

The [Project] section in the .mpf file

Sections within the .ini and .mpf files contain variable settings for libraries, the simulator, and compilers. The .mpf file includes an additional [Project] section located at the bottom of the file that contains one or more variables. These variables can be found in *Chapter B - ModelSim Variables*.

Project operations

The ModelSim user has the ability to perform the following operations on a project:

Create: **File > New > New Project**

New - Inherit default settings from the current .ini file (must specify project name and working directory). Creates a fresh project file. Opens the new project.

Copy - Use an existing project, but change working directory. Copies all dependant files/libraries that are specified relative to the working directory. Absolute library paths are unchanged in the copied project file. Opens the new project.

Open: **File > Open > Open Project**

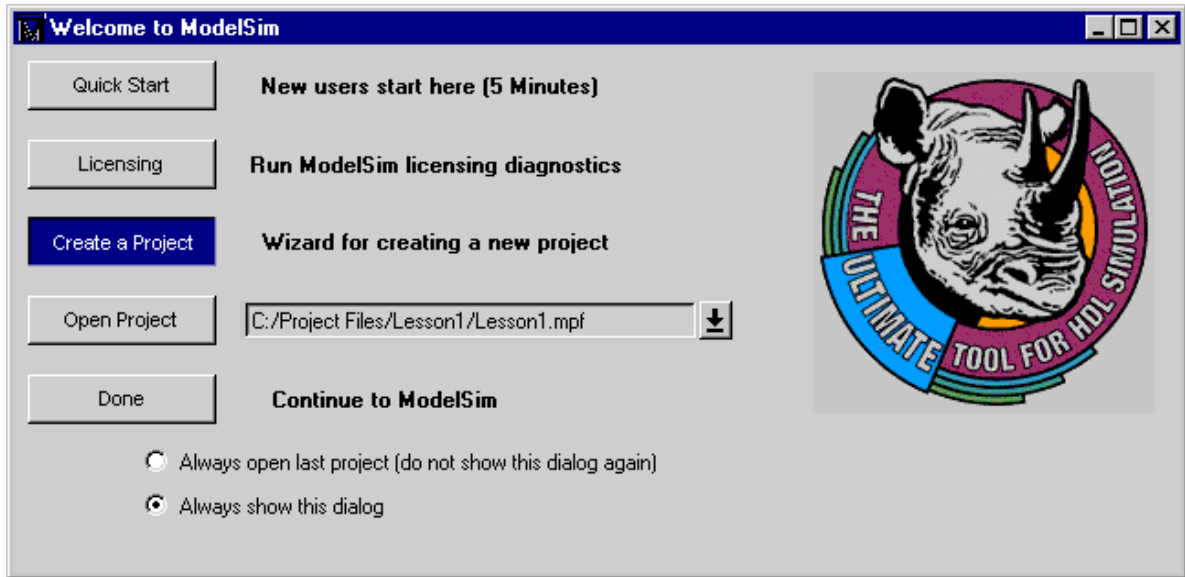
Open an existing project (change working directory, read settings from project file).

Delete: **File > Delete > Delete Project**

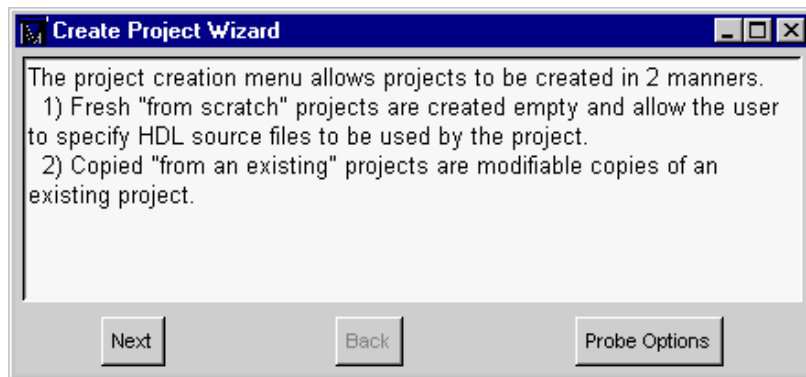
Deletes a specified project file. User must confirm relative file references and working directories. Absolute Library paths can also be optionally deleted.

Creating a Project

- 1 To get started fast, select the **Create a Project** button from the Welcome to ModelSim screen that opens the first time you start ModelSim 5.3. If this screen is not available, you can enable it by selecting **Help > Enable Welcome** from the Main window.



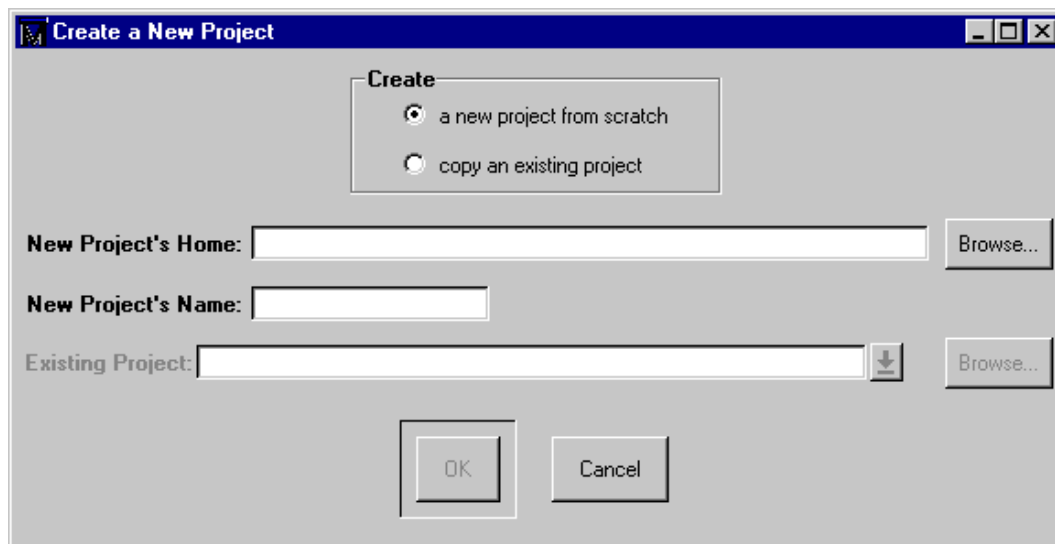
Clicking the **Create a Project** button opens the Create a New Project dialog box and a project creation wizard. The wizard guides you through each step of creating a new project, helping you create and load the project and providing the option of entering Verilog or VHDL descriptions.



Note: The Probe Options button allows you to probe the options within the Create a New Project dialog box. The Create Project Wizard displays option information as the cursor moves over each feature.

The Create a New Project dialog box can also be accessed by selecting **File > New > New Project** from the ModelSim Main window.

In the Create a New Project dialog box, you can elect to create a new project from scratch or copy an existing project.



If you select "create a new project from scratch," then:

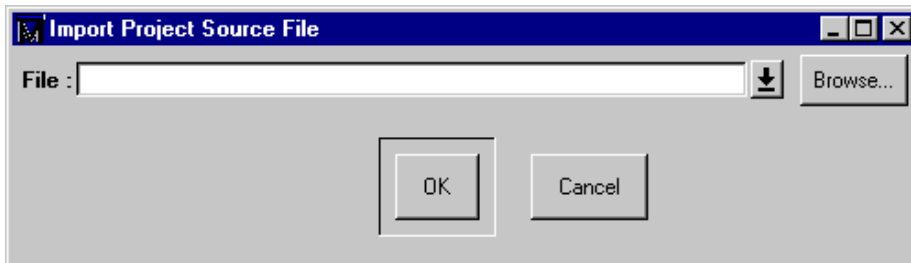
- 2 Specify the "New Project's Home," which is the directory under which this project's directory tree will reside.
- 3 Specify the "New Project's Name," which will act as the project's directory name. It is recommended that a unique name be given to each project.

If you select "copy an existing project," then:

- 4 Specify an "Existing Project" name, which is the full path to an existing project's .mpf file.

Note: A project's MPF file is always located in the project's directory.

Once you have specified enough information for the project creation. The OK button is selectable. Selecting OK causes the project directory to be created with a default working library. Once created the project is opened for use and the project wizard prompts you for the entry of a HDL source file.



Entry of a HDL source file name opens an editor session on the empty file. The source file must reside in the project's directory tree. When the editor session is complete you will be prompted to add the HDL source file to project's source list.

This completes the process of creating a project. The project will be open for use in the Main window. You can elect to leave *ModelSim* or edit the open project's HDL components until the project is completely specified and all files compile into libraries local to the project.

When you leave a *ModelSim* session, *ModelSim* will remember the last opened project so it can be reopened for your next session by simply clicking **Open Project** in the Welcome to *ModelSim* screen.

Working with a Project

Open a project

First, you must have a project open to work with it. To open a project select **File > Open > Open Project** from the Main window (cd'ing into projects directory won't work).

Once you have opened a project you can create HDL source files by selecting **File > New > New Source** from the Main window. When you create HDL files in the project's root directory you are prompted to add them to the project. HDL files for a given project must reside at or below the project's root directory.

Compile a project



To compile your project's HDL description with the project open, select **Design > Compile Project** from the Main window, or click the Compile icon, and select the files you want to compile. Each file will be compiled into your project's work library. Click Done when you are finished.

Simulating a project



To simulate an open project, select **Design > Load New Design** from the Main window or click the Load Design icon. On the Design tab of this menu you specify top level design unit for your project. On the VHDL and Verilog tabs you specify HDL specific simulator settings (these are described in the VSIM portion of the Reference Manual). On the SDF tab you can specify settings relating to the annotation of design timing from an SDF file (optional).

Modifying a project

There are four types of project settings that can be modified, each is modified with a different action..

- 1 Project-wide settings describe the make up of the project. These settings are changed from the **Options > Edit Project** pull down menu.
- 2 Project compiler settings specify HDL compiler options. These settings are changed from the **Options > Compile** pull down menu.

- 3 Project design simulation settings describe how a specific design unit is loaded and simulated. The simulation settings are edited from the **Design > Load New Design** pull down menu or by clicking the Load Design icon.
- 4 Project simulation settings describe simulation specific behavior. These settings are edited from the **Options > Simulation** pull down menu.

Project setting changes take place a different times. General Project editing (**Options > Edit_Project**) is disabled while compiling or simulating and project-wide settings become effected after their change. Compiler option editing (**Options > Compile**) is disabled while compiling and takes effect at the next compile. Project design simulation settings (**Design > Load New Design**) take effect at design load/reload. Simulation option edits (**Options > Simulation**) is enabled during simulation and take effect immediately.

Using project settings with the command line tools

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

The project command

The **project** command (CR-121) is used to perform common operations on new projects. The command is to be used outside of a simulation session. See "[ModelSim Commands](#)" (CR-9) for complete command details.

4 - VHDL Simulation

Chapter contents

Compiling VHDL designs 52
Creating a design library 52
Invoking the VHDL compiler 52
Dependency checking 53
Simulating VHDL designs 53
Invoking the simulator from the Main window. 53
Invoking Code Coverage with vsim 54
Using the TextIO package 55
Syntax for file declaration 55
Using STD_INPUT and STD_OUTPUT within ModelSim 56
TextIO implementation issues 56
Writing strings and aggregates 56
Reading and writing hexadecimal numbers 57
Dangling pointers 57
The ENDLINE function 58
The ENDFILE function. 58
Using alternative input/output files 58
Providing stimulus 58
Obtaining the VITAL specification and source code 59
VITAL packages. 59
ModelSim VITAL compliance 59
VITAL compliance checking 60
VITAL compliance warnings 60
Compiling and Simulating with accelerated VITAL packages 61

This chapter provides an overview of compilation and simulation for VHDL designs within the ModelSim/PLUS environment, using the TextIO package with ModelSim, and ModelSim's implementation of the VITAL (VHDL Initiative Towards ASIC Libraries) specification for ASIC modeling.

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

Compiling and simulating with the GUI

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

- [Compiling with the graphic interface](#) (10-244)
- [Simulating with the graphic interface](#) (10-251)

ModelSim variables

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

Compiling VHDL designs

Creating a design library

Before you can compile your design, you must create a library to store the compilation results. Use **vlib** (CR-198) to create a new library. For example:

```
vlib work
```

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

Note: The **work** library is actually a subdirectory named *work*. This subdirectory contains a special file named *_info*. Do not create libraries using UNIX, MS Windows, or DOS commands – always use the **vlib** command (CR-198).

See "[Design Libraries](#)" (2-31) for additional information on working with libraries.

Invoking the VHDL compiler

ModelSim compiles one or more VHDL design units with a single invocation of **vdel** (CR-175), the VHDL compiler. The design units are compiled in the order that they appear on the command line. For VHDL, the order of compilation is important – you must compile any entities or configurations before an architecture that references them.

You can simulate a design containing units written with both the 1076 -1987 and 1076 -1993 versions of VHDL. To do so you will need to compile units from each

VHDL version separately. The **vdel** (CR-175) command compiles units written with version 1076 -1987 by default; use the -93 option with **vdel** (CR-175) to compile units written with version 1076 -1993. You can also change the default by modifying the *modelsim.ini* file (see [Chapter 3 - Projects and system initialization](#) for more information).

Dependency checking

Dependent design units must be reanalyzed when the design units they depend on are changed in the library. **vdel** (CR-175) 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.

Simulating VHDL designs

After compiling the design units, you can proceed to simulate your designs with **vsim** (CR-208). This section includes a discussion of simulation from the UNIX or Windows/DOS command line. You can also use the graphic interface for simulation, see ["Simulating with the graphic interface"](#) (10-251).

Note: Simulation normally stops if a failure occurs, however, if a bounds check on a signal fails the simulator will continue running.

Invoking the simulator from the Main window

For VHDL, invoke **vsim** (CR-208) 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-208) on the entity **my_asic** and the architecture **structure**:

```
vsim my_asic structure
```

If a design unit name is not specified, **vsim** (CR-208) will present the **Load Design** dialog box from which you can choose a configuration or entity/architecture pair. See ["Simulating with the graphic interface"](#) (10-251) for more information.

Selecting the time resolution

The simulation time resolution is 1 ns by default. You can select a specific time resolution with the **vsim** (CR-208) **-t** option or from the **Load Design** dialog box. Available resolutions are: 1x, 10x or 100x of fs, ps, ns, us, ms, or sec.

For example, to run in picosecond resolution, or 10ps resolution respectively:

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

The default time resolution can also be changed by modifying the **resolution** (B-421) in the *modelsim.ini* file. You can view the current resolution by invoking the **report** command (CR-131) with the **simulator state** option.

See "[Projects and system initialization](#)" (3-43) for more information on modifying the *modelsim.ini* file.

vsim (CR-208) 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** (CR-208) with the **-sdfmin**, **-sdftyp** and **-sdfmax** options. Using the SDF file *fl.sdf* in the current work directory, the following invocation of **vsim** (CR-208) annotates maximum timing values for the design unit *my_asic*:

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

Timing check disabling

By default, the timing checks within VITAL models are enabled. They are also disabled with the **+notimingchecks** option.

For example:

```
vsim +notimingchecks topmod
```

Invoking Code Coverage with vsim

ModelSim's Code Coverage feature gives you graphical and report file feedback on how the source code is being executed. It allows line number execution statistics to be kept by the simulator. It can be used during any design phase and in all levels and types of designs. For complete details, see *Chapter 8 - ModelSim SE Code Coverage*.

To acquire code coverage statistics, the **-coverage** switch must be specified during the command-line invocation of the simulator.

```
vsim -coverage
```

This will allow you to use the various code coverage commands: **coverage clear** (CR-59), **coverage reload** (CR-60), and **coverage report** (CR-61).

Using the TextIO package

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

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

A simple example using the package TextIO is:

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

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

Syntax for file declaration

The VHDL'87 syntax for a file declaration is:

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

where "file_logical_name" must be a string expression.

The VHDL'93 syntax for a file declaration is:

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

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.

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

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

Using STD_INPUT and STD_OUTPUT within ModelSim

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

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

The standard VHDL'93 TextIO package contains these file declarations:

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

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

In ModelSim reading from the STD_INPUT file brings up a dialog box that allows you to enter text into the current buffer. The last line written to the STD_OUTPUT file appears as a prompt in this dialog box. Any text that is written to the STD_OUTPUT file is also echoed in the Transcript window.

TextIO implementation issues

Writing strings and aggregates

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

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

will cause the following error:

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

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

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

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

The following procedure call also generates an error:

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


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

There are two possible solutions to this problem:

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

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

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

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

The `WRITE_STRING` procedure simply defines the value to be a `STRING` and calls the `WRITE` procedure, but it serves as a shell around the `WRITE` procedure that solves the overloading problem. For further details, refer to the `WRITE_STRING` procedure in the `io_utils` package, which is located in the file `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 `io_utils.vhd`. To use these routines, compile the `io_utils` package and then include the following use clauses in your VHDL source code:

```
use std.textio.all;
use work.io_utils.all;
```

Dangling pointers

Dangling pointers are easily incurred when using the TextIO package, because `WRITELINE` deallocates the access type (pointer) that is passed to it. Following are examples of good and bad VHDL coding styles:

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

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

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

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

The ENDLINE function

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

Based on an ISAC-VASG recommendation the ENDLINE function has been removed from the TextIO package. The following test may be substituted for this function:

```
(L = NULL) OR (L'LENGTH = 0)
```

The ENDFILE function

In the *VHDL Language Reference Manuals, IEEE Std 1076-1987 and IEEE Std 1076-1993*, the ENDFILE function is listed as:

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

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

Using alternative input/output files

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

The VHDL'87 declaration is:

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

For VHDL'93 the declaration is:

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

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

Providing stimulus

You can create batch files containing **force** (CR-87) commands that provide stimulus for simulation. A VHDL test bench has been included with the ModelSim install files as an example; it illustrates how results can be generated by reading vectors from a file. Check for this file:

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

Obtaining the VITAL specification and source code

VITAL ASIC Modeling Specification

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

IEEE Customer Service
Hoes Lane
Tiscataway, NJ 08855-1331

Tel: (800)678-4333 ((908)562-5420 from outside the U.S.)

Fax: (908)981-9667

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`, or `/vital95` directories.

VITAL packages

VITAL v3.0 accelerated packages are pre-compiled into the **ieee** library in the installation directory.

Note: By default, ModelSim is optimized for VITAL v3.0. You can, however, revert to VITAL v2.2b by invoking **vsim** (CR-208) with the **-vital2.2b** option, and by mapping library **vital** to `<install_dir>/vital2.2b`.

ModelSim VITAL compliance

A simulator is VITAL compliant if it implements the SDF mapping and if it correctly simulates designs using the VITAL packages, as outlined in the VITAL Model Development Specification. ModelSim VSIM is compliant with the IEEE 1076.4 VITAL ASIC Modeling Specification. In addition, ModelSim accelerates the VITAL_Timing and VITAL_Primitives packages. The procedures in these packages are optimized and built into the simulator kernel. By default, **vsim** (CR-208) uses the optimized procedures. The optimized procedures are functionally equivalent to the IEEE 1076.4 VITAL ASIC Modeling Specification (VITAL v3.0).

VITAL compliance checking

Compliance checking is important in enabling VITAL acceleration; to qualify for global acceleration, an architecture must be VITAL-level-one compliant. **vdel** (CR-175) automatically checks for VITAL 3.0 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 **vdel** (CR-175) with the option **-novitalcheck**. It is, of course, possible to turn off compliance checking for VITAL 3.0 as well; 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 then the size of the DataIn parameter to VITALStateTable (1076.4 section 6.4.3.2.2)
- Signal q_w is read by the VITAL process but is NOT in the sensitivity list (1076.4 section 6.4.3)

The first two warnings are minor cases where the body of the VITAL 3.0 LRM is slightly stricter then the package portion of the LRM. Since either interpretation will provide the same simulation results, we chose to make these two cases just warnings.

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

You cannot control the visibility of VITAL compliance-check warnings in your **vdel** (CR-175) 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** (B-395) section.

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

Compiling and Simulating with accelerated VITAL packages

vdcl (CR-175) automatically recognizes that a VITAL function is being referenced from the **ieee** library and generates code to call the optimized built-in routines.

Optimization occurs on two levels:

- **VITAL Level-0 optimization**
This is a function-by-function optimization. It applies to all level-0 architectures, and any level-1 architectures that failed level-1 optimization.
- **VITAL Level-1 optimization**
Performs global optimization on a VITAL 3.0 level-1 architecture that passes the VITAL compliance checker. This is the default behavior.

Compiler options for VITAL optimization

Several **vdcl** (CR-175) options control and provide feedback on VITAL optimization:

-O0 | **-O4**

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

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

-debugVA

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

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

5 - Verilog Simulation

Chapter contents

Compilation 65
Incremental compilation 66
Library usage 68
Verilog-XL compatible compiler options 70
Verilog-XL 'uselib compiler directive 72
Simulation 74
Invoking the simulator 75
Simulation resolution limit 75
Event order issues 76
Verilog-XL compatible simulator options 78
Cell Libraries 80
SDF timing annotation 81
Delay modes 81
System Tasks 82
IEEE Std 1364-1995 system tasks 82
Verilog-XL compatible system tasks 85
Compiler Directives 87
IEEE Std 1364-1995 compiler directives 88
Verilog-XL compatible compiler directives 89
Using the Verilog PLI 90
Registering PLI applications 90
Compiling and linking PLI applications 92
The callback reason argument 97
The sizetf callback function. 99
Object handles 99
Third party PLI applications 99
Support for VHDL objects 100
IEEE Std 1364 ACC routines 101
IEEE Std 1364 TF routines 103
Verilog-XL compatible routines 104
PLI tracing 105

This chapter describes how to compile and simulate Verilog designs with ModelSim Verilog. ModelSim Verilog implements the Verilog language as defined by the IEEE Std 1364-1995, and it is recommended that you obtain this specification as a reference manual.

In addition to the functionality described in the IEEE Std 1364-1995, ModelSim Verilog includes the following features:

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

The following IEEE Std 1364-1995 functionality is not implemented in ModelSim Verilog:

- Array of instances (see section 7.1.5 in the IEEE Std 1364-1995).
- Verilog Procedural Interface (VPI) (see sections 22 and 23 in the IEEE Std 1364-1995).
- Macros (compiler ``define` directives) with arguments.

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

- [Compiling with the graphic interface](#) (10-244)
- [Simulating with the graphic interface](#) (10-251)

ModelSim variables

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

Compilation

Before you can simulate a Verilog design, you must first create a library and compile the Verilog source code into that library. This section provides detailed information on compiling Verilog designs. For information on creating a design library, see *Chapter 2 - Design Libraries*.

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

Contents of top.v:

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

Create the work library:

```
% vlib work
```

Compile the design:

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

```
Top level modules:
    top
```

View the contents of the work library (optional):

```
% vdir
MODULE top
```

Simulate the design:

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

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

Incremental compilation

By default, ModelSim Verilog supports incremental compilation of designs, thus saving compilation time when you modify your design. Unlike other Verilog simulators, there is no requirement that you compile the entire design in one invocation of the compiler (although, you may do so if desired).

You are not required to compile your design in any particular order because all module and UDP instantiations and external hierarchical references are resolved when the design is loaded by the simulator. Incremental compilation is made possible by deferring these bindings, and as a result some errors cannot be detected during compilation. Commonly, these errors include: modules that were referenced but not compiled, incorrect port connections, and incorrect hierarchical references.

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

Contents of top.v:

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

Contents of and2.v:

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

Contents of or2.v:

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

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

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

Top level modules:

    top
% vlog and2.v
-- Compiling module and2

Top level modules:

    and2
% vlog or2.v
-- Compiling module or2

Top level modules:

    or2
```

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

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

Top level modules:
    top
```

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

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

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

Top level modules:
    top
```

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

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

Top level modules:
    top
```

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

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

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** option to instruct the compiler to place the results in the asiclib library rather than the default work library.

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

- Search libraries specified with **-Lf** options in the order they appear on the command line.
- Search the library specified in "[Verilog-XL `uselib compiler directive](#)" (5-72).
- Search libraries specified with **-L** options 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.

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

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

Verilog-XL compatible compiler options

See [vlog](#) (CR-199) for a complete list of compiler options. The options described here are equivalent to Verilog-XL options. Many of these are provided to ease the porting of a design to ModelSim Verilog.

`+define+<macro_name>[=<macro_text>]`

This option allows you to define a macro from the command line that is equivalent to the following compiler directive:

```
'define <macro_name> <macro_text>
```

Multiple **+define** options are allowed on the command line. A command line macro overrides a macro of the same name defined with the `'define` compiler directive.

`+incdir+<directory>`

This option specifies directories to search for files included with **'include** compiler directives. By default, the current directory is searched first and then the directories specified by the **+incdir** options in the order they appear on the command line. You may specify multiple **+incdir** options as well as multiple directories separated by "+" in a single **+incdir** option.

`+delay_mode_distributed`

This option disables path delays in favor of distributed delays. See [Delay modes](#) (5-81) for details.

`+delay_mode_path`

This option sets distributed delays to zero in favor of path delays. See [Delay modes](#) (5-81) for details.

`+delay_mode_unit`

This option sets path delays to zero and non-zero distributed delays to one time unit. See [Delay modes](#) (5-81) for details.

`+delay_mode_zero`

This option sets path delays and distributed delays to zero. See [Delay modes](#) (5-81) for details.

`-f <filename>`

This option reads more command line arguments from the specified text file. Nesting of **-f** options is allowed.

+mindelays

This option selects minimum delays from the "min:typ:max" expressions. If preferred, you may defer delay selection until simulation time by specifying the same option on the simulator.

+typdelays

This option selects typical delays from the "min:typ:max" expressions. If preferred, you may defer delay selection until simulation time by specifying the same option on the simulator.

+maxdelays

This option selects maximum delays from the "min:typ:max" expressions. If preferred, you may defer delay selection until simulation time by specifying the same option on the simulator.

-u

This option treats all identifiers in the source code as all uppercase.

The following options support source libraries in the same manner as Verilog-XL. Note that these libraries are source libraries and are very different from the libraries that the ModelSim compiler uses to store compilation results. You may find it convenient to use these options if you are porting a design to ModelSim or if you are familiar with these options 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>

This option specifies a source library file containing module and UDP definitions. Modules and UDPs within the file are compiled only if they match previously unresolved references. Multiple **-v** options are allowed.

-y <directory>

This option specifies a source library directory containing module and UDP definitions. Files within this directory are compiled only if the file names match the names of previously unresolved references. Multiple **-y** options are allowed.

`+libext+<suffix>`

This option works in conjunction with the **-y** option. It specifies file extensions for the files in a source library directory. By default the compiler searches for files without extensions. If you specify the **+libext** option, then the compiler will search for a file with the suffix appended to an unresolved name. You may specify only one **+libext** option, but it may contain multiple suffixes separated by "+". The extensions are tried in the order they appear in the **+libext** option.

`+librescan`

This option changes how unresolved references are handled that are added while compiling a module or UDP from a source library. By default, the compiler attempts to resolve these references as it continues searching the source libraries. If you specify the **+librescan** option, then the new unresolved references are deferred until after the current pass through the source libraries. They are then resolved by searching the source libraries from the beginning in the order they are specified on the command line.

`+nolibcell`

By default, all modules compiled from a source library are treated as though they contain a **'celldefine** compiler directive. This option disables this default. The **'celldefine** directive only affects the PLI Access routines **acc_next_cell** and **acc_next_cell_load**.

The **-R** option is not a Verilog-XL option, but it is used by ModelSim Verilog to combine the compile and simulate phases together as you may be used to with Verilog-XL. It is not recommended that you regularly use this option because you will incur the unnecessary overhead of compiling your design for each simulation run. Mainly, it is provided to ease the transition to ModelSim Verilog.

`-R <simargs>`

This option instructs the compiler to invoke the simulator after compiling the design. The compiler automatically determines which top level modules are to be simulated. The command line arguments following **-R** are passed to the simulator, not the compiler. Place the **-R** option at the end of the command line or terminate the simulator command line arguments with a single "-" character to differentiate them from compiler command line arguments.

Verilog-XL 'uselib compiler directive

The **'uselib** compiler directive is an alternative source library management scheme to the **-v**, **-y**, and **+libext** compiler options. It has the advantage that a design may reference different modules having the same name. The **'uselib** compiler directive is not defined in the IEEE Std 1364-1995, but ModelSim supports it for compatibility with Verilog-XL.

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>
```

In Verilog-XL, the library references are equivalent to command line options 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 options:

```
-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.

For example, the following code fragment shows how two different modules that have the same name can be instantiated within the same design:

```
'uselib dir=/h/vendorA file=.v  
NAND2 u1(n1, n2, n3);  
'uselib dir=/h/vendorB file=.v  
NAND2 u2(n4, n5, n6);
```

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

ModelSim Verilog supports the **'uselib** directive in a different manner than Verilog-XL. The library files referenced by the **'uselib** directive are not automatically compiled by ModelSim Verilog. The reason for this is that an object library is not allowed to contain multiple modules having the same name, and the results of a single invocation of the compiler can be placed in only one object library. Because it is an important feature of **'uselib** to allow a design to reference multiple modules having the same name, independent compilation of the source libraries referenced by the **'uselib** directives is required. Each source library

should be compiled into its own object library. The compilation of the code containing the **'uselib** directives only records which object libraries to search for each module instantiation when the design is loaded by the simulator.

Because the **'uselib** directive is intended to reference source libraries, ModelSim Verilog must infer the object libraries from the library references. The rule is to assume an object library named **work** in the directory defined in the library reference **dir=<library_directory>** or the directory containing the file in the library reference **file=<library_file>**. The library reference **libext=<file_extension>** is ignored. For example, the following **'uselib** directives infer the same object library:

```
'uselib dir=/h/vendorA
'uselib file=/h/vendorA/libcells.v
```

In both cases ModelSim Verilog assumes that the library source is compiled into the object library **/h/vendorA/work**.

ModelSim Verilog also extends the **'uselib** directive to explicitly specify the object library with the library reference **lib=<library_name>**. For example,

```
'uselib lib=/h/vendorA/work
```

The library name can be a complete path to a library, or it can be a logical library name defined with the **vmap** command. Since this usage of **'uselib** is an extension, it may be desirable to qualify it with an **'ifdef** to make it portable to other Verilog systems. For example,

```
'ifdef MODEL_TECH
'uselib lib=vendorA
'else
'uselib dir=/h/vendorA libext=.v
'endif
```

The MODEL_TECH macro is automatically defined by the ModelSim compiler.

Simulation

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

Invoking the simulator

A Verilog design is ready for simulation after it has been compiled into one or more libraries. The simulator may then be invoked with the names of the top level modules (many designs contain only one top level module). For example, if your top level modules are "testbench" and "globals", then invoke the simulator as follows:

```
vsim testbench globals
```

If a top-level module name is not specified, VSIM will present the **Load Design** dialog box from which you can choose one or more top-level modules. See ["Simulating with the graphic interface"](#) (10-251) for more information.

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 up the ports and resolving hierarchical references. By default, all modules and UDPs are loaded from the library named **work**.

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 may also run for specific time periods, i.e., **run 100 ns**. Enter the **quit** command to exit the simulator.

Simulation 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 simulation resolution limit. The resolution limit defaults to the smallest time precision found among all of the 'timescale compiler directives in the design. The time precision is the second number in the 'timescale directive. For example, "10 ps" in the following:

```
`timescale 1 ns / 10 ps
```

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. If the design contains no 'timescale directives, then the resolution limit defaults to the "resolution" value specified in the *modelsim.ini* file (default is 1 ns). In any case, you may override the default resolution limit by specifying the -t option on the command line.

For example, to explicitly choose 100 ps resolution:

```
vsim -t 100ps top
```

This forces 100 ps resolution even if the design contains smaller time precisions. As a result, time values with finer precision are rounded to the nearest 100 ps.

Event order issues

The Verilog language is defined such that the simulator is not required to execute simultaneous events in any particular order. Unfortunately, some models are inadvertently written to rely on a particular event order, and these models may behave differently when ported to another Verilog simulator. A model with event order dependencies is ambiguous and should be corrected. For example, the following code is ambiguous:

```
module top;
    reg r;

    initial r = 0;
    initial r = 1;

    initial #10 $display(r);
endmodule
```

The value displayed for "r" depends on the order that the simulator executes the initial constructs that assign to "r". Conceptually, the initial constructs run concurrently and the simulator is allowed to execute them in any order. *ModelSim* Verilog executes the initial constructs in the order they appear in the module, and the value displayed for "r" is "1". Verilog-XL produces the same result, but a simulator that displays "0" is not incorrect because the code is ambiguous.

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

Tracking down event order dependencies is a tedious task, so *ModelSim* Verilog aids you with a couple of compiler options:

`-compat`

This option turns off optimizations that result in different event ordering than Verilog-XL. *ModelSim* Verilog generally duplicates Verilog-XL event ordering, but there are cases where it is inefficient to do so. Using this option does not help you find the event order dependencies, but it allows you to ignore them. Keep in mind that this option does not account for all event order discrepancies, and that using this option may degrade performance.

`-hazards`

This option detects event order hazards involving simultaneous reading and writing of the same register in concurrently executing processes. To enable hazard detection you must invoke `vlog` (CR-199) with the `-hazards` option when you compile your source code and you must also invoke `vsim` (CR-208) with the `-hazards` option when you simulate.

The `vsim` command (CR-208) 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. VSIM calls this a READ/WRITE hazard if it executed the read first.
- **WRITE/READ:**
Same as a READ/WRITE hazard except that VSIM executed the write first.

The `vsim` command (CR-208) 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-199) with the `-hazards` option when you compile your source code and you must also invoke `vsim` (CR-208) with the `-hazards` option when you simulate.

Limitations of hazard detection:

- Reads and writes involving bit and part selects of vectors are not considered for hazard detection. The overhead of tracking the overlap between the bit and part selects is too high.
- A WRITE/WRITE hazard is flagged even if the same value is written by both processes.
- A WRITE/READ or READ/WRITE hazard is flagged even if the write does not modify the variable's value.
- Glitches on nets caused by non-guaranteed event ordering are not detected.

Verilog-XL compatible simulator options

See [vsim](#) (CR-208) for a complete list of simulator options. The options described here are equivalent to Verilog-XL options. Many of these are provided to ease the porting of a design to ModelSim Verilog.

`+alt_path_delays`

Specify path delays operate in inertial mode by default. In inertial mode, a pending output transition is cancelled before a new output transition is scheduled, and the delay is selected based on the transition from the current value of the net to the new pending value. The result is that an output may have no more than one pending transition at a time, and that pulses narrower than the delay are filtered. The `+alt_path_delays` option modifies the inertial mode such that a delay is based on a transition from a pending output value rather than the current value of the net. This option has no effect in transport mode (see `+pulse_e` (5-79) and `+pulse_r` (5-79)).

`-l <filename>`

By default, the simulation log is written to the file "transcript". The `-l` option allows you to specify an alternate file.

`+maxdelays`

This option selects the maximum value in min:typ:max expressions. The default is the typical value. This option has no effect if the min:typ:max selection was determined at compile time.

`+mindelays`

This option selects the minimum value in min:typ:max expressions. The default is the typical value. This option has no effect if the min:typ:max selection was determined at compile time.

`+no_notifier`

This option disables the toggling of the notifier register argument of the timing check system tasks. By default, the notifier is toggled when there is a timing check violation, and the notifier usually causes a UDP to propagate an X. Therefore, the `+no_notifier` option suppresses X propagation on timing violations.

`+no_pulse_msg`

This option disables the warning message for specify path pulse errors. A path pulse error occurs when a pulse propagated through a path delay falls between the pulse rejection limit and pulse error limits set with the `+pulse_r` and `+pulse_e` options. A path pulse error results in a warning message, and the pulse is propagated as an X. The `+no_pulse_msg` option disables the warning message, but the X is still propagated.

`+no_tchk_msg`

This option disables error messages issued by timing check system tasks when timing check violations occur. However, notifier registers are still toggled any may result in the propagation of X's for timing check violations.

`+nosdfwarn`

This option disables warning messages during SDF annotation.

`+notimingchecks`

This option completely disables all timing check system tasks.

`+nowarn<mnemonic>`

This option disables the class of warning messages specified by `<mnemonic>`. This option only disables warning messages accompanied by a mnemonic enclosed in square brackets. For example,

```
# WARNING: test.v(2): [TFMPC] - Too few port connections.
```

This warning message can be disabled with the `+nowarnTFMPC` option.

`+pulse_e/<percent>`

This option controls how pulses are propagated through specify path delays, where `<percent>` is a number between 0 and 100 that specifies the error limit as a percentage of the path delay. A pulse greater than or equal to the error limit propagates to the output in transport mode (transport mode allows multiple pending transitions on an output). A pulse less than the error limit and greater than or equal to the rejection limit (see `+pulse_r` (5-79)) propagates to the output as an X. If the rejection limit is not specified, then it defaults to the error limit. For example, consider a path delay of 10 along with a `+pulse_e/80` option. The error limit is 80% of 10 and the rejection limit defaults to 80% of 10. This results in the propagation of pulses greater than or equal to 8, while all other pulses are filtered. Note that you can forcespecify path delays to operate in transport mode by using the `+pulse_e/0` option.

`+pulse_r/<percent>`

This option controls how pulses are propagated through specify path delays, where `<percent>` is a number between 0 and 100 that specifies the rejection limit as a percentage of the path delay. A pulse less than the rejection limit is suppressed from propagating to the output. If the error limit is not specified (see `+pulse_e` (5-79)), then it defaults to the rejection limit.

`+pulse_e_style_ondetect`

This option selects the "on detect" style of propagating pulse errors (see `+pulse_e` (5-79)). A pulse error propagates to the output as an X, and the "on detect" style is to schedule the X immediately, as soon as it has been detected that a pulse error

has occurred. The "on event" style is the default for propagating pulse errors (see **+pulse_e_style_onevent** (5-80)).

+pulse_e_style_onevent

This option selects the "on event" style of propagating pulse errors (see **+pulse_e** (5-79)). A pulse error propagates to the output as an X, and the "on event" style is to schedule the X to occur at the same time and for the same duration that the pulse would have occurred if it had propagated through normally. The "on event" style is the default for propagating pulse errors.

+sdf_nocheck_celltype

By default, the SDF annotator checks that the CELLTYPE name in the SDF file matches the module or primitive name for the CELL instance. It is an error if the names do not match. The **+sdf_nocheck_celltype** option disables this error check.

+sdf_verbose

This option displays a summary of the design objects annotated for each SDF file.

+transport_path_delays

By default, path delays operate in inertial mode (pulses smaller than the delay are filtered). The **+transport_path_delays** option selects transport mode for path delays. In transport mode, narrow pulses are propagated through path delays. Note that this option affects path delays only, and not primitives. Primitives always operate in inertial delay mode.

+typdelays

This option selects the typical value in min:typ:max expressions. The default is the typical value. This option has no effect if the min:typ:max selection was determined at compile time.

Cell Libraries

Model Technology is the first Verilog simulation vendor to pass the ASIC Council's Verilog test suite and achieve 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 the IEEE Std 1364-1995 along with some Verilog-XL compatible extensions.

SDF timing annotation

ModelSim Verilog supports timing annotation from Standard Delay Format (SDF) files. See *Chapter 11 - Standard Delay Format (SDF) Timing Annotation* for details.

Delay modes

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

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

    and(y, a, b);

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

In the above two-input "and" gate cell, the distributed delay for the "and" primitive is zero, and the actual delays observed on the module ports are taken from the path delays. This is typical for most cells, but a complex cell may require non-zero distributed delays to work properly. Even so, these delays are usually small enough that the path delays take priority over the distributed delays. The rule is that if a module contains both path delays and distributed delays, then the larger of the two delays for each path shall be used (as defined by the IEEE Std 1364-1995). This is the default behavior, but you can specify alternate delay modes with compiler directives and options. These options and directives are compatible with Verilog-XL. Compiler delay mode options take precedent 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 option or the **'delay_mode_distributed** compiler directive.

Path delay mode

In path delay mode the distributed delays are set to zero. Select this delay mode with the **+delay_mode_path** compiler option or the **'delay_mode_path** compiler directive.

Unit delay mode

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

System Tasks

The IEEE Std 1364-1995 defines many system tasks as part of the Verilog language, and ModelSim Verilog supports all of these along with several non-standard Verilog-XL system tasks. The system tasks listed in this chapter are built into the simulator, although some designs depend on user-defined system tasks implemented with the Programming Language Interface (PLI). If the simulator issues warnings regarding undefined system tasks, then it is likely that these system tasks are defined by a PLI application that must be loaded by the simulator.

IEEE Std 1364-1995 system tasks

The following system tasks are described in detail in the IEEE Std 1364-1995.

Timescale tasks

\$prinntimescale

\$timeformat

Simulator control tasks

\$finish

\$timeformat

Simulation time functions

\$realttime

\$stime

\$time

Probabilistic distribution functions

\$dist_chi_square

\$dist_erlang

\$dist_exponential

\$dist_normal

\$dist_poisson

\$dist_t

\$dist_uniform

Value change dump (VCD) file tasks

\$dumpall

\$dumpfile

\$dumpflush

\$dumplimit

\$dumpoff

\$dumpon

\$dumpvars

Conversion functions for reals

\$bitstoreal

\$itor

\$realtobits

\$rtoi

Display tasks

\$display

\$displayb

\$displayh

\$displayo

\$monitor

\$monitorb

\$monitorh

\$monitro

File I/O tasks

\$fclose

\$fdisplay

\$fdisplayb

\$fdisplayh

\$fdisplayo

\$fmonitor

\$fmonitorb

\$fmonitorh

PLA modeling tasks

\$async\$and\$array

\$async\$nand\$array

\$async\$or\$array

\$async\$nor\$array

\$async\$and\$plane

\$async\$nand\$plane

\$async\$or\$plane

\$async\$nor\$plane

Display tasks

\$monitoroff
\$monitoron
\$strobe
\$strobeb
\$strobeh
\$strobeo
\$write
\$writeb
\$writeh
\$writeo

File I/O tasks

\$fmonitoro
\$fopen
\$fstrobe
\$fstrobeb
\$fstrobeh
\$fstrobeo
\$fwrite
\$fwriteb
\$fwriteh
\$fwriteo
\$readmemb
\$readmemh

PLA modeling tasks

\$sync\$and\$array
\$sync\$nand\$array
\$sync\$or\$array
\$sync\$nor\$array
\$sync\$and\$plane
\$sync\$nand\$plane
\$sync\$or\$plane
\$sync\$nor\$plane

Timing check task

\$hold
\$nocharge
\$period
\$recovery
\$setup
\$setuphold
\$skew
\$width

Stochastic analysis tasks

\$q_add
\$q_exam
\$q_full
\$q_initialize
\$q_remove
\$random

Verilog-XL compatible system tasks

The following system tasks are provided for compatibility with Verilog-XL. Although they are not part of the IEEE standard, they are described in an annex of the IEEE Std 1364-1995.

```
$countdrivers
$getpattern
$sreadmemb
$sreadmemh
```

The following system tasks are also provided for compatibility with Verilog-XL, but they are not described in the IEEE Std 1364-1995.

```
$sdf_annotate
```

See: [The \\$sdf_annotate system task](#) (11-286)

```
$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");
```

```
$test$plusargs("plus argument")
```

This system function tests for the presence of a specific plus argument on the simulator's command line. It returns 1 if the plus argument is present; otherwise, it returns 0. For example, to test for **+verbose**:

```
if ($test$plusargs("verbose"))
    $display("Executing cycle 1");
```

```
$removal(reference_event, data_event, limit, [notifier])
```

The \$removal timing check issues a timing violation under the following condition:

$$0 < ((\text{time of reference event}) - (\text{time of data event})) < \text{limit}$$

```
$recrem(reference_event, data_event, recovery_limit, removal_limit, [notifier],
        [tstamp_cond], [tcheck_cond], {delayed-reference}, {delayed_data})
```

The \$recrem timing check is a combined \$recovery and \$removal timing check. It behaves very much like the \$setuphold timing check, along with the extensions for negative constraints and an alternate method of conditioning (see the description of \$setuphold below)

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

```
$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.

```
$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.

The following system tasks are Verilog-XL system tasks that are not implemented in ModelSim Verilog, but have equivalent simulator commands.

```
$input("filename")
```

This system task read command test from the specified filename. The equivalent simulator command is **do <filename>**.

```
$list[ (hierarchical_name) ]
```

This system task lists the source code for the specified scope. The equivalent functionality is provided by selecting a module in the graphic interface struture window. The corresponding source code is displayed in the source window.

```
$reset
```

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

```
$restart( "filename" )
```

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

```
$save( "filename" )
```

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

```
$scope(hierarchical_name)
```

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

```
$showscopes
```

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

```
$showvars
```

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

Compiler Directives

ModelSim Verilog supports all of the compiler directives defined in the IEEE Std 1364-1995 and some additional Verilog-XL compiler directives for compatibility.

Many of the compiler directives (such as **'define** and **'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 may be significant. For example, if you have a file that defines some common macros for the entire design, then you may 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 1364-1995):

```
'celldefine
'define_nettype
'delay_mode_distributed
'delay_mode_path
'delay_mode_unit
'delay_mode_zero
'timescale
'unconnected_drive
'uselib
```

ModelSim Verilog implicitly defines the following macro:

```
'define MODEL_TECH
```

IEEE Std 1364-1995 compiler directives

The following compiler directives are described in detail in the IEEE Std 1364-1995; however, the **'define** directive is not fully implemented as described - it does not support macro arguments.

```
'celldefine
'default_nettype
'define
'else
'endcelldefine
'endif
'ifdef
'include
'nounconnected_drive
'resetall
'timescale
'unconnected_drive
'undef
```


Verilog-XL compatible compiler directives

The following compiler directives are provided for compatibility with Verilog-XL.

``delay_mode_distributed`

This directive disables path delays in favor of distributed delays. See [Delay modes](#) (5-81) for details.

``delay_mode_path`

This directive sets distributed delays to zero in favor of path delays. See [Delay modes](#) (5-81) for details.

``delay_mode_unit`

This directive sets path delays to zero and non-zero distributed delays to one time unit. See [Delay modes](#) (5-81) for details.

``delay_mode_zero`

This directive sets path delays and distributed delays to zero. See [Delay modes](#) (5-81) for details.

``uselib`

This directive is an alternative to the **-v**, **-y**, and **+libext** source library compiler options. See [Verilog-XL 'uselib compiler directive](#) (5-72) for details.

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

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

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

```
'default_decay_time
'default_trireg_strength
'signed
'unsigned
```

Using the Verilog PLI

The Verilog PLI provides a mechanism for defining system tasks and functions that communicate with the simulator through a C procedural interface. There are many third party applications available that interface to Verilog simulators through the PLI interface (see [Third party PLI applications](#) (5-99)). In addition, you may write your own PLI applications.

ModelSim Verilog implements the PLI as defined in the IEEE Std 1364, with the exception of the VPI routines and the acc_handle_datapath routine. Currently, the VPI routines are not commonly used, although *ModelSim* will support them in a future release. The acc_handle_datapath routine is not implemented because the information it returns is more appropriate for a static timing analysis tool.

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

Registering PLI applications

Each PLI application must register its system tasks and functions with the simulator, providing the name of each system task and function and the associated callback routines. Since many PLI applications already interface to Verilog-XL, *ModelSim* Verilog PLI applications make use of the same mechanism to register information about each system task and function in an array of s_tfcell structures. This structure is declared in the veriusers.h include file as follows:

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

typedef struct t_tfcell {
    short type; /* USERTASK, USERFUNCTION, or USERREALFUNCTION */
    short data; /* passed as data argument of callback function */
    p_tffn checktf; /* argument checking callback function */
    p_tffn sizetf; /* function return size callback function */
    p_tffn calltf; /* task or function call callback function */
}
```

```

    p_tffn misctf; /* miscellaneous reason callback function */
    char *tfname; /* name of system task or function */

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

```

The various callback functions (checktf, sizetf, calltf, and misctf) are described in detail in Section 17 of 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 veriusert.h as follows:

```
void mti_RegisterUserTF(p_tfcell usertf);
```

The storage for each usertf entry passed to the simulator must persist throughout the simulation because the simulator dereferences the usertf pointer to call the callback functions. It is recommended that you define your entries in an array, with the last entry set to 0. If the array is named veriusertfs (as is the case for linking to Verilog-XL), then you don't have to provide an init_usertfs function, and the simulator will automatically register the entries directly from the array (the last entry must be 0). For example,

```

s_tfcell veriusertfs[] = {
    {usertask, 0, 0, 0, abc_calltf, 0, "$abc"},
    {usertask, 0, 0, 0, xyz_calltf, 0, "$xyz"},
    {0} /* last entry must be 0 */
};

```

Alternatively, you can add an `init_usertfs` function to explicitly register each entry from the array:

```
void init_usertfs()
{
    p_tfcell usertf = veriusertfs;
    while (usertf->type)
        mti_RegisterUserTF(usertf++);
}
```

It is an error if the PLI application does not contain a `veriusertfs` array or an `init_usertfs` function.

Since PLI applications are dynamically loaded by the simulator, you must specify which applications to load (each application must be a dynamically loadable library, see Compiling and linking PLI applications). The PLI applications are specified as follows:

- As a list in the Veriuser entry in the *modelsim.ini* file:

```
Veriuser = pliappl1.so pliapp2.so pliappn.so
```

- As a list in the PLIOBJS environment variable:

```
% setenv PLIOBJS "pliappl1.so pliapp2.so pliappn.so"
```

- As a `-pli` option to the simulator (multiple options are allowed):

```
-pli pliappl1.so -pli pliapp2.so -pli pliappn.so
```

The various various methods of specifying PLI applications may be used simultaneously.

Compiling and linking PLI applications

ModelSim Verilog uses operating system calls to dynamically load PLI applications when the simulator loads a design. Therefore, the PLI application must be compiled and linked for dynamic loading on a specific operating system. The PLI routines are declared in the include files located in the ModelSim *<install_dir>/modeltech/include* directory. The `acc_user.h` file declares the ACC routines (defined in Section 19 of the IEEE Std 1364) and the `veriusert.h` file declares the TF routines (defined in Section 21 of the IEEE Std 1364).

The following instructions assume that the PLI 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 instruction.

PLI Application Requirements

PLI applications are dynamically loaded into VSIM. A PLI application can consist of one or more dynamically loadable objects. Each of these objects must contain an entry point named `init_userdfs()` and a local `veriusertfs` table of user tasks and functions. There must be an entry in the table for each function in the object file that can be called externally. The `init_userdfs()` function must call `mti_RegisterUserTF()` for each entry in its local `veriusertfs` table.

PLI applications that use the TF functions should include the `veriusert.h` file.

Windows NT/95/98 platforms

Under Windows NT/95/98, VSIM loads a 32-bit dynamically linked library for each PLI application. The following compile and link steps are used to create the necessary `.dll` file (and other supporting files) using the Microsoft Visual C/C++ compiler.

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

Where `<C_init_function>` is the function name specified in the `FOREIGN` attribute (for FLI).

Note: The PLI interface has been tested with DLLs built using Microsoft Visual C/C++ compiler version 4.1 or greater.

Solaris platform

Under SUN Solaris, VSIM loads shared objects. Use these **gcc** or **cc** compiler commands to create a shared object:

gcc compiler:

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

cc compiler:

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

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

If *app.so* is in your current directory you must force Solaris to search the directory. There are two ways you can do this:

- Add “. / “ before *app.so* in the attribute specification, or
- Load the path as a UNIX shell environment variable:
`LD_LIBRARY_PATH= <library path without filename>`

SunOS 4 platform

Under SUN OS4, VSIM loads shared objects. Use the following commands to create a shared object:

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

HP700 platform

VSIM loads shared libraries on the HP700 workstation. A shared library is created by creating object files that contain position-independent code (use the **+z** compiler option) and by linking as a shared library (use the **-b** linker option). Use these **gcc** or **cc** compiler commands:

gcc compiler:

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

cc compiler:

```
cc -c +z -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.

for HP-UX 11.0 users

If you are building the FLI module under HP-UX 11.0, you should not specify the **"-lc"** option to the invocation of **ld**, since this will cause an incorrect version of the standard C library to be loaded with the module.

In other words, build modules like this:

```
cc -c +z -I/<install_dir>/modeltech/include app.c
ld -b -o app.sl app.o
```

If you receive the error "Exec format error" when the simulator is trying to load an FLI module, then you have most likely built under 11.0 and specified the **"-lc"** option. Just rebuild without **"-lc"** (or rebuild on an HP-UX 9.0/10.0 machine).

IBM RISC/6000 platform

VSIM loads shared libraries on the IBM RS/6000 workstation. The shared library must import VSIM's C interface symbols and it must export the C initialization function. VSIM's export file is located in the ModelSim installation directory in *rs6000/mti_exports*.

If your foreign module uses anything from a system library, you'll need to specify that library when you link your foreign module. For example, to use the standard C library, specify '-lc' to the 'ld' command.

The resulting object must be marked as shared reentrant using the compiler option appropriate for your version of AIX:

for AIX 4.1

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

for AIX 4.2 (choose gcc or cc compiler commands)

gcc compiler:

```
gcc -c -I/<install_dir>/modeltech/include app.c
ld -bPT:0x100000000 -bPD:0x20000000 -btextro -bnoelcsect\
    -o app.sl app.o -bI:/<install_dir>/modeltech/rs6000/mti_exports\
    -bnoentry
```

cc compiler:

```
cc -c -I/<install_dir>/modeltech/include app.c
cc -o app.sl app.o -bE:app.exp\
    -bI:/<install_dir>/modeltech/rs6000/mti_exports\
    -Wl-G -bnoentry
```

The *app.exp* file must export the C initialization function:

the function name specified in the FOREIGN attribute(for FLI)

Specifying the PLI file to load

Once your C application has been compiled it is ready to be loaded by VSIM. The name of the file to be loaded is specified in the *modelsim.ini* file by the Veriuser entry. The Veriuser entry must be in the [vsim] section of the file.

For example,

```
[vsim]
.
.
.
Veriuser = app.so
```

The Veriuser entry also accepts a list of shared objects. Each shared object is an independent PLI application that must contain an `init_usertfs()` entry point that registers the application's tasks and callback functions. An example entry in the *modelsim.ini* file is:

```
Veriuser = appl.so app2.so app3.so
```

VSIM also supports two alternative methods of specifying the PLI files to load: the **vsim** (CR-208) **-pli** command line option and the **PLIOBJS** (B-392) environment variable.

See also *Appendix B - ModelSim Variables* for more information on the *modelsim.ini* file.

PLI Example

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

hello.c:

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

hello.v:

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

Compile the PLI code for 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 callback reason argument

The second argument to a callback function is the reason argument. The values of the various reason constants are defined in the `veriusers.h` include file. See Section 17 of the IEEE Std 1364 for a description of the reason constants. The following details relate to ModelSim Verilog, and may not be obvious in the IEEE Std 1364. Specifically, the simulator passes the reason values to the `misctf` callback functions under the following circumstances:

`reason_endofcompile`

For the completion of loading the design.

`reason_finish`

For the execution of the `$finish` system task or the `quit` command.

`reason_startofsave`

For the start of execution of the checkpoint command, but before any of the simulation state has been saved. This allows the PLI application to prepare for the save, but it shouldn't save its data with calls to `tf_write_save` until it is called with `reason_save`.

`reason_save`

For the execution of the checkpoint command. This is when the PLI application must save its state with calls to `tf_write_save`.

`reason_startofrestart`

For the start of execution of the restore command, but before any of the simulation state has been restored. This allows the PLI application to prepare for the restore, but it shouldn't restore its data 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.

`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 omits 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".

Object handles

Many of the object handles returned by the ACC PLI routines are pointers to objects that naturally exist in the simulation data structures, and the handles to these objects are valid throughout the simulation, even after the `acc_close` routine is called. However, some of the objects are created on demand, and the handles to these objects become invalid after `acc_close` is called. The following object types are created on demand in *ModelSim* Verilog:

```
accOperator (acc_handle_condition)
accWirePath (acc_handle_path)
accTerminal (acc_handle_terminal, acc_next_cell_load, acc_next_driver, and
             acc_next_load)
accPathTerminal (acc_next_input and acc_next_output)
accTchkTerminal (acc_handle_tchkarg1 and acc_handle_tchkarg2)
accPartSelect (acc_handle_conn, acc_handle_pathin, and acc_handle_pathout)
accRegBit (acc_handle_by_name, acc_handle_tfarg, and acc_handle_itfarg)
```

If your PLI application uses these types of objects, then it is important to call `acc_close` to free the memory allocated for these objects when the application is done using them.

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 `veriusr.c` file. The `veriusr.c` file contains the registration information as

described above in "Registering PLI applications". To prepare the application for ModelSim Verilog, you must compile the veriuser.c file and link it to the object files to create a dynamically loadable object (see "Compiling and linking PLI applications"). 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
```

That's all there is to it. The PLI application is 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 modesim.ini file entry, the -pli simulator option, or the PLIOBS environment variable (see "Registering PLI applications").

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

Support for VHDL objects

The PLI ACC routines also provide limited support for VHDL objects in either an all VHDL design or a mixed VHDL/Verilog design. The following table lists the VHDL objects for which handles may be obtained and their type and fulltype constants:

Type	Fulltype	Description
accArchitecture	accArchitecture	instantiation of an architecture
accArchitecture	accEntityVitalLevel0	instantiation of an architecture whose entity is marked with the attribute VITAL_Level0
accArchitecture	accArchVitalLevel0	instantiation of an architecture which is marked with the attribute VITAL_Level0
accArchitecture	accArchVitalLevel1	instantiation of an architecture which is marked with the attribute VITAL_Level1

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

The type and fulltype constants for VHDL objects are defined in the *acc_vhdl.h* include file. All of these objects (except signals) are scope objects that define levels of hierarchy in the Structure window. Currently, the PLI ACC interface has no provision for obtaining handles to generics, types, constants, attributes, subprograms, and processes. However, some of these objects can be manipulated through the ModelSim VHDL foreign interface (mti_* routines). See ["FLI function descriptions"](#) (12-309).

IEEE Std 1364 ACC routines

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

acc_append_delays	acc_append_pulsere	acc_close
acc_collect	acc_compare_handles	acc_configure
acc_count	acc_fetch_argc	acc_fetch_argv
acc_fetch_attribute	acc_fetch_attribute_int	acc_fetch_attribute_str

acc_fetch_defname	acc_fetch_delay_mode	acc_fetch_delays
acc_fetch_direction	acc_fetch_edge	acc_fetch_fullname
acc_fetch_fulltype	acc_fetch_index	acc_fetch_location
acc_fetch_name	acc_fetch_paramtype	acc_fetch_paramval
acc_fetch_polarity	acc_fetch_precision	acc_fetch_pulsere
acc_fetch_range	acc_fetch_size	acc_fetch_tfarg
acc_fetch_itfarg	acc_fetch_tfarg_int	acc_fetch_itfarg_int
acc_fetch_tfarg_str	acc_fetch_itfarg_str	acc_fetch_timescale_info
acc_fetch_type	acc_fetch_type_str	acc_fetch_value
acc_free	acc_handle_by_name	acc_handle_calling_mod_m
acc_handle_condition	acc_handle_conn	acc_handle_hiconn
acc_handle_interactive_scope	acc_handle_loconn	acc_handle_modpath
acc_handle_notifier	acc_handle_object	acc_handle_parent
acc_handle_path	acc_handle_pathin	acc_handle_pathout
acc_handle_port	acc_handle_scope	acc_handle_simulated_net
acc_handle_tchk	acc_handle_tchkarg1	acc_handle_tchkarg2
acc_handle_terminal	acc_handle_tfarg	acc_handle_itfarg
acc_handle_tfinst	acc_initialize	acc_next
acc_next_bit	acc_next_cell	acc_next_cell_load
acc_next_child	acc_next_driver	acc_next_hiconn
acc_next_input	acc_next_load	acc_next_loconn
acc_next_modpath	acc_next_net	acc_next_output
acc_next_parameter	acc_next_port	acc_next_portout
acc_next_primitive	acc_next_scope	acc_next_specparam
acc_next_tchk	acc_next_terminal	acc_next_topmod

acc_object_in_typelist	acc_object_of_type	acc_product_type
acc_product_version	acc_release_object	acc_replace_delays
acc_replace_pulsere	acc_reset_buffer	acc_set_interactive_scope
acc_set_pulsere	acc_set_scope	acc_set_value
acc_vcl_add	acc_vcl_delete	acc_version

IEEE Std 1364 TF routines

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

io_mcdprintf	io_printf	mc_scan_plusargs
tf_add_long	tf_asynchoff	tf_iasynchoff
tf_asynchon	tf_iasynchon	tf_clearalldelays
tf_iclearalldelays	tf_compare_long	tf_copypvc_flag
tf_icopypvc_flag	tf_divide_long	tf_dofinish
tf_dostop	tf_error	tf_evaluatep
tf_ievaluatep	tf_exprinfo	tf_iexprinfo
tf_getcstringp	tf_igetcstringp	tf_getinstance
tf_getlongp	tf_igetlongp	tf_getlongtime
tf_igetlongtime	tf_getnextlongtime	tf_getp
tf_igetp	tf_getpchange	tf_igetpchange
tf_getrealp	tf_igetrealp	tf_getrealtime
tf_igetrealtime	tf_gettime	tf_gettime
tf_gettimeprecision	tf_gettimeprecision	tf_gettimeunit
tf_gettimeunit	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_inodeinfo	tf_nump	tf_inump
tf_propagatep	tf_ipropagatep	tf_putlongp
tf_iputlongp	tf_putp	tf_iputp
tf_putrealp	tf_iputrealp	tf_read_restart
tf_real_to_long	tf_rosynchronize	tf_irosynchronize
tf_scale_longdelay	tf_scale_realdelay	tf_setdelay
tf_isetdelay	tf_setlongdelay	tf_isetlongdelay
tf_setrealdelay	tf_isetrealdelay	tf_setworkarea
tf_isetworkarea	tf_sizep	tf_isizep
tf_spname	tf_ispname	tf_strdelputp
tf_istrdelputp	tf_strgetp	tf_istrgetp
tf_strgettime	tf_strlongdelputp	tf_istrlongdelputp
tf_strealdelputp	tf_istrrealdelputp	tf_subtract_long
tf_synchronize	tf_isynchronize	tf_testpvc_flag
tf_itestpvc_flag	tf_text	tf_typep
tf_itypep	tf_unscale_longdelay	tf_unscale_realdelay
tf_warning	tf_write_save	

Verilog-XL compatible routines

The following PLI routines are not define in the 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.

PLI tracing

The foreign interface tracing feature is available for tracing user foreign language calls made to the MTI Verilog PLI. 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 to replay what the foreign interface side did.

The purpose of tracing files

The purpose of the logfile is to aid you in debugging PLI code. The primary purpose of the replay facility is to send the replay file to MTI support for debugging co-simulation problems, or debugging PLI problems for which it is impractical to send the PLI code. MTI still would need the customer 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-208) with the **-trace_foreign** option:

Syntax

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

Arguments

<action>

Specifies one of the following actions:

Value	Action	Result
1	create log only	writes a local file called "mti_trace_<tag>"
2	create replay only	writes local files called "mti_data_<tag>.c", "mti_init_<tag>.c", "mti_replay_<tag>.c" and "mti_top_<tag>.c"
3	create both log and replay	

-tag <name>

Used to give distinct file names for multiple traces. Optional.

Examples

```
vsim -trace_foreign 1 mydesign
```

Creates a logfile.

```
vsim -trace_foreign 3 mydesign
```

Creates both a logfile and a set of replay files.

```
vsim -trace_foreign 1 -tag 2 mydesign
```

Creates a logfile with a tag of "2".

The tracing operations will provide tracing during all user foreign code-calls, including PLI user tasks and functions (calltf, checktf, sizetf and misctf routines), and Verilog VCL call-backs.

6 - Mixed VHDL and Verilog Designs

Chapter contents

Separate compilers, common libraries	108
Mapping data types	108
VHDL generics	108
Verilog parameters	109
VHDL and Verilog ports	109
Verilog states	110
VHDL instantiation of Verilog design units.	112
Verilog instantiation criteria	112
Component declaration	112
vgencomp component declaration	114
Verilog instantiation of VHDL design units.	116
VHDL instantiation criteria.	116
SDF annotation	117

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

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

Separate compilers, common libraries

VHDL source code is compiled by VCOM and the resulting compiled design units (entities, architectures, configurations, and packages) are stored in a library. Likewise, Verilog source code is compiled by VLOG and the resulting design units (modules and UDPs) are stored in a library.

Libraries can store any combination of VHDL and Verilog design units, provided the design unit names do not overlap (VHDL design unit names are changed to lower case).

See "[Design Libraries](#)" (2-31) for more information about library management and see the **vdel** (CR-175) and the **vlog** (CR-199) commands.

Mapping data types

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

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

VHDL generics

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

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

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

Physical and enumeration types receive a value that corresponds to the position number indicated by the Verilog number. In VHDL this is equivalent to TVAL(P), where T is the type, VAL is the predefined function attribute that returns a value given a position number, and P is the position number.

Verilog parameters

VHDL type	Verilog type
integer	integer
real	real
string	string

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

VHDL and Verilog ports

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

Allowed VHDL types
bit
bit_vector
std_logic
std_logic_vector
vl_logic
vl_logic_vector

The vl_logic type is an enumeration that defines the full state set for Verilog nets, including ambiguous strengths. The bit and std_logic types are convenient for most applications, but the vl_logic type is provided in case you need access to the

full Verilog state set. For example, you may wish to convert between `vl_logic` and your own user-defined type. The `vl_logic` type is defined in the `vl_types` package in the pre-compiled **verilog** library. This library is provided in the installation directory along with the other pre-compiled libraries (**std** and **ieee**). The source code for the `vl_types` package can be found in the files installed with *ModelSim*. (See `\modeltech\vhdl_src\verilog\vltypes.vhd`.)

Verilog states

Verilog states are mapped to `std_logic` and `bit` as follows:

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

Verilog	std_logic	bit
St1	'1'	'1'
StX	'X'	'0'
Su0	'0'	'0'
Su1	'1'	'1'
SuX	'X'	'0'

For Verilog states with ambiguous strength:

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

VHDL type bit is mapped to Verilog states as follows:

bit	Verilog
'0'	St0
'1'	St1

VHDL type std_logic is mapped to Verilog states as follows:

std_logic	Verilog
'U'	StX
'X'	StX
'0'	St0
'1'	St1
'Z'	HiZ
'W'	PuX

std_logic	Verilog
'L'	Pu0
'H'	Pu1
'_'	StX

VHDL instantiation of Verilog design units

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

Verilog instantiation criteria

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

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

Component declaration

A Verilog module that is compiled into a library can be referenced from a VHDL design as though the module is a VHDL entity. The interface to the module can be extracted from the library in the form of a component declaration by running **vgencomp** (CR-178). Given a library and module name, **vgencomp** (CR-178) writes a component declaration to standard output.

The default component port types are:

- std_logic
- std_logic_vector

Optionally, you can choose:

- bit and bit_vector

- vl_logic and vl_logic_vector

VHDL and Verilog identifiers

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

- The Verilog module was compiled with the -93 switch. This means **vgencomp** (CR-178) 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 and generic names are not unique unless case is preserved. In this event, **vgencomp** (CR-178) behaves as if the module was compiled with the -93 switch for those names only.

Examples

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

If the Verilog module is compiled with -93:

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

vgencomp component declaration

vgencomp (CR-178) generates a component declaration according to these rules:

Generic clause

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

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

Parameter value	Generic type
integer	integer
real	real
string literal	string

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

Examples

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

Port clause

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

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

Examples

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

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

Verilog instantiation of VHDL design units

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

VHDL instantiation criteria

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

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

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

Named port associations

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

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

An entity name is not case sensitive in Verilog instantiations. The entity default architecture is selected from the work library unless specified otherwise.

Verilog does not have the concept of architectures or libraries, so the escaped identifier is employed to provide an extended form of instantiation:

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

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

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

SDF annotation

A mixed VHDL/Verilog design can also be annotated with SDF. See ["SDF for Mixed VHDL and Verilog Designs"](#) (11-293) for more information.

7 - ModelSim SE Performance Analyzer

Chapter contents

Introducing Performance Analysis	120
A Statistical Sampling Profiler	120
Getting Started	121
Interpreting the data	122
Viewing Performance Analyzer Results	123
Interpreting the Name Field.	125
Interpreting the %Under and %In Fields	125
Differences in the Ranked and Hierarchical Views	126
Ranked/Hierarchical Window Features	128
The report option	129
Setting preferences with Tcl variables	130
Performance Analyzer commands	131

This chapter describes the Performance Analyzer feature of ModelSim SE.

Performance Analyzer is used to easily identify areas in your simulation where performance can be improved. The Performance Analyzer can be used at all levels of design simulation – Functional, RTL and Gate Level – and has the potential to save hours of regression test time in VHDL, Verilog and mixed HDL designs. In addition, ASIC and FPGA design flows benefit from the use of this tool.

This chapter can be used as a stand alone instruction tool. It is, however, most effective when used in conjunction with running the Performance Analyzer.

Note: Performance Analyzer is a component of the ModelSim SE. At this time, it works only with Windows NT and UNIX platforms. Please contact sales@model.com, or view our website at www.model.com for more information on ModelSim SE.

Introducing Performance Analysis

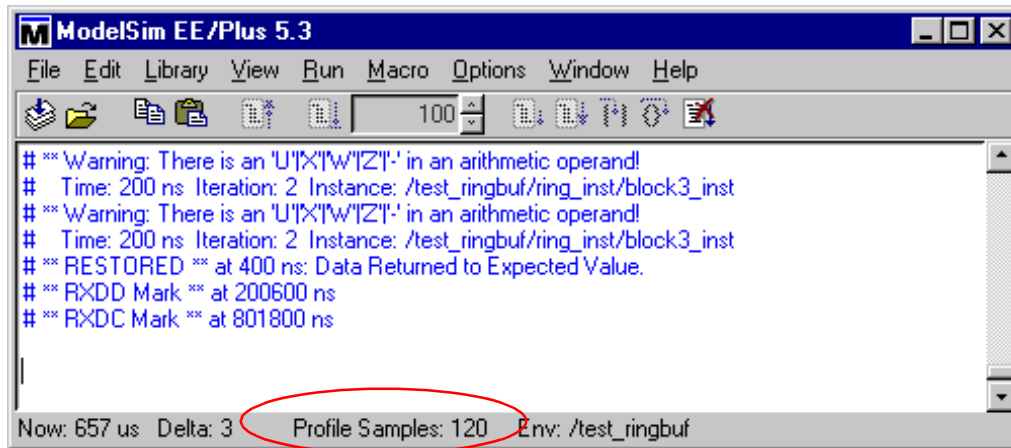
With the 5.3 release of ModelSim EE, a Performance Analyzer has been added as part of the simulation kernel engine. The Performance Analyzer provides a interactive graphical representation of where ModelSim is spending its time while running your design. This feature enables you to quickly determine what is impacting the design environment's simulation performance. Those familiar with the design and validation environment will be able to find first level improvements in a matter of minutes.

For example, the Performance Analyzer might show that a non-accelerated VITAL library cell is impacting simulation run time. Or, that a process is consuming more time than necessary because of non-required items on its sensitivity list. Or, that a testbench process is active even though it is not needed. Or, that a random number process is consuming simulation resources when in a test bench is run in non-random mode. With this information, you can make changes to the VHDL or Verilog source code that will speed up the simulation.

A Statistical Sampling Profiler

The Performance Analyzer feature in VSIM is a statistical sampling profiler. It periodically "wakes up" and samples the current simulation at a user-determined rate, and records what is executing in the simulation during the sample period. The advantage of statistical analysis is that an entire simulation may not have to be run to get good information from the Performance Analyzer. A few thousand samples, for example, can be accumulated before pausing the simulation to see where simulation time is being spent.

During sampling, the Samples field in the footer of the main VSIM window displays the number of profiling samples collected, and each sample becomes one data point in the simulation profile.



Getting Started

Performance analysis occurs during the ModelSim **run** command and is displayed graphically as a profile of simulator performance. To enable the Performance Analyzer, use the **profile on** command at the VSIM prompt. After this command is executed, all subsequent **run** commands will have profiling statistics gathered for them. With the Performance Analyzer enabled and a **run** command initiated, the simulator will provide a message indicating that profiling has started.

The Performance Analyzer is turned off by issuing the **profile off** command to the VSIM prompt. Any ModelSim **run** commands that follow will not be profiled.

Profiling results are cumulative. Therefore, each **run** command performed with profiling ON will add new information to the data being gathered. To clear this data, issue the **profile clear** command to the VSIM prompt.

Interpreting the data

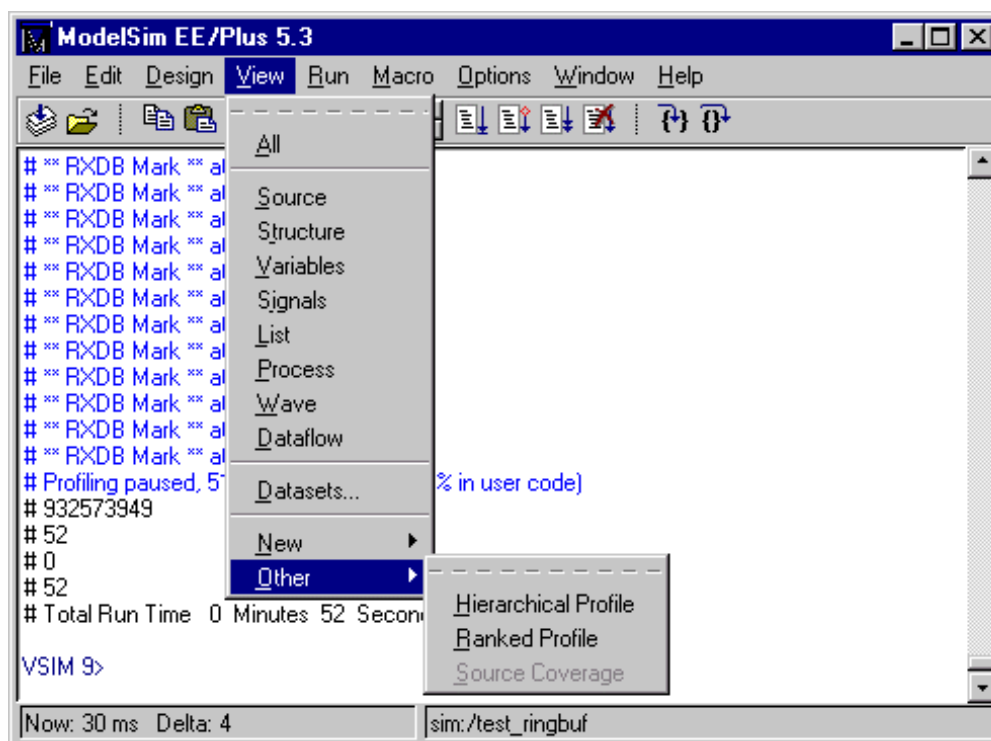
The Performance Analyzer provides insight into where the simulation is spending its time. It is most helpful in those situations where a very high percentage of simulation time is being spent in a particular module. For example, the Performance Analyzer may show that the simulation is spending, say, 60% of its time in module X. This information can be used to find where module X was implemented poorly and achieve a fix that runs several times faster. In such instances, performance analysis provides an invaluable guide to circuit optimization.

More commonly, the Performance Analyzer will tell you that 30% of simulation time was spent in model X, 25% in model Y, and 20% in model Z. In such situations, careful examination and improvement of each model may result in a significant overall simulation speed improvement.

There are times, however, when the Performance Analyzer tells you nothing better than that the simulation has executed in several hundred different models and has spent less than 1% of its time in any one of them. In such situations, the Performance Analyzer provides little helpful information and simulation improvement must come from a higher level examination of how the design can be changed or optimized.

Viewing Performance Analyzer Results

The Performance Analyzer provides two views of the collected data – a *hierarchical* and a *ranked* view. The hierarchical view is accessed by clicking **View > Other > Hierarchical Profile**. The ranked view is accessed through the menu bar by clicking **View > Other > Ranked Profile**.



The Hierarchical view can also be invoked by entering **view_profile** at the VSIM prompt. In the hierarchical view below, two lines (store.vhd:43 and retrieve.vhd:35) are taking the majority of the simulation time.

In the Hierarchical Profile window you can expand and collapse various levels to hide data that is not useful and/or is cluttering the data display. Click on a '-' box to collapse all levels beneath the entry. Click on the '+' box to expand an entry. By default, all levels are fully expanded.

Name	%Under	%In	%Parent
<<VHDL-Evaluation>>	97	4	97
store.vhd:43	42	11	43
E:/MTI_perf_53/modeltech/win32/.../vhd_src/synopsys/mti_std_logic_unsigned.vhd:429	31	31	74
control.vhd:98	1	0	1
E:/MTI_perf_53/modeltech/win32/.../vhd_src/synopsys/mti_std_logic_unsigned.vhd:424	1	1	76
retrieve.vhd:35	45	11	46
E:/MTI_perf_53/modeltech/win32/.../vhd_src/synopsys/mti_std_logic_unsigned.vhd:429	34	34	76
control.vhd:87	2	0	2
E:/MTI_perf_53/modeltech/win32/.../vhd_src/synopsys/mti_std_logic_unsigned.vhd:276	1	1	57
control.vhd:114	1	0	1
<<Process-Triggering>>	1	1	1
<<Internal-Functions>>	2	2	2

Similarly, the Ranked view can be invoked by entering **view_profile_ranked**.

Name	%Under	%In
E:/MTI_perf_53/modeltech/win32/.../vhd_src/synopsys/mti_std_logic_unsigned.vhd:429	65	65
retrieve.vhd:35	45	11
store.vhd:43	42	11
control.vhd:87	2	0
E:/MTI_perf_53/modeltech/win32/.../vhd_src/synopsys/mti_std_logic_unsigned.vhd:276	1	1
control.vhd:98	1	0
E:/MTI_perf_53/modeltech/win32/.../vhd_src/synopsys/mti_std_logic_unsigned.vhd:424	1	1
control.vhd:114	1	0

The modules and code lines are ranked in order of the amount of simulation time used. The two lines that are taking up most of the simulation time – `retrieve.vhd:35` and `store.vhd:43` – appear at the top of the list under the VHDL module that contains them.

Interpreting the Name Field

The *Name*, *%Under* and *%In* fields appear in both the ranked and hierarchical views. These fields are interpreted identically in both views. Typically a Name consist of a VHDL file and line number pair. Most useful names consist of a line of VHDL (or Verilog) source code. If you use a PLI or FLI routine, then the name of the C function that implements that routine can also appear in the name field.

Note: VSIM is a stripped executable file, so that any functions inside of it will be credited to the line of VHDL code that uses the function.

The *hierarchical view* opens with all levels displayed. You can collapse the hierarchical view by clicking the boxes next to the high level names.

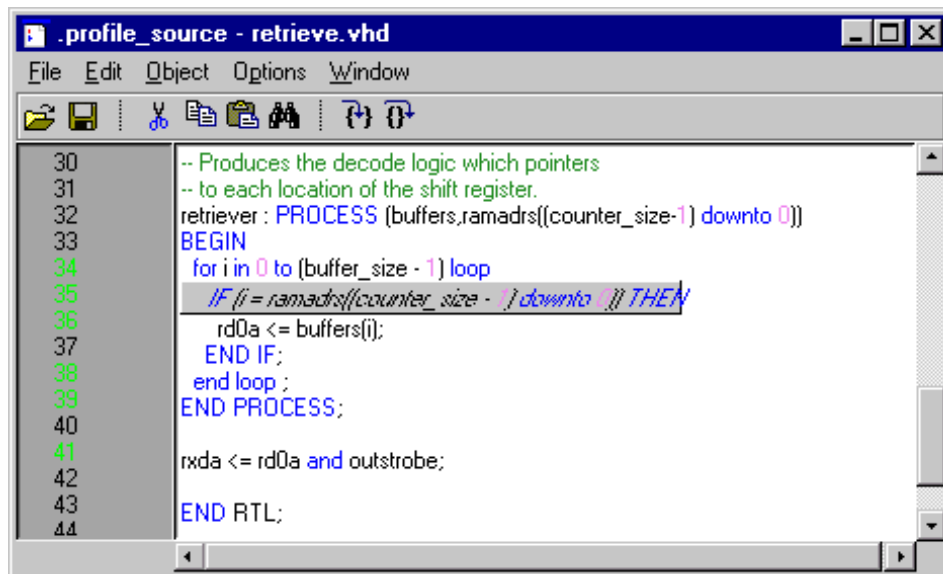
Note: At this time, the *hierarchical* view will not remember which levels are opened or closed when data is reloaded. By default, hierarchical levels are opened every time data is reloaded.

Interpreting the %Under and %In Fields

The *%In* and *%Under* columns describe the percentage of the total simulation time spent in and under a function listed in the Name field.

The distinction between *%In* and *%Under* is subtle but important. For the `retrieve.vhd:35` entry in the hierarchical and ranked views above, *%Under* is 45 and *%In* is 11. "*%Under*" means that this particular line and all support routines it needed took 45% of total simulation time. "*%In*" means that 11% of the total simulation time was actually spent executing this line of VHDL code.

In the body of the Hierarchical Profile or Ranked Profile windows, you can double-click on any VHDL/Verilog file and line-number pair to bring up that file in the Source Window with the selected line highlighted. In the diagram below, `retrieve.vhd:35` was selected in the Hierarchical Profile and, consequently, is highlighted in the Source window.



The actual line of VHDL code for *retrieve.vhd:35* is:

```
IF (i=ramadrs((counter_size-1)downto 0))THEN
```

Differences in the Ranked and Hierarchical Views

The hierarchical view differs from the ranked view in two important respects.

- Entries in the Names column are indented in order to show which functions or routines call which others.
- A %Parent column in the hierarchical view allows you to see what percentage of a parent routine's simulation time is used in which subroutines.

Indentation in the Name column of the hierarchical profile window indicates which line is calling a function. For example, in the hierarchical view above, the line *store.vhd:43* calls *E:MTI_perf_53/modeltech/win32/./vhd_src/synopsys/mti_std_logic_unsigned.vhd:429*.

The hierarchical view presents data in a call-graph style format that provides more context about where the simulation is spent than does the ranked view. For example, your models may contain several instances of a utility function that compute the maximum of 3-delay values. A ranked view might reveal that the simulation spent 60% of its time in this utility function, but would not tell you

what routine or routines were making the most use of it. The hierarchical view will reveal which line is calling the function most frequently. Using this information, you might decide that instead of calling the function every time to compute the maximum of the 3-delays, this spot in your VHDL code can be used to compute it just once. You can then store the maximum delay value in a local variable.

The *%Parent* column provides the percent of simulation time a given entry used of its parent's total simulation time. From this column, you can calculate the percentage of total simulation time taken up by any function. For example, if an particular parent entry used 10% of the total simulation time, and it called a routine that used 80% of its simulation time, then the percentage of total simulation time spent in that routine would be 80% of 10%, or 8%.

In addition to these differences, the ranked view displays any particular function only once, regardless of where it was used. In the hierarchical view, the function can appear multiple times – each time in the context of where it was used.

Ranked/Hierarchical Window Features

The ranked and hierarchical windows have a number of features that can manipulate the data displayed. Most of these features are contained in a toolbar in the header of the window, which displays an icon for each feature. Placing the mouse over an icon causes its function to be displayed.



The **Find Entry** icon provides access to a search function that can be used to search for a given string in the window. Press the return key or click on the binoculars to activate the search.

The **Under%** filter allows you to specify a cutoff percentage for displaying the data. By default, every entry in the profiling data that has spent at least 1% of the simulation time under that entry will be displayed.

The **hierCutoff** and **rankCutoff** variables provide a similar function. See "[Setting preferences with Tcl variables](#)" (7-130).

The **Update Data** icon causes the data to be reloaded from the simulator. If you change the cutoff percentage or do an additional simulation run the Ranked and Hierarchical Profile windows are not automatically updated. You should click on this button to update the data being displayed in these windows.

The **Save to File** icon allows the data to be saved to disk. You will be prompted for the output file name.

The **profile report** command (CR-120) provides another way to save profile data.

The report option

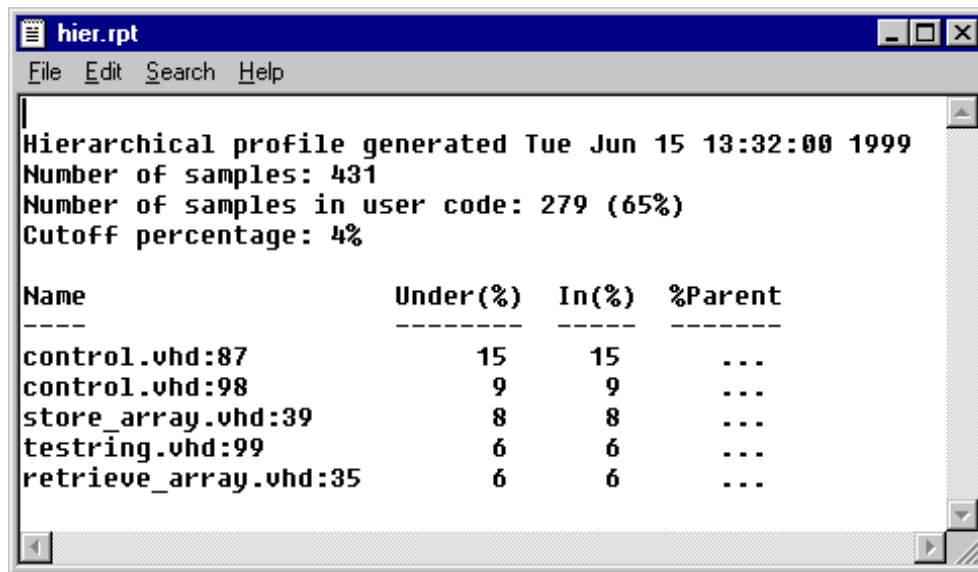
You can also use the tcl command **profile report** to save performance analyzer results.

```
profile report [<option>]
```

The arguments to the command are [-hierarchical | -ranked] [-file<filename>] [--cutoff <percentage>]. For example, the command

```
profile report -hierarchical -file hier.rpt -cutoff 4
```

will produce a profile report in a text file called *hier.rpt*, as shown here.



Setting preferences with Tcl variables

Various Tcl variables control how the Hierarchical Profile and Ranked Profile windows are displayed. They are:

`$PrefProfile(hierCutoff)`

hierCutoff is the minimum percent usage that will be listed in the Hierarchical Profile display. The default value is 1%. Any usage less than 1% will not be displayed.

`$PrefProfile(hierCutoffHighlight)`

hierCutoffHighlight is the minimum percent usage that will be highlighted (the `hotForeground` color) in the Hierarchical Profile display. The default value is 5%.

`$PrefProfile(rankCutoff)`

rankCutoff is the minimum percent usage that will be listed in the Ranked Profile display. The default value is 1%. Any usage less than 1% will not be displayed.

`$PrefProfile(rankCutoffHighlight)`

rankCutoffHighlight is the minimum percent usage that will be highlighted (the `hotForeground` color) in the Ranked Profile display. The default value is 5%.

`$PrefProfile(foreground)`

foreground is the color of the Hierarchical and Ranked Profile window borders that contain buttons and menus. The default color is blue.

`$PrefProfile(background)`

background is the color of the Hierarchical and Ranked Profile entry field. The default color is white.

`$PrefProfile(hotForeground)`

hotForeground is the highlight color of those lines that exceed the **rankCutoffHighlight** value. The default color is red.

`$PrefProfile(selectBackground)`

selectBackground is the highlighting color of the selected line in the Source window. The default color is Grey70.

Performance Analyzer commands

The table below provides a brief description of the profile commands, follow the links for complete commmand syntax.

See the [ModelSim EE/SE Command Reference](#) for complete command details.

Command	Description
profile clear command (CR-115)	used to clear any data that has been gathered during previous run commands. After this command is executed, all profiling data will be reset
profile interval command (CR-116)	allows you to select the frequency with which the profiler collects samples during a run command
profile off command (CR-117)	used to discontinue runtime profiling
profile on command (CR-118)	used to enable runtime analysis of where your simulation is spending its time
profile option command (CR-119)	allows various profiling options to be changed
profile report command (CR-120)	used to produce a textual output of the profiling statistics that have been gathered up to this point

8 - ModelSim SE Code Coverage

Chapter contents

Enabling Code Coverage	134
The coverage_summary window	136
Coverage Summary window preferences	137
Code Coverage commands	138

This chapter describes the Code Coverage feature of ModelSim SE.

Code Coverage gives you graphical and report file feedback on how the source code is being executed. This integrated feature provides three important benefits to the ModelSim user:

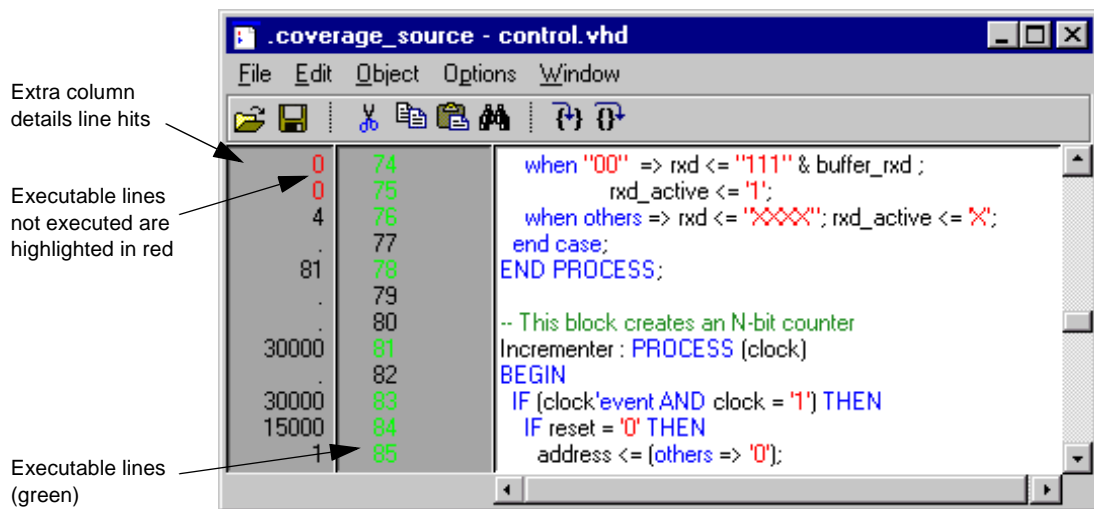
- 1 Because it's integrated into the ModelSim engine, it is totally non-intrusive – it doesn't require instrumented HDL code as do third party code coverage products.
- 2 It has very little impact on simulation performance (typically less than 5%).
- 3 There is no need to recompile to obtain code coverage statistics. The ModelSim version 5.3 library fully supports this feature.

The integrated Code Coverage capabilities can be used during any design phase in all levels and types of designs. With this simple, easy-to-use analysis tool you can develop more complete, robust, testbenches in less time.

Note: Code Coverage is a component of the ModelSim SE. Please contact sales@model.com, or view our website at www.model.com for more information on ModelSim SE.

Enabling Code Coverage

Begin simulation with the **-coverage** option to the **vsim** command (CR-208) to enable code coverage for the entire run. (The command can be also be invoked from the command line after simulation has begun.) When code coverage is enabled, an additional column appears on the left side of the source code window. This column indicates how many times each executable line of code has been executed during simulation (number of "hits"). Executable lines that are not executed (no hits) are highlighted with a red zero. (The default highlighting color is red and is controlled by the Tcl variable **\$PrefCoverage(zeroHitsColor)**).



In the source window you can find the next line with a zero hit count by using **Edit > Find > Next Coverage Miss**, or by simply pressing the Tab key. To view the maximum number of lines while doing code coverage use the **-O0** (capital O zero) compiler option.

The following commands to manipulate code coverage are enabled only when the **-coverage** invocation switch has been specified.

- **coverage clear** (CR-59)
Explicitly clears all the coverage line number counts.

- **coverage report [-file <filename>] [-lines]** (CR-61)

Reports the current coverage data. By default, the coverage report is output to the Main transcript window.

The **-file** option allows you to redirect the output. By default, a summary is output that only contains information on a "per file" basis -- i.e. file "foo.v" had 34 executable lines in it, of which 29 had hit counts that were greater than zero. This is essentially the same data that appears in the **view_coverage** window.

If you specify the **-lines** switch, then a verbose output is returned which lists every executable line with how many times it was executed.

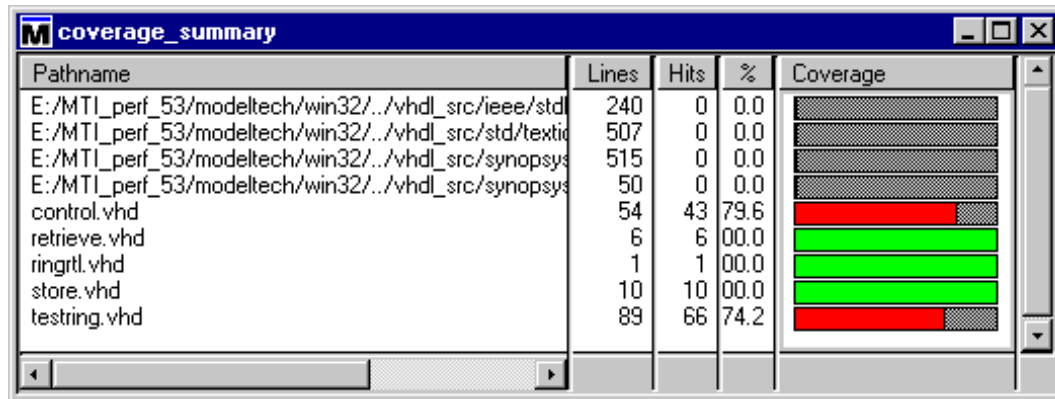
- **coverage reload <filename>** (CR-60)

Allows you to reload previous coverage data, and essentially "seed" the current results.

To accumulate code coverage results that span multiple **vsim** invocations you can use a combination of the **coverage report** and **coverage reload** commands. At the end of the first **vsim -coverage** run use the command: **coverage report -file your_file -lines**. At the beginning of the next run, use the command: **coverage reload <filename>**.

The coverage_summary window

The coverage_summary window is enabled by selecting **View > Other > Source Coverage** from the Main window or by entering **view_coverage** at the VSIM prompt. The **coverage_summary** provides a graphical summary of all execution coverage on a per file basis.



The file names in this window consist of all the design files that have executable lines of code.

- The Lines column contains the number of executable lines in that particular file.
- The Hits column indicates the number of executable lines that have been executed in the current simulation.
- The Percentage column is the current ratio of Hits to Lines. There is also a bar chart that graphically displays this percentage. If the coverage percentage is below a certain threshold, the bar chart is displayed in a highlighted color.

By default, the **coverage_summary** is sorted by Pathname and is linked to the source code. The summary may be sorted by any other column by simply clicking on the heading for that column (i.e.: Lines, Hits, %).

The **coverage_summary** includes a Coverage column that provides a bar graph representation of the number of times each executable line is executed or hit.

To view the source code of a particular file in the Source window, double click on a file Pathname in the coverage_summary window.

Coverage Summary window preferences

Coverage Summary window preferences control how the coverage data is displayed. The preferences can be set from in the ModelSim Main window with **Options > Edit Preferences > By Name** menu selection. Use the **Apply** button to view temporary changes, or **Save** the changes to a local *modelsim.tcl* file. Once saved, the preferences will be the default for subsequent simulations invoked from the same directory.

Each preference is saved as a tcl preference variable:

`$PrefCoverage(barColor)`

barColor is the color of the bar chart for files that meet the % Coverage threshold during execution (default color is green).

`$PrefCoverage(barHighlightColor)`

barHighlightColor is the color for those files that do not meet the % Coverage threshold during execution (default color is red).

`$PrefCoverage(cutoff)`

cutoff is the % Coverage threshold that must be met to not have the bar highlighted (default value is 90%).

`$PrefCoverage(background)`

background is the color of the Coverage Summary window background. The default color is white.

`$PrefCoverage(foreground)`

foreground is the color of the text in the Coverage Summary window. The default color is black.

`$PrefCoverage(selectBackground)`

selectBackground is the highlighting color of the selected line in the Coverage Summary window. The default color is Grey70.

Code Coverage commands

The commands below are available once Code Coverage is active. Enable code coverage with the **-coverage** option to the **vsim** command (CR-208).

The table below provides a brief description of the profile commands, follow the links for complete commmand syntax.

See the [ModelSim EE/SE Command Reference](#) for complete command details.

Command	Description
coverage clear command (CR-59)	used to clear all coverage data obtained during previous run commands
coverage reload command (CR-60)	used to seed the coverage statistics with the output of a previous coverage report command
coverage report command (CR-61)	used to produce a textual output of the coverage statistics that have been gathered up to this point

9 - Multiple logfiles, datasets and virtuals

Chapter contents

Multiple logfiles and datasets	139
Opening and viewing datasets	140
Using datasets with ModelSim commands	142
Restricting the dataset prefix display	143
Virtual Objects (User-defined buses, and more)	144
Virtual signals	145
Virtual functions	146
Virtual regions	146
Virtual types	147
Logfile and virtual commands reference table	147

A ModelSim simulation can be logged to a WLF file (formerly a WAV file) for future viewing or comparison to a current simulation. By default the logfile is named *vsim.wlf*.

With ModelSim release 5.3, you can open more than one logfile for simultaneous viewing in a single Wave window. You can also create virtual signals that are simple logical combinations of, or logical functions of, signals from different logfiles. This capability provides the basic mechanism for comparing simulations.

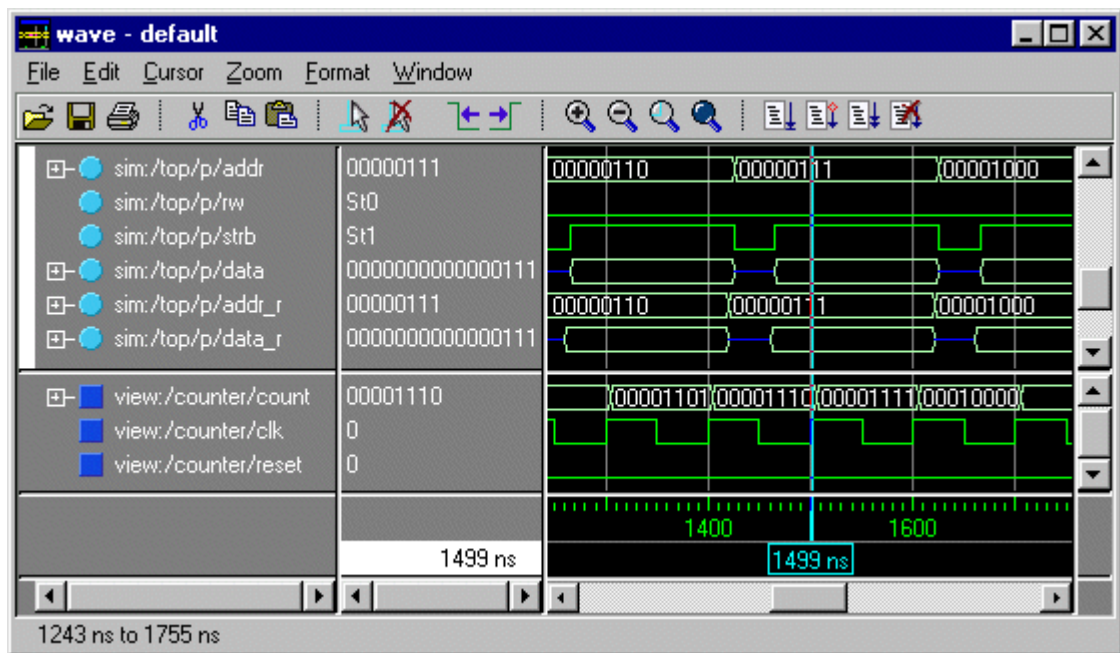
Multiple logfiles and datasets

When running a simulation, the logfile from a previous simulation may be opened in view mode and used as a reference. (View mode allows you to view, but not run, a previous simulation.)

If a logfile is viewed during an active simulation, a unique dataset name identifies each simulation. Two datasets are in view in the Wave window below; the current simulation is indicated by the "sim" prefix and a previous simulation is indicated by the "view" prefix. The default dataset names are:

- sim: – for the current active simulation
- view: – for the first view-mode file opened
- view<n>: – for the (n+1)-th view-mode file opened

Default dataset names may be changed in the *pref.tcl* file.

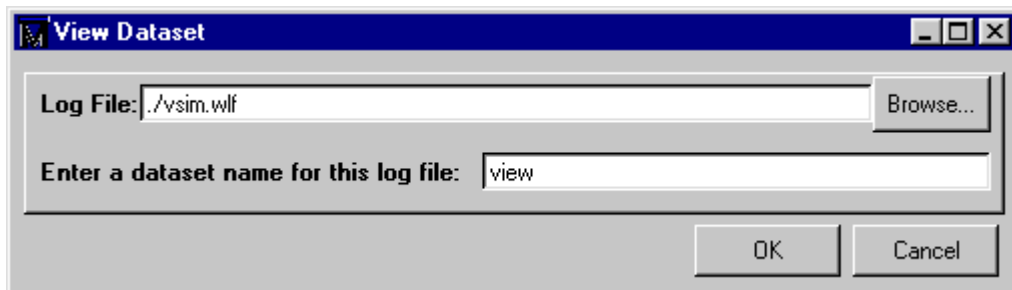


In the illustration above, the Wave window is split into two panes. The top pane shows the dataset of the current simulation. (The default dataset prefix is "sim".) The bottom pane shows a dataset in the view mode only. (The default dataset prefix is "view".) The view-mode dataset is located in the active pane, as indicated by the white bar in the left margin.

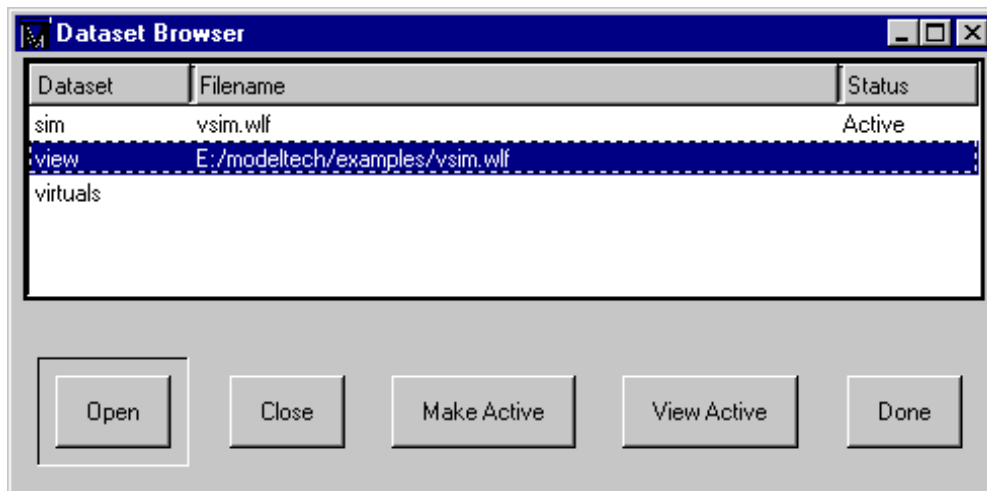
New panes are created with the **Wave > File > New Window Pane** menu selection. When multiple window panes exist, the **add_menusub** command (CR-28) adds signals to the active pane by default. Click in a pane to make it active.

Opening and viewing datasets

Logfiles may be opened using either the **Main > File > Open > Open Dataset** or **Wave > Open Dataset** menu selection. Opening datasets in this manner returns the View Dataset dialog box, allowing you to specify the dataset name for the logfile.



Once you've located the logfile, named, and OKed the dataset, it is ready for viewing. To view the dataset, use the **Main > View > Dataset** menu selection to open the Dataset Browser.



Select **Open** to browse for a dataset (this opens the View Dataset dialog box as well). Once the dataset is open, you can select it and choose **Make Active** to prepare it for viewing. To view the active dataset in the Structure and Signals windows click **View Active**. Add signals to the Wave window with [add_menucb](#) command (CR-28).

Make Active makes the selected dataset the "active" or default dataset. Default dataset means that if you type a region path as part of a command and omit the

dataset prefix, the default dataset will be assumed. It is equivalent to typing: `env <dataset>`: at the VSIM prompt.

Virtuals

ModelSim supports an additional, subterranean, dataset named "virtuals", which contains references to user-defined buses and other virtual objects (see below). Normally, you will not need to directly reference the virtuals dataset.

Using datasets with *ModelSim* commands

Multiple logfiles may be opened when the simulator is invoked by specifying more than one **vsim -view <filename>** option. Dataset prefixes for logfiles opened in this manner will be "view", "view2", "view3", etc..

A dataset name may also be specified as an optional qualifier to the **vsim -view** switch on the command line using the following syntax:

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

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

Design regions and signal names can be fully specified over multiple logfiles by using the dataset name as a prefix in the path. For example:

```
sim:/top/alu/out
view:/top/alu/out
golden:top.alu.out
```

Dataset prefixes are not required unless more than one dataset is open and you want to refer to something outside the default dataset. When more than one dataset is open, *ModelSim* will automatically prefix names in the Wave and List window with the dataset name. This may be changed using the **Props > Display Properties** dialog in those windows.

ModelSim designates one of the datasets to be the "current" dataset, and refers all names without dataset prefixes to that dataset. The current dataset is displayed in the context path at the bottom of the Main window. It can be displayed using the **environment** command (CR-80) with no arguments, or with the **View > Datasets** menu selection.

The Structure and Signals windows each have a current dataset to which they are sensitive. Being sensitive to a dataset means that the window will update when the content of that dataset changes. The dataset to which these windows are sensitive is set using the **File > Environment** menu selection in the respective window.

Additionally, a Structure or Signals window may be created sensitive to a specified dataset using the **-env <dataset>** switch to the **view** command. For example,

```
view -new signals -env view1
```

will create a new Signals window sensitive to the **view1** dataset.

ModelSim remembers a "current context" within each open dataset. You can toggle between the current context of each dataset using the **environment** command specifying the dataset without a path. For example:

```
env foo:
```

will set the current dataset to **foo** and the current context to the context last specified for **foo**.

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

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. You can use the Tcl **set** command to change the variable value from the ModelSim command line:

```
set PrefMain(DisplayDatasetPrefix) 0
```

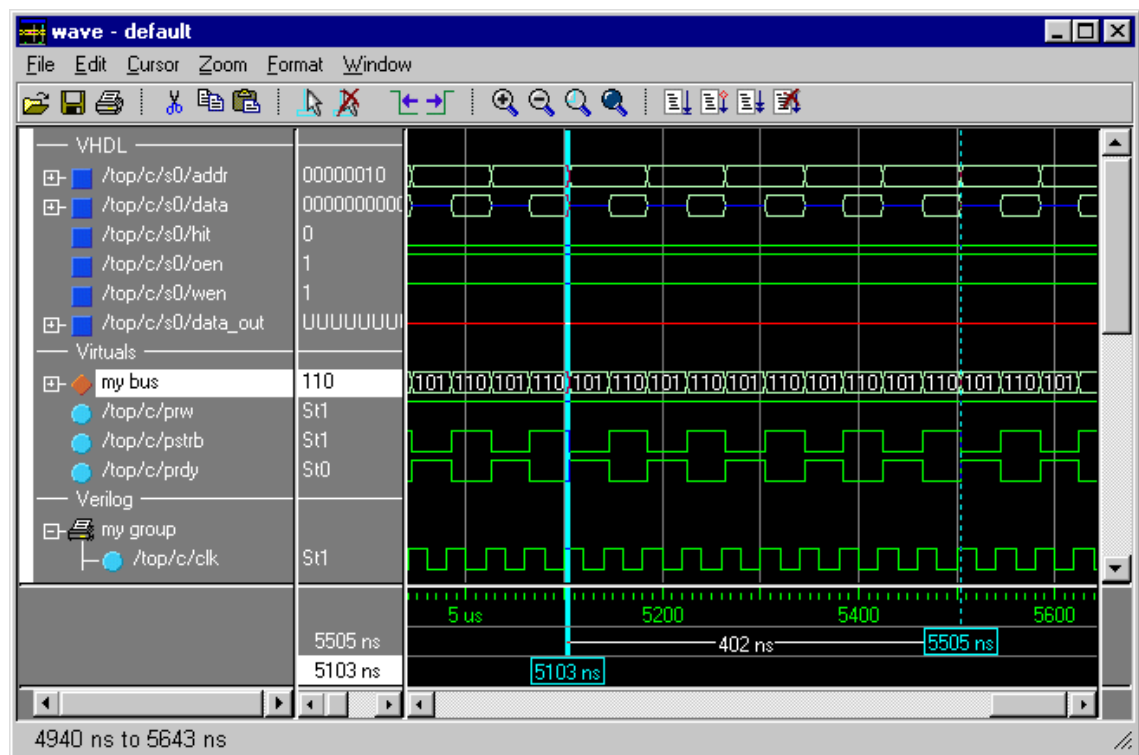
Additionally, you can restrict display of the dataset prefix if you use the **environment -nodataset** command to view a dataset. To display the prefix use the **environment** command (CR-80) with the **-dataset** option (you won't need to specify this option if the variable noted above is set to 1). The **environment** command line switches override the *pref.tcl* variable.

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 release 5.3 supports the following kinds of virtual objects:

- [Virtual signals](#) (9-145)
- [Virtual functions](#) (9-146)
- [Virtual regions](#) (9-146)
- [Virtual types](#) (9-147)

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



Virtual signals

Virtual signals are aliases for combinations or subelements of signals written to the logfile by the simulation kernel. They may be displayed in the Signals, List or Wave window, accessed by the **examine** command, and set using the **force** command. Virtual signals may be created by menu selections in the Signals, Wave or List windows, or with the **virtual signal** command described below. Virtual signals can also be dragged and dropped from the Signals window to the Wave or List window.

Virtual signals will be 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 should be installed. This may be used to install the virtual signal in a user-defined region in order to reconstruct the original RTL hierarchy when simulating and driving a post-synthesis gate-level implementation.

A virtual signal can be used to reconstruct RTL-level design buses that were broken down during synthesis. The **virtual hide** command can be used to hide the display of the broken-down bits if you don't want them cluttering up the Signals window (see below).

If the virtual signal has elements from more than one logfile, it will be automatically installed in the virtual region "virtuals:/Signals."

Virtual signals are not hierarchical – if two virtual signals are concatenated to become a third virtual signal, the resulting virtual signal will be a concatenation of all the subelements of the first two virtual signals.

The definitions of virtuals can be saved to a macro file using the **virtual save** command (see below). 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.

Implicit and explicit virtuals

There is one exception: "implicit virtuals" are automatically saved with the Wave or List format. An implicit virtual is a virtual signal that was automatically created by ModelSim without your knowledge and without you providing a name for it. An example would be if you expand a bus in the Wave window, then drag one bit out of the bus to display it separately. That action creates a one-bit virtual signal

whose definition is stored in a special location, and is not visible in the Signals window or to the normal virtual commands.

All other virtual signals are considered "explicit virtuals".

Virtual functions

Virtual functions behave in the GUI like signals but are not aliases of combinations or elements of signals logged by the kernel. They consist of logical operations on logged signals and may be dependent on simulation time. They may be displayed in the Signals, Wave or List windows, accessed by the **examine** command, but cannot be set by the **force** command.

Examples of virtual functions include the following:

- a function defined as the inverse of a given signal
- a function defined as the exclusive-OR of two signals
- a function defined as a repetitive clock
- a virtual 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 (see below).

Virtual functions are also implicitly created by ModelSim when referencing bit-selects or part-selects of Verilog registers in the GUI, or when expanding Verilog registers in the Signals, Wave or List windows. This is necessary because referencing Verilog register elements requires an intermediate step of shifting and masking of the Verilog "vreg" data structure.

Virtual regions

User-defined design hierarchy regions may be defined and attached to any existing design region or to the virtuals context tree. They may be used to reconstruct the RTL hierarchy in a gate-level design, and used to locate virtual signals. Thus, virtual signals and virtual regions may be used in a gate-level design to allow the RTL test bench with the gate-level design.

Virtual regions are created and attached using the **virtual region** command (see below).

Virtual types

User-defined enumerated types may 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.

Logfile and virtual commands reference table

The table below provides a brief description of the actions associated with logfile and virtual commands. For complete details about command syntax, arguments and usage, refer to the [ModelSim EE/SE Command Reference](#).

Command name	Action
searchLog (CR-144)	searches one or more of the currently open logfiles for a specified condition
virtual delete describe define (CR-182)	delete removes the matching virtuals; describe prints a complete description of the data type of one or more virtual signals; define prints the definition of the virtual signal or function in the form of a command that can be used to re-create the object
virtual expand (CR-183)	produces a list of all the non-virtual objects contained in the virtual signal(s)
virtual function (CR-184)	creates a new signal that consists of logical operations on existing signals and simulation time
virtual hide nohide (CR-188)	hide sets a flag in the specified real or virtual signals so that the signals do not appear in the Signals window; nohide resets the flag
virtual log nolog (CR-189)	log causes the sim-mode dependent signals of the specified virtual signals to be logged by the kernel; nolog causes the specified virtual signals to be un-logged by the kernel
virtual region (CR-190)	creates a new user-defined design hierarchy region
virtual save (CR-191)	saves the definitions of virtuals to a file

Command name	Action
virtual show count (CR-192)	show lists the full path names of all the virtuals explicitly defined; count counts the number of explicitly declared virtuals that have not been saved and that were not read in using a macro file
virtual signal (CR-193)	creates a new signal that consists of concatenations of signals and subelements
virtual type (CR-196)	creates a new enumerated type, used to convert signal values to character strings

10 - ModelSim EE Graphic Interface

Chapter contents

Window overview	150
Window features.	151
Main window	158
Dataflow window	170
List window.	174
Process window.	189
Signals window	192
Source window	200
Structure window	206
Variables window	209
Wave window	212
Compiling with the graphic interface	244
Setting default compile options	245
Simulating with the graphic interface	251
Setting default simulation options	260
ModelSim tools	263
Graphic interface commands	277
Customizing the interface	279

This chapter provides a detailed look at the debugging capabilities available within ModelSim's Tcl graphic interface.

The example graphics in this chapter illustrate ModelSim's graphic interface within a Windows 98/NT environment. ModelSim's interface is designed to provide consistency throughout all system environments. If you are using a UNIX or Windows version of ModelSim, your operating system provides the basic window-management frames, while ModelSim controls all internal window features such as menus, buttons, and scroll bars.

Because ModelSim's graphic interface is based on Tcl/Tk, you are able to customize your simulation environment. Easily-accessible preference variables and configuration commands give you control over the use and placement of windows, menus, menu options, and buttons.

Window overview

The ModelSim simulation and debugging environment consists of nine window types. Multiple windows of each type may be used during simulation (with the exception of the Main window). Make an additional window with the **View > New** menu selection in the Main window. A brief description of each window follows:

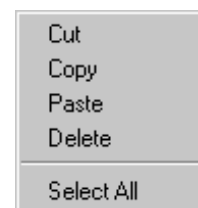
- [Main window](#) (10-158)
The main window from which all subsequent VSIM windows are available.
- [Dataflow window](#) (10-170)
Lets you trace signals and nets through your design by showing related processes.
- [List window](#) (10-174)
Shows the simulation values of selected VHDL signals, and Verilog nets and register variables in tabular format.
- [Process window](#) (10-189)
Displays a list of processes that are scheduled to run during the current simulation cycle.
- [Signals window](#) (10-192)
Shows the names and current values of VHDL signals, and Verilog nets and register variables in the region currently selected in the Structure window.
- [Source window](#) (10-200)
Displays the HDL source code for the design. (Your source code can remain hidden if you wish, see "[Source code security and -nodebug](#)" (E-443).)
- [Structure window](#) (10-206)
Displays the hierarchy of structural elements such as VHDL component instances, packages, blocks, generate statements, and Verilog model instances, named blocks, tasks and functions.
- [Variables window](#) (10-209)
Displays VHDL constants, generics, variables, and Verilog register variables in the current process and their current values.
- [Wave window](#) (10-212)
Displays waveforms, and current values for the VHDL signals, and Verilog nets and register variables you have selected. Current and past simulations can be compared side-by-side in one Wave window.

Window features

ModelSim's graphic interface provides many features that add to its usability; features common to many of the windows are described below.

Feature	Feature applies to these windows
Quick access toolbars (10-152)	Main, Source, and Wave
Drag and Drop (10-152)	Dataflow, List, Signals, Source, Structure, Variables, and Wave windows
Command history (10-153)	Main window command line
Automatic window updating (10-153)	Dataflow, Process, Signals, and Structure
Finding names, searching for values, and locating cursors (10-154)	various windows
Sorting HDL items (10-154)	Process, Signals, Source, Structure, Variables and Wave windows
Multiple window copies (10-154)	all windows except the Main window
Menu tear off (10-154)	all windows
Customizing menus and buttons (10-155)	all windows
Combine signals into a user-defined bus (10-155)	List and Wave windows
Tree window hierarchical view (10-155)	Structure, Signals, Variables, and Wave windows

- Press the <ESC> key to cancel any dialog box.
- Cut/Copy/Paste/Delete into any entry box by clicking the right mouse button in the entry box.
- Standard cut/copy/paste shortcut keystrokes – ^X/^C/^V – will work in all entry boxes.
- When the focus changes to an entry box, the contents of that box are selected (highlighted). This allows you to replace the current contents of the entry box with new contents with a simple paste command, without having to delete the old value.
- Dialog boxes will appear on top of their parent window (instead of the upper left corner of the screen)



- The transcript window now includes an edit popup menu activated via with the right mouse button.
- The middle mouse button will allow you to paste the following into the transcript window:
 - text currently selected in the transcript window,
 - a current primary X-Windows selection (may be from another application), or
 - contents of the clipboard.



Transcript edit popup

Note: Selecting text in the transcript window makes it the current primary X-Windows selection. This way you can copy transcript window selections to other X-Windows windows (xterm, emacs, etc.).

- The **Edit > Paste** operation in the transcript window will ONLY paste from the clipboard.
- All menus highlight their accelerator keys.

Quick access toolbars



Buttons on the Main, Source, and Wave windows provide access to commonly used commands and functions. See, "[The Main window tool bar](#)" (10-165), "[The Source window tool bar](#)" (10-203), and "[Wave window tool bar](#)" (10-220).

Drag and Drop

Drag and drop of HDL items is possible between the following windows. Using the left mouse button, click and release to select an item, then click and hold to drag it.

- **Drag items from these windows:**
Dataflow, List, Signals, Source, Structure, Variables, and Wave windows

- **Drop items into these windows:**
List and Wave windows

Note: Drag and drop works to rearrange items *within* the List and Wave windows as well.

Command history

Avoid entering long commands twice; use the down and up keyboard arrows to move through the command history for the current simulation.

Automatic window updating

Selecting an item in the following windows automatically updates other related ModelSim windows as indicated below:

Select an item in this window	To update these windows
Dataflow window (10-170) (with a process selected in the center of the window)	Process window (10-189)
	Signals window (10-192)
	Source window (10-200)
	Structure window (10-206)
	Variables window (10-209)
Process window (10-189)	Dataflow window (10-170)
	Signals window (10-192)
	Structure window (10-206)
	Variables window (10-209)
Signals window (10-192)	Dataflow window (10-170)
Structure window (10-206)	Signals window (10-192)
	Source window (10-200)

Finding names, searching for values, and locating cursors

- **Find** HDL item names with the **Edit > Find** menu selection in these windows: List, Process, Signals, Source, Structure, Variables, and Wave windows.
- **Search** for HDL item values with the **Edit > Search** menu selection in these windows: List, and Wave windows.

You can also:

- **Locate** time markers in the List window with the **Markers > Goto** menu selection.
- **Locate** time cursors in the Wave window with the **Cursor > Goto** menu selection.

In addition to the menu selections above, the virtual event <<**Find**>> is defined for all windows. The default binding is to <**Key-F19**> in most windows (the Find key on a Sun keyboard). You can bind <<**Find**>> to other events with the Tcl/Tk command **event add**. For example,

```
event add <<Find>> <control-Key-F>
```

Sorting HDL items

Use the **Edit > Sort** menu selection in the windows below to sort HDL items in ascending, descending or declaration order.

Process, Signals, Source, Structure, Variables and Wave windows

Names such as net_1, net_10, and net_2 will sort numerically in the Signals and Wave windows.

Multiple window copies

Use the **View > New** menu selection from the [Main window](#) (10-158) to create multiple copies of the same window type. The new window will become the default window for that type.

Menu tear off

All window menus may be "torn off " to create a separate menu window. To tear off, click on the menu, then select the dotted-line button at the top of the menu.

Customizing menus and buttons

Menus can be added, deleted, and modified in all windows. Custom buttons can also be added to window tool bars. See

- ["Customizing the interface"](#) (10-279),
- ["Customizing menus and buttons"](#) (10-155), and
- ["The Button Adder"](#) (10-265) more information.

Combine signals into a user-defined bus

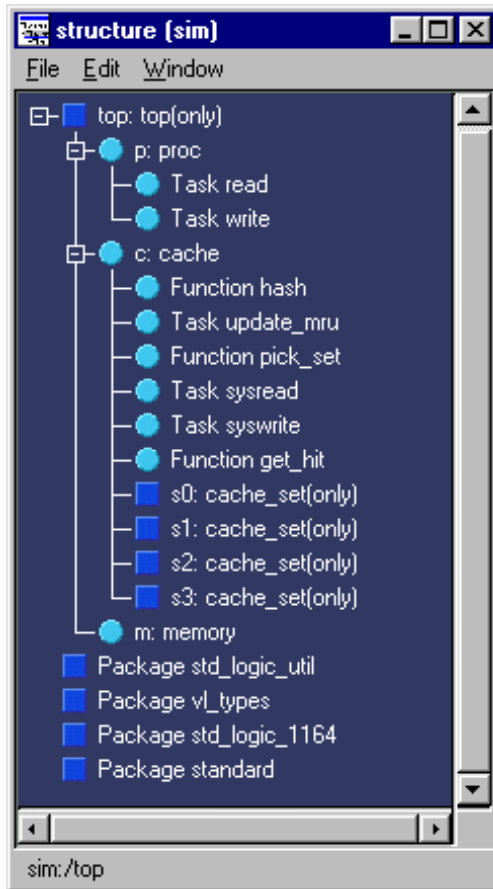
You can collect selected items in the [List window](#) (10-174) and [Wave window](#) (10-212) displays and combine them into a bus named by you. In the List window, the **Edit > Combine** menu selection allows you to move the selected items to the new bus as long as they are all scalars or arrays of the same base type (records are not yet supported).

In the [Wave window](#) (10-212), the **Edit > Combine** menu selection requires all selected items to be either all scalars or all arrays of the same size. The benefit of this added restriction is that the bus can be expanded to show each element as a separate waveform. Using the **flatten** option allows scalars and various array sizes to be mixed, but foregoes display of child waveforms.

The **keep** option in both windows copies the signals rather than moving them.

Tree window hierarchical view

ModelSim provides a hierarchical, or "tree view" of some aspect of your design in the Structure, Signals, Variables, and Wave windows.



HDL items you can view

Depending on which window you are viewing, one entry is created for each of the following VHDL and Verilog HDL item within the design:

VHDL items

(indicated by a dark blue square icon)
signals, variables, component instantiation, generate statement, block statement, and package

Verilog items

(indicated by a lighter blue circle icon)
parameters, registers, nets, module instantiation, named fork, named begin, task, and function

Virtual items

(indicated by an orange diamond icon)
virtual signals, buses, and functions, see ["Virtual Objects \(User-defined buses, and more\)"](#) (9-144) for more information

Viewing the hierarchy

Whenever you see a tree view, as in the Structure window displayed here, you can use the mouse to collapse or expand the hierarchy. Select the symbols as shown below to change the view of the structure.

Symbol	Description
[+]	click a plus box to expand the item and view the structure
[-]	click a minus box to hide a hierarchy that has been expanded

Finding items within tree windows

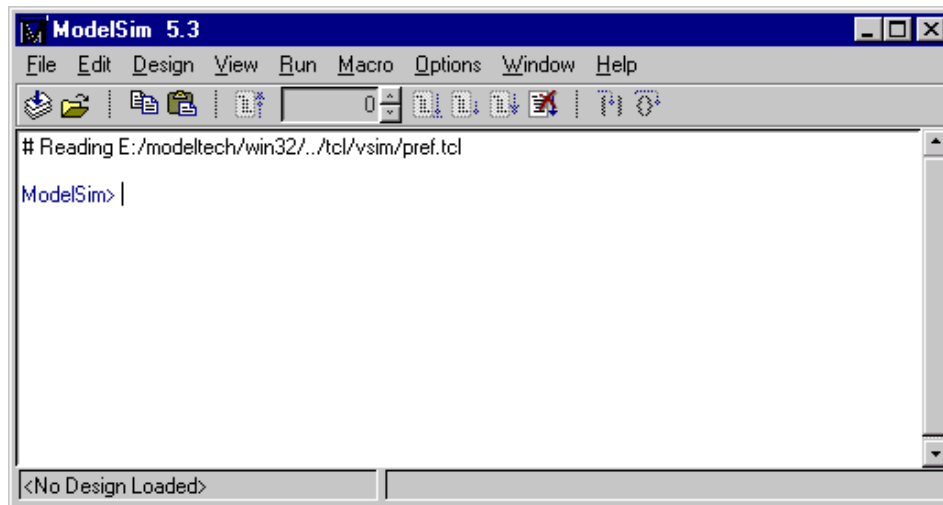
You can open the find dialog box within all windows (except the Main, and Source windows) by using **<control-f>**.

Options within the Find dialog box allow you to search unique text-string fields within the specific window. See also,

- ["Finding items by name in the List window"](#) (10-184),
- ["Finding HDL items in the Signals window"](#) (10-198), and
- ["Finding items by name or value in the Wave window"](#) (10-231).

Main window

The Main window is pictured below as it appears when VSIM is first invoked. Note that your operating system graphic interface provides the window-management frame only; ModelSim handles all internal-window features including menus, buttons, and scroll bars.



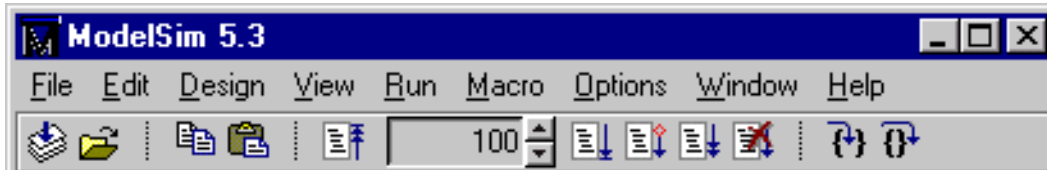
The menu bar at the top of the window provides access to a wide variety of simulation commands and ModelSim preferences. The status bar at the bottom of the window gives you information about the data in the active ModelSim window. The tool bar provides buttons for quick access to the many common commands.

When a simulation is running, the Main window displays a VSIM prompt, allowing you to enter command-line commands from within the graphic interface. Messages output by VSIM during simulation are also displayed in this window. You can scroll backward and forward through the current work history by using the vertical scrollbar. You can also copy and paste using the mouse within the window, see ["Editing the command line, the current source file, and notepads"](#) (10-167).

The Main window menu bar, tool bar, and status bar are detailed below.

The Main window menu bar

The menu bar at the top of the Main window lets you access many *ModelSim* commands and features. The menus are listed below with brief descriptions of the command's use.



File menu

New	provides the following four options: New Folder – create a new folder in the current directory New Source – create a project source file Import Source – import a project source file New Project – create a new project from scratch or copy an existing project
Open	provides the following three options: Open Source – opens the selected project source file Open Project – opens the selected .mpf project file Open Dataset – view new design simulation logfile (.wlf); enter short name for this design view
Delete	provides the following two options: Delete Project – delete the selected .mpf project file Delete Source – delete the selected project source file
Change Directory	change to a different working directory
Save Transcript	save the current contents of the transcript window to the file indicated with a "Save Transcript as" selection (this selection is not initially available because the transcript is written to the <i>transcript</i> file by default), see "Saving the Main window transcript file" (10-164)
Save Transcript as...	save the current contents of the transcript window to a file
Clear Transcript	clear the Main window transcript display

Options (all options are set for the current session only)	Transcript File: sets a transcript file to save for this session only Command History: file for saving command history only, no comments Save File: sets filename for Save Main, and Save Main as Saved Lines: limits the number of lines saved in the transcript (default is all) Line Prefix: specify the comment prefix for the transcript Update Rate: specify the update frequency for the Main status bar ModelSim Prompt: change the title of the ModelSim prompt VSIM Prompt: change the title of the VSIM prompt Paused Prompt: change the title of the Paused prompt
<path list>	Windows only - a list of the most recent working directory changes
Quit	quit ModelSim (returns to the command line if UNIX)

Edit menu

Copy	copy the selected text
Paste	paste the previously cut or copied item to the left of the currently selected item
Select All	select all text in the Main window transcript
Unselect All	deselect all text in the Main window transcript
Find	search the transcript forward or backward for the specified text string

Design menu

Browse Libraries	browse all libraries within the scope of the design; see also " Viewing and deleting library contents " (2-35)
Create a New Library	create a new library or map a library to a new name; see " Library management commands " (2-33), and " Assigning a logical name to a design library " (2-37)
View Library Contents	view or delete the contents of a library; see also " Viewing and deleting library contents " (2-35)
FPGA Library Manager	build and install libraries provided by FPGA vendors for use within ModelSim, see the " The FPGA Library Manager " (10-274) for more information
Compile	compiles HDL source files into the current project's work library

Compile Project	recompile all of the previously compiled files in the current project
Load New Design	initiate simulation by specifying top level design unit in the Design tab; specify HDL specific simulator settings with the VHDL and Verilog tabs; specify setting relating to the annotation of design timing with the SDF tab
End Simulation	end the simulation (returns to the ModelSim command line)

View menu

All	open all VSIM windows
Source	open and/or view the Source window (10-200)
Structure	open and/or view the Structure window (10-206)
Variables	open and/or view the Variables window (10-209)
Signals	open and/or view the Signals window (10-192)
List	open and/or view the List window (10-174)
Process	open and/or view the Process window (10-189)
Wave	open and/or view the Wave window (10-212)
Dataflow	open and/or view the Dataflow window (10-170)
Datasets	opens the Dataset Browser for selecting the current Dataset
New	create a new VSIM window of the specified type
Other	if the Performance Analyzer and/or Code Coverage is turned on (available only in ModelSim EE Special Edition), this selection will allow viewing of: Hierarchical Profile, Ranked Profile and Source Coverage

Run menu

Run <default>	run simulation for one default run length; change the run length with Options > Simulation, or use the Run Length list on the tool bar
Run -All	run simulation until you stop it; see also the run command (CR-139)
Continue	continue the simulation; see also the run command (CR-139) and the -continue option

Run -Next	run to the next event time
Step	single-step the simulator; see also the step command (CR-151)
Step-Over	execute without single-stepping through a subprogram call
Restart	reloads the design elements and resets the simulation time to zero; only design elements that have changed are reloaded; see also the restart command (CR-133)

Macro menu

Execute Macro	allows you to browse for and execute a DO file (macro)
Execute Old PE Macro...	calls and executes old PE 4.7 macro without changing the macro to EE 5.3; backslashes may be selected as pathname delimiters
Convert Old PE Macro...	converts old PE 4.7 macro to EE 5.3 macro without changing the file; backslashes may be selected as pathname delimiters
Macro Helper	UNIX only - invokes the Macro Helper tool; see also " The Macro Helper " (10-266)
Tcl Debugger	invokes the Tcl debugger, TDebug; see also " The Tcl Debugger " (10-267)
TclPro Debugger	invokes TclPro Debugger by Scriptics® if installed. TclPro Debugger can be acquired from Scriptics at www.scriptics.com .

Options menu

Compile	returns the Compile Options dialog box; options cover both VHDL and Verilog compile options; see also " Setting default compile options " (10-245)
Simulation	returns the Simulation Options dialog box; options include: default radix, default force type, default run length, iteration limit, warning suppression, and break on assertion specification; see also " Setting default simulation options " (10-260)
Edit Preferences...	returns the Preferences dialog box; color preferences can be set for window background, text and graphic items (i.e., waves in the Wave window)

Edit Project	allows modification of project-wide settings that describe the makeup of the project
Save Preferences	save current ModelSim settings to a Tcl preference file; saves preferences as Tcl arrays, see " Preference variable arrays " (B-407)

Window menu

Initial Layout	restore all windows to the size and placement of the initial full-screen layout
Cascade	cascade all open windows
Tile Horizontally	tile all open windows horizontally
Tile Vertically	tile all open windows vertically
Icon Children	icon all but the Main window
Icon All	icon all windows
Deicon All	deicon all windows
Customize	use to define and add a button to either the menu bar, tool bar, or status bar of the specified window
<window_name>	lists the currently open windows; select a window name to switch to, or show that window if it is hidden; when the source window is available, the source file name is also indicated; open additional windows from the " View menu " (10-161) in the Main window, or use the view command (CR-180)

Help menu

About ModelSim	display ModelSim application information
Release Notes	view current release notes with the ModelSim notepad (CR-104)
Enable Welcome	enables Welcome screen for starting a new project or opening an existing project when ModelSim is initiated
Quick Start Menu	enables " Quick Start " (10-264), which explains the process of setting up a project, compiling project source files, simulating a project and modifying it; also includes three step by step simulation examples
Information about Help	view the readme file pertaining to ModelSim's online documentation

EE Documentation	open and read the ModelSim documentation in PDF format; PDF files can be read with a free Adobe Acrobat reader available through www.adobe.com
Tcl Help	open the Tcl command reference (man pages) in Windows help format
Tcl Syntax	open and read Tcl syntax details in HTML format
Tcl Man Pages	open and read Tcl /Tk 8.0 manual in HTML format
Technotes	select a technical note to view from the drop-down list

Saving the Main window transcript file

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

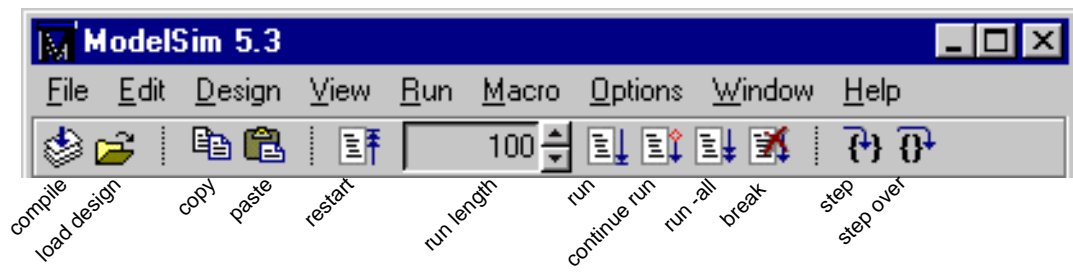
If you would like to save an additional copy of the transcript with a different filename, you can use the File > Save Main As, or File > Save Main menu items. The initial save must be made with the Save Main As selection, which stores the filename in the Tcl variable PrefMain(saveFile). Subsequent saves can be made with the Save Main selection. Since no automatic saves are performed for this file, it is written only when a Save... menu selection is made. The file is written to the current working directory and records the contents of the transcript at the time of the save.





Using the saved transcript as a macro (DO file)








Saved transcript files can be used as macros (DO files), see, or the **do** command (CR-67) for more information.


The Main window tool bar

Buttons on the Main window tool bar give you quick access to these ModelSim commands and functions.

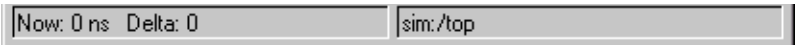


Main window tool bar buttons		
Button	Menu equivalent	Command equivalents
 Compile open the Compile HDL Source Files dialog box to select files for compilation	none, however, Options > Compile opens the Compile Options dialog box	vcom <arguments>, or vlog <arguments> see: vdel (CR-175) or vlog (CR-199)
 Load Design open the Load a Design dialog box to initiate simulation	File > Load New Design	vsim <arguments> see: vsim (CR-208)
 Copy copy the selected text within the Main window transcript	Edit > Copy	see: "Editing the command line, the current source file, and notepads" (10-167)
 Paste paste the copied text to the cursor location	Edit > Paste	see: "Editing the command line, the current source file, and notepads" (10-167)

Main window tool bar buttons		
Button	Menu equivalent	Command equivalents
 Restart restart the current simulation with the option of use current formatting, breakpoints, and logfile	File > Restart	restart <arguments> see: restart (CR-133)
 Run Length specify the run length for the current simulation	none	run <specific run length> see: run (CR-139)
 Run run the current simulation for the default time length	Run > Run <default_run_length>...	run (no arguments) see: run (CR-139)
 Continue Run continue the current simulation run	Run > Continue	run -continue see: run (CR-139)
 Run -All run to current simulation forever, or until it hits a breakpoint or specified break event *	Run > Run -All	run -all see: run (CR-139), * see " Assertion settings page " (10-262)
 Break stop the current simulation run	none	none
 Step steps the current simulation to the next HDL statement	Run > Step....	step see: step (CR-151)

Main window tool bar buttons		
Button	Menu equivalent	Command equivalents
 Step Over HDL statements are executed but treated as simple statements instead of entered and traced line by line	Run > Step Over....	step -over see: step (CR-151)

The Main window status bar



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

Field	Description
Now	the current simulation time, using the default resolution units specified in "Simulating with the graphic interface" (10-251), or a larger time unit if one can be used without a fractional remainder
Delta	the current simulation iteration number
<dataset name>	name of the current dataset (item selected in the Structure window (10-206))

Editing the command line, the current source file, and notepads

The following mouse actions and special keystrokes can be used to edit commands in the entry region of the Main window. They can also be used in editing the file

displayed in the Source window and all Notepad windows (enter the **notepad** command within ModelSim to open the Notepad editor).

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

Keystrokes - UNIX	Keystrokes - Windows	Result
< left right - arrow >		move the insertion cursor
< up down - arrow >		scroll through command history
< control - p >		move insertion cursor to previous line
< control - n >		move insertion cursor to next line
< control - f >		move insertion cursor forward
< control - b >		move insertion cursor backward
< backspace >		delete character to the left
< control - d >		delete character to the right
< control - k >		delete to the end of line

Keystrokes - UNIX	Keystrokes - Windows	Result
< control - a >	< control - a >, <Home>	move insertion cursor to beginning of line
< control - e >	< control - e >, <End>	move insertion cursor to end of line
< * meta - "<" >	none	move insertion cursor to beginning of file
< * meta - ">" >	none	move insertion cursor to end of file
< control - x >		cut selection
< control - c >		copy selection
< control - v >		insert clipboard

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.

*** UNIX only**

Which keyboard key functions as the meta key depends on how your X-windows KeySym mapping is set up. You may need help from your system administrator to map a particular key, such as the <alt> key, to the meta KeySym.

Dataflow window

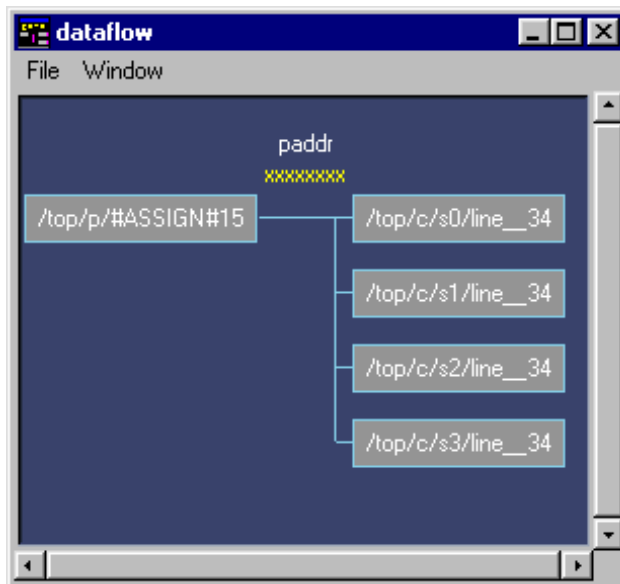
The Dataflow window allows you to trace VHDL signals or Verilog nets through your design. Double-click an item with the left mouse button to move it to the center of the Dataflow display.

VHDL signals or processes in the Dataflow window:

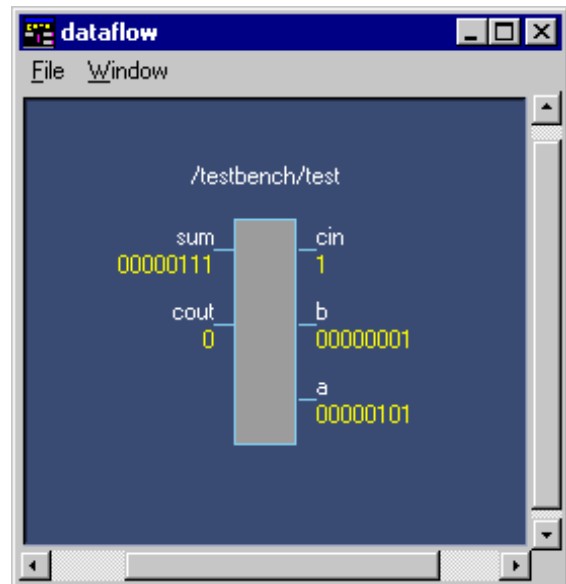
- A signal displays in the center of the window with all the processes that drive the signal on the left, and all the processes that read the signal on the right, or
- a process is displayed with all the signals read by the process shown as inputs on the left of the window, and all the signals driven by the process on the right.

Verilog nets or processes in the Dataflow window:

- A net displays in the center of the window with all the processes that drive the net on the left, and all the processes triggered by the net on the right, or
- a process is displayed with all the nets that trigger the process shown as inputs on the left of the window, and all the nets driven by the process on the right.



signal or net



process

The Dataflow window menu bar

The following menu commands and button options are available from the Dataflow window menu bar.

File menu

Save Postscript	save the current dataflow view as a Postscript file; see "Saving the Dataflow window as a Postscript file" (10-173)
Selection	Selection > Follow Selection updates window when the Process window (10-189) or Signals window (10-192) changes; Selection > Fix Selection freezes the view selected from within the Dataflow window
Close	close this copy of the Dataflow window; you can create a new window with View > New from the "The Main window menu bar" (10-159)

Window menu

Initial Layout	restore all windows to the size and placement of the initial full-screen layout
Cascade	cascade all open windows
Tile Horizontally	tile all open windows horizontally
Tile Vertically	tile all open windows vertically
Icon Children	icon all but the Main window
Icon All	icon all windows
Deicon All	deicon all windows
Customize	use The Button Adder (10-265) to define and add a button to either the menu bar, tool bar, or status bar of the specified window
<window_name>	lists the currently open windows; select a window name to switch to, or show that window if it is hidden; when the source window is available, the source file name is also indicated; open additional windows from the "View menu" (10-161) in the Main window, or use the view command (CR-180)

Tracing HDL items with the Dataflow window

The Dataflow window is linked with the [Signals window](#) (10-192) and the [Process window](#) (10-189). To examine a particular process in the Dataflow window, click on the process name in the Process window. To examine a particular HDL item in the Dataflow window, click on the item name in the Signals window.

with a signal in center of the Dataflow window, you can:

- click once on a process name in the Dataflow window to make the Source and Variable windows update to show that process,
- click twice on a process name in the Dataflow window to move the process to the center of the Dataflow window

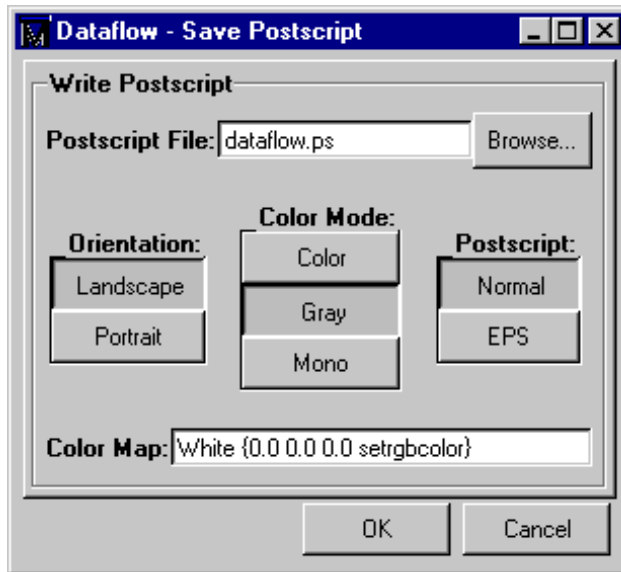
with a process in center of the Dataflow window, you can:

- click once on an item name to make the Source and Signals windows update to show that item,
- click twice on an item name to move that item to the center of the Dataflow window.

The Dataflow window will display the current process when you single-step or when VSIM hits a breakpoint.

Saving the Dataflow window as a Postscript file

Use this Dataflow window menu selection: **File > Save Postscript** to save the current Dataflow view as a Postscript file. Configure the Postscript output with the following dialog box, or use the Preferences dialog box from this Main window selection: **Option > Edit Preferences**.

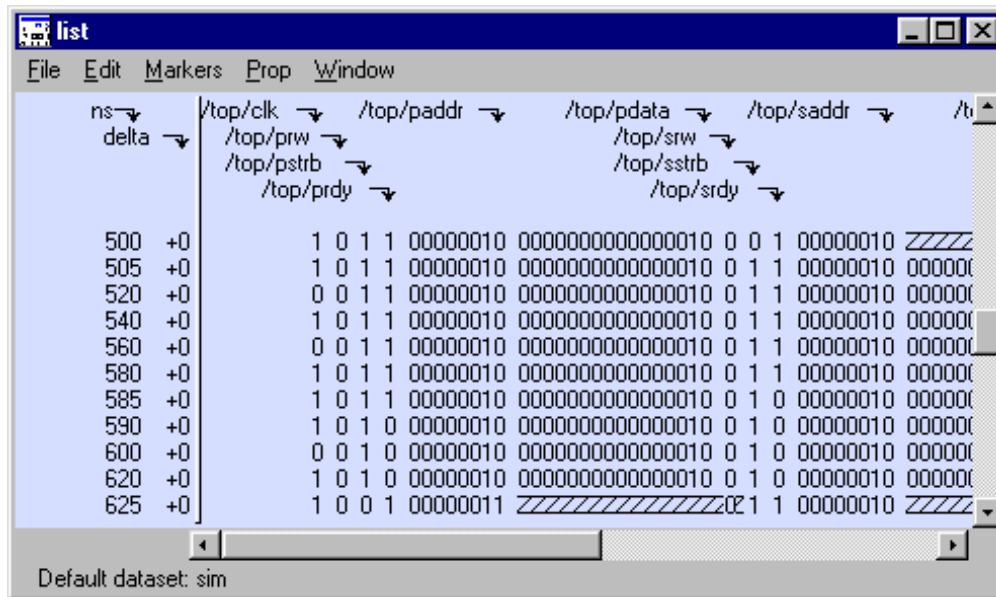


The dialog box has the following options:

- **Postscript File**
specify the name of the file to save, default is *dataflow.ps*
- **Orientation**
specify **Landscape** (horizontal) or **Portrait** (vertical) orientation
- **Color Mode**
specify **Color** (256 colors), **Gray** (gray-scale) or **Mono** color mode
- **Postscript**
specify Normal Postscript or EPS (Encapsulated Postscript) file type
- **Color Map**
specify the color mapping from current Dataflow window colors to Postscript colors

List window

The List window displays the results of your simulation run in tabular format. The window is divided into two adjustable panes, which allow you to scroll horizontally through the listing on the right, while keeping time and delta visible on the left.



HDL items you can view

One entry is created for each of the following VHDL and Verilog HDL items within the design:

- *VHDL items*
signals and process variables
- *Verilog items*
nets and register variables

Note: Constants, generics, parameters, and memories are not viewable in the List or Wave windows.

The List window menu bar

The following menu commands and button options are available from the List window menu bar.

File menu

Write List (format)	save the listing as a text file in one of three formats: tabular, events, or TSSI
Load Format	run a List window format DO file previously saved with Save Format
Save Format	saves the current List window display and signal preferences to a do (macro) file; running the DO file will reformat the List window to match the display as it appeared when the DO file was created
Close	close this copy of the List window; you can create a new window with View > New from the "The Main window menu bar" (10-159)

Edit menu

Cut	cut the selected item field from the listing; see "Editing and formatting HDL items in the List window" (10-181)
Copy	copy the selected item field
Paste	paste the previously cut or copied item to the left of the currently selected item
Delete	delete the selected item field
Combine	combine the selected fields into a user-defined bus; keep copies of the original items rather than moving them; see "Combine signals into a user-defined bus" (10-155)
Select All	select all signals in the List window
Unselect All	deselect all signals in the List window
Find...	find specified item label within the List window
Search...	search the List window for a specified value, or the next transition for the selected signal

Markers menu

Add Marker	add a time marker at the top of the listing page
Delete Marker	delete the selected marker from the listing
Goto	choose the time marker to go to from a list of current markers

Prop menu

Display Props	set display properties for all items in the window: delta settings, trigger on selection, strobe period, and label size
Signal Props	set label, radix, trigger on/off, and field width for the selected item

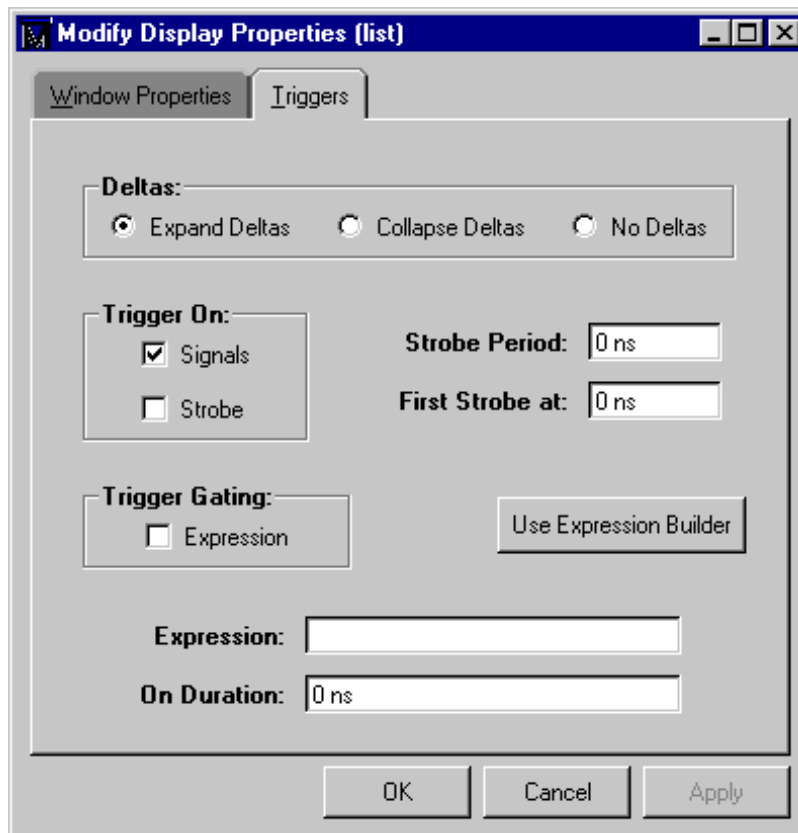
Window menu

Initial Layout	restore all windows to the size and placement of the initial full-screen layout
Cascade	cascade all open windows
Tile Horizontally	tile all open windows horizontally
Tile Vertically	tile all open windows vertically
Icon Children	icon all but the Main window
Icon All	icon all windows
Deicon All	deicon all windows
Customize	use The Button Adder (10-265) to define and add a button to either the menu bar, tool bar, or status bar of the specified window
<window_name>	lists the currently open windows; select a window name to switch to, or show that window if it is hidden; when the source window is available, the source file name is also indicated; open additional windows from the "View menu" (10-161) in the Main window, or use the view command (CR-180)

Setting List window display properties

Before you add items to the List window you can set the window's display properties. To change when and how a signal is displayed in the List window, make this selection from the List window menu bar: **Prop > Display Props**. The resulting Modify Display Properties dialog box has the following options.

Trigger settings page



The Triggers page controls the triggering for the display of new lines in the List window. You can specify whether an HDL item trigger or a strobe trigger is used to determine when the List window displays a new line. If you choose **Trigger on: Signals**, then you can choose between collapsed or expanded delta displays. You can also choose a combination of signal or strobe triggers. To use gating, Signals or Strobe or both must be selected.

The Triggers page includes the following options:

- **Deltas:Expand Deltas**
When selected with the **Trigger on: Signals** check box, displays a new line for

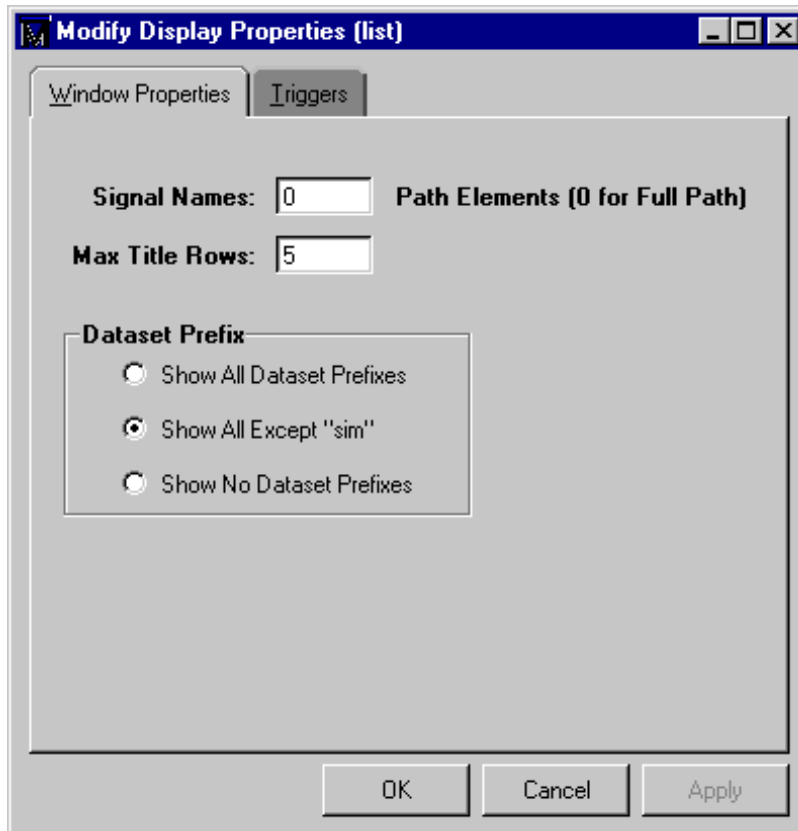
each time step on which items change, including deltas within a single unit of time resolution.

- **Deltas:Collapse Deltas**
Displays only the final value for each time unit in the List window.
- **Deltas:No Deltas**
No simulation cycle (delta) column is displayed in the List window.
- **Trigger On: Signals**
Triggers on signal changes. Defaults to all signals. Individual signals may be excluded from triggering by using the **Prop > Signals Props** dialog box or by originally adding them with the **-nottrigger** option to the **add list** command (CR-22).
- **Trigger On: Strobe**
Triggers on the **Strobe Period** you specify; specify the first strobe with **First Strobe at:**.
- **Trigger Gating: Expression**
Enables triggers to be gated on and off by an overriding expression, much like a hardware signal analyzer might be set up to start recording data on a specified setup of address bits and clock edges. Affects the display of data, not the acquisition of the data.
- **Use Expression Builder** (button)
Opens the Expression Builder to help you write a gating expression. See "[The GUI Expression Builder](#)" (10-272)
- **Expression**
Enter the expression for trigger gating into this field, or use the Expression Builder (select the Use Expression Builder button). The expression is evaluated when the List window would normally have displayed a row of data (given the trigger on signals and strobe settings above).
- **On Duration**
The duration for gating to remain open after the last list row in which the expression evaluates to true; expressed in x number of default timescale units. Gating is level-sensitive rather than edge-triggered.

List window gating information is saved as configuration statements when the list format is saved. The gating portion of a configuration statement might look like this:

```
.list.tbl config -usegating 1
.list.tbl config -gateduration 100
.list.tbl config -gateexpr {<expression>}
```

Window Properties page



The **Window Properties** page includes these options:

- **Signal Names**
Allows you to determine the number of path elements to be shown in the List window. For example, "0" shows the full path. "1" shows only the leaf element.

- **Max Title Rows**
The maximum number of rows in the name pane.

- **Dataset Prefix: Show All Dataset Prefixes**
Display the dataset prefix associated with each signal pathname. Useful for displaying signals from multiple datasets.

- **Dataset Prefix: Show All Except "sim"**

Display all dataset prefixes except the one associated with the current simulation – "sim." Useful for displaying signals from multiple datasets.

- **Dataset Prefix: Show No Dataset Prefixes**
Do not display any dataset prefixes.

Adding HDL items to the List window

Before adding items to the List window you may want to set the window display properties (see ["Setting List window display properties"](#) (10-177)). You can add items to the List window in several ways.

Adding items with drag and drop

You can drag and drop items into the List window from the Process, Signals, or Structure window. Select the items in the first window, then drop them into the List window. Depending on what you select, all items or any portion of the design may be added.

Adding items from the Main window command line

Invoke the **add list** (CR-22) command to add one or more individual items; separate the names with a space:

```
add list <item_name> <item_name>
```

You can add all the items in the current region with this command:

```
add list *
```

Or add all the items in the design with:

```
add list -r \*
```

Adding items with a List window format file

To use a List window format file you must first save a format file for the design you are simulating. The saved format file can then be used as a DO file to recreate the List window formatting.

- add HDL items to your List window
- edit and format the items to create the view you want, see ["Editing and formatting HDL items in the List window"](#) (10-181)
- save the format to a file with the List window menu selection:
File > Save Format

To use the format (do) file, start with a blank List window, and run the DO file in one of two ways:

- use the **do** (CR-67) command on the command line:

```
do <my_list_format>
```
- select **File > Load Format** from the List window menu bar

Use **Edit > Select All** and **Edit > Delete** to remove the items from the current List window or create a new, blank List window with the **View > New > List** selection from the "Main window" (10-158). You may find it useful to have two differently formatted windows open at the same time, see "Examining simulation results with the List window" (10-183).

Note: List window format files are design-specific; use them only with the design you were simulating when they were created. If you try to the wrong format file, ModelSim will advise you of the HDL items it expects to find.

Editing and formatting HDL items in the List window

Once you have the HDL items you want in the List window, you can edit and format the list to create the view you find most useful. (See also, "Adding HDL items to the List window" (10-180))

To edit an item:

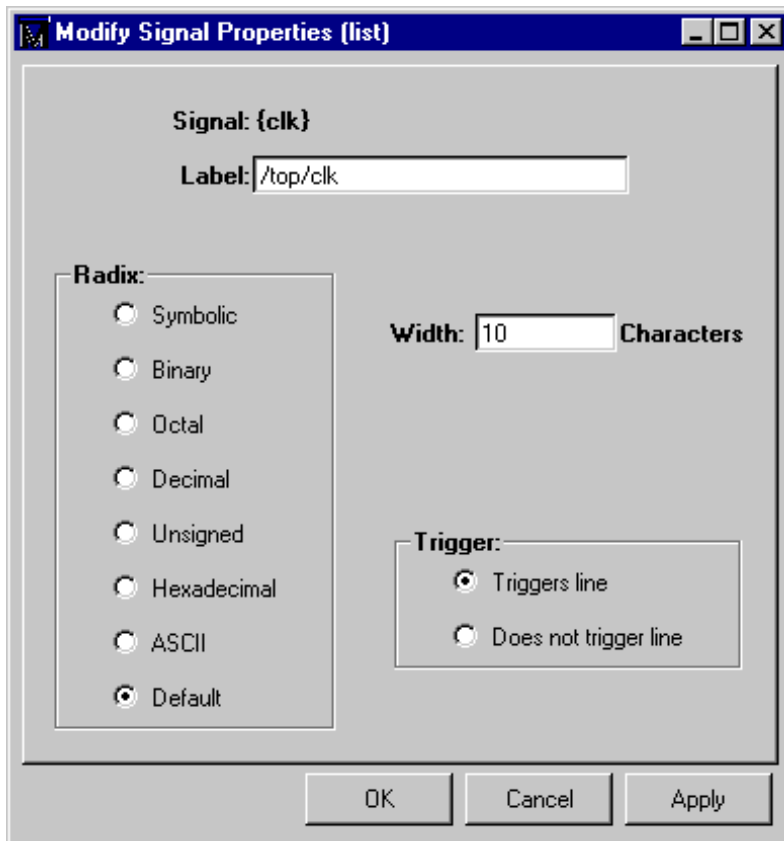
Select the item's label at the top of the List window or one of its values from the listing. Move, copy or remove the item by selecting commands from the List window **Edit menu** (10-175) menu.

You can also click+drag to move items within the window:

- to select several contiguous items:
click+drag to select additional items to the right or the left of the original selection
- to select several items randomly:
Control+click to add or subtract from the selected group
- to move the selected items:
re-click on one of the selected items, hold and drag it to the new location

To format an item:

Select the item's label at the top of the List window or one of its values from the listing, then use the **Prop > Signal Props** menu selection. The resulting Modify Signal Properties dialog box allows you to set the item's label, label width, triggering, and radix.



The **Modify Signal Properties** dialog box includes these options:

- **Signal**

Shows the signal you selected with the mouse with its dataset prefix.

- **Label**

Allows you to specify the label that is to appear at the top of the List window column for the specified item.

- **Radix**

Allows you to specify the radix (base) in which the item value is expressed. The default radix is symbolic, which means that for an enumerated type, the List window lists the actual values of the enumerated type of that item.

For the other radices - binary, octal, decimal, unsigned, hexadecimal, or ASCII - the item value is converted to an appropriate representation in that radix. In the system initialization file, *modelsim.tcl*, you can specify the list translation rules for arrays of enumerated types for binary, octal, decimal, unsigned decimal, or hexadecimal item values in the design unit.

- **Width**

Allows you to specify the desired width of the column used to list the item value. The default is an approximation of the width of the current value.

- **Trigger: Triggers line**

Specifies that a change in the value of the selected item causes a new line to be displayed in the List window.

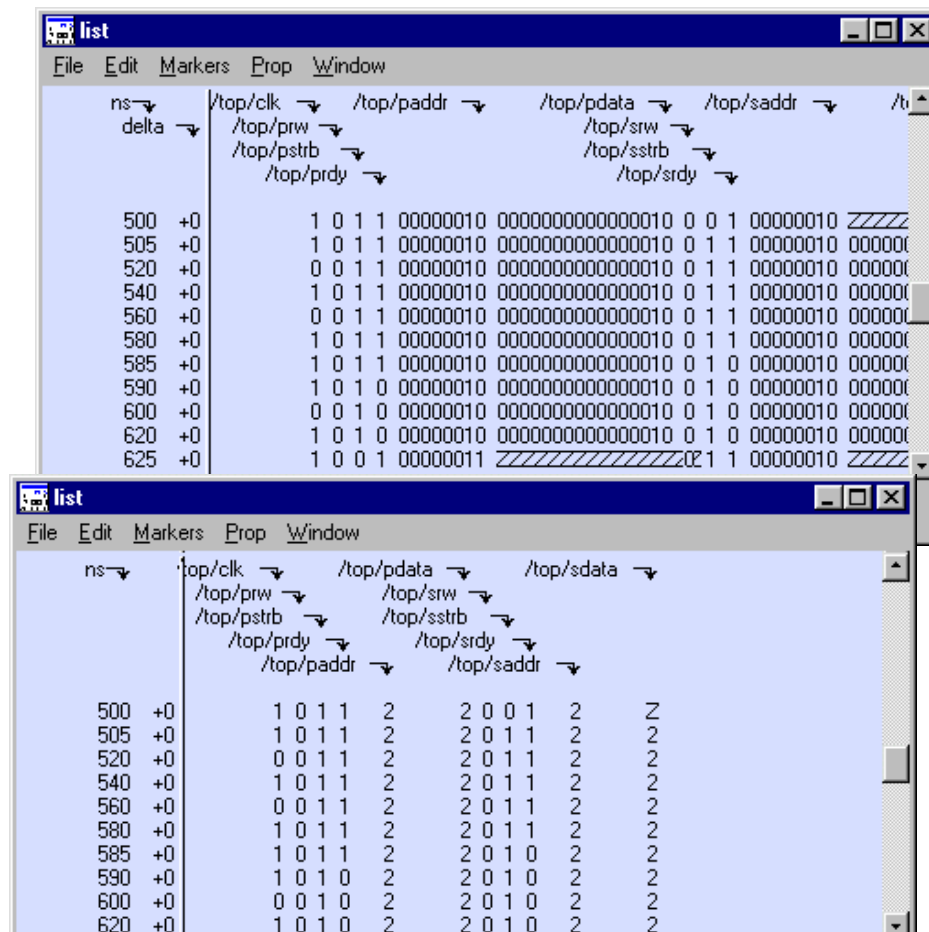
- **Trigger: Does not trigger line**

Selecting this option in the List Signals window specifies that a change in the value of the selected item does not affect the List window.

The trigger specification affects the trigger property of the selected item. See also, "[Setting List window display properties](#)" (10-177).

Examining simulation results with the List window

Because you can use the Main window [View menu](#) (10-161) to create a second List window, you can reformat another List window after the simulation run if you decide a different format would reveal the information you're after. Compare the two illustrations.



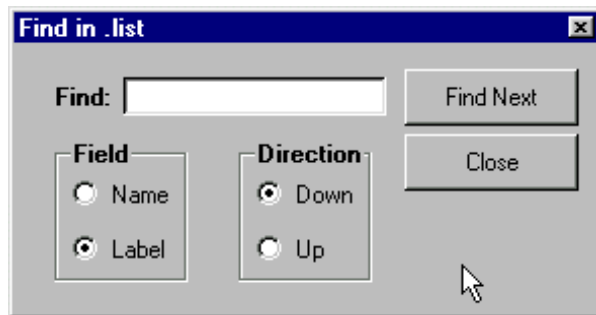
The divider bar separates resolution and delta from values; signal values are listed in symbolic format; and an item change triggers a new line.

Signal values are listed in decimal format; a 20ns strobe triggers a new line

In the first List window, the HDL items are formatted as symbolic and use an item change to trigger a line; the field width was changed to accommodate the default label width. The window divider maintains the time and delta in the left pane; signals in the right pane may be viewed by scrolling. For the second listing, the specification for triggering was changed to a 100-ns strobe, and the item radix for **a**, **b**, **cin**, and **sum** is now decimal.

Finding items by name in the List window

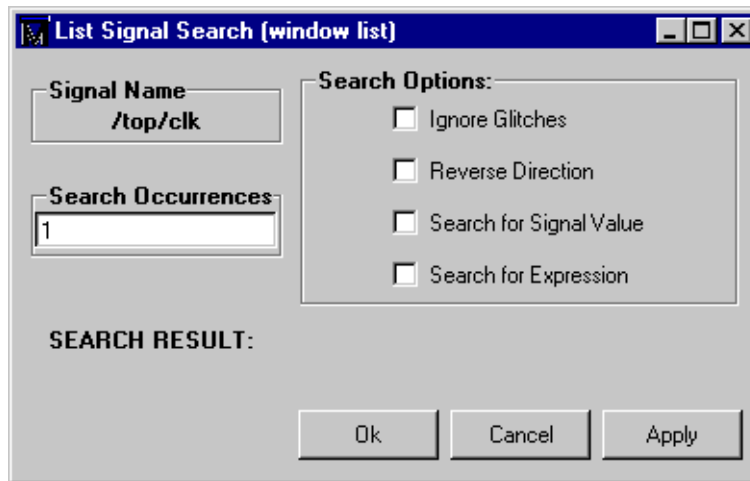
The Find dialog box allows you to search for text strings in the List window. From the List window select **Edit > Find** to bring up the Find dialog box.



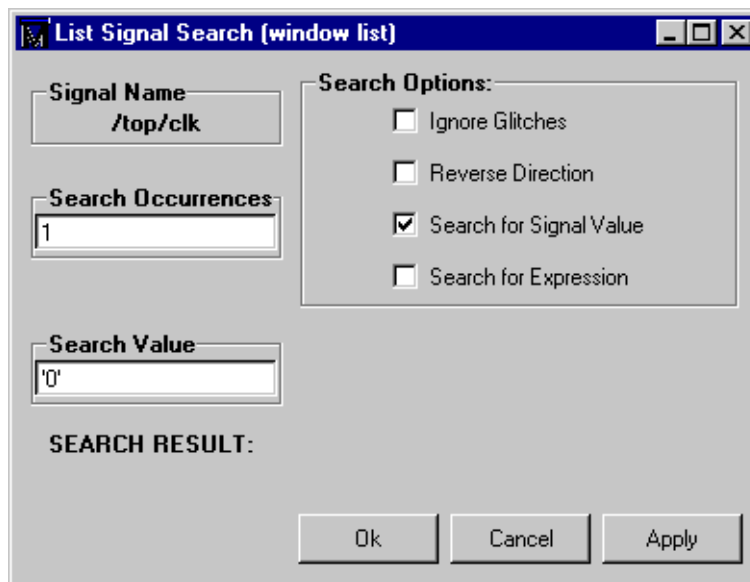
Enter an item label and **Find** it by searching **Forward** (right) or **Reverse** (left) through the List window display. The column number of the item found displays at the bottom of the dialog box. Note that you can change an item's label, see ["Setting List window display properties"](#) (10-177).

Searching for item values in the List window

Select an item in the List window. From the List window menu bar select **Edit > Search** to bring up the List Signal Search dialog box.



The List Signal Search dialog expands with Search for Signal Value selected (shown below)



The List Signal Search dialog box includes these options:

- **Signal Name <item_label>**

This indicates the item currently selected in the List window; the subject of the search.

- **Search Options: Ignore Glitches**

Ignore zero width glitches in VHDL signals and Verilog nets.

- **Search Options: Reverse Direction**

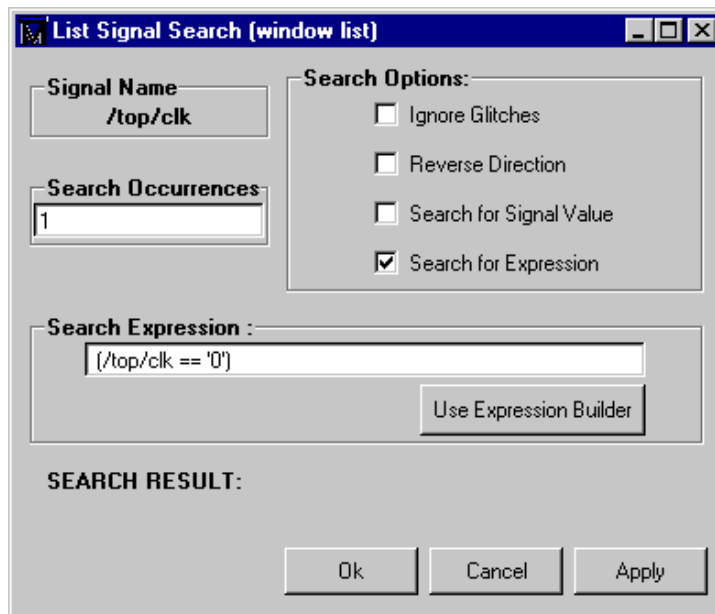
Select to search the list from bottom to top. Deselect to search from top to bottom.

- **Search Options: Search for Signal Value**

Reveals the Search Value field; search for the value specified in the Search Value field (the value must be formatted in the same radix as the display). If no value is specified look for transitions.

- **Search Options: Search for Expression**

Reveals the Search Expression field and the Use Expression Builder button; searches for the expression specified in the Search Expression field evaluating to a boolean true.



List Signal Search dialog box with Search for Expression selected

indicates the number of transitions or matches for which to search.

The result of your search is indicated at the bottom of the dialog box.

Setting time markers in the List window

From the List window select **Markers > Add Marker** to tag the selected list line with a marker. The marker is indicated by a thin box surrounding the marked line. The selected line uses the same indicator, but its values are highlighted. Delete markers by first selecting the marked line, then making the **Markers > Delete Marker** menu selection.

The expression may involve more than one signal but is limited to signals logged in the List window.

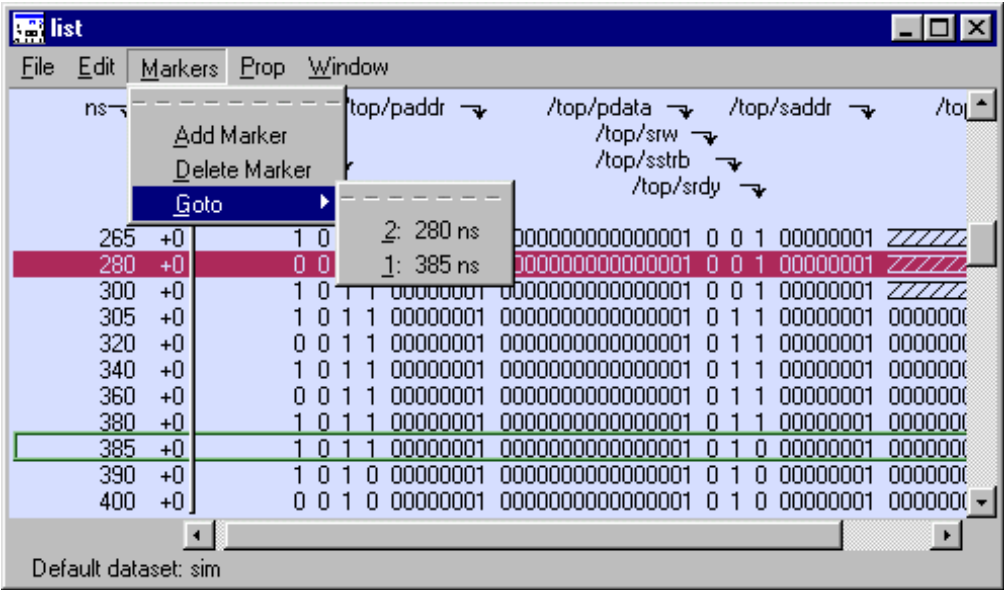
Expressions may include constants, variables and macros. If no expression is specified, the search will give an error. See ["GUI_expression_format"](#) (CR-250) for information on creating an expression.

To help build the expression, click the **Use Expression Builder** button to open ["The GUI Expression Builder"](#) (10-272).

• Search Occurrences

You can search for the n-th transition or the n-th match on value or expression; Search Occurrences

Finding a marker



Choose a specific marked line to view with **Markers > Goto** menu selection. The marker name (on the **Goto** list) corresponds to the simulation time of the selected line.

List window keyboard shortcuts

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

Key	Action
<arrow up>	scroll listing up
<arrow down>	scroll listing down
<arrow left>	scroll listing left
<arrow right>	scroll listing right
<page up>	scroll listing up by page
<page down>	scroll listing down by page

Key	Action
<tab>	searches forward (down) to the next transition on the selected signal
<shift-tab>	searches backward (up) to the previous transition on the selected signal (does not function on HP workstations)
<control-f>	opens the find dialog box; find the specified item label within the list display

Saving List window data to a file

From the List window select **Edit > Write List (format)** to save the List window data in one of these formats:

- **tabular**
writes a text file that looks like the window listing

```
ns      delta      /a      /b      /cin      /sum      /cout
0       +0         X       X       U       X       U
0       +1         0       1       0       X       U
2       +0         0       1       0       X       U
```

- **event**
writes a text file containing transitions during simulation

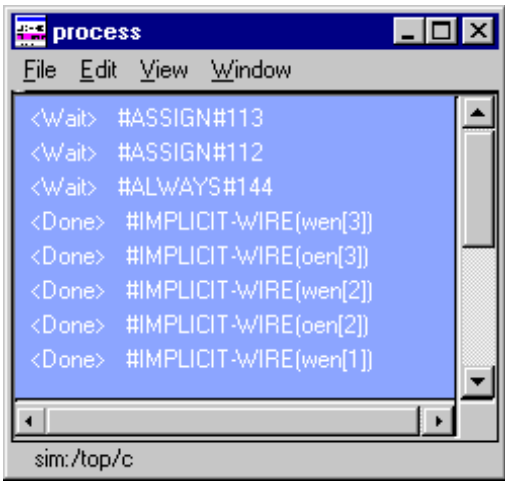
```
@0 +0
/a X
/b X
/cin U
/sum X
/cout U
@0 +1
/a 0
/b 1
/cin 0
```

- **TSSI**
writes a file in standard TSSI format; see also, the [write tssi](#) command (CR-236)

```
0 000000000000000010????????
2 000000000000000010???????1?
3 000000000000000010???????010
4 000000000000000010000000010
100 00000001000000010000000010
```

Process window

The Process window displays a list of processes (either active or in region) and indicates the pathname of the instance in which the process is located.



Each HDL item in the scrollbox is preceded by one of the following indicators:

- **<Ready>**
Indicates that the process is scheduled to be executed within the current delta time.
- **<Wait>**
Indicates that the process is waiting for a VHDL signal or Verilog net or variable to change or for a specified time-out period.
- **<Done>**
Indicates that the process has executed a VHDL wait statement without a time-out or a sensitivity list. The process will not restart during the current simulation run.

If you select a "Ready" process, it will be executed next by the simulator.

When you click on a process in the Process window the following windows are updated:

Window updated	Result
Structure window (10-206)	shows the region in which the process is located
Variables window (10-209)	shows the VHDL variables and Verilog register variables in the process
Source window (10-200)	shows the associated source code
Dataflow window (10-170)	shows the process, the signals and nets the process reads, and the signals and nets driven by the process.

The Process window menu bar

The following menu commands and button options are available from the Process window menu bar.

File menu

Save As	save the process tree to a text file viewable with the ModelSim notepad (CR-104)
Environment	Follow Process Selection: update the window based on the selection in the Structure window (10-206); Fix to Process: maintain the current view, do not update
Close	close this copy of the Process window; you can create a new window with View > New from the " The Main window menu bar " (10-159)

Edit menu

Copy	copy the selected process
Sort	sort the process list in either ascending, descending, or declaration order
Select All	select all signals in the Process window
Unselect All	deselect all signals in the Process window
Find...	find specified text string within the structure tree; choose the Status (ready, wait or done) or Process label to search and the search direction: forward or reverse

View menu

Active	Displays all the processes that are scheduled to run during the current simulation cycle.
In Region	Displays any processes that exist in the region that is selected in the Structure window.

Window menu

Initial Layout	restore all windows to the size and placement of the initial full-screen layout
Cascade	cascade all open windows
Tile Horizontally	tile all open windows horizontally

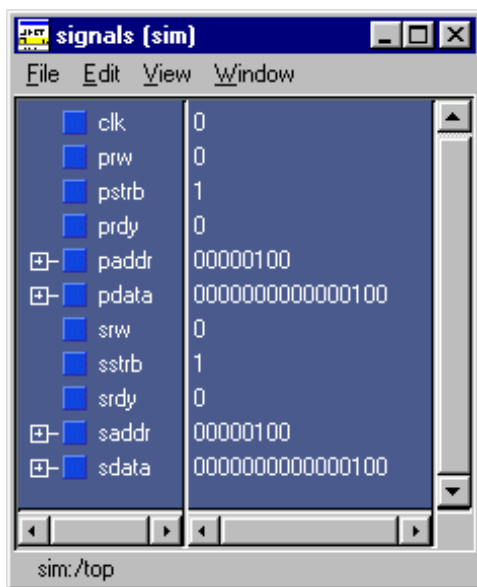
Tile Vertically	tile all open windows vertically
Icon Children	icon all but the Main window
Icon All	icon all windows
Deicon All	deicon all windows
Customize	use The Button Adder (10-265) to define and add a button to either the menu bar, tool bar, or status bar of the specified window
<window_name>	lists the currently open windows; select a window name to switch to, or show that window if it is hidden; when the source window is available, the source file name is also indicated; open additional windows from the "View menu" (10-161) in the Main window, or use the view command (CR-180)

Signals window

The Signals window is divided into two window panes. The left pane shows the names of HDL items in the current region (which is selected in the Structure window). The right pane shows the values of the associated HDL item pathname at the end of the current run. The data in this pane is similar to that shown in the [Wave window](#) (10-212), except that the values do not change dynamically with movement of the select Wave window cursor.

Horizontal scroll bars for each window pane allow scrolling to the right or left in each pane individually. The vertical scroll bar will scroll both panes together.

The HDL items may be sorted in ascending, descending, or declaration order.



HDL items you can view

One entry is created for each of the following VHDL and Verilog HDL items within the design:

VHDL items

signals

Verilog items

nets, register variables, and named events

Virtual items

(indicated by an orange diamond icon) virtual signals, see "[Virtual signals](#)" (9-145) for more information

The names of any VHDL composite types (arrays and record types) are shown in a hierarchical fashion.

Hierarchy also applies to Verilog nets and vector memories. (Verilog vector registers do not have hierarchy because they are not internally represented as arrays.)

Hierarchy is indicated in typical ModelSim fashion with plus (expandable), minus (expanded), and blank (single level) boxes.

See "[Tree window hierarchical view](#)" (10-155) for more information.

The Signals window menu bar

The following menu commands are available from the Signals window menu bar.

File menu

Save As	save the signals tree to a text file viewable with the ModelSim notepad (CR-104)
Environment	Follow Environment: update the window based on the selection in the Structure window (10-206); Fix to Context: maintain the current view, do not update
Close	close this copy of the Signals window; you can create a new window with View > New from the " The Main window menu bar " (10-159)

Edit menu

Copy	copy the current selection in the Signals window
Sort	sort the signals tree in either ascending, descending, or declaration order
Select All	select all items in the Signals window
Unselect All	unselect all items in the Signals window
Expand Selected	expands the hierarchy of the selected item
Collapse Selected	collapses the hierarchy of the selected item
Expand All	expands the hierarchy of all items that can be expanded
Collapse All	collapses the hierarchy of all expanded items
Force...	apply stimulus to the specified Signal Name; specify Value, Kind (Freeze/Drive/Deposit), Delay, and Repeat; see also the force command (CR-87)
Noforce	removes the effect of any active force command (CR-87) on the selected HDL item; see also the noforce command (CR-101)
Clock...	defines clock signals by Signal Name, Period, Duty Cycle, Offset, and whether the first edge is rising or falling, see " Defining clock signals " (10-198)

Justify Values	justify values to the left or right margins of the window pane
Find...	find specified text string within the Signals window; choose the Name or Value field to search and the search direction: forward or reverse; see also the search and next command (CR-141)

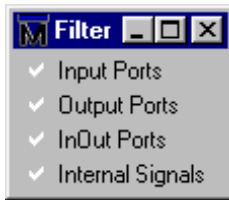
View menu

Wave/List/Log	place the Selected Signals, Signals in Region, or Signals in Design in the Wave window (10-212), List window (10-174), or logfile
Filter	choose the port and signal types to view (Input Ports, Output Ports, InOut Ports and Internal Signals) in the Signals window

Window menu

Initial Layout	restore all windows to the size and placement of the initial full-screen layout
Cascade	cascade all open windows
Tile Horizontally	tile all open windows horizontally
Tile Vertically	tile all open windows vertically
Icon Children	icon all but the Main window
Icon All	icon all windows
Deicon All	deicon all windows
Customize	use The Button Adder (10-265) to define and add a button to either the menu bar, tool bar, or status bar of the specified window
<window_name>	lists the currently open windows; select a window name to switch to, or show that window if it is hidden; when the source window is available, the source file name is also indicated; open additional windows from the " View menu " (10-161) in the Main window, or use the view command (CR-180)

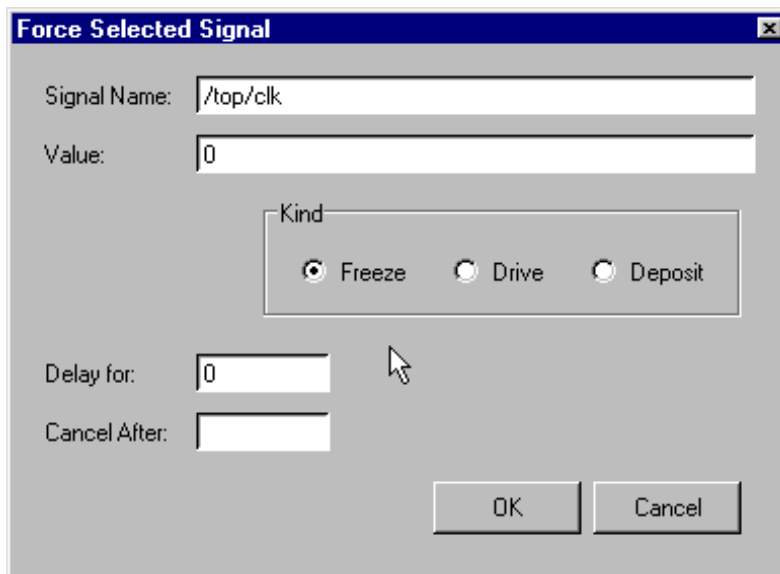
Selecting HDL item types to view



The **View > Filter...** menu selection allows you to specify which HDL items are shown in the Signals window. Multiple options may be selected.

Forcing signal and net values

The **Edit > Force** menu selection displays a dialog box that allows you to apply stimulus to the selected signal or net. You can specify that the stimulus is to repeat at a regular time interval, expressed in the time units set in the Startup window when you invoked the simulator. Multiple signals may be selected and forced; the force dialog box remains open until all of the signals are either forced, skipped, or you close the dialog box. See also the [force](#) command (CR-87).



The **Force** dialog box includes these options:

- **Signal Name**

Specify the signal or net for the applied stimulus.

- **Value**

Initially displays the current value, which can be changed by entering a new value into the field. A value can be specified in radices other than decimal by using the form (for VHDL and Verilog, respectively):

```
base#value -or-
b|o|d|h'value
```

16#EE or h'EE, for example, specifies the hexadecimal value EE.

- **Kind: Freeze**

Freezes the signal or net at the specified value until it is forced again or until it is unforced with a **noforce** command (CR-101).

Freeze is the default for Verilog nets and unresolved VHDL signals and **Drive** is the default for resolved signals.

If you prefer **Freeze** as the default for resolved and unresolved signals, you can change the default force kind in the *modelsim.ini* file; see "[Projects and system initialization](#)" (3-43).

- **Kind: Drive**

Attaches a driver to the signal and drives the specified value until the signal or net is forced again or until it is unforced with a **noforce** command (CR-101). This value is illegal for unresolved VHDL signals.

- **Kind: Deposit**

Sets the signal or net to the specified value. The value remains until there is a subsequent driver transaction, or until the signal or net is forced again, or until it is unforced with a **noforce** command (CR-101).

- **Delay for**

Allows you to specify how many time units from the current time the stimulus is to be applied.

- **Cancel After**

Cancels the **force** command (CR-87) after the specified period of simulation time.

- **OK**

When you click the OK button, a **force** command (CR-87) is issued with the parameters you have set, and is echoed in the Main window. If more than one signal is selected to force, the next signal down appears in the dialog box each time the OK button is selected. Unique force parameters may be set for each signal.

Adding HDL items to the Wave and List windows or a logfile

Before adding items to the List or Wave window you may want to set the window display properties (see "[Setting List window display properties](#)" (10-177)). Once display properties have been set, you can add items to the windows or logfile in several ways.

Adding items from the Main window command line



Use the **View** menu with either the **Wave**, **List**, or **Log** selection to add HDL items to the [Wave window](#) (10-212), [Saving the Dataflow window as a Postscript file](#) (10-173) or a logfile, respectively.

The logfile is written as an archive file in binary format and is used to drive the List and Wave window at a later time. Once signals are added to the logfile they cannot be removed. If you begin a simulation by invoking [vsim](#) (CR-208) with the -view <logfile_name> option, VSIM reads the logfile to drive the Wave and List windows.

Choose one of the following options (ModelSim opens the target window for you):



- **Selected signal**
Lists only the item(s) selected in the Signals window.
- **Signals in region**
Lists all items in the region that is selected in the Structure window.
- **Signals in design**
Lists all items in the design.

Adding items from the Main window command line

Another way to add items to the Wave or List window or the logfile is to enter the one of the following commands at the VSIM prompt (choose either the [add list](#) (CR-22), or [log](#) (CR-93) command):

```
add list | add wave | log <item_name> <item_name>
```

You can add all the items in the current region with this command:

```
add list | add wave | log *
```

Or add all the items in the design with:

```
add list | add wave | log -r /*
```

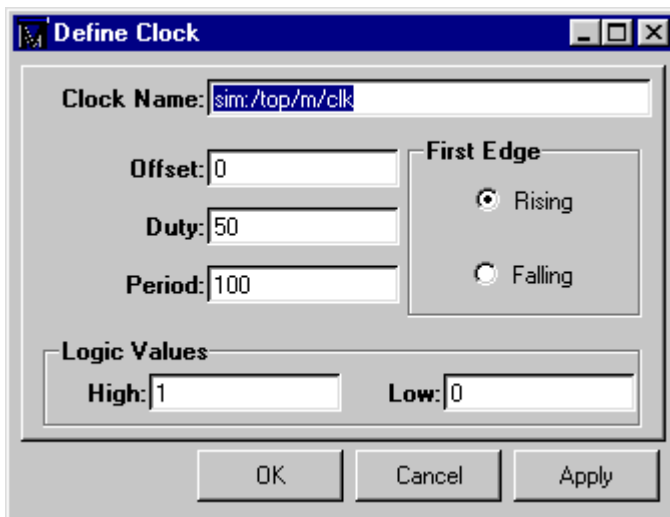
If the target window (Wave or List) is closed, ModelSim opens it when you when you invoke the command.

Finding HDL items in the Signals window



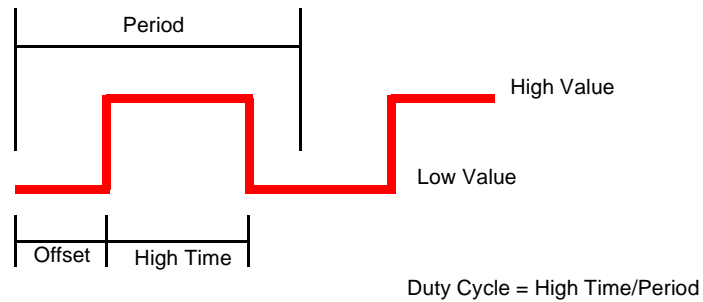
Find the specified text string within the Signals window; choose the **Name** or **Value** field to search and the search direction: **Forward** or **Reverse**.

Defining clock signals



Selecting Clock from the Edit menu allows you to define clock signals by Name, Period, Duty Cycle, Offset, and whether the first rising edge is rising or falling.

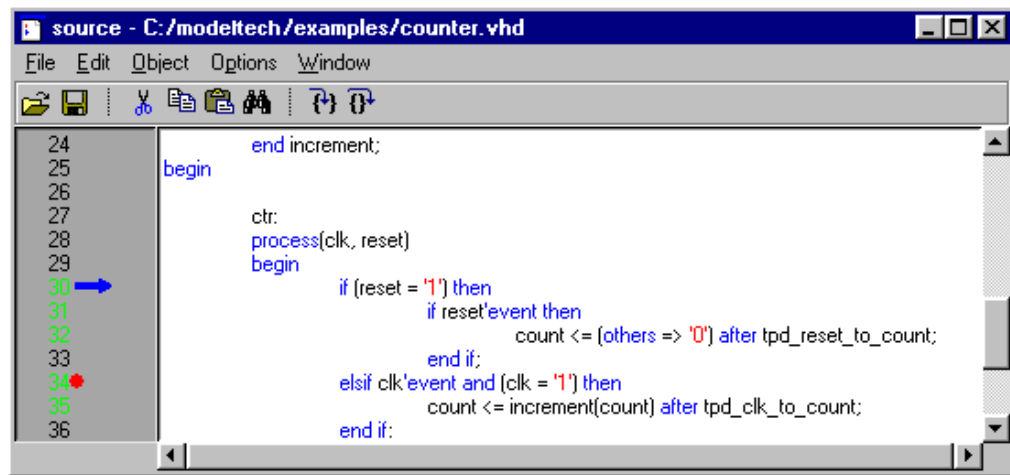
For clock signals starting on the rising edge, the definition for Period, Offset, and Duty Cycle is as follows:



If the signal type is `std_logic`, `std_ulogic`, `bit`, `verilog wire`, `verilog net`, or any other logic type where 1 and 0 are valid, then 1 is the default High Value and 0 is the default Low Value. For other signal types, you will need to specify a High Value and a Low Value for the clock.

Source window

The Source window allows you to view and edit your HDL source code. Select an item in the [Structure window](#) (10-206) or use the **File** menu to add a source file to the window, then select a process in the [Process window](#) (10-189) to view that process; an arrow next to the line numbers indicates the selected process. (Your source code can remain hidden if you wish, see "[Source code security and -nodebug](#)" (E-443). A dot next to a line number indicates a breakpoint. Breakpoints can only occur at executable lines, which are indicated by green line numbers.



If any breakpoints have been set, each is signified by a colored dot next to a line number at the left side of the window pane. To set a breakpoint, click at or near the line number in the numbered area at the left side of the window. The breakpoints are toggles, so you can click again to delete an existing breakpoint. There is no limit to the number of breakpoints you can set. See also the **bp** command (CR-40) (breakpoint) command.

To look at a file that is not currently being displayed, use the [Structure window](#) (10-206) to select a different design unit or use the Source menu selection: **File > Open**. The pathname of the source file is indicated in the header of the Source window.

You can copy and paste text between the Source window and the [Main window](#) (10-158); select the text you want to copy, then paste it into the Main window with the middle button (3-button mouse), right button (2-button mouse).

The Source window menu bar

The following menu commands are available from the Source window menu bar.

File menu

New	edit a new (VHDL, Verilog or Other) source file
Open	select a source file to open
Use Source	specifies an alternative file to use for the current source file; this alternative source mapping exists for the current simulation only
Source Directory	add to a list of directories (the SourceDir variable in modelsim.tcl) to search for source files
Save	save the current source file
Save_As	save the current source file with a different name
Compile	compiles HDL source files
Close	close this copy of the Source window; you can create a new window with View > New from the "The Main window menu bar" (10-159)

Edit menu

To edit a source file, make sure the **Read Only** option in the Source Options dialog box is *not* selected (use the Source menu **Edit > read only** selection).

<editing option>	basic editing options include: Cut, Copy, Paste, Select All, and Unselect All; see: "Editing the command line, the current source file, and notepads" (10-167)
Find...	find the specified text string within the source file; match case option
read only	toggles the read-only status of the current source file

Object menu

Describe	displays information about the selected HDL item; same as the describe command (CR-63); the item name is shown in the title bar
Examine	displays the current value of the selected HDL item; same as the examine (CR-81) command; the item name is shown in the title bar

Options menu

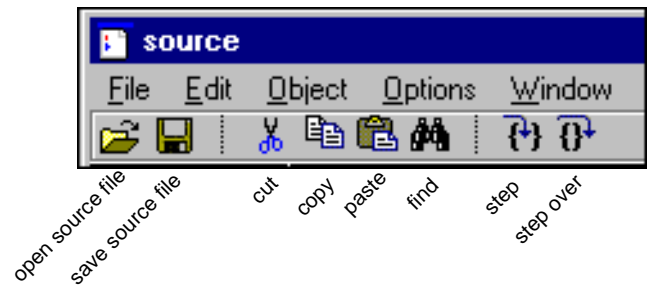
Options	open the Source Options dialog box, see "Setting Source window options" (10-205)
---------	--





Window menu





Initial Layout	restore all windows to the size and placement of the initial full-screen layout
Cascade	cascade all open windows
Tile Horizontally	tile all open windows horizontally
Tile Vertically	tile all open windows vertically
Icon Children	icon all but the Main window
Icon All	icon all windows
Deicon All	deicon all windows
Customize	use The Button Adder (10-265) to define and add a button to either the menu bar, tool bar, or status bar of the specified window
<window_name>	lists the currently open windows; select a window name to switch to, or show that window if it is hidden; when the source window is available, the source file name is also indicated; open additional windows from the "View menu" (10-161) in the Main window, or use the view command (CR-180)

The Source window tool bar

Buttons on the Source window tool bar gives you quick access to these ModelSim commands and functions.



Source window tool bar buttons		
Button	Menu equivalent	Other equivalents
 Open Source File open the Open dialog box (you can open any text file for editing in the Source window)	File > Open	select an HDL item in the Structure window, the associated source file is loaded into the Source window
 Save Source File save the file in the Source window	File > Save	none
 Cut cut the selected text within the Source window	Edit > Cut	see: "Editing the command line, the current source file, and notepads" (10-167)
 Copy copy the selected text within the Source window	Edit > Copy	see: "Editing the command line, the current source file, and notepads" (10-167)

Source window tool bar buttons		
Button	Menu equivalent	Other equivalents
 Paste paste the copied text to the cursor location	Edit > Paste	see: "Editing the command line, the current source file, and notepads" (10-167)
 Find find the specified text string within the source file; match case option	Edit > Find	none
 Step steps the current simulation to the next HDL statement	none	use step command at the VSIM prompt see: step (CR-151) command
 Step Over HDL statements are executed but treated as simple statements instead of entered and traced line by line	none	use the step -over command at the VSIM prompt see: step (CR-151) command

Editing the source file in the Source window

Several tool bar buttons (shown above), mouse actions, and special keystrokes can be used to edit the source file in the Source window. See ["Editing the command line, the current source file, and notepads"](#) (10-167) for a list of mouse and keyboard editing options.

Checking HDL item values and descriptions

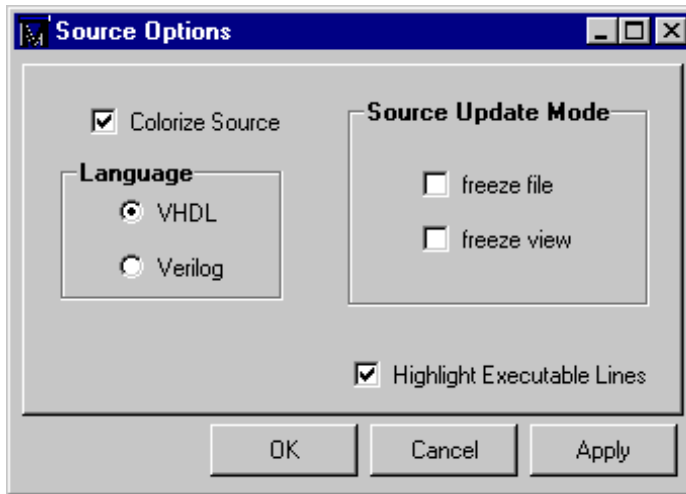
There are two quick methods to determine the value and description of an HDL item displayed in the Source window:

- select an item, then chose **Object > Examine** or **Object > Description** from the Source window menu
- select an item with the right mouse button to view examine pop-up (select "now" to examine the current simulation time in VHDL code)

You can also invoke the **examine** (CR-81) and/or **describe** (CR-63) command on the command line or in a macro.

Setting Source window options

Access the Source window options with this Source menu selection: **Options > Options.**

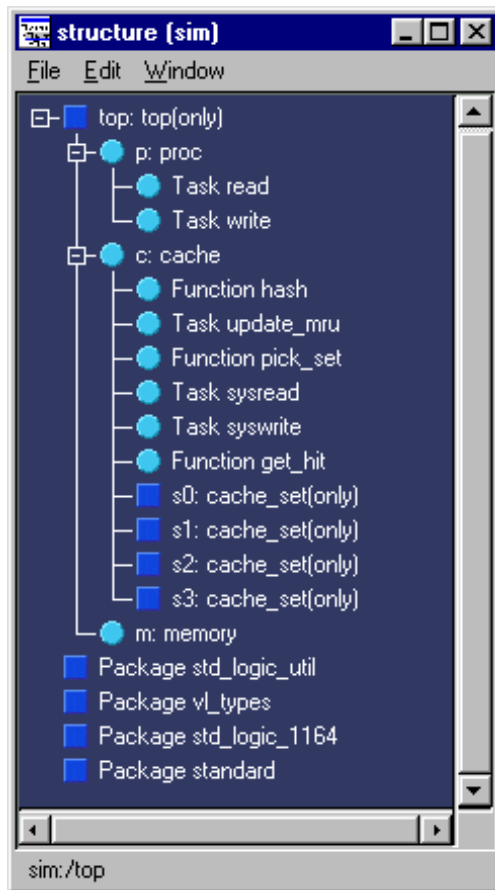


The **Source Options** dialog box includes these options:

- **Language**
select either **VHDL** or **Verilog**;
sets language for key word colorizing
- **Source Update Mode**
select **freeze file** to maintain the same source file in the Source window (useful when you have two Source windows open; one can be updated from the [Structure window](#) (10-206), the other frozen) or **freeze view** to disable updating the source view from the [Process window](#) (10-189)
- **Colorize Source**
colorize key words, variables and comments
- **Highlight Executable Lines**
highlights the line number of executable lines

Structure window

The Structure window provides a hierarchical view of the structure of your design. An entry is created by each HDL item within the design. (Your design structure can remain hidden if you wish, see ["Source code security and -nodebug"](#) (E-443).)



HDL items you can view

The following HDL items for VHDL and Verilog are represented by hierarchy within Structure window.

VHDL items

(indicated by a dark blue square icon)
signals, variables, component instantiation, generate statement, block statement, and package

Verilog items

(indicated by a lighter blue circle icon)
parameters, registers, nets, module instantiation, named fork, named begin, task, and function

Virtual items

(indicated by an orange diamond icon)
virtual signals, buses, and functions, see ["Virtual Objects \(User-defined buses, and more\)"](#) (9-144) for more information.

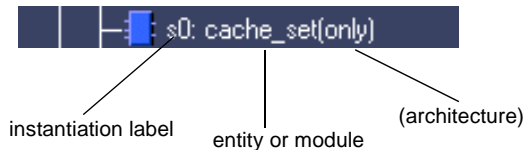
You can expand and contract the display to view the hierarchical structure by clicking on the boxes that contain "+" or "-". Clicking "+" expands the hierarchy so the subelements of

that item can be seen. Clicking "-" contracts the hierarchy.

The first line of the Structure window indicates the top-level design unit being simulated. By default, this is the only level of the hierarchy that is expanded upon opening the Structure window.

Instance name components in the Structure window

An instance name displayed in the Structure window consists of the following parts:



where:

- **instantiation label**

Indicates the label assigned to the component or module instance in the instantiation statement.

- **entity or module**

Indicates the name of the entity or module that has been instantiated.

- **architecture**

Indicates the name of the architecture associated with the entity (not present for Verilog).

When you select a region in the Structure window, it becomes the *current region* and is highlighted; the [Source window](#) (10-200) and [Signals window](#) (10-192) change dynamically to reflect the information for that region. This feature provides a useful method for finding the source code for a selected region because the system keeps track of the pathname where the source is located and displays it automatically, without the need for you to provide the pathname.

Also, when you select a region in the Structure window, the [Process window](#) (10-189) is updated if **In Region** is selected in that window; the Process window will in turn update the [Variables window](#) (10-209).

The Structure window menu bar

The following menu commands are available from the Structure window menu bar.

File menu

Save_As	save the structure tree to a text file viewable with the ModelSim notepad (CR-104)
Environment	changing of current environment between open datasets; or, establish a New Context by opening a new dataset
Close	close this copy of the Structure window; you can create a new window with View > New from the "The Main window menu bar" (10-159)

Edit menu

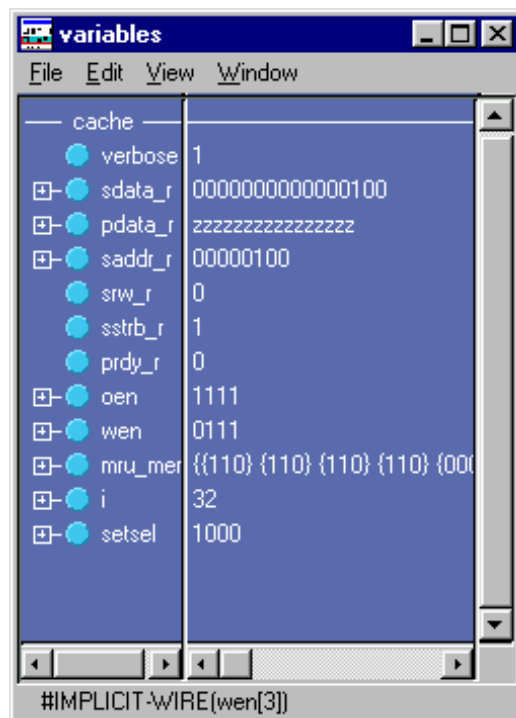
Copy	copy the current selection in the Structure window
Sort	sort the structure tree in either ascending, descending, or declaration order
Expand Selected	expands the hierarchy of the selected item
Collapse Selected	collapses the hierarchy of the selected item
Expand All	expands the hierarchy of all items that can be expanded
Collapse All	collapses the hierarchy of all expanded items
Find...	find specified text string within the structure tree; choose the label for instance, entity/module or architecture to search for and the search direction: forward or reverse

Window menu

Initial Layout	restore all windows to the size and placement of the initial full-screen layout
Cascade	cascade all open windows
Tile Horizontally	tile all open windows horizontally
Tile Vertically	tile all open windows vertically
Icon Children	icon all but the Main window
Icon All	icon all windows
Deicon All	deicon all windows
Customize	use The Button Adder (10-265) to define and add a button to either the menu bar, tool bar, or status bar of the specified window
<window_name>	lists the currently open windows; select a window name to switch to, or show that window if it is hidden; when the source window is available, the source file name is also indicated; open additional windows from the " View menu " (10-161) in the Main window, or use the view command (CR-180)

Variables window

The Variables window is divided into two window panes. The left pane lists the names of HDL items within the current process. The right pane lists the current value(s) associated with each name. The pathname of the current process is displayed at the bottom of the window. (The internal variables of your design can remain hidden if you wish, see ["Source code security and -nodebug"](#) (E-443).)



HDL items you can view

The following HDL items for VHDL and Verilog are viewable within the Variables window.

VHDL items

constants, generics, and variables

Verilog items

register variables

The names of any VHDL composite types (arrays and record types) are shown in a hierarchical fashion. Hierarchy also applies to Verilog vector memories. (Verilog vector registers do not have hierarchy because they are not internally represented as arrays.) Hierarchy is indicated in typical ModelSim fashion with plus (expandable) and minus (expanded). See ["Tree window hierarchical view"](#) (10-155) for more information.

To change the value of a VHDL variable, constant, generic or Verilog register variable, move the pointer to the desired name and click to highlight the selection. Then select **Edit > Change** from the Variables window menu. This brings up a dialog box that lets you specify a new value. Note that "Variable Name" is a term that is used loosely in this case to signify VHDL constants and generics as well as VHDL and Verilog register variables. You can enter any value that is valid for the variable. An array value must be specified as a string (without surrounding quotation marks). To modify the values in a record, you need to change each field separately.

Click on a process in the Process window to change the Variables window.

The Variables window menu bar

The following menu commands are available from the Variables window menu bar.

File menu

Save As	save the variables tree to a text file viewable with the ModelSim notepad (CR-104)
Environment	Follow Process Selection: update the window based on the selection in the Structure window (10-206); Fix to Process: maintain the current view, do not update
Close	close this copy of the Variables window; you can create a new window with View > New from the " The Main window menu bar " (10-159)

Edit menu

Copy	copy the selected items in the Variables window
Sort	sort the variables tree in either ascending, descending, or declaration order
Select All	select all items in the Variables window
Unselect All	deselect all items in the Variables window
Expand Selected	expands the hierarchy of the selected item
Collapse Selected	collapses the hierarchy of the selected item
Expand All	expands the hierarchy of all items that can be expanded
Collapse All	collapses the hierarchy of all expanded items
Change	change the value of the selected HDL item
Justify Values	justify values to the left or right margins of the window pane
Find...	find specified text string within the variables tree; choose the Name or Value field to search and the search direction: forward or reverse

View menu

Wave/List/Log	place the Selected Variables or Variables in Region in the Wave window (10-212), List window (10-174), or logfile
---------------	---

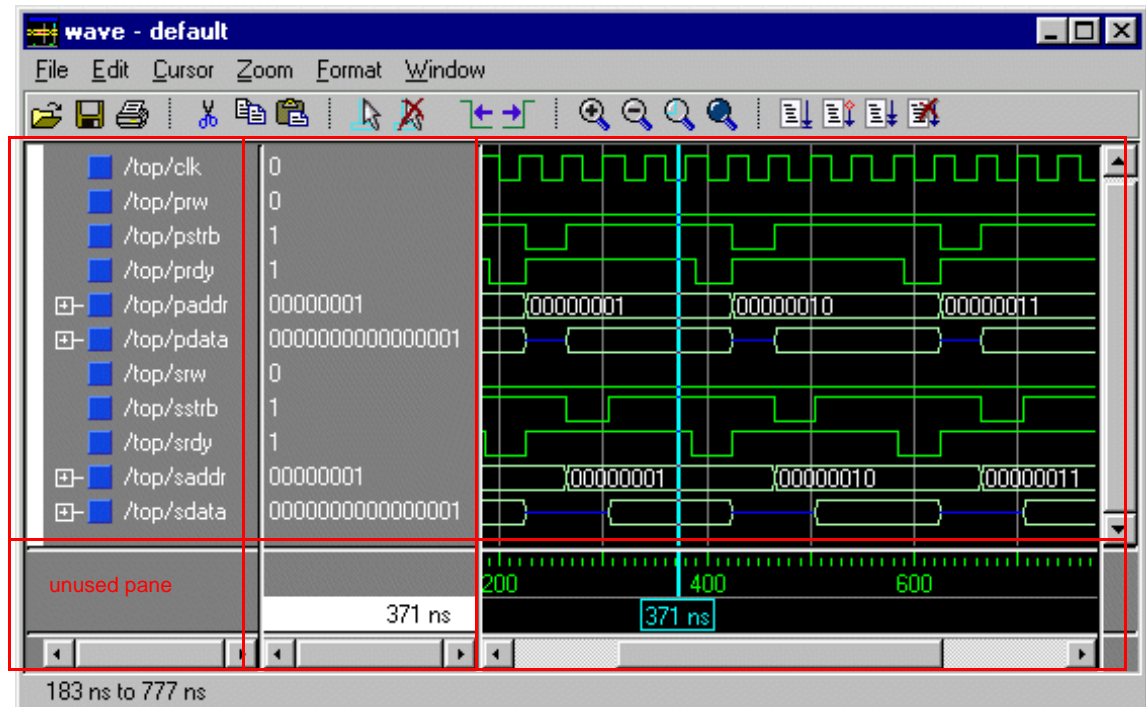
Window menu

Initial Layout	restore all windows to the size and placement of the initial full-screen layout
Cascade	cascade all open windows
Tile Horizontally	tile all open windows horizontally
Tile Vertically	tile all open windows vertically
Icon Children	icon all but the Main window
Icon All	icon all windows
Deicon All	deicon all windows
Customize	use The Button Adder (10-265) to define and add a button to either the menu bar, tool bar, or status bar of the specified window
<window_name>	lists the currently open windows; select a window name to switch to, or show that window if it is hidden; when the source window is available, the source file name is also indicated; open additional windows from the "View menu" (10-161) in the Main window, or use the view command (CR-180)

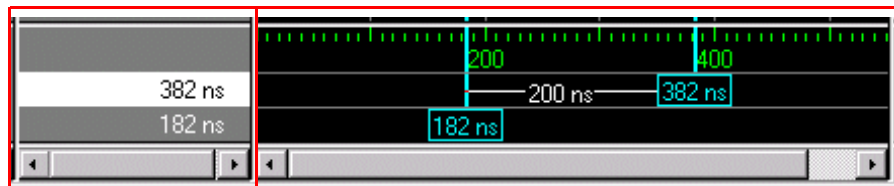
Wave window

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

The Wave window is divided into a number of window panes.



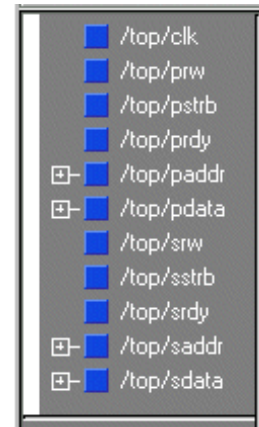
There are two cursor panes, as shown below. The left pane shows the time value for each cursor. The selected cursor's value is highlighted. The right pane shows the absolute time value for each cursor and relative time between cursors. Up to 20 cursors may be displayed.



two cursor panes

The pathname pane displays signal pathnames. Signals may be displayed with full pathnames, as shown here, or with only the leaf element displayed. The selected signal is highlighted.

The white bar along the left margin indicates the selected Waveset (see [Wave window panes](#) (10-215)).



pathnames pane

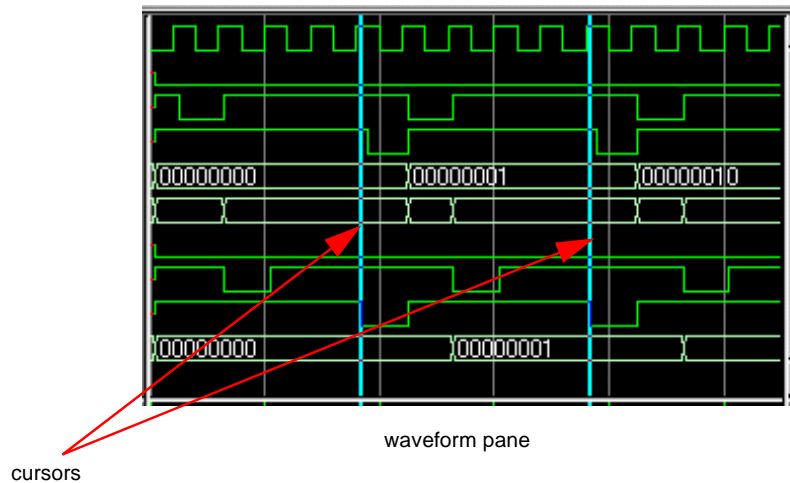


values pane

A values pane displays the values of the displayed signals. Signal values may be displayed in analog step, analog interpolated, analog backstep, literal, logic, and event formats. Each signal may be formatted individually. The default format is logic.

The radix for each signal may be symbolic, binary, octal, decimal, unsigned, hexadecimal, ASCII or default. The default radix may be set by selecting **File > Options > Simulation** in the Main window (see ["Setting default simulation options"](#) (10-260)).

The data in this pane is similar to that shown in the [Signals window](#) (10-192), except that the values change dynamically whenever a cursor in the waveform pane (below) is moved.



The waveform pane displays the waveforms that correspond to the displayed signal pathnames. It also displays up to 20 cursors.

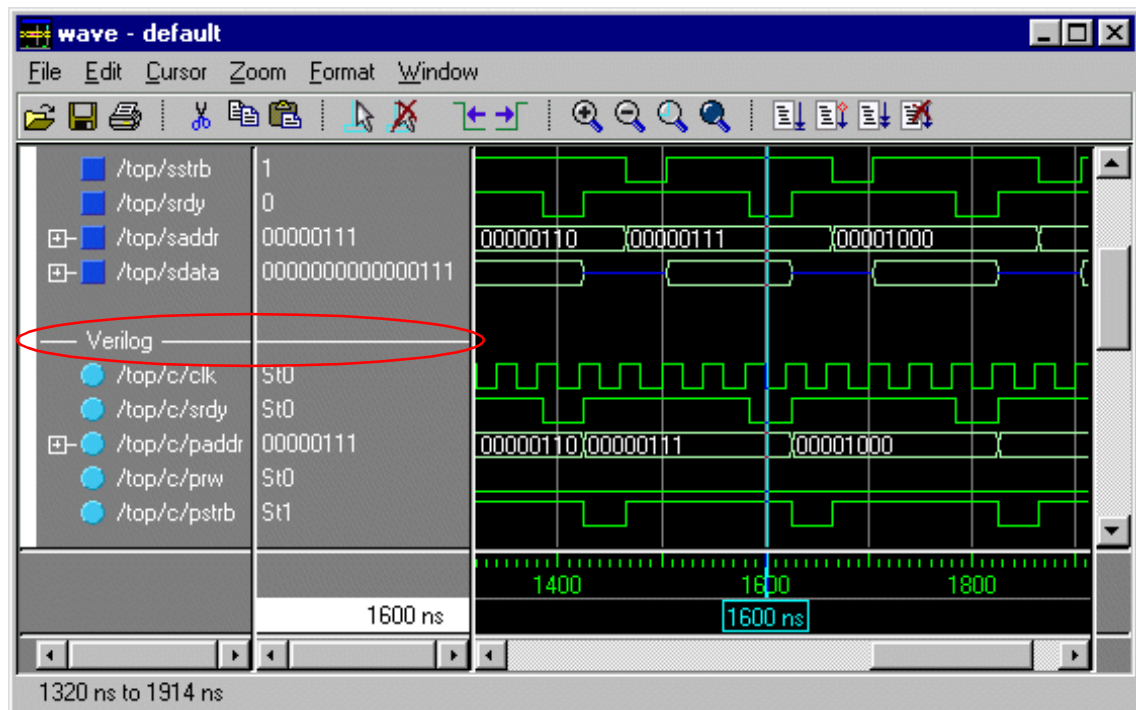
The window pane below the pathnames window pane and to the left of the cursor panes is unused at this time.

All window panes in the Wave window may be resized by clicking and dragging the bar between any two panes.

Using Dividers

Dividing lines may be placed in the pathname and values window panes by selecting **File > New Divider**. Dividers serve as a visual aid to signal debugging, allowing you to separate signals and waveforms for easier viewing.

Dividing lines can be assigned any name, or no name at all. The default name is "New Divider." In the illustration below, VHDL signals have been separated from Verilog signals with a Divider called "Verilog." Notice that the waveforms in the waveform window pane have been separated by the divider as well.



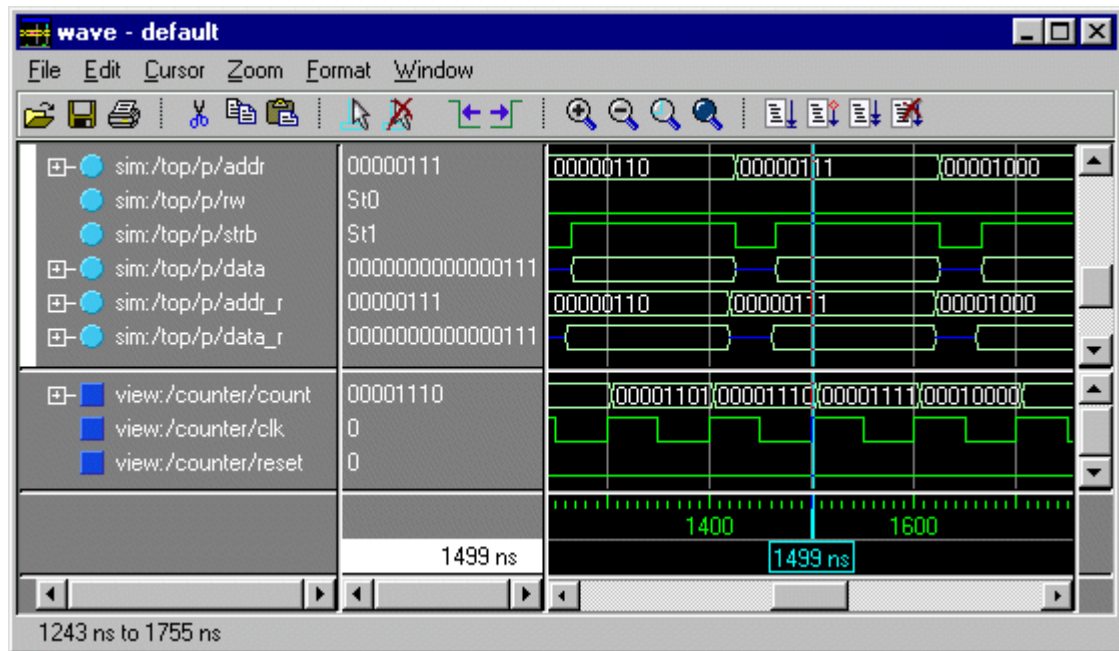
Using dividers

Wave window panes

The pathnames, values and waveforms window panes of the Wave window display may be split to accommodate signals from one or more datasets. Selecting **File > New Window Pane** creates a space below the selected waveset and makes the new window pane the selected pane. (The selected wave window pane is indicated by a white bar along the left margin of the pane.)

In the illustration below, the Wave window is split, showing the current active simulation with the prefix "sim," and a view mode simulation, with the prefix "view."

For more information on datasets see *Chapter 9 - Multiple logfiles, datasets and virtuals*.



HDL items you can view

VHDL items

(indicated by a dark blue square)
signals and process variables

Verilog items

(indicated by a lighter blue circle)
nets, register variables, and named events

Virtual items

(indicated by an orange diamond)
virtual signals, buses, and functions, see
["Virtual Objects \(User-defined buses, and more\)"](#) (9-144)
for more information

Note: Constants, generics, parameters, and memories are not viewable in the List or Wave windows.

The data in the item values windowpane is very similar to the Signals window, except that the values change dynamically whenever a cursor in the waveform windowpane is moved.

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

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

Waveform and signal-name formatting are easily changed via the [Format menu](#) (10-219). You can reuse any formatting changes you make by saving a Wave window format file, see ["Adding items with a Wave window format file"](#) (10-224).

The Wave window menu bar



The following menu commands and button options are available from the Wave window menu bar. If you see a dotted line at the top of a drop-down menu, click and drag the dotted line to create a separate menu window.

File menu

Open Dataset	opens a new .wlf dataset file
New Divider	inserts divider at current location
New Group	allows setup of new group element – a container for other items that can be moved, cut and pasted like other objects

Save Format	saves the current Wave window display and signal preferences to a .do (macro) file; running the .do file will reformat the Wave window to match the display as it appeared when the .do file was created
Load Format	run a Wave window format (.do) file previously saved with Save Format
Page Setup	allow setup of page for printing; options include: paper size, margins, label width, cursors, color, scaling and orientation
Print	sends contents of Wave window to a selected printer; options include: All – prints all signals Current View – prints signals in current view for the time displayed Custom – prints all or current view signals for user-designated time
Print Postscript	save or print the waveform display as a Postscript file; options include: All – prints all signals Current View – prints signals in current view for the time displayed Custom – prints all or current view signals for user-designated time
New Window Pane	splits the pathname, values and waveform window panes to provide room for a new waveset; gives the option of inserting the new waveset above or below the current, selected, waveset
Remove Window Pane	unplits window and removes active waveset
Refresh Display	clears the Wave window, empties the file cache, and the rebuilds the window from scratch
Close	close this copy of the Wave window; you can create a new window with View > New from the "The Main window menu bar" (10-159)

Edit menu

Cut	cut the selected item from the wave name pane; see "Editing and formatting HDL items in the Wave window" (10-226)
Copy	copy the selected item and waveform
Paste	paste the previously cut or copied item above the currently selected item
Delete	delete the selected item and its waveform

Select All Unselect All	select, or unselect, all item names in name pane
Combine	combine the selected fields into a user defined bus
Sort	sort the top-level items in the name pane; sort with full path name or viewed name; use ascending, descending or declaration order
Find...	find specified item label within the Wave name window
Search...	search the waveform display for a specified value, or the next transition for the selected signal; see: "Searching for item values in the Wave window" (10-231)
Display Properties	set display properties for signal name width, cursor snap distance (in pixels); set dataset prefix to: show all dataset prefixes, show all dataset prefixes if 2 or more, or, show no dataset prefixes; set value justification to left or right margin
Signal Properties	set label, height, color, radix, and format for the selected item (use the Format menu selections below to quickly change individual properties)

Cursors menu

Add Cursor	add a cursor to the center of the waveform window
Delete Cursor	delete the selected cursor from the window
Goto	choose a cursor to go to from a list of current cursors

Zoom menu

Zoom <selection>	selection: Full, In, Out, Last, Area with mouse button 1, or Range to change the waveform display range
------------------	---

Format menu

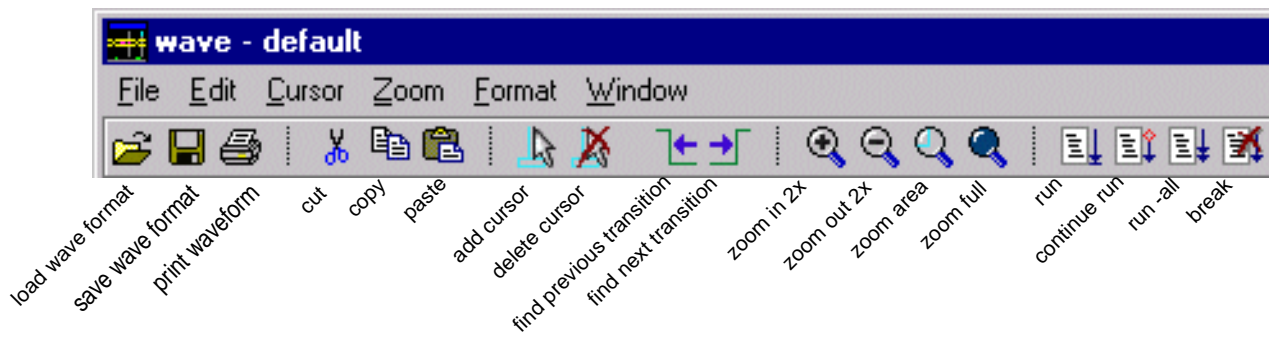
Radix	set the selected item's radix
Format	set the waveform format for the selected item – Literal, Logic, Analog
Color	set the color for the selected item from a color palette
Height	set the waveform height in pixels for the selected item





Window menu








Initial Layout	restore all windows to the size and placement of the initial full-screen layout
Cascade	cascade all open windows
Tile Horizontally	tile all open windows horizontally
Tile Vertically	tile all open windows vertically
Icon Children	icon all but the Main window
Icon All	icon all windows
Deicon All	deicon all windows
Customize	use The Button Adder (10-265) to define and add a button to either the menu bar, tool bar, or status bar of the specified window
<window_name>	lists the currently open windows; select a window name to switch to, or show that window if it is hidden; when the source window is available, the source file name is also indicated; open additional windows from the "View menu" (10-161) in the Main window, or use the view command (CR-180)





Wave window tool bar

The Wave window tool bar gives you quick access to these ModelSim commands and functions.



Wave window tool bar buttons		
Button	Menu equivalent	Other options
 Load Wave Format run a Wave window format (DO) file previously saved with Save Format	File > Load Format	none
 Save Wave Format saves the current Wave window display and signal preferences to a do (macro) file	File > Save Format	none
 Print Waveform prints a user-selected range of the current Wave window display to a printer or a file	File > Print	none
 Cut cut the selected signal within the Wave window	Edit > Cut	none
 Copy copy the selected signal in the signal-name pane	Edit > Copy	none
 Paste paste the copied signal above another selected signal	Edit > Paste	none
 Add Cursor add a cursor to the center of the waveform pane	Cursor > Add Cursor	none

Wave window tool bar buttons			
Button		Menu equivalent	Other options
	Delete Cursor delete the selected cursor from the window	Cursor > Delete Cursor	none
	Find Previous Transition locate the previous signal value change for the selected signal	Edit > Find > Reverse	Keyboard: Shift + Tab
	Find Next Transition locate the next signal value change for the selected signal	Edit > Find > Forward	Keyboard: Tab
	Zoom in 2x zoom in by a factor of two from the current view	Zoom > Zoom In	Keyboard: i I or +
	Zoom out 2x zoom out by a factor of two from current view	Zoom > Zoom Out	Keyboard: o O or -
	Zoom area with mouse button 1 use the cursor to outline a zoom area	Zoom > Zoom Range	Keyboard: r or R
	Zoom Full zoom out to view the full range of the simulation from time 0 to the current time	Zoom > Zoom Full	Keyboard: f or F

Wave window tool bar buttons			
Button		Menu equivalent	Other options
	Run run the current simulation for the default time length	Main menu: Run > Run <default_length>	use the run command at the VSIM prompt see: run (CR-139)
	Continue Run continue the current simulation run	Main menu: Run > Continue	use the run -continue command at the VSIM prompt see: run (CR-139)
	Run -All run to current simulation forever, or until it hits a breakpoint or specified break event*	Main menu: Run > Run -All	use run -all command at the VSIM prompt see: run (CR-139), also see "Assertion settings page" (10-262)
	Break stop the current simulation run	none	none

Adding HDL items in the Wave window

Before adding items to the Wave window you may want to set the window display properties (see ["Setting Wave window display properties"](#) (10-230)). You can add items to the Wave window in several ways.

Adding items from the Signals window with drag and drop

You can drag and drop items into the Wave window from the Process, Signals, or Structure window. Select the items in the first window, then drop them into the Wave window. Depending on what you select, all items or any portion of the design may be added.

Adding items from the Main window command line

To add specific HDL items to the window, enter (separate the item names with a space):

```
add wave <item_name> <item_name>
```

You can add all the items in the current region with this command:

```
add wave *
```

Or add all the items in the design with:

```
add wave -r /*
```

Adding items with a Wave window format file

To use a Wave window format file you must first save a format file for the design you are simulating.

- add the items you want in the Wave window with any other method shown above
- edit and format the items, see "[Editing and formatting HDL items in the Wave window](#)" (10-226) to create the view you want
- save the format to a file with the Wave window menu selection: **File > Save Format**

To use the format file, start with a blank Wave window and run the DO file in one of two ways:

- use the do command on the command line:

```
do <my_wave_format>
```
- select **File > Load Format** from the Wave window menu bar

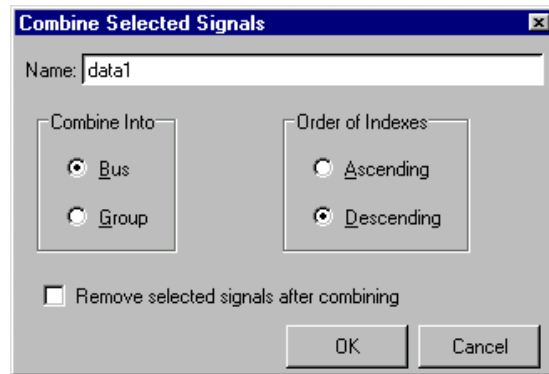
Use **Edit > Select All** and **Edit > Delete** to remove the items from the current Wave window, use the [delete](#) command (CR-62) with the **wave** option, or create a new, blank Wave window with the **View > New > Wave** selection from the [Main window](#) (10-158).

Note: Wave window format files are design-specific; use them only with the design you were simulating when they were created.

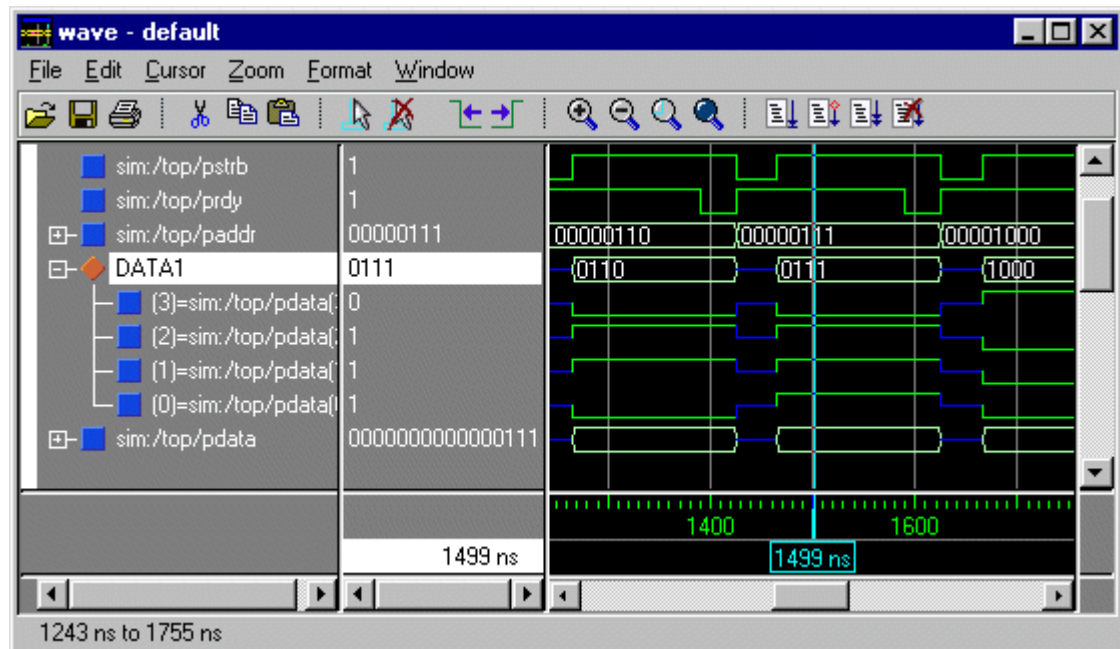
Combining and grouping items in the Wave window

The Wave window allows you to combine signals into busses or groups. Use the **Edit > Combine** menu selections to call up the Combine Selected Signals Dialog box.

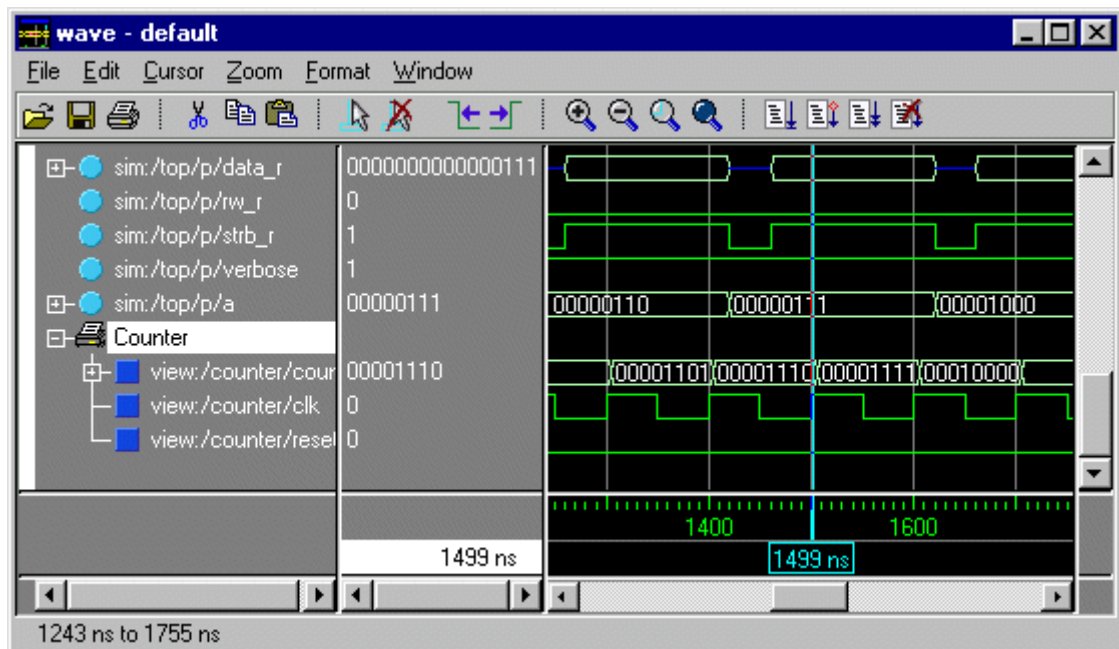
A bus is a collection of signals concatenated in a specific order to create a new virtual signal with a specific value.



In the illustration below, four data signals have been combined to form a new bus called DATA1. Notice, the new bus has a value that is made up of the values of its component signals arranged in a specific order. Virtual objects are indicated by an orange diamond.



A group is simply a container for any number of signals. It has no value, and the signals contained within it may be arranged in any order. In the illustration below, the signals counter/count, counter/clk, and counter/reset have been combined in a group called Counter. Notice that the Counter group has no value associated with it. The counter, clk and reset signals may be arranged in any order.



Other virtual items in the Wave window

See ["Virtual Objects \(User-defined buses, and more\)"](#) (9-144) for information about other virtual item viewable in the Wave window.

Editing and formatting HDL items in the Wave window

Once you have the HDL items you want in the Wave window, you can edit and format the list in the name/value pane to create the view you find most useful. (See also, ["Setting Wave window display properties"](#) (10-230).)

To edit an item:

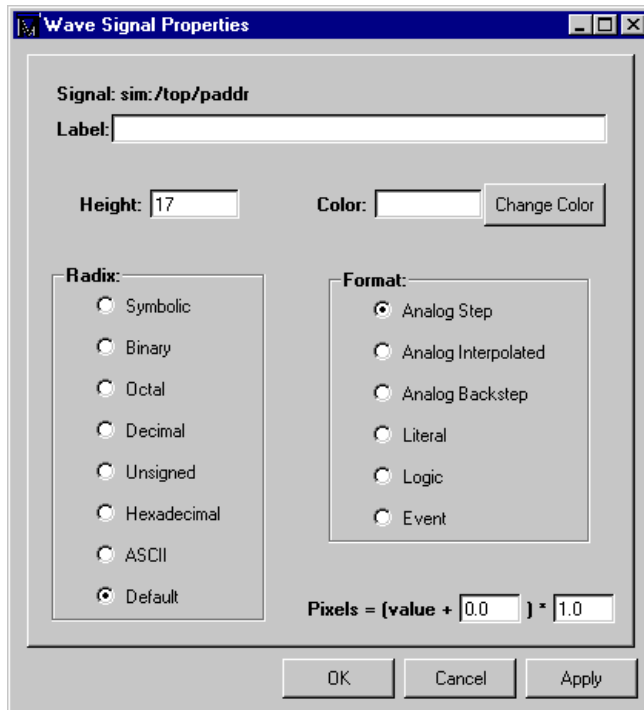
Select the item's label in the left name/value windowpane or its waveform in the right windowpane. Move, copy or remove the item by selecting commands from the Wave window [Edit menu](#) (10-218) menu.

You can also **click+drag** to move items within the name/value windowpane:

- to select several contiguous items:
click+drag to select additional items above or below the original selection
- to select several items randomly:
control+click to add or subtract from the selected group
- to move the selected items:
re-click and hold on one of the selected items, then drag to the new location

To format an item:

Select the item's label in the left pathname pane or its waveform in the waveform pane, then use the **Edit > Signal Properties** menu selection. The resulting Wave Signal Properties dialog box allows you to set the item's height, color, format, range, and radix.



The **Wave Signal Properties** dialog box includes these options:

- **Signal**

Indicates the name of the currently selected signal.

- **Label**

Allows you to specify a new label (in the pathname pane) for the selected item.

- **Height**

Allows you to specify the height (in pixels) of the waveform.

- **Color**

Lets you override the default color of a waveform by selecting a new color from the color palette, or by entering an X-Windows color name.

- **Pixels = (value + <offset>) * <scale factor>**

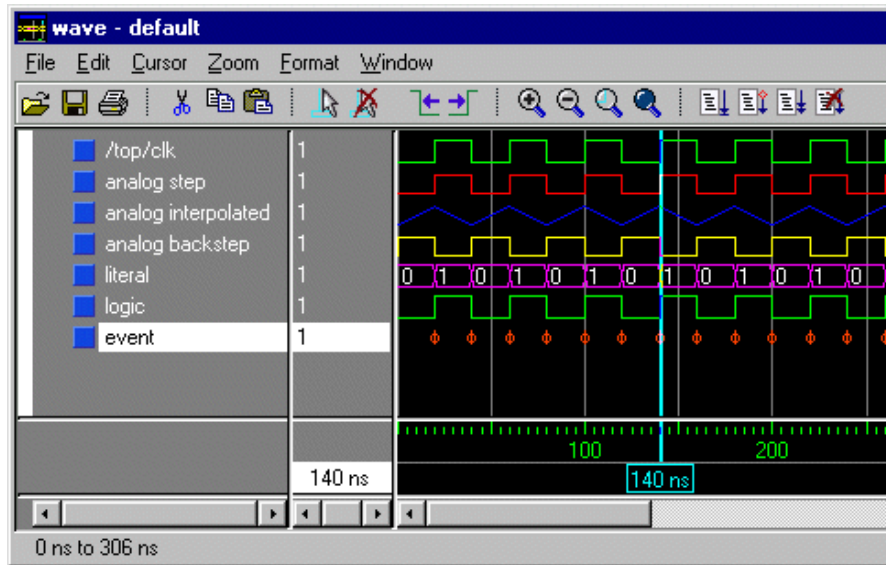
This choice works with analog items only and allows you to decide on the

scale of the item as it is seen on the display. Value is the value of the signal at a given time, <offset> is the number of pixels offset from zero. The <scale factor> reduces (if less than 1) or increases (if greater than 1) the number of pixels displayed.

- **Radix**

The explicit choices are Symbolic, Binary, Octal, Decimal, Unsigned, Hexadecimal, and ASCII. If you select Default the signal's radix changes whenever the default is changed using the **radix** command (CR-129). Item values are not translated if you select Symbolic.

- **Format: Analog [Step | Interpolated | Backstep]**



All signals in this illustration are the same */top/clk* signal. Starting with "analog step", the */top/clk* signal has been relabeled to illustrate each different wave formats.

Analog Step

Displays a waveform in step style.

Analog Interpolated

Displays the

waveform in interpolated style.

Analog Backstep

Displays the waveform in backstep style. Often used for power calculations.

Only the following types are supported in Analog format:

VHDL types:

All vectors - std logic vectors, bit vectors, and vectors derived from these types

Scaler integers

Scaler reals

Scaler time

Verilog types:

All vectors

Scaler real

Scalar integers

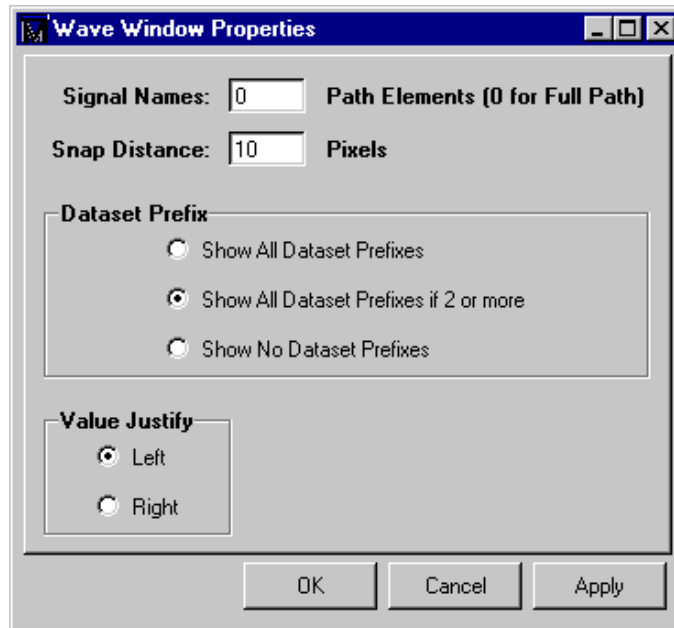
Wave height and offset can be set from the Wave window menu Prop > Signal props by adjusting the parameters to the Pixels equation.

- **Format: Literal**

Displays the waveform as a box containing the item value (if the value fits the space available). This is the only format that can be used to list a record.

- **Format: Logic**
Displays values as 0, 1, X, Z, H, L, U, or -.
- **Format: Event**
Marks each transition during the simulation run.

Setting Wave window display properties



You can define the display properties of the pathname and values window panes by selecting **Edit > Display Properties** in the Wave window.

The **Wave Window Properties** dialog box includes the following options:

- **Signal Names**

This selection allows you to display the full pathname of each signal (i.e.: *sim:/top/clock*), or its leaf element only (i.e.: *sim:clock*). The default is Full Path.

- **Snap Distance**

Specifies the distance the cursor needs to be placed from an item edge to jump to that edge (a 0 specification turns off the snap). The value displayed in the item value windowpane is updated to reflect the snap.

- **Dataset Prefix**

With this selection you can determine how signals from different datasets are displayed.

Show All Dataset Prefixes

All dataset prefixes will be displayed along with the dataset prefix of the current simulation ("sim").

Show All Dataset Prefixes if 2 or more

Displays all dataset prefixes if 2 or more datasets are displayed. "sim" is the default prefix for the current simulation.

Show No Dataset Prefixes

No dataset prefixes will be display. This selection is useful if you are only running a single simulaiton.

- **Value Justify**

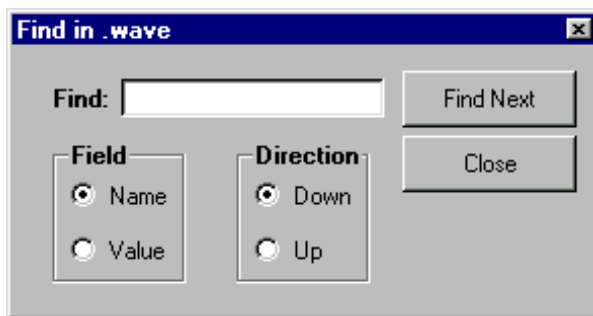
Specifies whether the signal values will be justified to the left margin or the right margi in the values window pane.

Sorting a group of HDL items

Use the **Edit > Sort** menu selection to sort the items in the name/value pane.

Finding items by name or value in the Wave window

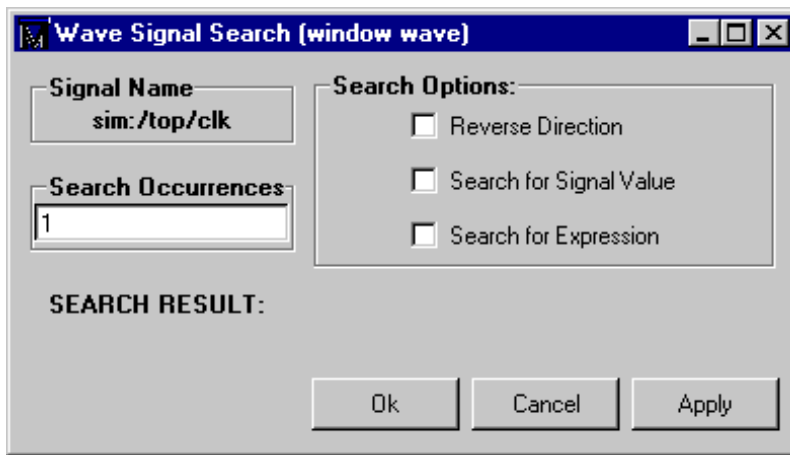
The Find dialog box allows you to search for text strings in the Wave window. From the Wave window select **Edit > Find** to bring up the Find dialog box.



Choose either the Name or Value field to search from the drop-down menu, and enter the value to search for in the Find field. **Find** the item by searching **Forward** (down) or **Reverse** (up) through the Wave window display.

Searching for item values in the Wave window

Select an item in the Wave window. From the Wave window menu bar select **Edit > Search** to bring up the Wave Signal Search dialog box.



The **Wave Signal Search** dialog box includes these options:

- **Signal Name**
<item_label>

This indicates the item currently selected in the Wave window; the subject of the search.

- **Search Options:**
Reverse Direction

Search the list from right to left. Deselect to search from left to right.

- **Search Options: Search for Signal Value**

Reveals the Search Value field; search for the value specified in the Search Value field (the value must be formatted in the same radix as the display). If no value is specified the search will look for transitions.

- **Search Options: Search for Expression**

Reveals the Search Expression field and the Use Expression Builder button; searches for the expression specified in the Search Expression field evaluating to a boolean true.

The expression may involve more than one signal but is limited to signals logged in the List window. Expressions may include constants, variables, and macros. If no expression is specified, the search will give an error. See ["GUI_expression_format"](#) (CR-250) for information on creating an expression.

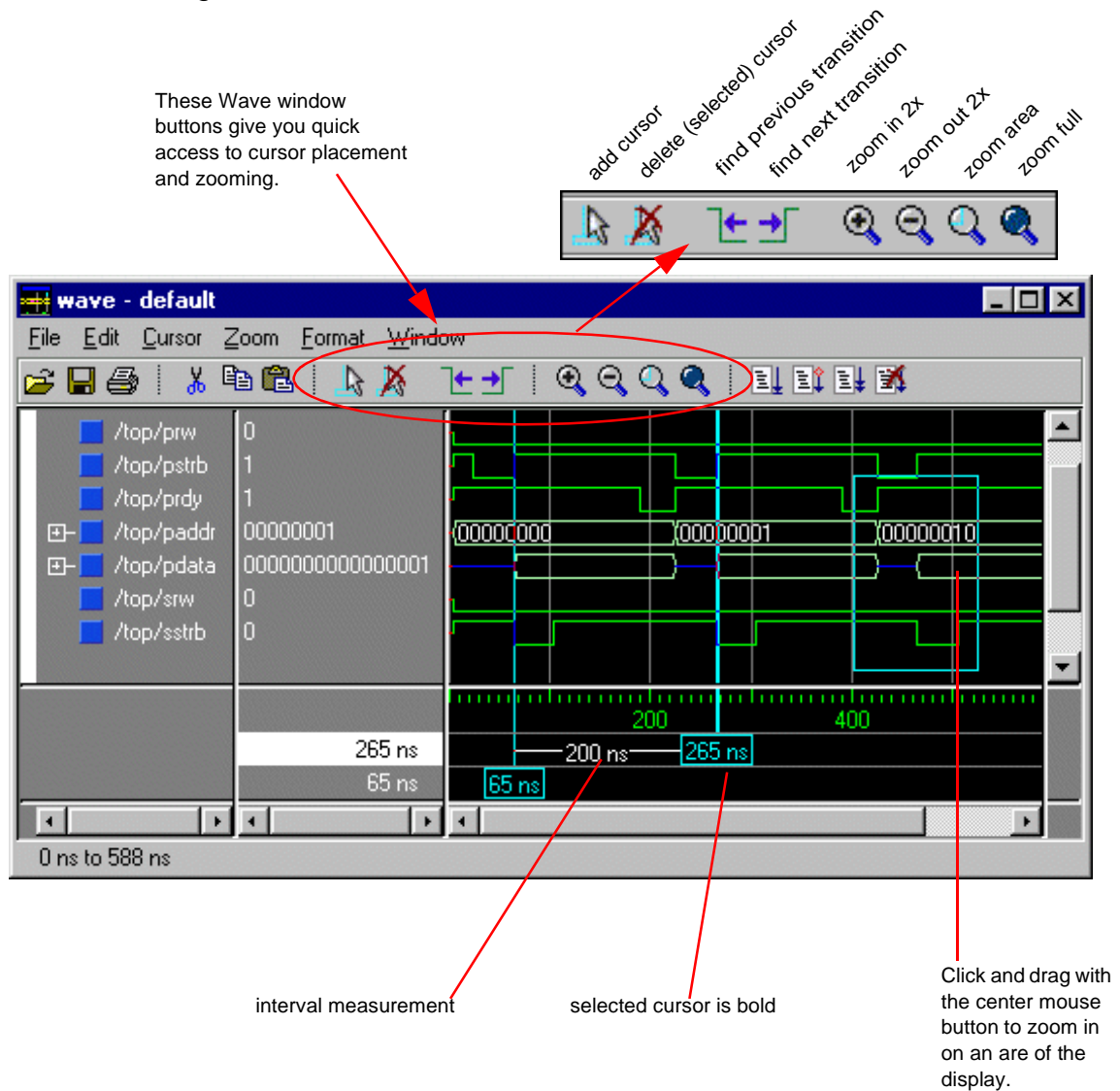
To help build the expression, click the **Use Expression Builder** button to open ["The GUI Expression Builder"](#) (10-272).

- **Search Occurrences**

You can search for the n-th transition or the n-th match on value or expression; Search Occurrences indicates the number of transitions or matches for which to search.



- The **SEARCH RESULT** is indicated at the bottom of the dialog box.

Using time cursors in the Wave window



When the Wave window is first drawn, there is one cursor located at time zero. Clicking anywhere in the waveform display brings that cursor to the mouse location. You can add additional cursors to the waveform pane with the **Cursor > Add Cursor** menu selection (or the Add Cursor button shown below). The

selected cursor is drawn as a bold solid line; all other cursors are drawn with thin solid lines. Remove cursors by selecting them and choosing using the **Cursor > Delete Cursor** menu selection (or the Delete Cursor button shown below).

	Add Cursor add a cursor to the center of the waveform window		Delete Cursor delete the selected cursor from the window
---	--	---	--

Finding a cursor

The cursor value (on the **Goto** list) corresponds to the simulation time of that cursor. Choose a specific cursor view with **Cursor > Goto** menu selection.

Making cursor measurements



Each cursor is displayed with a time box showing the precise simulation time at the bottom. When you have more than one cursor, each time box appears in a separate track at the bottom of the display. VSIM also adds a delta measurement showing the time difference between the two cursor positions.

If you click in the waveform display, the cursor closest to the mouse position is selected and then moved to the mouse position. Another way to position multiple cursors is to use the mouse in the time box tracks at the bottom of the display. Clicking anywhere in a track selects that cursor and brings it to the mouse position.

The cursors are designed to snap to the closest wave edge to the left on the waveform that the mouse pointer is positioned over. You can control the snap distance from "Wave category" in the dialog box available from the Wave window **Prop > Display Props** menu selection.

You can position a cursor without snapping by dragging in the area below the waveforms.

You can also move cursors to the next transition of a signal with these toolbar buttons:

	Find Previous Transition locate the previous signal value change for the selected signal		Find Next Transition locate the next signal value change for the selected signal
---	--	---	--

Zooming - changing the waveform display range

Zooming lets you change the simulation range in the windowpane display. You can zoom with either the **Zoom** menu, toolbar buttons, mouse, keyboard, or VSIM commands.

Using the Zoom menu

You can use the Wave window menu bar, or call up the **Zoom** menu by clicking the right mouse button (of a three-button mouse) in the right windowpane.





Note: The right mouse button of a two-button mouse will not open the **Zoom** menu. It will, however, allow you to create a zoom area by dragging left to right while holding down the button.

The Zoom menu options include:

- **Zoom Full**
Redraws the display to show the entire simulation from time 0 to the current simulation time.
- **Zoom In**
Zooms in by a factor of two, increasing the resolution and decreasing the visible range horizontally, cropping the view on the right. The starting time is held static.
- **Zoom Out**
Zooms out by a factor of two, decreasing the resolution and increasing the visible range horizontally, extending the view on the right. The starting time is held static.
- **Zoom Last**
Restores the display to where it was before the last zoom operation.
- **Zoom Area with Mouse Button 1**
Use mouse button 1 to create a zoom area. Position the mouse cursor to the left side of the desired zoom interval, press mouse button 1 and drag to the right. Release when the box has expanded to the right side of the desired zoom interval.
- **Zoom Range**
Brings up a dialog box that allows you to enter the beginning and ending times for a range of time units to be displayed.

Zooming with the toolbar buttons

These zoom buttons are available on the toolbar:

	Zoom in 2x zoom in by a factor of two from the current view		Zoom area use the cursor to outline a zoom area
	Zoom out 2x zoom out by a factor of two from current view		Zoom Full zoom out to view the full range of the simulation from time 0 to the current time

Zooming with the mouse

To zoom with the mouse, position the mouse cursor to the left side of the desired zoom interval, press the middle mouse button (three-button mouse), or right button (two-button mouse), and while continuing to press, drag to the right and then release at the right side of the desired zoom interval.

Zooming keyboard shortcuts

See ["Wave window keyboard shortcuts"](#) (10-237) for a complete list of Wave window keyboard shortcuts.

Zooming with VSIM commands

The **.wave.tree zoomfull** provides the same function as the **Zoom > Zoom Full** menu selection and the **.wave.tree zoomrange** provides the same function as the **Zoom > Zoom Range** menu selection.

Use this syntax with the **.wave.tree zoomrange** command:

Syntax

```
.wave.tree zoomrange f1 f2
```

Arguments

f1 f2
Sets the waveform display to zoom from time *f1* to *f2*, where *f1* and *f2* are floating point numbers.

Wave window keyboard shortcuts

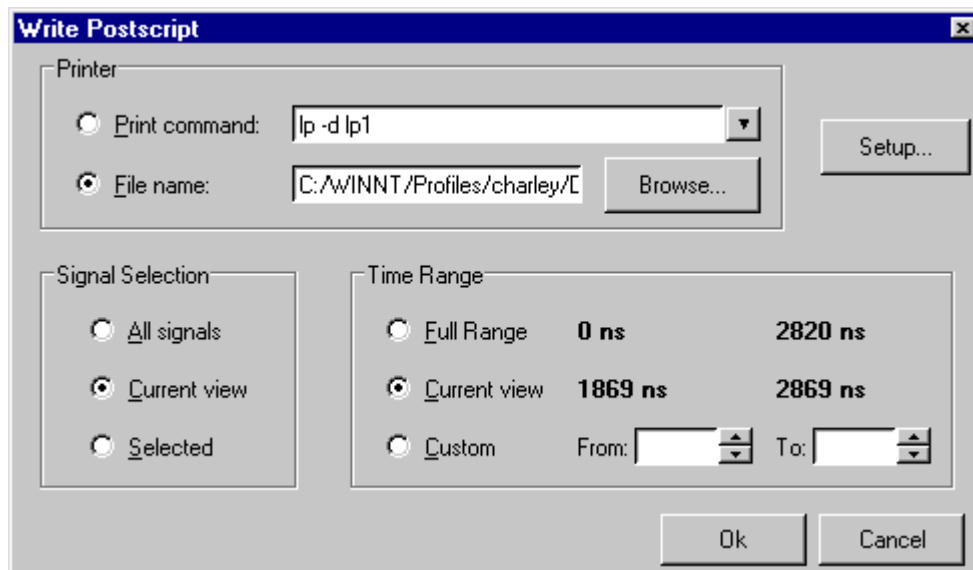
Using the following keys when the mouse cursor is within the Wave window will cause the indicated actions:

Key	Action
i I or +	zoom in
o O or -	zoom out
f or F	zoom full
l or L	zoom last
r or R	zoom range
<arrow up>	scroll waveform display up
<arrow down>	scroll waveform display down
<arrow left>	scroll waveform display left
<arrow right>	scroll waveform display right
<page up>	scroll waveform display up by page
<page down>	scroll waveform display down by page
<tab>	searches forward (right) to the next transition on the selected signal - finds the next edge
<shift-tab>	searches backward (left) to the previous transition on the selected signal - finds the previous edge
<control-f>	opens the find dialog box; search within the specified field in the wave-name pane for text strings

Printing and saving waveforms

Saving a .eps file and printing under UNIX

Use the **File > Print Postscript** menu selection in the 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. Printing and writing preferences are controlled by the dialog box shown below.



The **Write Postscript** dialog box includes these options:

Printer

- **Print command**
enter a UNIX print command to print the waveform in a UNIX environment
- **File name**
enter a filename for the encapsulated Postscript (.eps) file to be created; or browse to a previously created .eps file and use that filename.

Signal Selection

- **All signals**
prints all signals
- **Current View**
prints signals in current view
- **Selected**
prints all selected signals

Time Range

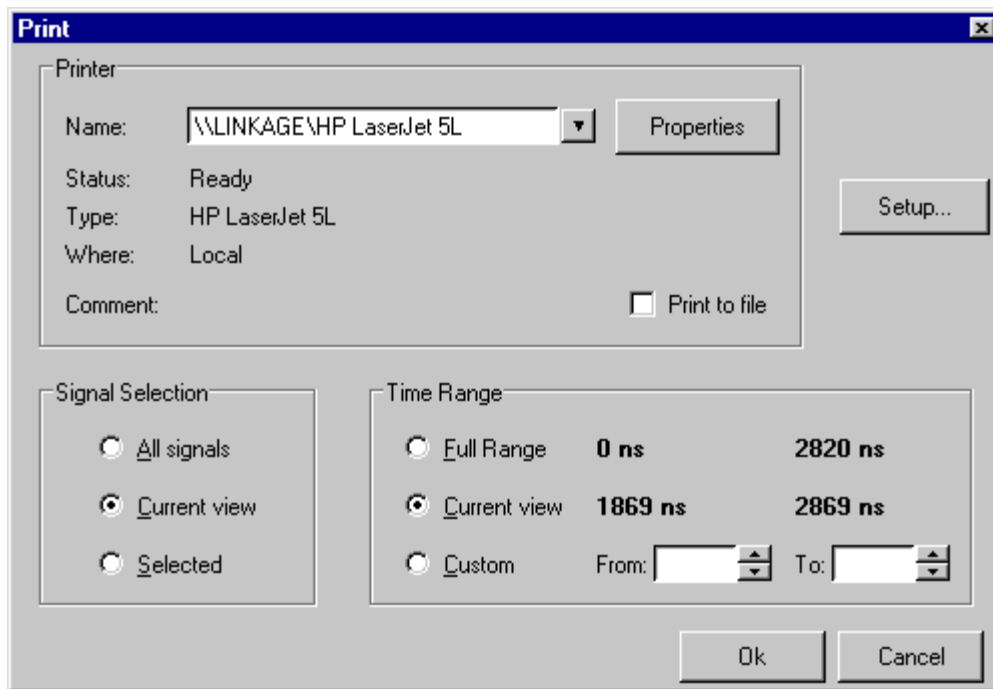
- **Full Range**
prints all specified signals in the full simulation range
- **Current view**
prints specified signals for the viewable time range
- **Custom**
prints the specified signals for user-designated **From** and **To** time

Setup button

See ["Printer Page Setup"](#) (10-242)

Printing on Windows platforms

Use the **File > Print** menu selection in the Wave window to print all or part of the waveform in the current Wave window, or save the waveform as a printer file (a Postscript file for Postscript printers). Printing and writing preferences are controlled by the dialog box shown below.

*Printer*

- **Name**
Choose the printer from the drop-down menu. Set printer properties with the *Properties* button.
- **Status**
Indicates the availability of the selected printer.
- **Type**
Printer driver name for the selected printer. The driver determines what type of file is output if "Print to file" is selected.

- **Where**
The printer port for the selected printer.
- **Comment**
The printer comment from the printer properties dialog box.
- **Print to file**
Make this selection to print the waveform to a file instead of a printer. The printer driver determines what type of file is created. Postscript printers create a Postscript (.ps) file, non-Postscript printers print a .prn or printer control language file. To create an encapsulated Postscript file (.eps) use the File > Print Postscript menu selection.

Signal Selection

- **All signals**
prints all signals
- **Current View**
prints signals in current view
- **Selected**
prints all selected signals

Time Range

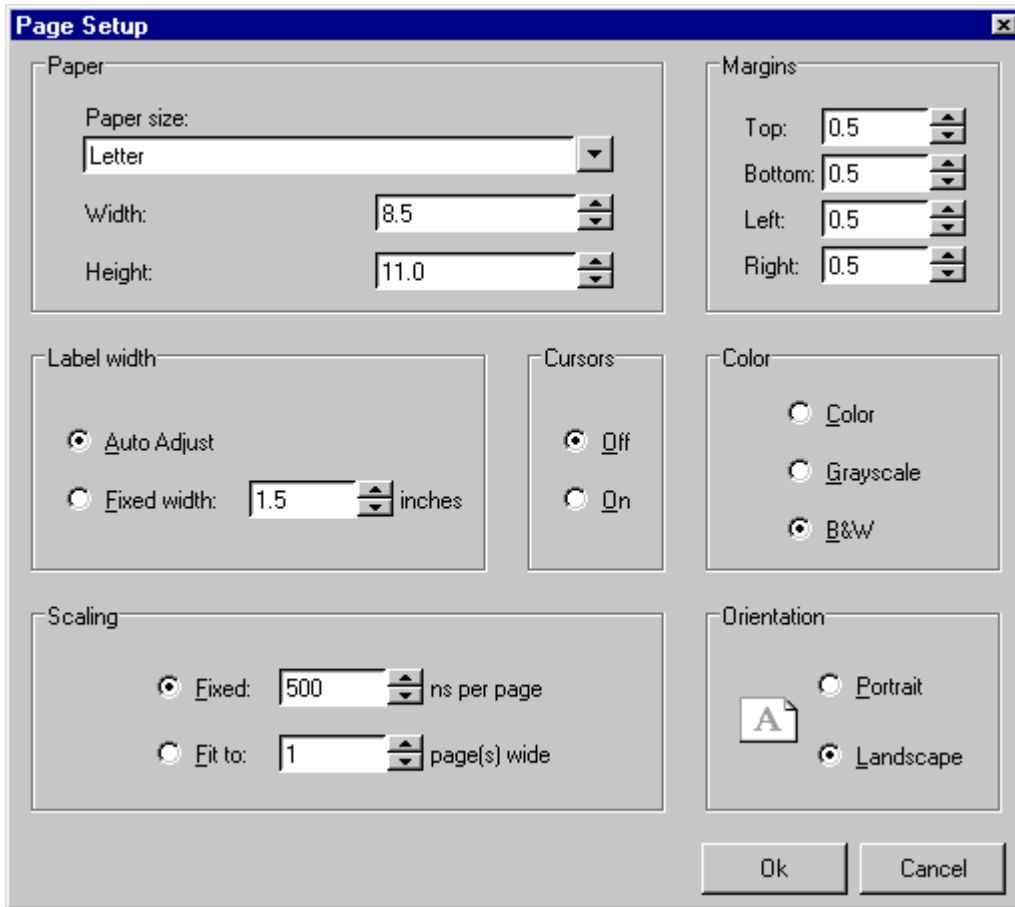
- **Full Range**
prints all specified signals in the full simulation range
- **Current view**
prints specified signals for the viewable time range
- **Custom**
prints the specified signals for user-designated **From** and **To** time

Setup button

See ["Printer Page Setup"](#) (10-242)

Printer Page Setup

Clicking the Setup button in the Write Postscript or Print dialog box allows you to define the following options (this is the same dialog that opens with File > Page setup menu selection).



- **Paper Size**
select your output page size from a number of options; also choose the paper width and height
- **Margins**
specify the margin in inches or centimeters (units are controlled by the inches/cm selection); changing the **Margin** will change the **Scale** and **Page** specifications

- **Label width**
specify Auto Adjust to accomodate any length label, or set a fixed lable with
- **Cursors**
turn printing of cursors on or off
- **Color**
select full color printing, grayscale or black and white
- **Scaling**
specify a **Fixed** output time width in nanoseconds per page – the number of pages output is automatically computed; or, select **Fit to** to define the number of pages to be output based on the paper size and time settings; if set, the time-width per page is automatically computed
- **Orientation**
select the output page orientation, **Portrait** or **Landscape**

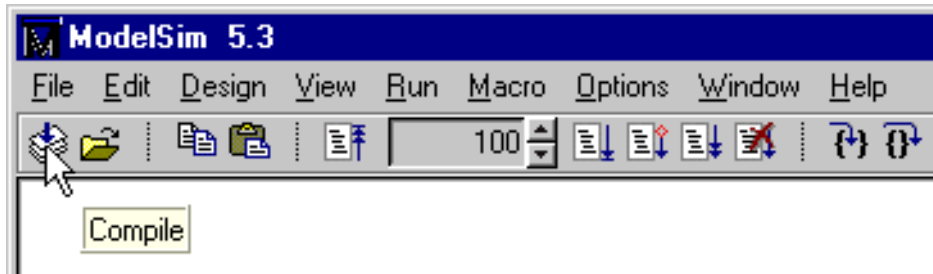
The vsim.ps include file

The Postscript file that VSIM creates (if you print to a file) includes a file named *vsim.ps* that is shipped with ModelSim. The file is "included" in the Postscript output from the waveform display; it sets the fonts, spacing, and print header and footer (the font and spacing is based on those used in the Wave window). If you want to change any of the Postscript defaults, you can do it by making changes in this file. Note that you should copy the file before making your changes so that you can save the original.

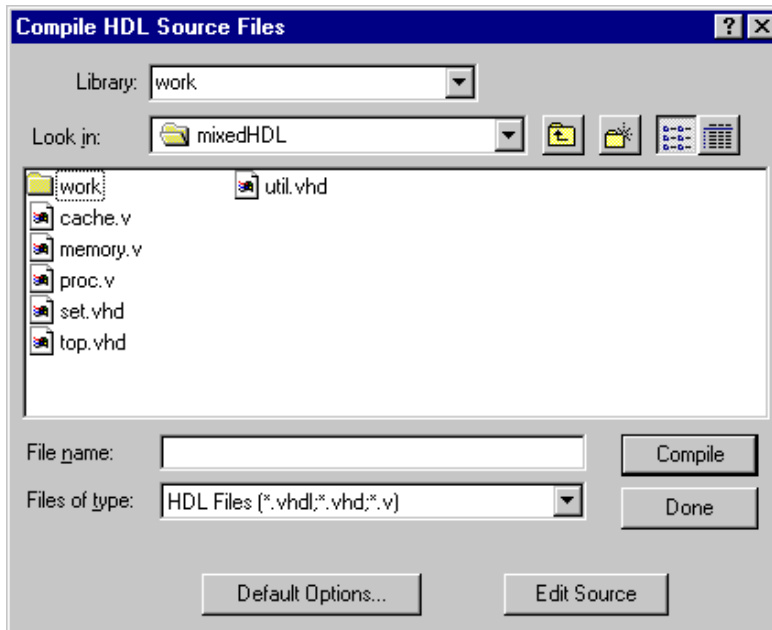
```
% Copyright 1993 Model Technology Incorporated.
% All rights reserved.
% A(#)vsim.ps 1.2 13 Mar 1994
%
% This file is 'included' in the postscript output from the
% waveform display.
%
% pick the fonts
/fontheight 10 def
/mainfont {/Helvetica-Narrow findfont fontheight scalefont setfont} def
/smallfont {/Helvetica-Narrow findfont fontheight 3 sub scalefont setfont} def
mainfont
3 10 div setlinewidth
/signal_spacing fontheight 9 add def
/one_ht fontheight 2 sub def
/z_ht one_ht 2 div def
/ramp 2 def
/hz_tick_len 4 def
...
```

Compiling with the graphic interface

To compile either VHDL or Verilog designs, select the **Compile** button on the Main window toolbar.



The **Compile HDL Source Files** dialog box opens as shown below.



From the **Compile HDL Source Files** dialog box you can:

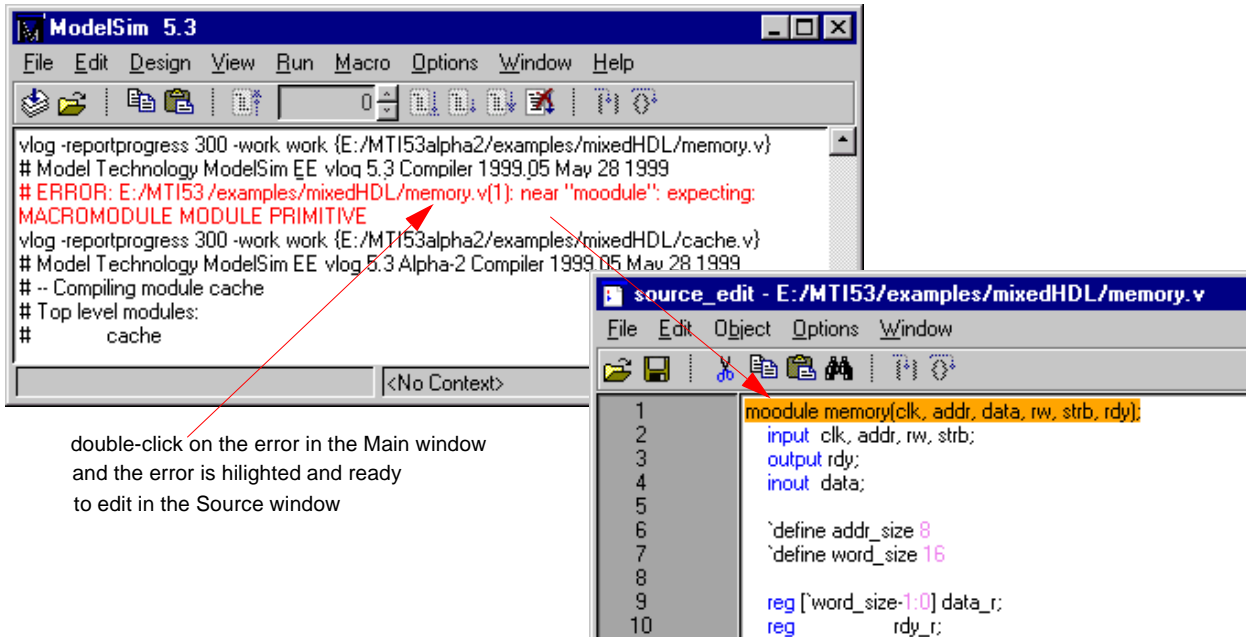
- select source files to compile in any language combination
- specify the target library for the compiled design units
- select among the compiler options for either VHDL or Verilog

Select the **Default Options** button to change the compiler options, see ["Setting default compile options"](#) (10-245) for details. The same Compiler Options dialog box can also be accessed with the **Options > Compile** Main window menu selection.

Select the **Edit Source** button to view or edit a source file via the Compile dialog box. See ["Source window"](#) (10-200), and ["Editing the command line, the current source file, and notepads"](#) (10-167) for additional source file editing information.

Locating source errors during compilation

If a compiler error occurs during compilation, a red error message is printed in the Main transcript. Double-click on the error message to open the source file in an editable Source window with the error highlighted.



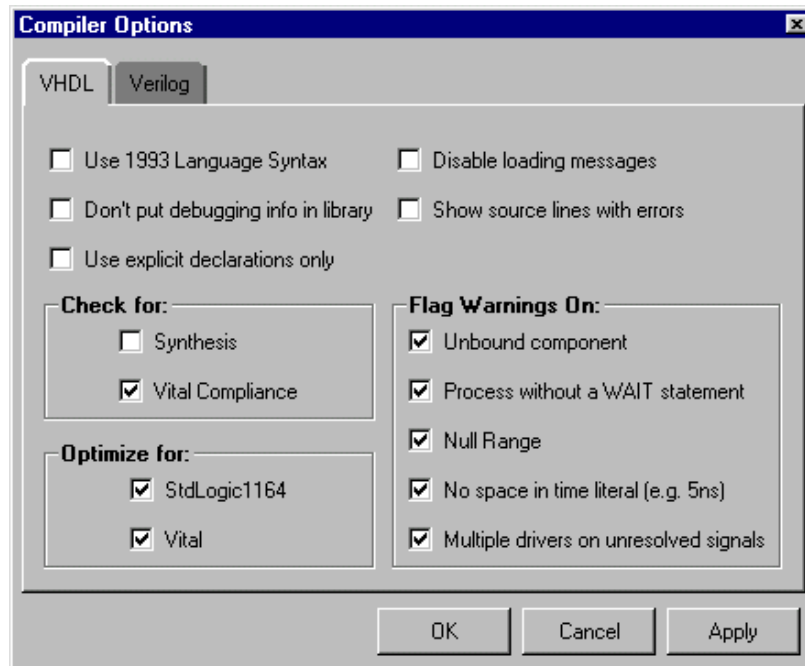
double-click on the error in the Main window and the error is highlighted and ready to edit in the Source window

Setting default compile options

Use the **Options > Compile** menu selection to bring up the **Compile Options** dialog box shown below. **OK** accepts the changes made and closes the dialog box. **Apply** makes the changes with the dialog box open so you can test your settings. **Cancel** closes the dialog box and makes no changes. The options found on each page of the dialog box are detailed below.

Note: Changes made in the **Compile Options** dialog box are the default for the current simulation only. Options can be saved as the default for future simulations by editing the simulator control variables in the *modelsim.ini* file; the variables to edit are noted in the text below. You can use the **notepad** (CR-104) to edit the variables in *modelsim.ini* if you wish. See also, "[Projects and system initialization](#)" (3-43) for more information.

VHDL compiler options pag



- Use 1993 language syntax**
 Specifies the use of VHDL93 during compilation. The 1987 standard is the default. Same as the **-93** switch for the **vdel** command (CR-175). Edit the **VHDL93** (B-395) in the *modelsim.ini* to set a permanent default.
- Don't put debugging info in library**
 Models compiled with this option do not use any of the ModelSim debugging features. Consequently, your user will not be able to see into the model. This also means that you cannot set breakpoints or single step within this code. Don't compile with this option until you're done debugging. Same as the **-nodebug** switch for the **vdel** command (CR-175). See "[Source code security and -nodebug](#)" (E-443) for more details. Edit the **NoDebug** (B-396) in the *modelsim.ini* to set a permanent default.
- Use explicit declarations only**
 Used to ignore an error in packages supplied by some other EDA vendors; directs the compiler to resolve ambiguous function overloading in favor of the

explicit function definition. Same as the **-explicit** switch for the **vdcl** command (CR-175). Edit the **Explicit** (B-396) in the *modelsim.ini* to set a permanent default.

Although it is not intuitively obvious, the = operator is overloaded in the **std_logic_1164** package. All enumeration data types in VHDL get an “implicit” definition for the = operator. So while there is no explicit = operator, there is an implicit one. This implicit declaration can be hidden by an explicit declaration of = in the same package (LRM Section 10.3). However, if another version of the = operator is declared in a different package than that containing the enumeration declaration, and both operators become visible through **use** clauses, neither can be used without explicit naming, i.e.,

```
ARITHMETIC."="(left, right)
```

This option allows the explicit = operator to hide the implicit one.

- **Disable loading messages**

Disables loading messages in the Transcript window. Same as the **-quiet** switch for the **vdcl** command (CR-175). Edit the **Quiet** (B-396) in the *modelsim.ini* to set a permanent default.

- **Show source lines with errors**

Causes the compiler to display the relevant lines of code in the transcript. Same as the **-source** switch for the **vdcl** command (CR-175). Edit the **Show_source** (B-395) in the *modelsim.ini* to set a permanent default.

Flag Warnings on:

- **Unbound Component**

Flags any component instantiation in the VHDL source code that has no matching entity in a library that is referenced in the source code, either directly or indirectly. Edit the **Show_Warning1** (B-395) in the *modelsim.ini* to set a permanent default.

- **Process without a wait statement**

Flags any process that does not contain a wait statement or a sensitivity list. Edit the **Show_Warning2** (B-395) in the *modelsim.ini* to set a permanent default.

- **Null Range**

Flags any null range, such as 0 down to 4. Edit the **Show_Warning3** (B-396) in the *modelsim.ini* to set a permanent default.

- **No space in time literal (e.g. 5ns)**

Flags any time literal that is missing a space between the number and the time unit. Edit the **Show_Warning4** (B-396) in the *modelsim.ini* to set a permanent default.

- **Multiple drivers on unresolved signal**

Flags any unresolved signals that have multiple drivers. Edit the [Show_Warning5](#) (B-396) in the *modelsim.ini* to set a permanent default.

Check for:

- **Synthesis**

Turns on limited synthesis-rule compliance checking. Edit the [CheckSynthesis](#) (B-396) in the *modelsim.ini* to set a permanent default.

- **Vital Compliance**

Toggle Vital compliance checking. Edit the [NoVitalCheck](#) (B-396) in the *modelsim.ini* to set a permanent default.

Optimize for:

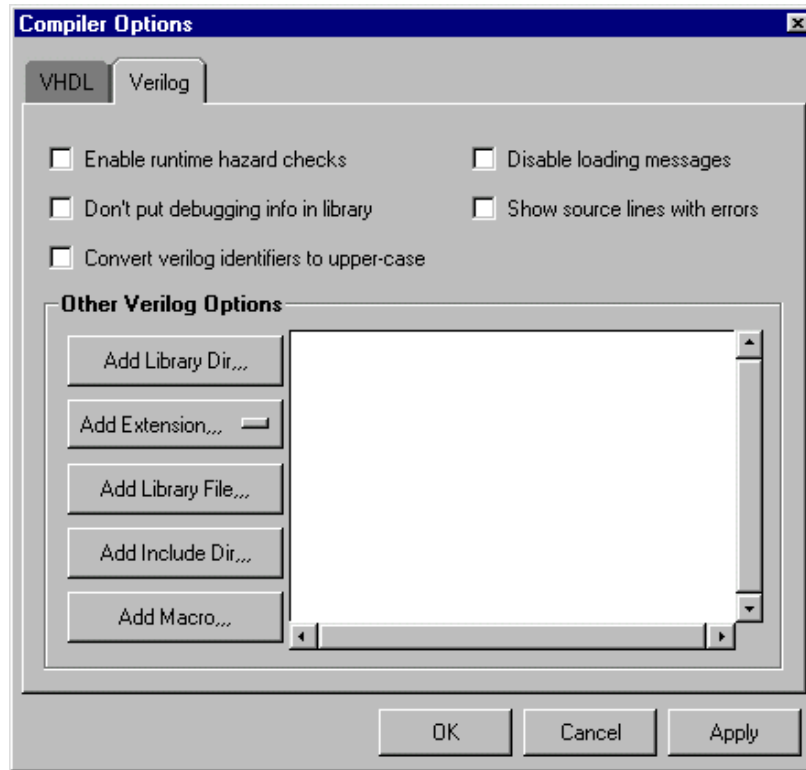
- **std_logic_1164**

Causes the compiler to perform special optimizations for speeding up simulation when the multi-value logic package `std_logic_1164` is used. Unless you have modified the `std_logic_1164` package, this option should always be checked. Edit the [Optimize_1164](#) (B-396) in the *modelsim.ini* to set a permanent default.

- **Vital**

Toggle acceleration of the Vital packages. Edit the [NoVital](#) (B-396) in the *modelsim.ini* to set a permanent default.

Verilog compiler options page



- Enable run-time hazard checks**
 Enables the run-time hazard checking code. Same as the **-hazards** switch for the **vlog** command (CR-199). Edit the **Hazard** (B-396) in the *modelsim.ini* to set a permanent default.
- Don't put debugging info in library**
 Models compiled with this option do not use any of the ModelSim debugging features. Consequently, your user will not be able to see into the model. This also means that you cannot set breakpoints or single step within this code. Don't compile with this option until you're done debugging. Same as the **-nodebug** switch for the **vlog** command (CR-199). See "[Source code security and -nodebug](#)" (E-443) for more details. Edit the **NoDebug** (B-396) in the *modelsim.ini* to set a permanent default.
- Convert Verilog identifiers to upper-case**
 Converts regular Verilog identifiers to uppercase. Allows case insensitivity for

module names. Same as the **-u** switch for the **vlog** command (CR-199). Edit the **UpCase** (B-397) in the *modelsim.ini* to set a permanent default.

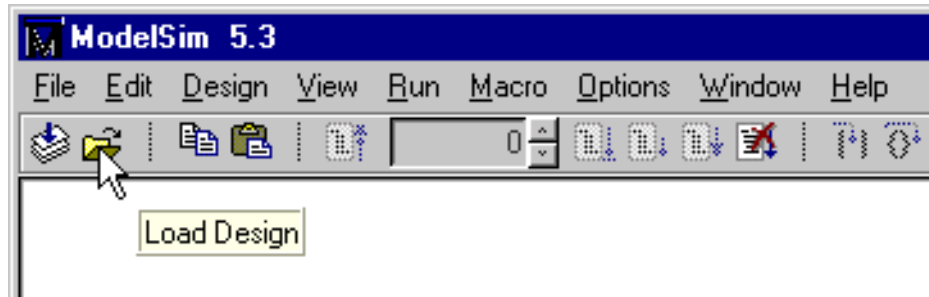
- **Disable loading messages**
Disables loading messages in the Transcript window. Same as the **-quiet** switch for the **vlog** command (CR-199). Edit the **Quiet** (B-396) in the *modelsim.ini* to set a permanent default.
- **Show source lines with errors**
Causes the compiler to display the relevant lines of code in the transcript. Same as the **-source** switch for the **vlog** command (CR-199). Edit the **Show_source** (B-395) in the *modelsim.ini* to set a permanent default.

Other Verilog Options:

- **Add Library Dir**
Specifies the Verilog source library directory to search for undefined modules. Same as the **-y <library_directory>** switch for the **vlog** command (CR-199).
- **Add Extension**
Specifies the suffix of files in library directory. Multiple suffixes may be used. Same as the **+libext+<suffix>** switch for the **vlog** command (CR-199).
- **Add Library File**
Specifies the Verilog source library file to search for undefined modules. Same as the **-v <library_file>** switch for the **vlog** command (CR-199).
- **Add Include Dir**
Search specified directory for files included with the **'include filename** compiler directive. Same as the **+incdir+<directory>** switch for the **vlog** command (CR-199).
- **Add Macro**
Define a macro to execute during compilation. Same as compiler directive: **'define macro_name macro_text**. Also the same as the **+define+<macro_name> [=<macro_text>]** switch for the **vlog** command (CR-199).

Simulating with the graphic interface

The **Load Design** dialog box is activated when the Load Design button is selected from the Main window toolbar.



Four pages - **Design**, **VHDL**, **Verilog**, and **SDF** - allow you to select various simulation options.

You can switch between pages to modify settings, then begin simulation by selecting the **Load** button. If you select **Cancel**, all selections remain unchanged and you are returned to the Main VSIM window; the **Exit** button (only active before simulation) closes ModelSim. The **Save Settings** button allows you to save the preferences on all pages to a do (macro) file.

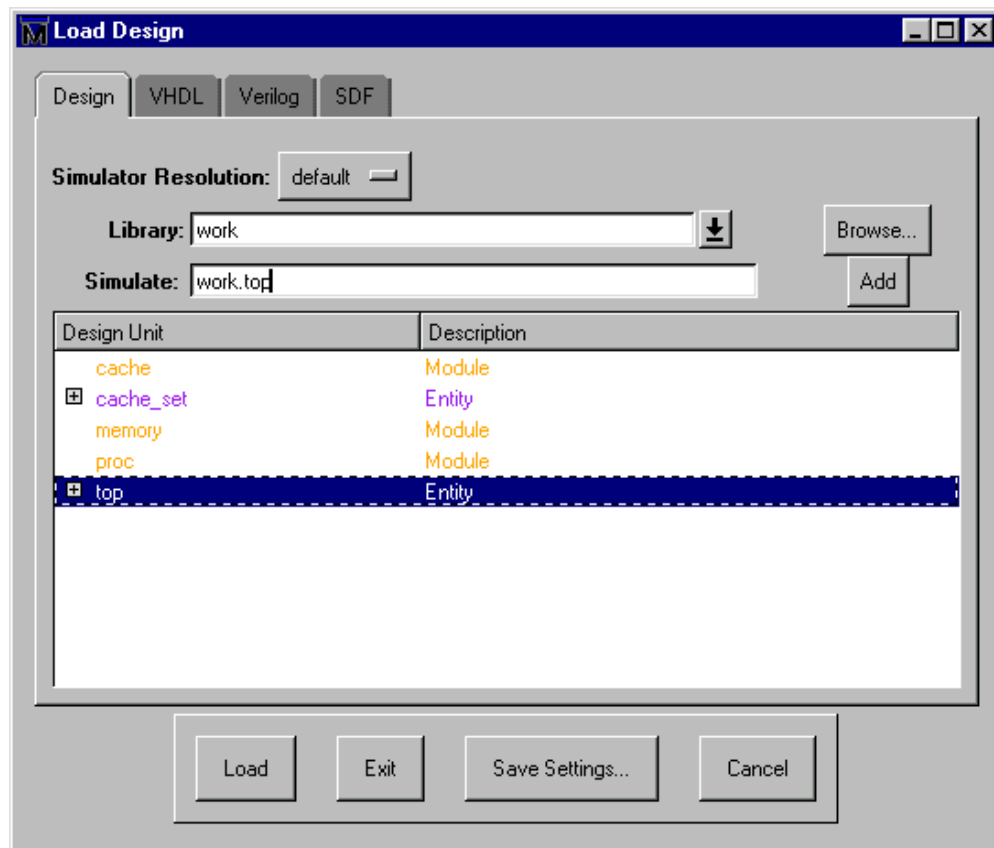
Compile before you simulate

To begin simulation you must have compiled design units located in a design library, see "[Creating a design library](#)" (4-52).

VSIM command options

Options that correspond to **vsim** (CR-208) commands are noted within parentheses in the text below, i.e., **Simulator Resolution** (-t [<multiplier>]<time_unit>).

Design selection page



Note: The Exit button closes the Load Design dialog box and quits ModelSim.

The **Design** page includes these options:

- **Simulator Resolution**
(-time [<multiplier>]<time_unit>)
The drop-down menu sets the simulator time units (original default is ns).
- **Library**
Specifies a library for viewing in the **Design Unit** list box. You can use the drop-down list (click the arrow) to select a "mapped" library or you can type in

a library name. You can also use the **Browse** button to locate a library among your directories. Make certain your selection is a valid ModelSim library - it must include an *_info* file and must have been created from ModelSim's **vlib** command (CR-198). Once the library is selected you can view its design units within the **Design Unit** list box.

- **Simulate** (<configuration> | <module> | <entity> [(<architecture>)])
Specifies the design unit(s) to simulate. You can simulate several Verilog top-level modules or a VHDL top-level design unit in one of three ways:
 1. Type a design unit name (configuration, module, or entity) into the field, separate additional names with a space. Specify library/design units with the following syntax:

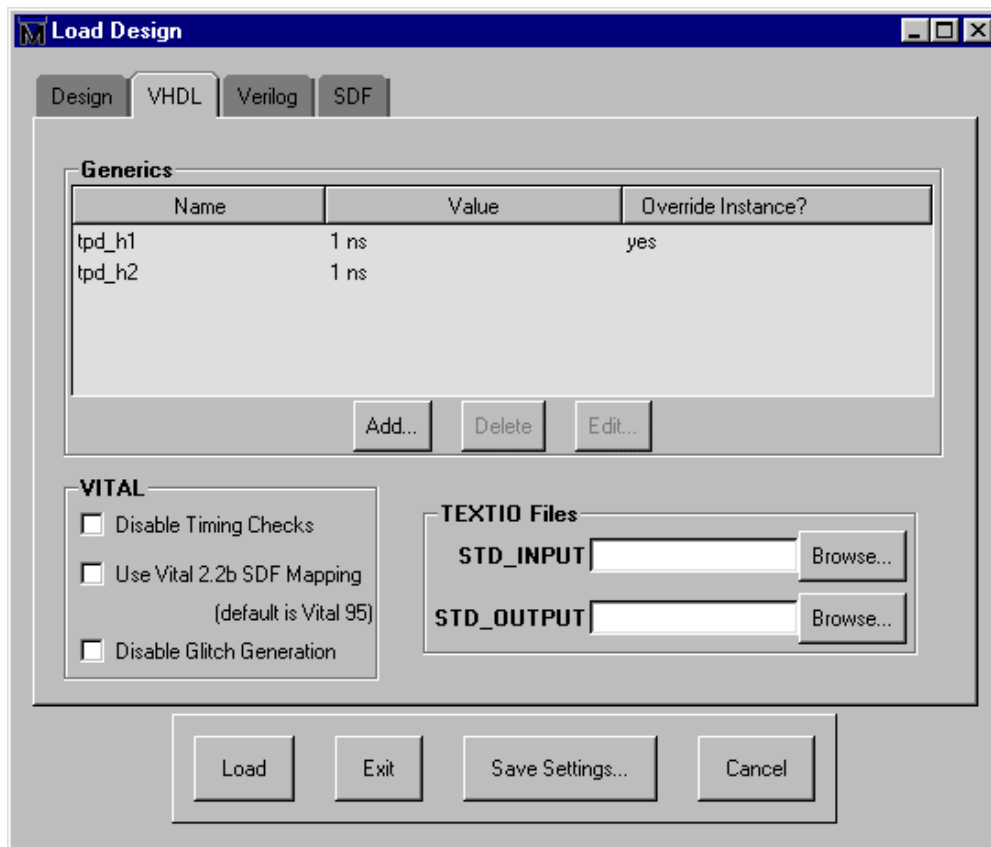
```
[<library_name>.<design_unit>
```
 2. Click on a name in the **Design Unit** list below and click the **Add** button.
 3. Leave this field blank and click on a name in the **Design Unit** list (single unit only).
- **Design Unit/Description**
This hierarchal list allows you to select one top-level entity or configuration to be simulated. All entities and configurations that exist in the specified library are displayed in the list box. Architectures may be viewed by selecting the "+" box before any name.

Simulator time units may be expressed as any of the following:

Simulation time units	
1fs, 10fs, or 100fs	femtoseconds
1ps, 10ps, or 100ps	picoseconds
1ns, 10ns, or 100ns	nanoseconds
1us, 10us, or 100us	microseconds
1ms, 10ms, or 100ms	milliseconds
1sec, 10sec, or 100sec	seconds

See also, "[Selecting the time resolution](#)" (4-54).

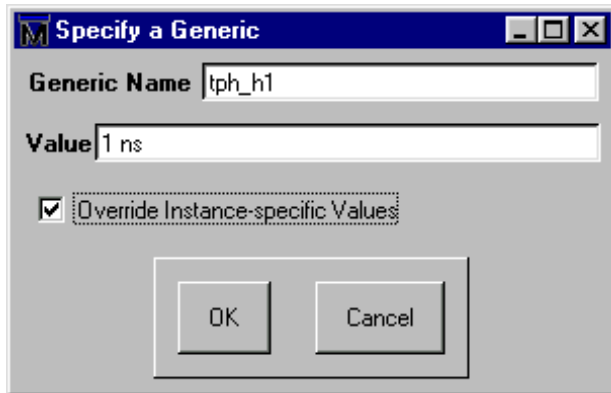
VHDL settings page



The **VHDL** page includes these options:

Generics

The **Add** button opens a dialog box that allows you to specify the value of generics within the current simulation; generics are then added to the **Generics** list. You may also select a generic on the listing to **Delete** or **Edit** (opens the dialog box below).



From **Specify a Generic** dialog box you can set the following options.

- **Generic Name** (-g <Name=Value>)

The name of the generic parameter. You can make a selection from the drop-down menu or type it in as it appears in the VHDL source (case is ignored).

- **Value**

Specifies a value for all generics in the design with the given name (above)

that have not received explicit values in generic maps (such as top-level generics and generics that would otherwise receive their default value). Value is an appropriate value for the declared data type of the generic parameter. No spaces are allowed in the specification (except within quotes) when specifying a string value.

- **Override Instance - specific Values** (-G <Name=Value>)

Select to override generics that received explicit values in generic maps. The name and value are specified as above. The use of this switch is indicated in the **Override Instance** column of the **Generics** list.

The **OK** button adds the generic to the **Generics** listing; **Cancel** dismisses the dialog box without changes.

VITAL

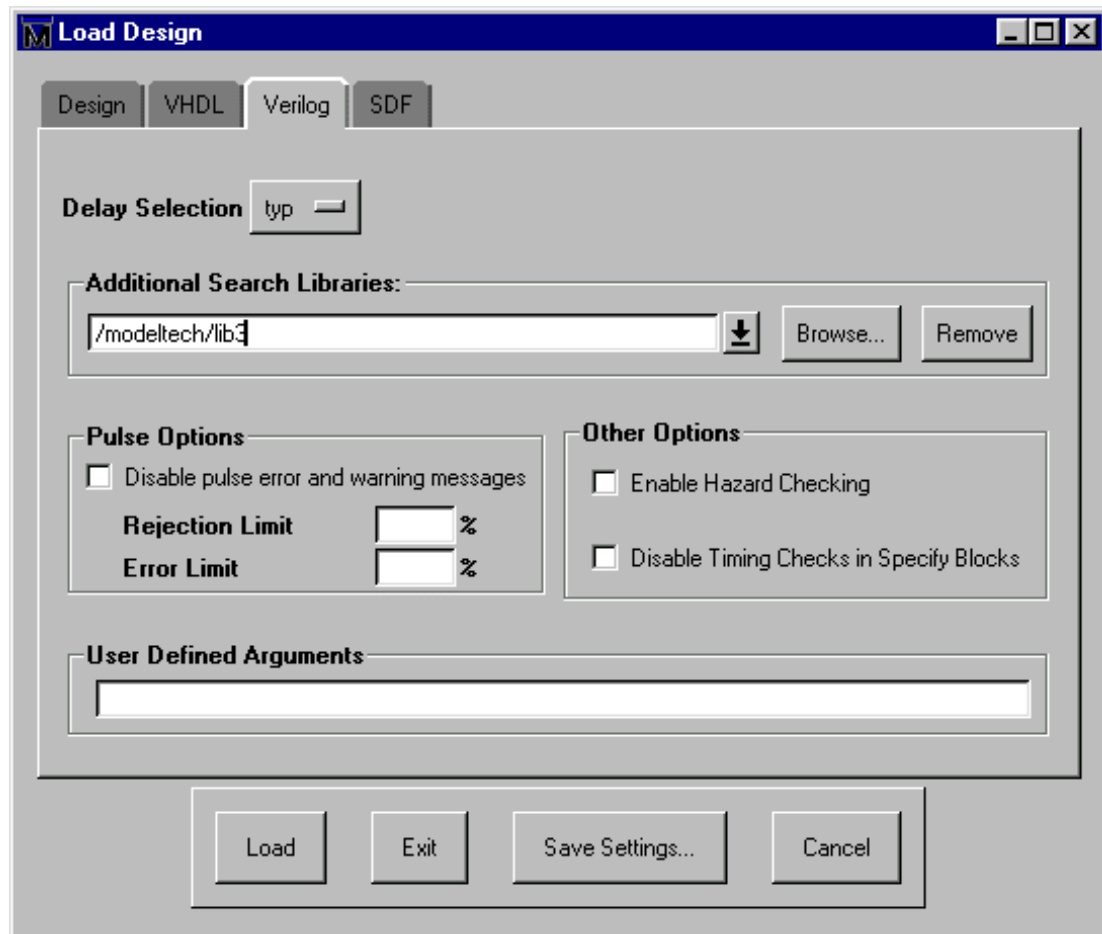
- **Disable Timing Checks** (+notimingchecks)
Disables timing checks generated by VITAL models.
- **Use Vital 2.2b SDF Mapping** (-vital2.2b)
Selects SDF mapping for VITAL 2.2b (default is Vital95).
- **Disable Glitch Generation** (-noglitch)
Disables VITAL glitch generation.

TEXTIO files

- **STD_INPUT** (-std_input <filename>)
Specifies the file to use for the VHDL textio STD_INPUT file. Use the **Browse** button to locate a file within your directories.

- **STD_OUTPUT** (-std_output <filename>)
Specifies the file to use for the VHDL textio STD_OUTPUT file. Use the **Browse** button to locate a file within your directories.

Verilog settings page



The **Verilog** page includes these options:

- **Delay Selection** (+mindelays | +typdelays | +maxdelays)
Use the drop-down menu to select timing for min:typ:max expressions.
Also see: "[Timing check disabling](#)" (4-54).
- **Additional Search Libraries** (-L <library_name>)
Specifies one or more libraries to search for the design unit(s) you wish to simulate. Type in a library name or use the **Browse** button to locate a library within your directories. All specified libraries are added to the drop-down list; remove the currently selected library from the list with the **Remove** button. Make certain your selection is a valid ModelSim library - it must include an *_info* file and must have been created from ModelSim's **vlib** command (CR-198).

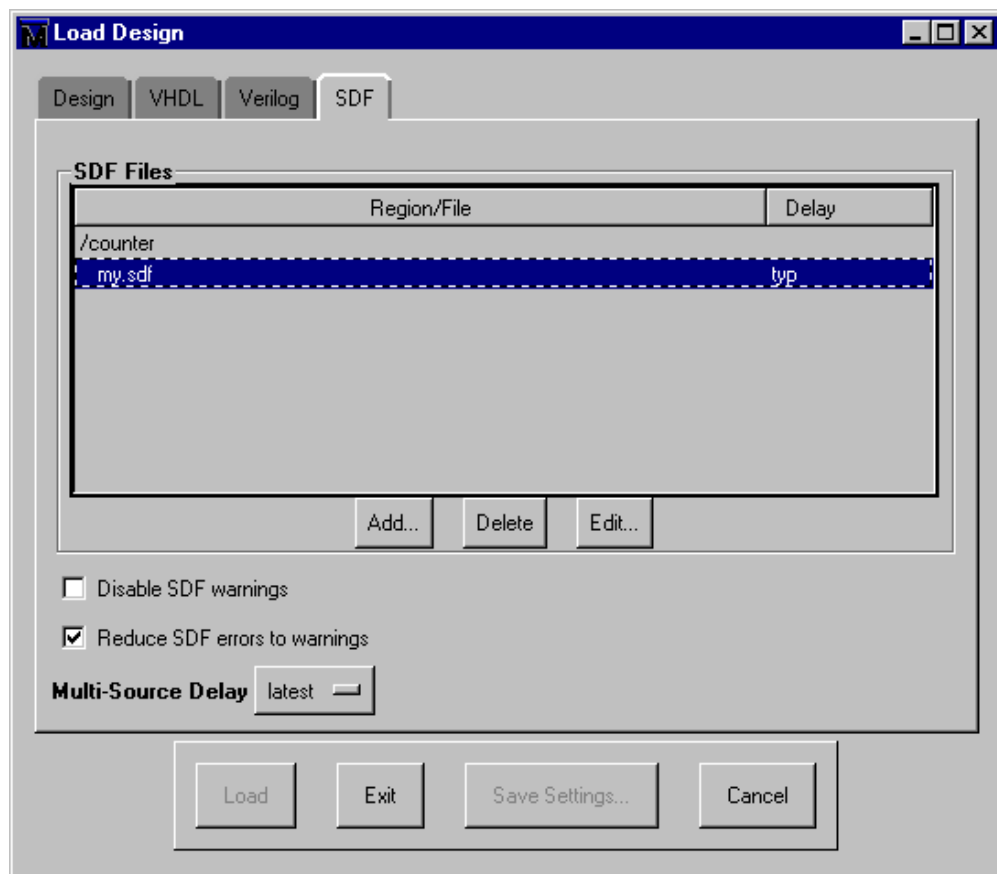
Pulse Options

- **Disable pulse error and warning messages** (+no_pulse_msg)
Disables path pulse error warning messages.
- **Rejection Limit** (+pulse_r/<percent>)
Sets module path pulse rejection limit as percentage of path delay.
- **Error Limit** (+pulse_e/<percent>)
Sets module path pulse error limit as percentage of path delay.

Other Options

- **Enable Hazard Checking** (-hazards)
Enables hazard checking in Verilog modules.
- **Disable Timing Checks in Specify Blocks** (+notimingchecks)
Disables the timing check system tasks (\$setup, \$hold,...) in specify blocks.
- **User Defined Arguments** (+<plusarg>)
Arguments are preceded with "+", making them accessible by the Verilog PLI routine **mc_scan_plusargs**. The values specified in this field must have a "+" preceding them or VSIM may incorrectly parse them.

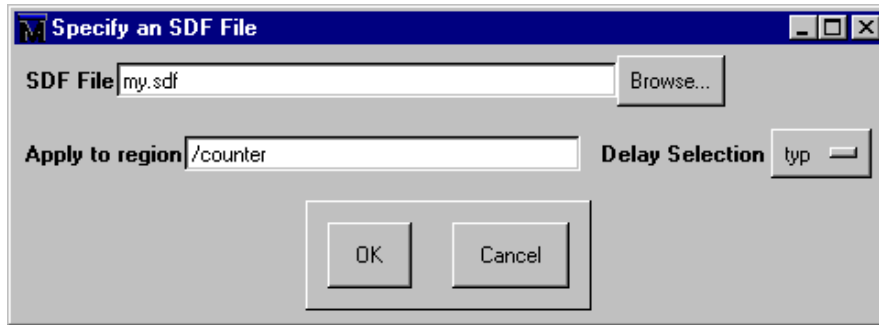
SDF settings page



The **SDF** (Standard Delay Format) page includes these options:

SDF Files

The **Add** button opens a dialog box that allows you to specify the SDF files to load for the current simulation; files are then added to the **Region/File** list. You may also select a file on the listing to **Delete** or **Edit** (opens the dialog box below).



From the **Specify an SDF File** dialog box you can set the following options.

- **SDF file** ([<region>] = <sdf_filename>)
Specifies the SDF file to use for annotation. Use the **Browse** button to locate a file within your directories.
- **Apply to region**
Specifies the design region to use with the selected SDF options.
- **Delay Selection** (-sdfmin | -sdftyp | -sdfmax)
Drop-down menu selects delay timing (min, typ or max) to be used from the specified SDF file. See also, ["Specifying SDF files for simulation"](#) (11-282).
The **OK** button places the specified SDF file and delay on the **Region/File** list; **Cancel** dismisses the dialog box without changes.

and

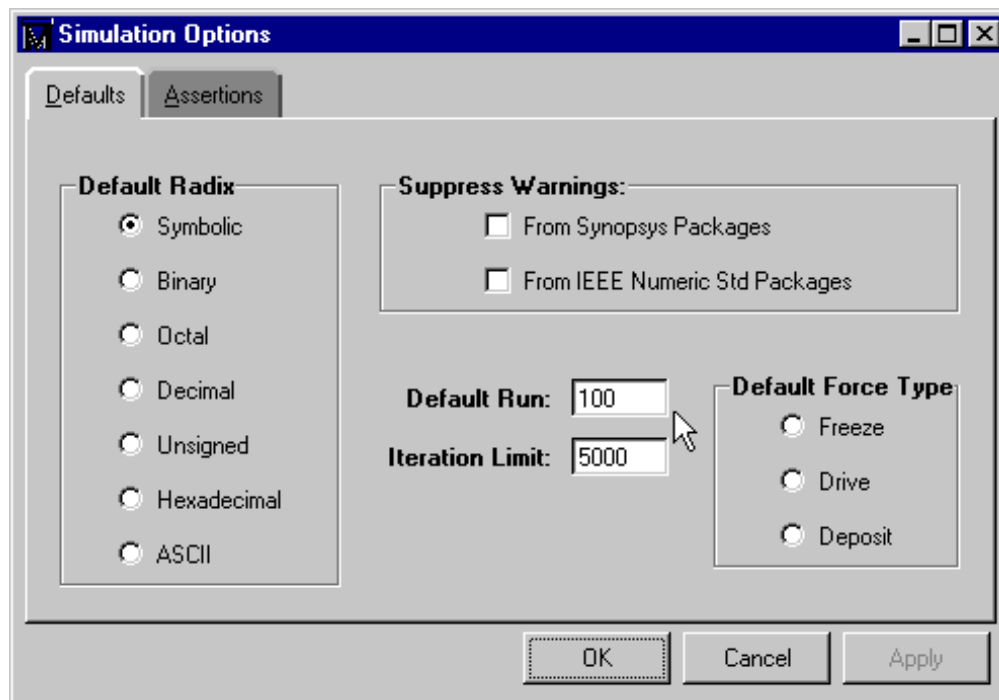
- **Disable warnings from SDF reader** (-sdfnowarn)
Select to disable warnings from the SDF reader.
- **Reduce SDF errors to warnings** (-sdfnoerror)
Change SDF errors to warnings so the simulation can continue.
- **Multi-Source Delay** (-multisource_delay <sdf_option>)
Drop-down menu allows selection of **max**, **min** or **latest** delay. Controls how multiple PORT or INTERCONNECT constructs that terminate at the same port are handled. By default, the Module Input Port Delay (MIPD) is set to the **max** value encountered in the SDF file. Alternatively, you may choose the **min** or **latest** of the values.

Setting default simulation options

Use the **Options > Simulation...** menu selection to bring up the **Simulation Options** dialog box shown below. Options you may set for the current simulation include: default radix, default force type, default run length, iteration limit, warning suppression, and break on assertion specifications. **OK** accepts the changes made and closes the dialog box. **Apply** makes the changes with the dialog box open so you can test your settings. **Cancel** closes the dialog box and makes no changes. The options found on each page are detailed below.

Note: Changes made in the **Simulation Options** dialog box are the default for the current simulation only. Options can be saved as the default for future simulations by editing the simulator control variables in the *modelsim.ini* file; the variables to edit are noted in the text below. You can use the [notepad](#) (CR-104) to edit the variables in *modelsim.ini* if you wish. See also, ["Projects and system initialization"](#) (3-43) for more information.

Default settings page



The **Default** page includes these options:

- **Default Radix**

Sets the default radix for the current simulation run. You can also use the **radix** (CR-129) command to set the same temporary default. A permanent default can be set by editing the **DefaultRadix** (B-399) in the *modelsim.ini* file. The chosen radix is used for all commands (**force** (CR-87), **examine** (CR-81), **change** (CR-43) are examples) and for displayed values in the Signals, Variables, Dataflow, List, and Wave windows.

- **Default Force Type**

Selects the default force type for the current simulation. Edit the **DefaultForceKind** (B-398) in the *modelsim.ini* to set a permanent default.

- **Suppress Warnings**

Selecting **From IEEE Numeric Std Packages** suppresses warnings generated within the accelerated numeric_std and numeric_bit packages. Edit the **NumericStdNoWarnings** (B-400) in the *modelsim.ini* to set a permanent default.

Selecting **From Synopsys Packages** suppresses warnings generated within the accelerated Synopsys std_arith packages. The permanent default can be set in the *modelsim.ini* file with the **StdArithNoWarnings** (B-400).

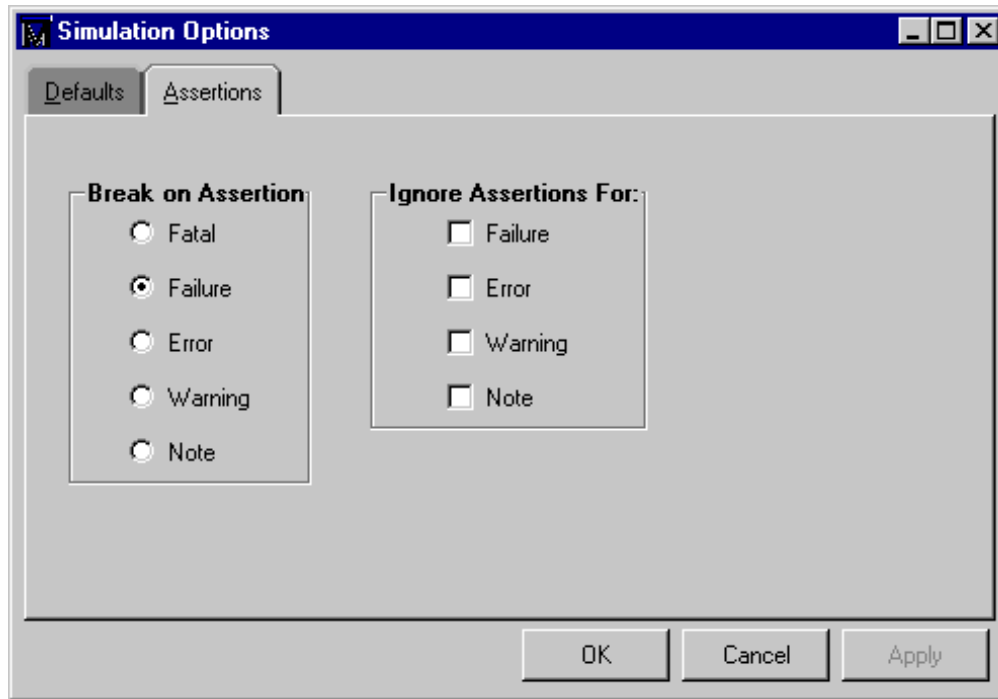
- **Default Run**

Sets the default run length for the current simulation. A permanent default can be set in the *modelsim.ini* file with the **RunLength** (B-400).

- **Iteration Limit**

Sets a limit on the number of deltas within the same simulation time unit to prevent infinite looping. A permanent iteration limit default can be set in the *modelsim.ini* file with the **IterationLimit** (B-399).

Assertion settings page



The **Assertions** page includes these options:

- **Break on Assertion**

Selects the assertion severity that will stop simulation. Edit the [BreakOnAssertion](#) (B-398) in the *modelsim.ini* to set a permanent default.

- **Ignore Assertions For**

Selects the assertion type to ignore for the current simulation. Multiple selections are possible. Edit the [IgnoreFailure](#), [IgnoreError](#), [IgnoreWarning](#), or [IgnoreNote](#) (B-399) variables in the *modelsim.ini* to set a permanent default.

When an assertion type is ignored, no message will be printed, nor will the simulation halt (even if break on assertion is set for that type).

Note: Assertions that appear within an instantiation or configuration port map clause or resolution function will not stop the simulation regardless of the severity level of the assertion.

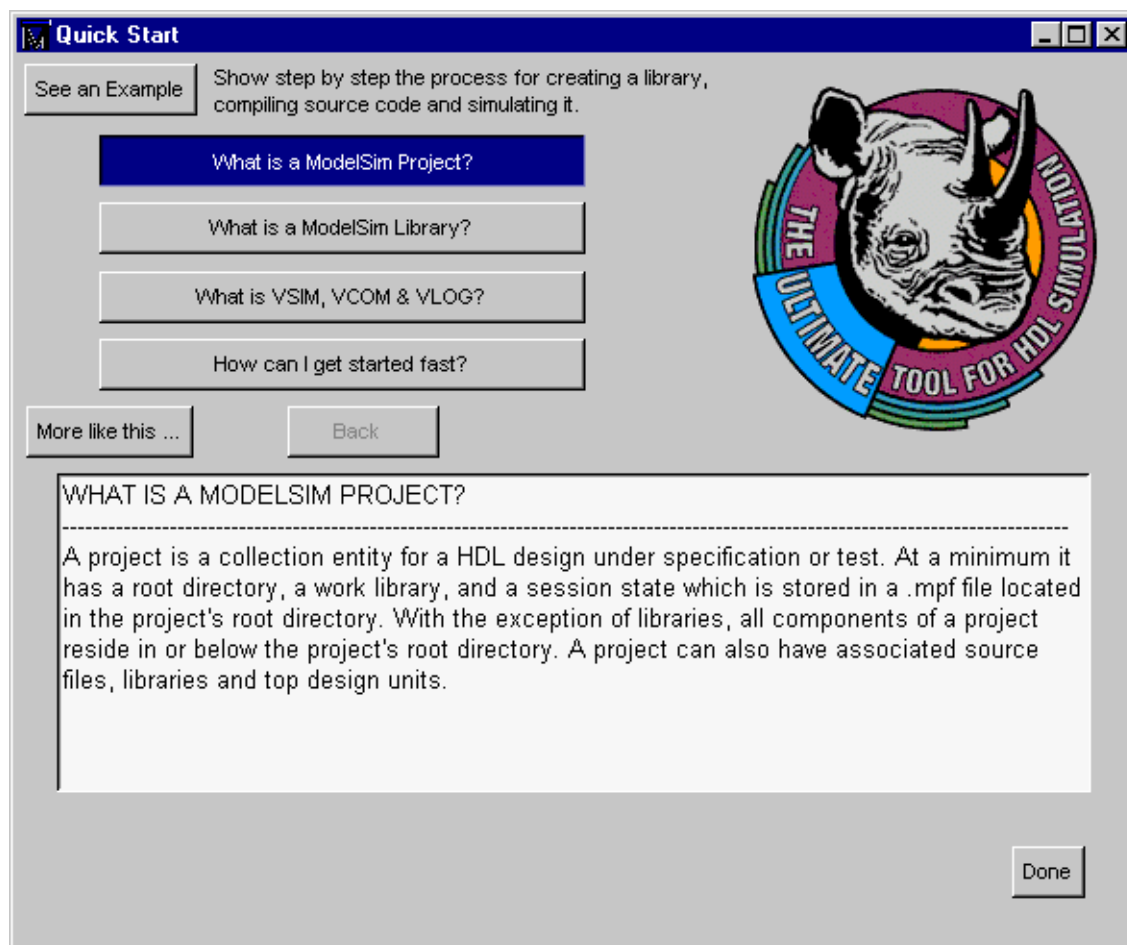
ModelSim tools

Several tools are available from ModelSim menus. The menu selections to locate the tool are below the tool name. Follow the links for more information on each tool.

- ["Quick Start"](#) (10-264)
Main: Help > QuickStart Menu
Explains the process of setting up a project, compiling project source files, simulating a project and modifying it; also includes three step by step simulation examples. Quick Start is also available from the ModelSim Welcome screen.
- ["The Button Adder"](#) (10-265)
Main: Window > Customize
Allows you to add a temporary function button or tool bar to any window.
- ["The Macro Helper"](#) (10-266)
Main: Macro > Macro Helper
Creates macros by recording mouse movements and key strokes. UNIX only.
- ["The Tcl Debugger"](#) (10-267)
Main: Macro > Tcl Debugger
Helps you debug your Tcl procedures.
- ["The GUI Expression Builder"](#) (10-272)
List or Wave: Edit > Search > Search for Expression > Expression Builder
Helps you build logical expressions for use in Wave and List window searches and several simulator commands. For expression format syntax see ["GUI_expression_format"](#) (CR-250).
- ["The FPGA Library Manager"](#) (10-274)
Main: Design > FPGA Library Manager
Helps you build and install libraries provided by FPGA vendors for use within ModelSim.

Quick Start

The ModelSim Quick Start guide is available by clicking the **Quick Start** button in the Welcome to ModelSim window or by selecting **Help > Quick Start Menu** from the Main window. The Quick Start guide includes three examples (select the See an Example button) that shows the process for creating a project, creating a library, compiling source code and simulating the code.



Use the Quick Start guide to find online answers to the following questions:

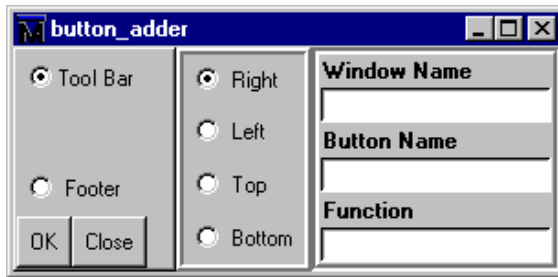
- What is a ModelSim Project?
- What is a ModelSim Library?
- What is VSIM, VCOM & VLOG?
- How can I get started fast?
- How do I create a project in ModelSim?
- How do I specify design components of a project?
- How do I compile my project?
- How do I simulate my project?
- How do I stop working on a project?
- How do I modify a project?
- When do project setting changes take effect?
- What does enabling auto update of project settings mean?
- How do I use a project's settings with the command line tools?

The Button Adder

The ModelSim Button Adder creates a single button, or a combined button and tool bar in any currently opened VSIM window. The button exists until you close the window. (See [Buttons the easy way](#) (10-279).)

Note: When a button is created with the Button Adder, the commands that created the button are echoed in the transcript. The transcript can then be saved and used as a DO file, allowing you to reuse the commands to recreate the button from a startup DO file. See ["Using a startup file"](#) (B-404) for additional information.

Invoke the Button Adder from any ModelSim window menu: **Window > Customize.**



You have the following options for adding a button:

- **Window Name** is the name of the VSIM window to which you wish to add the button.
- **Button Name** is the button's label.
- **Function** can be any command or macro you might execute from the VSIM command line. For example, you might want to add a

Run or **Step** button to the Wave window.

Locate the button within the window with these selections:

- **Tool Bar** places the button on a new tool bar.
- **Footer** adds the button to the Main window's status bar.

Justify the button within the menu bar/tool bar with these selections:

- **Right** places the button on the right side of the menu/tool bar.
- **Left** adds the button on the left side of the menu/tool bar.
- **Top** places the button at the top/center of the menu bar or tool bar.
- **Bottom** places the button at the bottom/center of the menu bar or tool bar.

The Macro Helper

This tool is available for UNIX only.

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-110) 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, VSIM **run** (CR-139) commands cannot be recorded with the Macro Helper but can be invoked as part of a complex macro.



Access the Macro Helper with this Main window menu selection: **Macro > Macro Helper**.

- **Record a macro**
by typing a new macro file name into the field provided, then press **Record**. Use the **Pause** and **Stop** buttons as shown in the table below.
- **Play a macro**
by entering the file name of a Macro Helper file into the field and pressing **Play**.

Files created by the Macro Helper can be viewed with the [notepad](#) (CR-104).

Button	Description
Record/Stop	Record begins recording and toggles to Stop once a recording begins
Insert Pause	inserts a .5 second pause into the macro file; press the button more than once to add more pause time; the pause time can subsequently be edited in the macro file
Play	plays the Macro Helper file specified in the file name field

See the [macro_option](#) command (CR-97) 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

TDebug is installed with *ModelSim* and is configured to run from the **Macro > Tcl Debugger** menu selection. 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

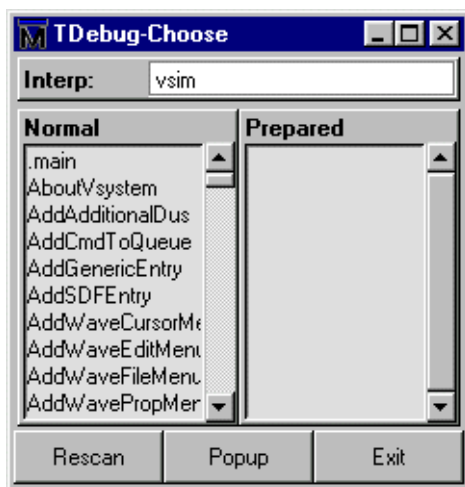
TDebug works by parsing and redefining Tcl/Tk-procedures, inserting calls to ``td_eval'` at certain points, which takes care of the display, stepping, breakpoints, variables etc. The advantages are that TDebug knows which statement in what procedure is currently being executed and can give visual feedback 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

Open the TDebug chooser with the **Macro > Tcl Debugger** menu selection from the Main ModelSim window.



The TDebug chooser has three parts. At the top the current interpreter, *vsim.op_*, is shown. In the main section there are two list boxes. All currently defined procedures are shown in the left list box. By clicking the left mouse button on a procedure name, the procedure gets prepared for debugging and its name is moved to the right list box. Clicking a name in the right list box returns a procedure to its normal state.

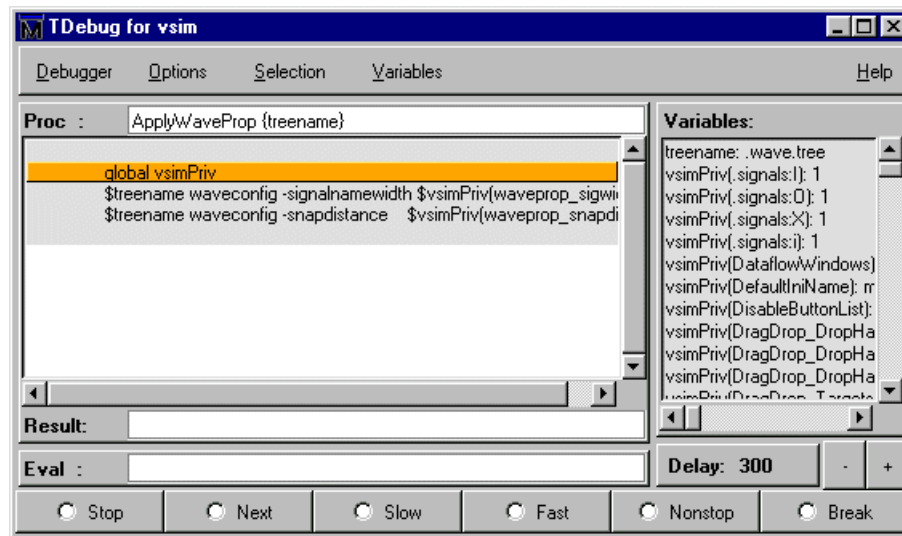
Press the right mouse button on a procedure in either list box to get its program code displayed in the main debugger window.

The three buttons at the bottom let you force a **Rescan** of the available procedures, **Popup** the debugger window or **Exit** TDebug. Exiting from

TDebug doesn't terminate ModelSim, it merely detaches from *vsim.op_*, restoring all prepared procedures to their unmodified state.

The Debugger

Select the **Popup** button in the Chooser to open the debugger window.



The debugger window is divided into the main region with the name of the current procedure (**Proc**), a listing in which the expression just executed is highlighted, the **Result** of this execution and the currently available **Variables** and their values, an entry to **Eval** expressions in the context of the current procedure and some button controls for the state of the debugger.

A procedure listing displayed in the main region will have a darker background on all lines that have been prepared. You can prepare or restore additional lines by selecting a region (<Button-1>, standard selection) and choosing **Selection > Prepare Proc** or **Selection > Restore Proc** from the debugger menu (or by pressing ^P or ^R).

When using 'Prepare' and 'Restore', try to be smart about what you intend to do. If you select just a single word (plus some optional white space) it will be interpreted as the name of a procedure to prepare or restore. Otherwise, if the selection is owned by the listing, the corresponding lines will be used.

Be careful with partial prepare or restore! If you prepare random lines inside a 'switch' or 'bind' expression, you may get surprising results on execution, because the parser doesn't know about the surrounding expression and can't try to prevent problems.

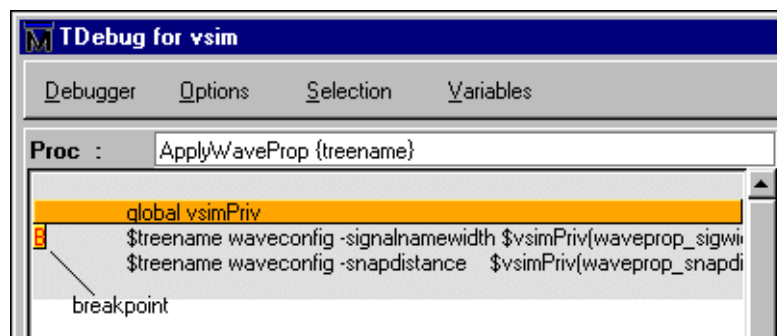
There are seven possible debugger states, one for each button and an 'idle' or 'waiting' state when no button is active. The button-activated states are:

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

Closing the debugger doesn't quit it, it only does 'wm withdraw'. The debugger window will pop up the next time a prepared procedure is called. Make sure you close the debugger with **Debugger > Close**.

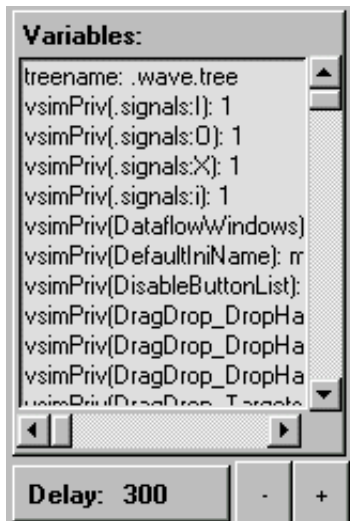
Breakpoints

To set/unset a breakpoint, double-click inside the listing. The breakpoint will be set at the innermost available expression that contains the position of the click. There's no support for conditional or counted breakpoints.



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 at 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.



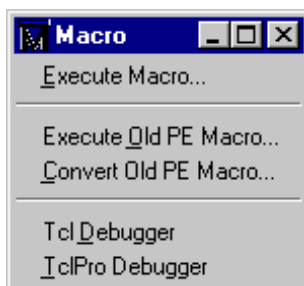
Try entering the line 'global td_priv' and watch the **Variables** box (with global and array variables enabled of course).

Configuration

You can customize TDebug by setting up a file named .tdebugrc in your home directory. See the TDebug README at **Help > Technotes > tdebug** more information on the configuration of TDebug.

TclPro Debugger

The Macro menu in the Main window contains a selection for the TclPro Debugger from Scriptics Corporation. This debugger can be acquired from Scriptics at their web site: www.scriptics.com. Once acquired, do the following steps to use the TclPro Debugger:



- 1 Launch TclPro Debugger
- 2 Launch ModelSim
- 3 From the Main window, select **Macro > TclPro Debugger**

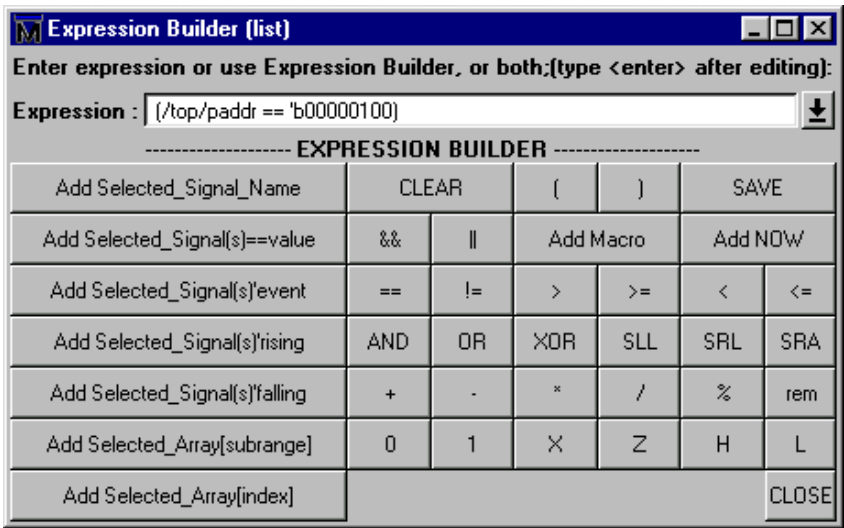
This will connect ModelSim to the Scriptics TclPro Debugger.

The GUI Expression Builder

The GUI Expression Builder is a feature of the Wave and List Signal Search dialog boxes, and the List trigger properties dialog box. It aids in building a search expression that follows the "GUI_expression_format" (CR-250).

To locate the Builder:

- select **Edit > Search** in either the List or Wave window
- select the **Search for Expression** option in the resulting dialog box
- select the **Expression Builder** button



The Expression Builder dialog box provides an array of buttons that help you build a GUI expression. For instance, rather than typing in a signal name, you can select the signal in the associated Wave or List window and press Add Selected_Signal_Name in the Expression Builder. The result will be the full signal name added to the expression field. All Expression Builder buttons correspond to the "Expression syntax" (CR-254).

To search for when a signal reaches a particular value

Select the signal in the Wave window and press this Expression Builder button:

Add Selected_Signal(s) ==value.

This will enter a subexpression consisting of (<signal_name> == <current_value>). You may then edit the value string to be the value of your choice. After editing type <enter> to save the change.

To evaluate only on clock edges

Press the **&&** button to AND this condition with the rest of the expression, then select the clock in the Wave window and press **Add Selected_Signal(s)'rising**. You may also select the falling edge or both edges.

To reference array subranges

The Add Selected_Array[subrange] and Add Selected_Array[index] buttons assist in referencing arrays by bringing up the declared range of the array. You may then edit the range to select a subrange or index.

Operators

Other buttons will add operators of various kinds (see ["Expression syntax"](#) (CR-254)), or you can type them in.

To hand edit the expression

Click with left mouse button in the expression entry box and delete or add characters. Type <enter> to accept the change.

To save the expression as a Tcl variable

The Save button will allow you to save the expression to a Tcl variable. It also saves the expression in the drop-down entry box.

See ["Setting up a List trigger with Expression Builder"](#) (E-456) for an additional Expression builder example.

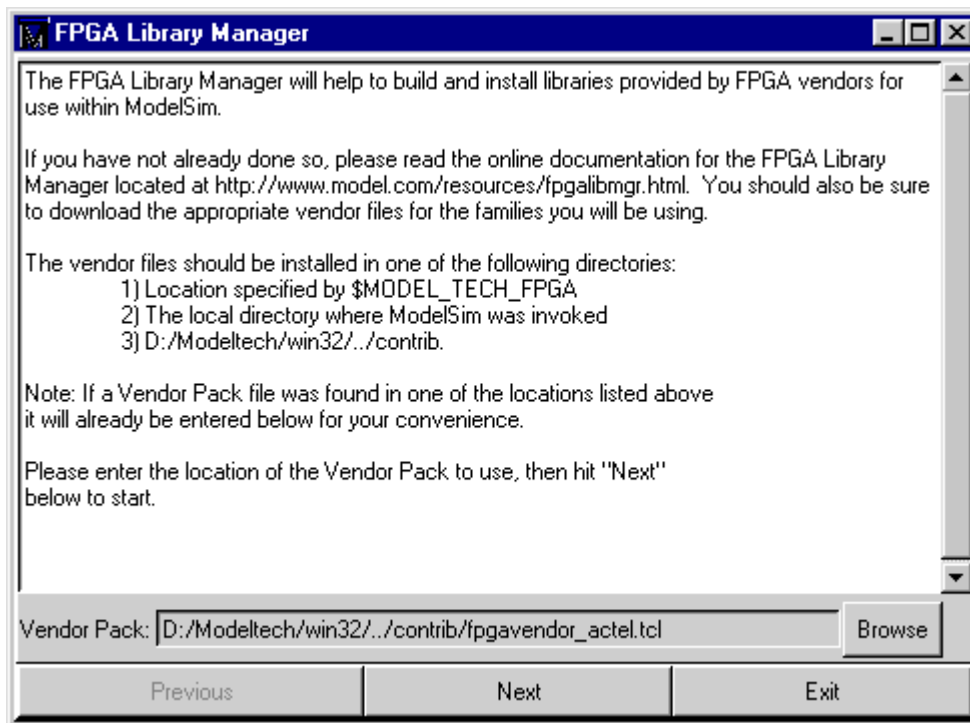
The FPGA Library Manager

The FPGA Library Manager helps you build and install libraries provided by FPGA vendors for use within ModelSim. The Library Manager is accessed from ModelSim's Main window with the **Design > FPGA Library Manager** menu selection.

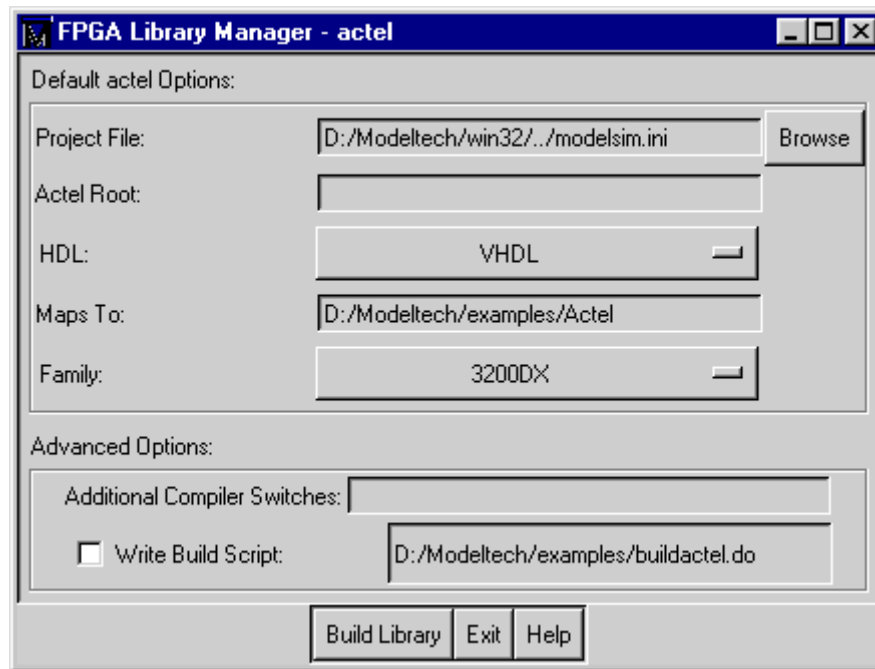
To prepare the Library Manager for use, you will need to download the appropriate vendor files for the design families you will be using, and install the files in one of the following directories:

- location specified by the [MODEL_TECH_FPGA](#) (B-391) environment variable
- the local directory where ModelSim was invoked
- Modeltech/win32/./contrib.

The initial Library Manager dialog box provides a brief introduction and allows you to specify the **Vendor Pack** to use. (If a vendor pack is found it will be entered for you.)



Once you have specified the vendor pack, click **Next** to open a vendor-specific dialog box similar to the one below.



The options in this dialog box will vary depending on the FPGA vendor, for example, in addition to the options shown above, the Xilinx version of the Library Manager includes timing, GSR, testbench, and design instance fields.

Select the **Help** button for a description of all fields included in the dialog box.

Options common to all versions of the FPGA Library Manager include:

Option	Description
Project File	When a library is mapped using the vmap command (CR-207), it defines a relationship between a logical library name and a directory by modifying the appropriate <i>modelsim.ini</i> file. This field allows you to define the INI file that will be modified to reflect this mapping.
<vendor> Root	This is the location of the <vendor> installation. If the <vendor_variable> environment variable is set, this entry will contain that value by default. You can override the default value with this entry.
HDL	Select the appropriate HDL being used. Choices are VHDL or Verilog.
Maps To	Enter the location where the FPGA library will be built. This should be a fully qualified path to a directory. It should not already exist and the parent directory should be writeable. Please see the description of the vlib command (CR-198) for additional information.
Family	Select the <vendor> family to be built. Current supported selections are: <list of family options>.
Additional Compiler Switches	Add any additional switches for the compiler here. Please see the vdcl command (CR-175) and the vlog command (CR-199) for additional information on VHDL and Verilog compiler switches.
Write Build Script	If you select this option and enter a valid file name into the entry box, a script will be written with the commands used to build the library. This script can be run to rebuild the library outside the FPGA Library Manager.

When the Library Manager options are set, click **Build Library** to compile the library. **Exit** will return you to the original Library Manager dialog box where you can select another vendor pack to compile, or exit the Library Manager.

Graphic interface commands

The following commands provide control and feedback during simulation as well as the ability to edit, and add menus and buttons to the interface. Only brief descriptions are provided here; for more information and command syntax see the [ModelSim EE/SE Command Reference](#).

Window control and feedback commands	Description
batch_mode (CR-38)	returns a 1 if VSIM is operating in batch mode, otherwise returns a 0; it is typically used as a condition in an if statement
configure (CR-54)	invokes the List or Wave widget configure command for the current default List or Wave window
down up (CR-70)	moves the active marker in the List window down or up to the next or previous transition on the selected signal that matches the specifications
getactivecursortime (CR-90)	gets the time of the active cursor in the Wave window
getactivemarkertime (CR-91)	gets the time of the active marker in the List window
.main clear (CR-99)	clears the Main window transcript
notepad (CR-104)	a simple text editor; used to view and edit ascii files or create new files
play (CR-110)	UNIX only - replays a sequence of keyboard and mouse actions, which were previously saved to a file with the record command (CR-130)
property list (CR-123)	change properties of an HDL item in the List window display
property wave (CR-124)	change properties of an HDL item in the waveform or signal name display in the Wave window
record (CR-130)	UNIX only - starts recording a replayable trace of all keyboard and mouse actions
right left (CR-136)	searches for signal transitions or values in the specified Wave window

Window control and feedback commands	Description
search and next (CR-141)	search the specified window for one or more items matching the specified pattern(s)
seetime (CR-146)	scrolls the List or Wave window to make the specified time visible
transcribe (CR-157)	displays a command in the Main window, then executes the command
.wave.tree zoomfull (CR-221)	zoom waveform display to view from 0 to the current simulation time
.wave.tree zoomrange (CR-222)	zoom waveform display to view from one specified time to another
write preferences (CR-233)	saves the current GUI preference settings to a Tcl preference file

Window menu and button commands	Description
add button (CR-20)	adds a user-defined button to the Main window button bar
add_menu (CR-26)	adds a menu to the menu bar of the specified window
add_menub (CR-28)	creates a checkbox within the specified menu of the specified window
add_menuitem (CR-30)	creates a menu item within the specified menu of the specified window
add_separator (CR-31)	adds a separator as the next item in the specified menu path in the specified window
add_submenu (CR-32)	creates a cascading submenu within the specified menu_path of the specified window
change_menu_cmd (CR-44)	changes the command to be executed for a specified menu item label, in the specified menu, in the specified window

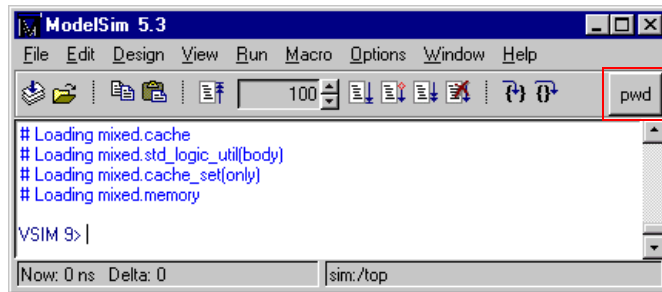
Window menu and button commands	Description
disable_menu (CR-65)	disables the specified menu within the specified window; useful if you want to restrict access to a group of ModelSim features
disable_menuitem (CR-66)	disables a specified menu item within the specified menu_path of the specified window; useful if you want to restrict access to a specific ModelSim feature
enable_menu (CR-78)	enables a previously-disabled menu
enable_menuitem (CR-79)	enables a previously-disabled menu item

Customizing the interface

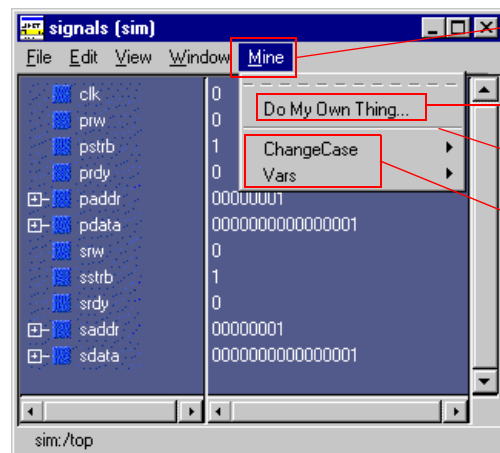
Try customizing ModelSim's interface yourself; use the command examples for [add button](#) (CR-20) and [add_menu](#) (CR-26) to add a button to the Main window, and a new menu to the [Signals window](#) (10-192). Results of the button and menu commands are shown below

Buttons the easy way

"[The Button Adder](#)" (10-265) tool makes adding buttons easy. Use the **Window > Customize** menu selection in any window to access the Button Adder. Buttons you create are not permanent; they exist only during the current session. To reuse a button, save the Main transcript (**File > Save Main as**) after the button is created. Edit the file to contain only button-creation commands, then pass the filename as an argument to the [do](#) command (CR-67) to recreate the button.



- The **pwd** button was added to the Main window with the **add_button** command (CR-20). Buttons can be added to the menu bar and status bar as well.

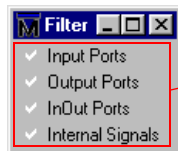


- The Mine menu was added to the Signals window with the **add_menu** command (CR-26).

- The Do My Own Thing menu item was added with the **add_menuitem** command (CR-30)

- The menu separator was added with the **add_separator** command (CR-31).

- The ChangeCase and Vars submenus were added with the **add_submenu** command (CR-32).



- You can also add a menu checkbox (like those in this menu tearoff) with the **add_menucheckbox** command (CR-28).

11 - Standard Delay Format (SDF) Timing Annotation

Chapter contents

Specifying SDF files for simulation	282
Instance specification	282
SDF specification with the GUI	283
Errors and warnings.	283
VHDL VITAL SDF	284
SDF to VHDL generic matching	284
Resolving errors.	285
Verilog SDF	286
The \$sdf_annotate system task	286
SDF to Verilog construct matching.	287
Optional edge specifications	291
Optional conditions	292
Rounded timing values	292
SDF for Mixed VHDL and Verilog Designs	293
Interconnect delays	293
Troubleshooting	294
Obtaining the SDF specification.	296

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 may 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 vendor's also provide instructions on using their SDF files and libraries with ModelSim.

The SDF specification was originally created for Verilog designs, but it has also been adopted for VHDL VITAL designs. In general, the designer does not need to be familiar with the details of the SDF specification because the cell library provider has already supplied tools that create SDF files that match their libraries.

Note: In order to conserve disk space ModelSim will read sdf files that were compressed using the standard

unix/gnu file compression algorithm. The filename must end with the suffix ".Z" for the decompress to work.

Specifying SDF files for simulation

ModelSim supports SDF versions 1.0 through 3.0. The simulator's built-in SDF annotator automatically adjusts to the version of the file. Use the following **vsim** (CR-208) 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 that the SDF is applied to. 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:

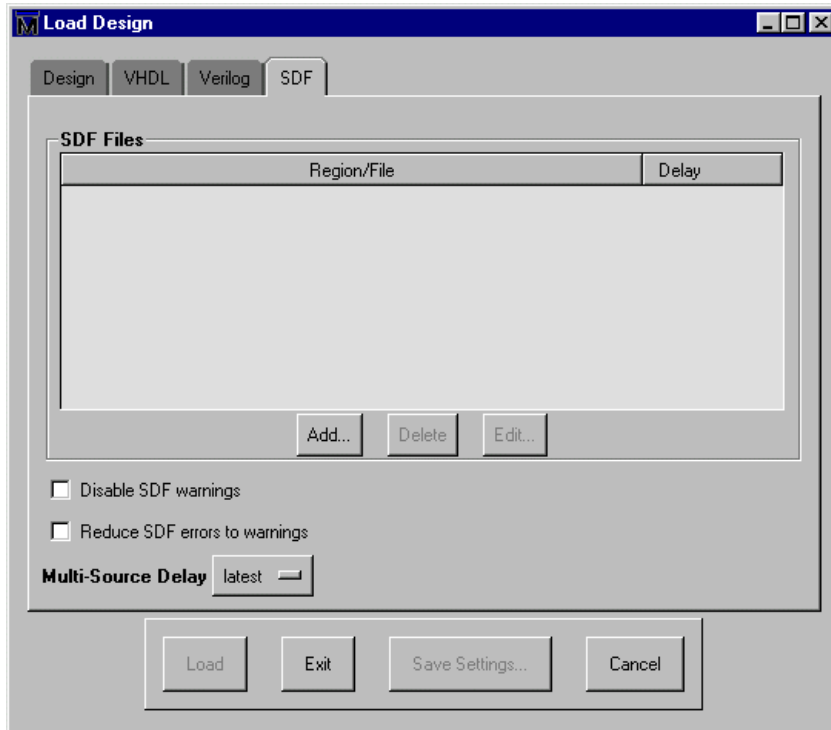
```
vsim -sdfmax /testbench/u1=myasic.sdf testbench
```

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 may 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 may specify SDF files in the **Load Design** dialog box under the SDF tab.



This dialog box is presented if you invoke the simulator without any arguments or if you select "Load New Design..." under the simulator's file menu. For Verilog designs, you may also specify SDF files by using the **\$sdf_annotate** system task. See ["The \\$sdf_annotate system task"](#) (11-286) 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-208) 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** page from the **Load Design** dialog box (shown above). Select **Disable SDF warnings** (-sdfnowarn, or +nosdfwarn) to disable warnings, or select **Reduce SDF errors to warnings** (-sdfnoerror) to change errors to warnings.

See ["Troubleshooting"](#) (11-294) 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 is provided to help understand simulator error messages in case of user error or in case the vendor's SDF does not match the VITAL cells. For additional VITAL specification information see ["Obtaining the VITAL specification and source code"](#) (4-59).

SDF to VHDL generic matching

An SDF file contains delay and timing constraint data for cell instances in the design. The annotator must locate the cell instances and the placeholders (VHDL generics) for the timing data. Each type of SDF timing construct is mapped to the name of a generic as specified by the VITAL modeling specification. The annotator locates the generic and updates it with the timing value from the SDF file. It is an error if the annotator fails to find the cell instance or the named generic. The following are examples of SDF constructs and their associated generic names:

SDF construct	Matching VHDL generic name
(IOPATH a y (3))	tpd_a_y
(IOPATH (posedge clk) q (1) (2))	tpd_clk_q_posedge
(INTERCONNECT u1/y u2/a (5))	tipd_a
(SETUP d (posedge clk) (5))	tsetup_d_clk_noedge_posedge
(HOLD (negedge d) (posedge clk) (5))	thold_d_clk_negedge_posedge
(SETUPHOLD d clk (5) (5))	tsetup_d_clk & thold_d_clk
(WIDTH (COND (reset==1'b0) clk) (5))	tpw_clk_reset_eq_0

Resolving errors

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

```
ERROR: 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 directly or use the simulator's user interface to locate the information:

- Open the [Structure window](#) (10-206) and navigate to the instance named in the error message (you could try the Edit > Find menu option). Alternatively, use the [enable_menuitem](#) command (CR-79) to select the instance. For example:

```
env /testbench/dut/u1
```

- Open the [Process window](#) (10-189) and select the "In Region" mode (the default mode is "Active").
- Select a process in the Process window (usually the process named "vitalbehavior").
- Open the [Variables window](#) (10-209) to see all of the generics and their current values.

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-208) with the **-vital2.2b** option:

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

For more information on resolving errors see ["Troubleshooting"](#) (11-294).

Verilog SDF

Verilog designs may 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 that 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  
    ([ "<sdf_file>" ], [ <instance> ], [ "<config_file>" ], [ "<log_file>" ],  
    [ "<mtm_spec>" ], [ "<scale_factor>" ], [ "<scale_type>" ] );
```

Arguments

"<sdf_file>"

String that specifies the SDF file. Required.

<instance>

Hierarchical name of the instance to be annotated. Optional. Defaults to the instance where the \$sdf_annotate call is made.

"<config_file>"

String that specifies the configuration file. Optional. Currently not supported, this argument is ignored.

"<log_file>"

String that specifies the logfile. Optional. Currently not supported, this argument is ignored.

"<mtm_spec>"

String that specifies 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 may 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 it applies to:

```
$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 may have multiple matches, in which case each matching specify statement is updated with the SDF timing value. SDF constructs are matched to Verilog constructs as follows:

IOPATH is matched to specify path delays or primitives:

SDF	Verilog
(IOPATH (posedge clk) q (3) (4))	(posedge clk => q) = 0;
(IOPATH a y (3) (4))	buf u1 (y, a);

The IOPATH construct usually annotates path delays. If the module contains no path delays, then all primitives that drive the specified output port are annotated.

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

SDF	Verilog
(INTERCONNECT u1.y u2.a (5))	input a;
(PORT u2.a (5))	inout a;

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

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

SDF	Verilog
(PATHPULSE a y (5) (10))	(a => y) = 0;
(GLOABLPATHPULSE a y (30) (60))	(a => y) = 0;

If the input and output ports are omitted in the SDF, then all path delays are matched in the cell.

DEVICE is matched to primitives or specify path delays:

SDF	Verilog
(DEVICE y (5))	and u1(y, a, b);
(DEVICE y (5))	(a => y) = 0; (b => y) = 0;

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

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

SDF	Verilog
(SETUP d (posedge clk) (5))	\$setup(d, posedge clk, 0);
(SETUP d (posedge clk) (5))	\$setuphold(posedge clk, d, 0, 0);

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

SDF	Verilog
(HOLD d (posedge clk) (5))	\$hold(posedge clk, d, 0);
(HOLD d (posedge clk) (5))	\$setuphold(posedge clk, d, 0, 0);

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

SDF	Verilog
(SETUPHOLD d (posedge clk) (5) (5))	\$setup(d, posedge clk, 0);
(SETUPHOLD d (posedge clk) (5) (5))	\$hold(posedge clk, d, 0);
(SETUPHOLD d (posedge clk) (5) (5))	\$setuphold(posedge clk, d, 0, 0);

RECOVERY is matched to \$recovery:

SDF	Verilog
(RECOVERY (negedge reset) (posedge clk) (5))	\$recovery(negedge reset, posedge clk, 0);

REMOVAL is matched to \$removal:

SDF	Verilog
(REMOVAL (negedge reset) (posedge clk) (5))	\$removal(negedge reset, posedge clk, 0);

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

SDF	Verilog
(RECREM (negedge reset) (posedge clk) (5) (5))	\$recovery(negedge reset, posedge clk, 0);
(RECREM (negedge reset) (posedge clk) (5) (5))	\$removal(negedge reset, posedge clk, 0);
(RECREM (negedge reset) (posedge clk) (5) (5))	\$recrem(negedge reset, posedge clk, 0);

SKEW is matched to \$skew:

SDF	Verilog
(SKEW (posedge clk1) (posedge clk2) (5))	\$skew(posedge clk1, posedge clk2, 0);

WIDTH is matched to \$width:

SDF	Verilog
(WIDTH (posedge clk) (5))	\$width(posedge clk, 0);

PERIOD is matched to \$period:

SDF	Verilog
(PERIOD (posedge clk) (5))	\$period(posedge clk, 0);

NOCHANGE is matched to \$nochange:

SDF	Verilog
(NOCHANGE (negedge write) addr (5) (5))	\$nochange(negedge write, addr, 0, 0);

Optional edge specifications

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

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

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

SDF	Verilog
(SETUP data (posedge clock) (5))	\$setup(posedge data, posedge clk, 0);
(SETUP data (posedge clock) (5))	\$setup(negedge data, posedge clk, 0);

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

SDF	Verilog
(SETUP (posedge data) (posedge clock) (4))	\$setup(data, posedge clk, 0);
(SETUP (negedge data) (posedge clock) (6))	\$setup(data, posedge clk, 0);

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

Timing check edge specifiers may also use explicit edge transitions instead of posedge and negedge. However, the SDF file is limited to posedge and negedge. The explicit edge specifiers are 01, 0x, 10, 1x, x0, and x1. The set of [01, 0x, x1] is equivalent to posedge, while the set of [10, 1x, x0] is equivalent to negedge. A

match occurs if any of the explicit edges in the specify port match any of the explicit edges implied by the SDF port. For example,

SDF	Verilog
(SETUP data (posedge clock) (5))	\$setup(data, edge[01, 0x] clk, 0);

Optional conditions

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

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

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

SDF	Verilog
(SETUP data (COND (reset!=1) (posedge clock)) (5))	\$setup(data, posedge clk &&& (reset==0), 0);

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

SDF	Verilog
(COND (r1 r2) (IOPATH clk q (5)))	if (r1 r2) (clk => q) = 5; // matches
(COND (r1 r2) (IOPATH clk q (5)))	if (r2 r1) (clk => q) = 5; // does not match

The annotator does not match the second condition above because the order of r1 and r2 are reversed.

Rounded timing values

The SDF **TIMESCALE** construct specifies time units of values in the SDF file. The annotator rounds timing values from the SDF file to the time precision of the module that is annotated. For example, if the SDF TIMESCALE is 1ns and a value of .016 is annotated to a path delay in a module having a time precision of 10ps

(from the timescale directive), then the path delay receives a value of 20ps. The SDF value of 16ps is rounded to 20ps. Interconnect delays are rounded to the time precision of the module that contains the annotated MIPD.

SDF for Mixed VHDL and Verilog Designs

Annotation of a mixed VHDL and Verilog design is very flexible. VHDL VITAL cells and Verilog cells may 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-208) 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. This type of delay is modeled in the receiving device as a delay from an input port to an internal node. In VHDL VITAL this node is explicitly declared, whereas in Verilog it is automatically created by the simulator and is not visible to the user interface.

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.

Since an interconnect delay is modeled as a single delay between an input port and an internal node, there is no convenient way to handle interconnect delays from multiple outputs to a single input. For both VHDL VITAL and Verilog the default is to use the value of the maximum encountered delay in the SDF file. Optionally, you may choose the minimum or latest value of the multiple delays with the [vsim](#) (CR-208) **-multisource_delay** option:

```
-multisource_delay min|max|latest
```

Troubleshooting

Several common mistakes in SDF annotation are outlined below.

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 almost always 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. A common example for both VHDL and Verilog test benches is provided below. For simplicity, the test benches do nothing more than instantiate a model that has no ports.

VHDL testbench

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

Verilog testbench

```
module testbench;  
    myasic dut();  
endmodule
```

The name of the model is *myasic* and the instance label is *dut*. For either testbench, an appropriate simulator invocation might be:

```
vsim -sdfmax /testbench/dut=myasic.sdf testbench
```

Optionally, you may leave off the name of the top-level:

```
vsim -sdfmax /dut=myasic.sdf testbench
```

The important point is to select the instance that the SDF is intended for. If the model is deep within the design hierarchy, an easy way to find the instance name is to first invoke the simulator without SDF options, open the structure window, navigate to the model instance, select it, and enter the **environment** command (CR-80). 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: myasic.sdf:
The design does not have an instance named '/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: myasic.sdf:
Failed to find INSTANCE '/testbench/u1'
ERROR: myasic.sdf:
Failed to find INSTANCE '/testbench/u2'
ERROR: myasic.sdf:
Failed to find INSTANCE '/testbench/u3'
ERROR: myasic.sdf:
Failed to find INSTANCE '/testbench/u4'
ERROR: myasic.sdf:
Failed to find INSTANCE '/testbench/u5'
WARNING: myasic.sdf:
This file is probably applied to the wrong instance.
WARNING: 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: myasic.sdf:
Failed to find any of the 358 instances from this file.
WARNING: 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"](#) (11-285) for specific VHDL VITAL SDF troubleshooting.

Obtaining the SDF specification

SDF specification is available from Open Verilog International:

Lynn Horobin

phone: (408)358-9510

fax: (408)358-3910

email: info@ovi.org

home page: <http://www.ovi.org>

12 - VHDL Foreign Language Interface

Chapter contents

Using the VHDL FLI	298
Using the VHDL FLI with foreign architectures	298
Using the VHDL FLI with foreign subprograms	300
Using checkpoint/restore with the FLI	305
Support for Verilog instances	308
Support for Windows platforms	308
FLI function descriptions	309
Mapping to VHDL data types	328
VHDL FLI examples	330
Compiling and linking FLI applications.	331
Windows NT/95/98 platforms	331
Solaris platform	331
SunOS 4 platform	332
HP700 platform	332
IBM RISC/6000 platform	333
FLI tracing	334
The purpose of tracing files.	334
Invoking a trace.	334

This chapter covers ModelSim's VHDL FLI (Foreign Language Interface), which allows you to replace VHDL architectures and subprogram bodies with code written in C.

Note: The Tcl C interface is included in the FLI; *tcl.h* is found in the *<install_dir>/modeltech/include* directory. Tk and Tix are not included in the FLI because the FLI is in the kernel, not the user interface. You can FTP Tcl from: <http://www.scripts.com>.

Using the VHDL FLI

Using the VHDL FLI with foreign architectures

To use the foreign language interface with C models, you first create and compile an architecture with the FOREIGN attribute. The string value of the attribute is used to specify the name of a C initialization function and the name of an object file to load. When VSIM elaborates the architecture, the initialization function is called. Parameters to the function include a list of ports and a list of generics. See: ["Mapping to VHDL data types"](#) (12-328).

Declaring the FOREIGN attribute

Starting with VHDL93, the FOREIGN language attribute is declared in package STANDARD. With the 1987 version, you need to declare the attribute yourself. You can declare it in a separate package, or you can declare it in the architecture that you are replacing. (This will also work with VHDL93).

The FOREIGN attribute string

The value of the FOREIGN attribute is a string containing three parts. For the following declaration:

```
ATTRIBUTE foreign OF arch_name : ARCHITECTURE IS "app_init app.so; parameter";
```

the attribute string parses this way:

`app_init`

The name of the initialization function for this architecture. This part is required. See ["The C initialization function"](#) (12-299).

`app.so`

The path to the shared object file to load. This part is required. See ["Location of shared object files"](#) (12-299).

`parameter`

A string that is passed to the initialization function. This part is preceded by a semicolon and is optional.

If the initialization function has a leading '+' or '^', the VHDL architecture body will be elaborated in addition to the foreign module. If '+' is used (as in the example below), the VHDL will be elaborated first. If '^' is used, the VHDL will be elaborated after the foreign initialization function is called.

UNIX environment variables can also be used within the string as in this example:

```
ATTRIBUTE foreign OF arch_name : ARCHITECTURE IS "+app_init $CAE/app.so";
```

Location of shared object files

VSIM searches for object files in the following order:

- \$MGC_WD/<so> or ./<so> (If MGC_WD is not set, then it will use ".")
- <so>
- within \$LD_LIBRARY_PATH (\$SHLIB_PATH on HP only)
- \$MGC_HOME/lib/<so>
- \$MODEL_TECH/<so>
- \$MODEL_TECH/./<so>

In the search information above "<so>" refers to the path specified in the FOREIGN attribute string. MGC_WD and MGC_HOME are user-definable environment variables. MODEL_TECH is set by the application to the directory where the **vsim** executable resides.

Note: The *.so* extension will work on all platforms (it is not necessary to use the *.sl* extension on HPs).

The C initialization function

This is the entry point into the foreign C model. The initialization function typically:

- allocates memory to hold variables for the instance
- registers a callback function to free the memory when VSIM is restarted
- saves the handles to the signals in the port list
- creates drivers on the ports that will be driven
- creates one or more processes (a C function that can be called when a signal changes)
- sensitizes each process to a list of signals

The declaration of an initialization function is:

```
init_func(region, param, generics, ports)
    mtiRegionIdT region;
    char *param;
    mtiInterfaceListT *generics;
    mtiInterfaceListT *ports;
```

The function specified in the foreign attribute is called during elaboration. The first parameter is a region ID that can be used to determine the location in the design for this instance. The second parameter is the last part of the string in the foreign attribute. The third parameter is a linked list of the generic values for this instance. The list will be NULL if there are no generics. The last parameter is a linked list of the ports for this instance. The typedef `mtiInterfaceListT` in *mti.h* describes the entries in these lists.

Restrictions on generics

VSIM does not allow you to read RECORD generics through the foreign language interface.

Using the VHDL FLI with foreign subprograms

Declaring a subprogram in VHDL

To call a foreign C subprogram, you must write a VHDL subprogram declaration that has the equivalent VHDL parameters and return type. Then use the FOREIGN attribute to specify which C function and module to load. The syntax of the FOREIGN attribute is almost identical to the syntax used for foreign architectures.

Example

```
procedure in_params(  
    vhdl_integer : IN integer;  
    vhdl_enum    : IN severity_level;  
    vhdl_real    : IN real;  
    vhdl_array   : IN string);  
  
attribute FOREIGN of in_params : procedure is "in_params app.so";
```

You must also write a subprogram body for the subprogram, but it will never be called.

Example

```
procedure in_params(  
    vhdl_integer : IN integer;  
    vhdl_enum    : IN severity_level;  
    vhdl_real    : IN real;  
    vhdl_array   : IN string) is  
begin  
    report "ERROR: foreign subprogram in_params not called";  
end;
```

Matching VHDL parameters with C parameters

Use the table below to match the C parameters in your foreign C subprogram to the VHDL parameters in your VHDL package declaration. The parameters must match in order as well as type.

	Parameters of class CONSTANT OR VARIABLE		Parameters of class SIGNAL
VHDL Type	IN	INOUT/OUT	IN
Integer	int	int *	mtiSignalIdT
Enumeration	int	char *	mtiSignalIdT
Real	double *	double *	mtiSignalIdT
Time	time64 *	time64 *	mtiSignalIdT
Array	mtiVariableIdT	mtiVariableIdT	mtiVariableIdT
File	-- Not supported --		
Record	-- Not supported --		
Access Integer	int	int *	-- Not supported --
Access Enumeration	int	int *	-- Not supported --
Access Real	double *	double *	-- Not supported --
Access Array	mtiVariableIdT	mtiVariableIdT	-- Not supported --
Access File	-- Not supported --		
Access Record	-- Not supported --		

Definitions

Enumeration

Everything that is declared in VHDL as an enumeration with 256 or fewer elements. For example: BIT, BOOLEAN, CHARACTER, STD_LOGIC.

Array

Everything that is declared in VHDL as an array. For example: STRING, BIT_VECTOR, STD_LOGIC_VECTOR. Arrays are not NULL terminated.

Array SIGNAL parameters are passed as an mtiVariableIdT type representing an array of mtiSignalIdT types.

Matching VHDL return types with C return types

Use the table below to match the C return types in your foreign C subprogram to the VHDL return types in your VHDL code.

VHDL Return Type	C Return Type
Integer	int
Enumeration	int
Real	-- Not supported --
Time	-- Not supported --
Array	-- Not supported --
File	-- Not supported --
Record	-- Not supported --
Access	-- Not supported --

C code and VHDL examples

The following examples illustrate the association between C functions and VHDL procedures. The C function is connected to the VHDL procedure through the FOREIGN attribute specification.

C subprogram example

Functions declared in this code, **in_params()** and **out_params()**, have parameters and return types that match the procedures in the subsequent package declaration (**pkg**).

```
#include <stdio.h>
#include "mti.h"

char *severity[] = {"NOTE", "WARNING", "ERROR", "FAILURE"};
static char *get_string(mtiVariableIdT id);

void in_params (
    int          vhdl_integer, /* IN integer          */
    int          vhdl_enum,    /* IN severity_level */
    double       *vhdl_real,   /* IN real           */
    mtiVariableIdT vhdl_array /* IN string         */
)
{
    printf("Integer    = %d\n", vhdl_integer);
    printf("Enum      = %s\n", severity[vhdl_enum]);
    printf("Real       = %g\n", *vhdl_real);
    printf("String    = %s\n", get_string(vhdl_array));
}

void out_params (
    int          *vhdl_integer, /* OUT integer        */
    char         *vhdl_enum,    /* OUT severity_level */
    double       *vhdl_real,   /* OUT real           */
    mtiVariableIdT vhdl_array /* OUT string         */
)
{
    char *val;
    int i, len, first;

    *vhdl_integer += 1;

    *vhdl_enum += 1;
    if (*vhdl_enum > 3){
        *vhdl_enum = 0;
    }

    *vhdl_real += 1.01;

    /* rotate the array */
    val = mti_GetArrayVarValue(vhdl_array, NULL);
    len = mti_TickLength(mti_GetVarType(vhdl_array));
    first = val[0];
    for (i = 0; i < len - 1; i++){
        val[i] = val[i+1];
    }
    val[len - 1] = first;
}
```

```

}

/* Convert a VHDL String array into a NULL terminated string */
static char *get_string(mtiVariableIdT id)
{
    static char buf[1000];
    mtiTypeIdT type;
    int len;

    mti_GetArrayVarValue(id, buf);
    type = mti_GetVarType(id);
    len = mti_TickLength(type);
    buf[len] = 0;
    return buf;
}

```

Package (pkg) example

The FOREIGN attribute specification links the C functions (declared above) to VHDL procedures (**in_params()** and **out_params()**) in **pkg**.

```

package pkg is
    procedure in_params(
        vhdl_integer : IN integer;
        vhdl_enum     : IN severity_level;
        vhdl_real      : IN real;
        vhdl_array     : IN string);
    attribute foreign of in_params : procedure is "in_params test.sl";

    procedure out_params(
        vhdl_integer : OUT integer;
        vhdl_enum     : OUT severity_level;
        vhdl_real      : OUT real;
        vhdl_array     : OUT string);
    attribute foreign of out_params : procedure is "out_params test.sl";
end;

package body pkg is

    procedure in_params(
        vhdl_integer : IN integer;
        vhdl_enum     : IN severity_level;
        vhdl_real      : IN real;
        vhdl_array     : IN string) is
    begin
        report "ERROR: foreign subprogram in_params not called";
    end;

```



```

procedure out_params(
    vhdl_integer : OUT integer;
    vhdl_enum    : OUT severity_level;
    vhdl_real    : OUT real;
    vhdl_array   : OUT string) is
begin
    report "ERROR: foreign subprogram out_params not called";
end;
end;

```

Entity (test) example

The VHDL model **test** contains calls to procedures (**in_params()** and **out_params()**) that are declared in **pkg** and linked to functions in the original C subprogram.

```

entity test is end test;
use work.pkg.all;
architecture only of test is
begin
    process
        variable int : integer := 0;
        variable enum : severity_level := note;
        variable r    : real := 0.0;
        variable s    : string(1 to 5) := "abcde";
    begin
        for i in 1 to 10 loop
            in_params(int, enum, r, s);
            out_params(int, enum, r, s);
        end loop;
        wait;
    end process;
end;

```

Using checkpoint/restore with the FLI

In order to use checkpoint/restore with the FLI, any data structures that have been allocated in foreign models and certain IDs passed back from mti function calls must be explicitly saved and restored. We have provided a number of features to make this as painless as possible.

The main feature is a set of memory allocation function calls. Memory allocated by such a function call will be automatically restored for you to the same location in memory, ensuring that pointers into the memory will still be valid.

The second feature is a collection of explicit calls to save and restore data. You will need to use these for any pointers to your data structures, and for IDs returned from mti routines. Pointers that you save and restore must be global, not variables on the stack. If you choose not to use the MTI provided memory allocation functions, you will have to explicitly save and restore your allocated memory structures as well.

You must code your model assuming that the code could reside in a different memory location when restored. This requires that you also reset all callback functions after a restore.

The following is a C model of a two-input AND gate (the same model as provided in `<install_dir>/modeltech/examples/foreign/gates.c`) adapted for checkpoint/restore. The lines added for checkpoint/restore are marked with comments.

```
#include <stdio.h>
#include "mti.h"

typedef struct {
    mtiSignalIdT in1;
    mtiSignalIdT in2;
    mtiDriverIdT out1;
} inst_rec;

do_and(ip)
    inst_rec *ip;

{
    int val1, val2;
    int result;
    char buf[128];

    val1 = mti_GetSignalValue(ip->in1);
    val2 = mti_GetSignalValue(ip->in2);
    result = val1 & val2;
    mti_ScheduleDriver(ip->out1, result, 0, MTI_INERTIAL);
}

and_gate_init(region, param, generics, ports)
    mtiRegionIdT region;
    char *param;
    mtiInterfaceListT *generics;
    mtiInterfaceListT *ports;
{
    inst_rec *ip;
    mtiSignalIdT outp;
    mtiProcessIdT proc;
```

```

if (mti_IsRestore( )){/* new */
    ip = (inst_rec *)mti_RestoreLong( ); /* new */
    proc = (mtiProcessIdT)mti_RestoreLong( ); /* new */
    mti_RestoreProcess(proc, "pl", do_and, ip); /* new */
} else if (mti_IsFirstInit( )) {
    ip = (inst_rec *)mti_Malloc(sizeof(inst_rec));
                                /* malloc changed to mti_Malloc */
    ip->in1 = mti_FindPort(ports, "in1");
    ip->in2 = mti_FindPort(ports, "in2");
    outp = mti_FindPort(ports, "out1");
    ip->out1 = mti_CreateDriver(outp);
    proc = mti_CreateProcess("pl", do_and, ip);
    mti_Sensitize(proc, ip->in1, MTI_EVENT);
    mti_Sensitize(proc, ip->in2, MTI_EVENT);
} else {
    /* do whatever you might want to do for restart */
    /*...*/
}
mti_AddSaveCB(mti_SaveLong, ip); /* new */
mti_AddSaveCB(mti_SaveLong, proc); /* new */
mti_AddRestartCB(mti_Free, ip);
                                /* free changed to mti_Free */
}

```

The above example displays the following features:

- Malloc and free calls have been replaced by mti_Malloc() and mti_Free().
- Callbacks are added using mti_AddSaveCB() to save the ip and proc pointers.
- The mti_IsRestore() flag is checked for restore.
- When a restore is being done, mti_RestoreLong() is used to restore the ip and proc pointers because mti_SaveLong() was used to save them.

Note: The restores must be performed in the same order as the saves.

- mti_RestoreProcess() is used to update mti_CreateProcess() with the possibly new address of the do_and() function. (This is in case the foreign code gets loaded into a different memory location.)
- All callbacks are added on first init and on restore. The restore does not restore callbacks because the routines might be located at different places after the restore operation.

Support for Verilog instances

The FLI functions are designed to work with VHDL designs and VHDL objects. However, the functions for traversing the design hierarchy also recognize Verilog instances.

The following functions operate on Verilog instances as indicated:

`mti_GetTopRegion()`

Gets the first top-level module. Use `mti_NextRegion()` to get additional top-level modules.

`mti_GetPrimaryName()`

Gets the module name.

`mti_GetSecondaryName()`

Returns NULL for Verilog modules.

The following functions operate on Verilog instances in the same manner that they operate on VHDL instances:

<code>mti_CreateRegion</code> (12-312)	<code>mti_GetRegionFullName</code> (12-317)
<code>mti_FindRegion</code> (12-313)	<code>mti_GetRegionName</code> (12-317)
<code>mti_FirstLowerRegion</code> (12-314)	<code>mti_GetRegionSourceName</code> (12-317)
<code>mti_GetCurrentRegion</code> (12-315)	<code>mti_HigherRegion</code> (12-321)
<code>mti_GetLibraryName</code> (12-316)	<code>mti_NextRegion</code> (12-322)

All other FLI functions operate only on VHDL instances and objects. Specifically, the functions that operate on VHDL signals and drivers cannot be used on Verilog nets and drivers. For example, a call to `mti_FirstSignal()` on a Verilog region always returns NULL. You must use the PLI functions to operate on Verilog objects.

Support for Windows platforms

Under Windows, sockets are separate objects from files and pipes, which require the use of different system calls. There is no way to determine if a given descriptor is for a file or a socket. This necessitates the use of different callback functions for sockets under Windows NT/95/98. The following functions work specifically with sockets. While these functions are required for use with Windows, they are optional for use on UNIX platforms.

- `mti_AddSocketInputReadyCB` (12-310)
- `mti_AddSocketOutputReadyCB` (12-310)

FLI function descriptions

The FLI functions described below are in alphabetic order by function name. The function declarations are in the *mti.h* header file located in the *ModelSim* installation directory.

```
void mti_AddCommand(char *cmd_name, mtiVoidFuncPtrT func)
```

Adds a simulator command. The `cmd_name` case is significant. The simulator command interpreter subsequently recognizes the command and calls the user-supplied function whenever the command is recognized. The entire command line (the command and any arguments) are passed to the user function as a single `char *` argument.

It is legal to add a command with the same name as a previously added command (or even a standard simulator command), but only the command added last has any effect.

```
void mti_AddEnvCB(mtiVoidFuncPtrT func, void *param)
```

Causes the specified function to be called whenever the environment is changed; for example, when the **environment** command (CR-80) is used. When the function is called, it is passed the parameter specified by "param".

```
void mti_AddInputReadyCB(int file_desc, mtiVoidFuncPtrT func, void *param)
```

Causes the specified function to be called when the specified file descriptor has data available for reading. This is similar to the function `XtAppAddInput` in X11R5. When the function is called, it is passed the parameter specified by "param".

```
void mti_AddLoadDoneCB(mtiVoidFuncPtrT func, void *param)
```

Causes the specified function to be called when elaboration of the entire design tree is complete. When the function is called, it is passed the parameter specified by "param".

```
void mti_AddOutputReadyCB(int file_desc, mtiVoidFuncPtrT func, void *param)
```

Causes the specified function to be called when the specified file descriptor is available for writing. The function is passed the parameter specified by "param".

```
void mti_AddQuitCB(mtiVoidFuncPtrT func, void *param)
```

Causes the specified function to be called when the simulator exits. The function is passed the parameter specified by "param".

```
void mti_AddRestartCB(mtiVoidFuncPtrT func, void *param)
```

Causes the specified function to be called before the simulator is restarted. The function is passed the parameter specified by “param”. This function should free any memory that was allocated.

```
void mti_AddRestoreCB(mtiVoidFuncPtrT func, void *param)
```

Causes the specified function to be called on a warm restore. This function cannot be used for cold restores (vsim -restore) as function entry points might be located at different places after a cold restore. The function is passed the parameter specified by “param”.

```
void mti_AddSaveCB(mtiVoidFuncPtrT func, void *param)
```

Causes the specified function to be called when a checkpoint operation is performed. The function is passed the parameter specified by “param”.

```
void mti_AddSimStatusCB(mtiVoidFuncPtrT func, void *param)
```

Causes the specified function to be called when the simulator RUN status changes. For this callback, the function will be called with two arguments: the user specified param, and a second argument of type int, which is 1 when the simulator is about to start a run and 0 when the run completes.

```
void mti_AddSocketInputReadyCB(int socket_desc, mtiVoidFuncPtrT func,  
void *param)
```

Causes the specified function to be called when the specified socket descriptor has data available for reading. The function is passed the parameter specified by “param”.

```
void mti_AddSocketOutputReadyCB(int socket_desc, mtiVoidFuncPtrT func,  
void *param)
```

Causes the specified function to be called when the specified socket descriptor is available for writing. The function is passed the parameter specified by “param”.

```
void mti_AddTclCommand(char *cmd_name, mtiVoidFuncPtrT func, void *param,  
mtiVoidFuncPtrT func_delete_cb)
```

Gives access to Tcl_CreateCommand(). Similar to mti_AddCommand() with the difference that the parameter "param" will be passed to the command function.

```
int mti_AskStdin(char *buf, char *prompt)
```

Asks for an input string using a prompt created by concatenating the "prompt" parameter with a '>' (for example, "input>"). The parameter "buf" must be allocated by the caller. It will be used to return the user input. This function returns -1 if the buf parameter is NULL; otherwise it returns 0.

```
void mti_Break(void)
```

Requests the simulator to halt the simulation and issue an assertion message with the text "Halt requested". The break request is satisfied after the caller returns control to the simulator.

The simulation can be continued after being halted with `mti_Break()`. This function cannot be called during elaboration.

```
int mti_Cmd(char *cmd)
```

Executes the simulator command specified by the parameter "cmd". The string must contain the command just as it would be typed at the VSIM prompt. A Tcl interp status (TCL_OK or TCL_ERROR) is returned. (See `mti_Interp()` (12-321).) The results are not transcribed.

Any command that changes the state of simulation (such as run, restart, etc.) cannot be sent from a foreign architecture or subprogram.

```
void mti_Command(char *cmd)
```

Executes the simulator command specified by the parameter "cmd." The string must contain the command just as it would be typed at the VSIM prompt. The results are transcribed.

Any command that changes the state of simulation (such as run, restart, etc.) cannot be sent from a foreign architecture or subprogram.

```
mtiTypeIdT mti_CreateArrayType(mtiInt32T left, mtiInt32T right, mtiTypeIdT  
elem_type)
```

Creates a new type ID that describes an ARRAY type. The parameters "left" and "right" specify the bounds of the array. The parameter "elem_type" specifies the type of the elements of the array.

```
mtiDriverIdT mti_CreateDriver(mtiSignalIdT sig)
```

Creates a driver on a signal. A driver must be created for a resolved signal in order to be able to drive values onto that signal and have the values be resolved. Multiple drivers can be created for a resolved signal, but no more than one driver can be created for an unresolved signal. Alternately, an unresolved signal's value can be changed using `mti_SetSignalValue()`.

A driver cannot be created on a RECORD signal, but drivers can be created on non-record subelements.

When using `mti_CreateDriver()` it is necessary to follow up with a call to `mti_SetDriverOwner()` (12-326), otherwise the "drivers" command and the Dataflow window may give unexpected or incorrect information regarding the newly created driver.

`mtiTypeIdT mti_CreateEnumType(mtiInt32T size, mtiInt32T count, char **literals)`
Creates a new type ID that describes an enumeration type. The parameter "count" indicates how many values are in the type. The parameter "literals" is an array of strings that defines the enumeration literals for the type. The number of elements in the array must equal count. The first element of the array is the left-most value of the enumerated type.

The "size" parameter specifies how many bytes of storage the simulator must use for values of this type. If count is greater than 256, then size must be 4. Otherwise, size may be either 1 or 4.

`mtiProcessIdT mti_CreateProcess(char *name, mtiVoidFuncPtrT func, void *param)`
Creates a new process. The parameter "name" is the name that will appear in the simulator's process window. The specified function will be called at time 0 after all the signals have been initialized. The `mti_Sensitize()` and `mti_ScheduleWakeup()` functions can be used to cause the function to be called at other times. When the function is called, it is passed the parameter specified by "param".

`mtiTypeIdT mti_CreateRealType(void)`
Creates a new type ID that describes a VHDL REAL type.

`mtiRegionIdT mti_CreateRegion(mtiRegionIdT parent, char *name)`
Creates a new subregion with the specified name in the specified parent region. The name will be forced to lower case. The region ID for the new region is returned. The operation is the same for Verilog instances.

`mtiTypeIdT mti_CreateScalarType(mtiInt32T left, mtiInt32T right)`
Creates a new type ID that describes an integer scalar. The range of valid values is bounded by "left" and "right". The type ID for the new type is returned.

`mtiSignalIdT mti_CreateSignal(char *name, mtiRegionIdT region, mtiTypeIdT type)`
Creates a new signal of the specified type in the specified region. If the parameter "name" is not NULL, the signal will appear in the simulator's signal window. The name will be forced to lower case. If there is no error, the signal ID for the new signal is returned; otherwise a NULL is returned.

`mtiInt32T mti_Delta(void)`
Returns the simulator iteration count for the current time step.

`void mti_Desensitize(mtiProcessIdT proc)`
Disconnects a C process from the signals to which it is sensitive. That is, it undoes the connection created by the `mti_Sensitize()` call.

```
mtiTypeIdT mti_ElementType(mtiTypeIdT type)
```

See `mti_GetArrayElementType()`.

```
void mti_FatalError(void)
```

Causes the simulator to immediately halt the simulation and issue an assertion message with the text "*** Fatal: Foreign module requested halt". A call to `mti_FatalError()` does not return control to the caller. The simulation cannot continue after being halted with `mti_FatalError()`.

```
mtiDriverIdT mti_FindDriver(mtiSignalIdT sig)
```

Returns a driver ID (either scalar or array) for the specified signal ID. Returns NULL if there is no driver found for a scalar signal, or if any element of an array does not have a driver.

```
mtiSignalIdT mti_FindPort(mtiInterfaceListT *list, char *name)
```

This function searches linearly through the interface list and returns the signal ID of the port whose name matches the one specified. It returns NULL if it does not find the port. The search is not case-sensitive.

```
char *mti_FindProjectEntry(char *section, char *name, int expand)
```

Returns the value of an entry in the project file (*modelsim.ini*). The parameter "section" identifies the project file section in which the entry resides. The entry is identified by the parameter "name". If the parameter "expand" is nonzero, then environment variables are expanded; otherwise they are not expanded. For example, if *modelsim.ini* contains:

```
[myconfig]
myentry = $abc/xyz
```

then the following call returns the string "\$abc/xyz":

```
entry_value = mti_FindProjectEntry("myconfig", "myentry", 0)
```

The function returns NULL if the project entry does not exist. The caller is responsible for freeing the returned pointer with the `free()` C-library function.

```
mtiRegionIdT mti_FindRegion(char *name)
```

Returns the region ID for the specified region name. The region name can be either a full hierarchical name or a relative name. A relative name is relative to the region set by the **VSIM environment** command (CR-80). The default is the top level region. NULL is returned if the region is not found. The operation is the same for Verilog instances.

```
mtiSignalIdT mti_FindSignal(char *name)
```

Returns the signal ID for the specified signal name. The signal name can be either a full hierarchical name or a relative name. A relative name is relative to the region set by the

VSIM **environment** command (CR-80). The default is the top level region. NULL is returned if a matching signal is not found.

Note: mti_FindSignal() cannot be called to get signal IDs for composite subelements. That is, the name cannot be a subscripted array element or a selected record field.

```
mtiVariableIdT mti_FindVar(char *name)
```

Returns a variable ID for the specified variable, generic or constant name. The name can be either a full hierarchical name or a relative name. For variables, the name must include the process label. A relative name is relative to the region set by the VSIM **environment** command (CR-80). The default is the top level region. NULL is returned if the object is not found. For variables, mti_FindVar() can be called successfully only after the end of elaboration. (See mti_AddLoadDoneCB() (12-309).)

Note: mti_FindVar() cannot be called to get variable IDs for composite subelements. That is, the name cannot be a subscripted array element or a selected record field.

```
mtiRegionIdT mti_FirstLowerRegion(mtiRegionIdT reg)
```

Returns the region ID of the first subregion in the specified region or NULL if there are no subregions. mti_NextRegion() can be used to get the subsequent subregions in the region. The operation is the same for Verilog instances.

```
mtiProcessIdT mti_FirstProcess(mtiRegionIdT reg)
```

Returns the process ID of the first process in the specified region or NULL if there are no processes. mti_NextProcess() can be used to get the subsequent processes in the region.

```
mtiSignalIdT mti_FirstSignal(mtiRegionIdT reg)
```

Returns the signal ID of the first signal in the specified region or NULL if there are no signals in the region. mti_NextSignal() can be used to get the subsequent signals in the region.

```
mtiVariableIdT mti_FirstVar(mtiProcessIdT proc)
```

Returns the variable ID of the first variable in the specified process or NULL if there are no variables. mti_NextVar() can be used to get the subsequent variables in the process. This function can be called successfully only after the end of elaboration.

```
void mti_Free(void *ptr)
```

Returns the specified block of memory allocated by mti_Malloc() to the MTI memory allocator. mti_Free() cannot be used for memory allocated with direct calls to malloc(). Also, direct calls to free() cannot be used to free memory allocated with mti_Malloc().

```
void mtiTypeIdT mti_GetArrayElementType(mtiTypeIdT type)
```

If the "type" parameter is an ARRAY type, returns the type ID for the elements in the array; otherwise, returns NULL.

```
void *mti_GetArraySignalValue(mtiSignalIdT sig, void *buf)
```

Gets the value of an ARRAY type signal. If the buf parameter is NULL, this function allocates memory for the value and returns a pointer to it. The caller is responsible for freeing this memory with the *free()* C-library function. If the buf parameter is not NULL, this function copies the value into buf and returns buf. See [Mapping to VHDL data types](#) (12-328) for information on how to decode the return value.

```
void *mti_GetArrayVarValue(mtiVariableIdT var, void * buf)
```

Gets the value of an ARRAY type variable. If the buf parameter is NULL, this function returns a pointer to the value, which must be treated as read-only data and must not be freed. If the buf parameter is not NULL, this function copies the value into buf and returns buf. See [Mapping to VHDL data types](#) (12-328) for information on how to decode the return value.

```
mtiRegionIdT mti_GetCallingRegion(void)
```

During elaboration, returns the region ID of the current elaboration region. During simulation, returns the region ID of the current active process or signal resolution function context. This function can be called by a foreign subprogram to determine from which process it was called. If there is currently no active process or signal resolution function, *mti_GetCallingRegion()* returns the current environment set by the [environment](#) command (CR-80). The operation is the same for Verilog instances.

```
char *mti_GetCheckpointFilename(void)
```

Returns the filename specified with the most recent checkpoint command or NULL if no checkpoint has been done. The returned pointer must not be freed.

```
mtiRegionIdT mti_GetCurrentRegion(void)
```

During elaboration, returns the region ID of the current elaboration region. During simulation, returns the region ID of the current environment set by the [environment](#) command (CR-80). The operation is the same for Verilog instances.

```
mtiDriverIdT *mti_GetDriverSubelements(mtiDriverIdT driv, mtiDriverIdT *buf)
```

Returns an array of driver IDs for each of the subelements of the ARRAY type driver "driv". If the buf parameter is NULL, this function allocates memory for the value and returns a pointer to it. The caller is responsible for freeing this memory with the *free()* C-library function. If the buf parameter is not NULL, this function copies the value into buf and returns buf.

```
char **mti_GetEnumValues(mtiTypeIdT type)
```

Returns a pointer to an array of enumeration literals for the specified enumeration type. The pointer must not be freed. The number of elements in the array can be found by calling `mti_TickLength()`. The first element in the array is the left-most value of the enumerated type. This function returns NULL if the type is not an enumeration type.

```
mtiInterfaceListT *mti_GetGenericList(mtiRegionIdT reg)
```

Returns a list of the generics defined for the specified region. This list is in the same interface format as the C initialization function generics list. See ["The C initialization function"](#) (12-299). Each element in the list returned by this function can be freed using `mti_Free()`. In order to use this function on a foreign architecture region, there must be a "+" in front of the initialization function name in the foreign attribute. See [Using the VHDL FLI with foreign architectures](#) (12-298).

This function returns NULL if the regionID indicates a foreign architecture, unless the initialization function name is preceded with a "+" in the foreign attribute string.

```
char *mti_GetLibraryName(mtiRegionIdT reg)
```

Returns the logical name of the library that contains the design unit identified by the region "reg". If the region is not a design unit, then the parent design unit is used. The operation is the same for Verilog instances. The returned pointer must not be freed.

```
int mti_GetNextEventTime(mtiTime64T *timep)
```

Use is limited to cosimulation with foreign simulators or a simulator backplane. Not to be used within a VHDL process. Returns the time the next simulation event will occur. The return value indicates one of the following:

- 0 - There are no pending events; *timep is set to the current time.
- 1 - There are pending events; *timep is set to the maturity time.
- 2 - There are pending postponed processes for the last delta of the current time; *timep is set to the maturity time of the next event (which is in the future).

```
int mti_GetNextNextEventTime(mtiTime64T *timep)
```

Used by a VHDL process to get the time the next simulation event will occur. The return value is the same as for `mti_GetNextEventTime()`.

```
long mti_GetNumRecordElements(mtiTypeIdT type)
```

Returns the number of elements in a record type.

```
mtiPhysicalData* mti_GetPhysicalData(mtiTypeIDT type)
```

If the parameter is a physical type, this function returns a pointer to a linked list of structures

each describing the name and position of a unit in the physical type. The linked list is traversed by using the next pointer in each structure. Traversal is terminated by a NULL pointer. The caller is responsible for freeing each structure in the list with `mti_Free()`. If the parameter is not a physical type, a NULL pointer is returned.

```
char *mti_GetPrimaryName(mtiRegionIdT reg)
```

Returns the primary name of the region (that is, an entity, package, or configuration name). If the region is not a primary design unit, then the parent primary design unit is used. Returns the module name for Verilog instances. The returned pointer must not be freed.

```
char *mti_GetProcessName(mtiProcessIdT proc)
```

Returns the name of the specified process. The returned pointer must not be freed.

```
char *mti_GetProductVersion(void)
```

Returns the name and version of the product. The returned pointer must not be freed.

```
char *mti_GetRegionFullName(mtiRegionIdT reg)
```

Returns the full hierarchical name of the specified region. The caller is responsible for freeing the returned pointer with the *free()* C-library function. The operation is the same for Verilog instances.

```
int mti_GetRegionKind(mtiRegionIdT reg)
```

Returns the kind (Verilog or VHDL) of the specified region. The value returned is one of the values defined in *acc_user.h* or *acc_vhdl.h*.

```
char *mti_GetRegionName(mtiRegionIdT reg)
```

Returns the name of the specified region. This pointer must not be freed. The operation is the same for Verilog instances.

```
char *mti_GetRegionSourceName(mtiRegionIdT reg)
```

Returns the name of the VHDL source file which contains the specified region. This pointer must not be freed. The operation is the same for Verilog instances.

```
int mti_GetResolutionLimit(void)
```

Returns the simulator resolution limit in log10 seconds. The value ranges from -15 (1fs) to 2 (100 sec). For example, the function returns *n* from the expression: `time_scale = 1*10^n` seconds; a time scale of 1 ns returns -9 and a time scale of 100 ps returns -10.

```
char *mti_GetSecondaryName(mtiRegionIdT reg)
```

Returns the secondary name of the specified region (that is, an architecture or package body name). If the region is not a secondary design unit, then the parent secondary design unit is used. Returns NULL for Verilog modules. The returned pointer must not be freed.

```
mtiDirectionT mti_GetSignalMode(mtiSignalIdT sig)
```

Returns the port mode of the specified signal. The mode will be: MTI_DIR_IN, MTI_DIR_OUT, MTI_DIR_INOUT or MTI_INTERNAL. MTI_INTERNAL means that the signal is not a port.

```
char *mti_GetSignalName(mtiSignalIdT sig)
```

Returns the name of the specified signal. If the signal is a composite subelement, the name returned is the name of the top level composite. To get the name of a composite subelement signal, use mti_GetSignalNameIndirect(). The returned pointer must not be freed.

```
char *mti_GetSignalNameIndirect(mtiSignalIdT sig, char *buf, int length)
```

Returns the name of the specified signal including array indices and record fields. If the buf parameter is NULL, this function allocates space for the name and returns a pointer to it. The caller is responsible for freeing this memory with the *free()* C-library function. If the buf parameter is not NULL, this function copies the name into buf up to the length of the length parameter and returns buf.

```
mtiRegionIdT mti_GetSignalRegion(mtiSignalIdT sig)
```

Returns the region ID for the specified signal.

```
mtiSignalIdT *mti_GetSignalSubelements(mtiSignalIdT sig, mtiSignalIdT *buf)
```

Returns an array of signal IDs for each of the subelements of the specified composite signal. If the buf parameter is NULL, this function allocates memory for the value and returns a pointer to it. The caller is responsible for freeing this memory with the *free()* C-library function. If the buf parameter is not NULL, this function copies the value into buf and returns buf. Returns NULL if the sig parameter is a scalar.

The internal representation of multi-dimensional arrays is the same as arrays of arrays. For example, array a(x,y,z) is accessed in the same manner as a(x)(y)(z). In order to get to the scalar subelements of an array of arrays, mti_GetSignalSubelements() must be used on each level of the array until reaching the scalar subelements.

For example, if you have the following:

```
type mem_type is array (0 to 127) of std_logic_vector(7 downto 0);
signal mem1 : mem_type;
```

then calling mti_GetSignalSubelements() on signal mem1 will return an array of signal IDs for each of the std_logic_vector subelements. You must then call mti_GetSignalSubelements() on each of the std_logic_vector subelements to get the signal IDs of the std_logic scalar subelements. You can use mti_TickLength() to get the length of the array at each level.

```
mtiTypeIdT mti_GetSignalType(mtiSignalIdT sig)
```

Returns the type ID for the specified signal.

```
mtiInt32T mti_GetSignalValue(mtiSignalIdT sig)
```

Returns the value of any scalar signal except those of types REAL and TIME. For composite, REAL, and TIME type signals, use `mti_GetSignalValueIndirect()`.

```
void *mti_GetSignalValueIndirect(mtiSignalIdT sig, void *buf)
```

Returns the value of a signal of any type except RECORD. This function must be used for signals of type REAL and TIME. If the `buf` parameter is NULL, this function allocates memory for the value and returns a pointer to it. The caller is responsible for freeing this memory with the `free()` C-library function. If the `buf` parameter is not NULL, this function copies the value into `buf` and returns `buf`.

See "Mapping to VHDL data types" (p480) for information on how to decode the return value.

Note: In order to get the value of a record signal, use `mti_GetSignalSubelements()` to get the signal IDs of the subelements and then use `mti_GetSignalValue()`, `mti_GetSignalValueIndirect()`, or `mti_GetArraySignalValue()` for each of the subelements.

```
mtiRegionIdT mti_GetTopRegion(void)
```

Returns the region ID for the top of the design tree. This can be used to traverse the signal hierarchy from the top. For Verilog, gets the first top-level module. (Use `mti_NextRegion()` to get additional top-level Verilog modules.)

```
int mti_GetTraceLevel(void)
```

Returns the trace level if **vsim** (CR-208) was invoked with the `-trace_foreign` option; otherwise, returns 0. Use to detect whether FLI/PLI tracing is on.

```
mtiTypeKindT mti_GetTypeKind(mtiTypeIdT type)
```

Returns the kind of the type described by the type ID; that is, one of the enum values listed for the `mtiTypeKindT` type in `mti.h`.

Note: `MTI_TYPE_SCALAR` indicates an integer type.

```
void mti_GetVarAddr(char *name)
```

Returns the address of the specified variable. The variable name can be either a full hierarchical name or a relative name. The variable name must include the process label. A

relative name is relative to the region set by the VSIM **environment** (CR-80) command. The top level region is the default. NULL is returned if the variable is not found.

The caller can read or modify the value of the variable provided the type of the variable is known. Given the variable's data type, the caller can interpret the storage pointed to by the returned pointer. This pointer must not be freed. This function can be called successfully only after the end of elaboration.

Note: mti_GetVarAddr() cannot be called to get the address of a composite subelement. That is, the name cannot be a subscripted array element or a selected record field.

```
char *mti_GetVarImage(char *name)
```

Returns a pointer to a static buffer containing the string image of the specified variable. The variable name may be either a full hierarchical name or a relative name. The variable name must include the process label. A relative name is relative to the region set by the VSIM **environment** (CR-80) command. The top level region is the default. NULL is returned if the variable is not found. The returned string is valid only until the next call to any FLI function. This pointer must not be freed. This function can be called successfully only after the end of elaboration.

The image is the same as would be returned by the VHDL 1076-1993 attribute 'IMAGE.

Note: mti_GetVarImage() cannot be called to get the image of a composite subelement. That is, the name cannot be a subscripted array element or a selected record field.

```
char *mti_GetVarImageById(mtiVariableIdT var)
```

Returns a pointer to a static buffer containing the string image of the specified variable. The returned string is valid only until the next call to any FLI function. This pointer must not be freed. This function can be called successfully only after the end of elaboration.

The image is the same as would be returned by the VHDL 1076-1993 attribute 'IMAGE.

```
char *mti_GetVarName(mtiVariableIdT var)
```

Returns the simple name of the specified variable, or NULL if no information can be found. The returned pointer must not be freed. This function cannot be used with variable IDs passed as foreign subprogram parameters.

```
mtiVariableIdT *mti_GetVarSubelements(mtiVariableIdT var, mtiVariableIdT *buf)
```

Returns an array of variable IDs for the subelements of the specified composite variable. If

buf is NULL, then this function allocates memory for the value and returns a pointer to it. The caller is responsible for freeing this memory with the *free()* C-library function. If buf is not NULL, then the value is copied into buf and buf is returned. This function cannot be used with variable IDs passed as foreign subprogram parameters.

```
mtiTypeIdT mti_GetVarType(mtiVariableIdT var)
```

Returns the type ID for the specified variable.

```
long mti_GetVarValue(mtiVariableIdT var)
```

Returns the value of any scalar variable except those of types REAL and TIME. For composite, REAL and TIME type variables, use mti_GetVarValueIndirect().

```
void *mti_GetVarValueIndirect(mtiVariableIdT var, void *buf)
```

Returns the value of any variable of any type except RECORD. This function must be used for variables of type TIME and REAL. If the buf parameter is NULL, this function returns a pointer to the value, which must be treated as read-only data. This pointer must not be freed. If the buf parameter is not NULL, this function copies the value into buf and returns buf.

See "Mapping to VHDL data types" (p480) for information on how to decode the return value.

Note: In order to get the value of a record variable, use mti_GetVarSubelements() to get the variable IDs of the subelements and then use mti_GetVarValue(), mti_GetVarValueIndirect(), or mti_GetArrayVarValue() for each of the subelements.

```
mtiRegionIdT mti_HigherRegion(mtiRegionIdT reg)
```

Returns the parent region of the specified region. The operation is the same for Verilog instances.

```
char *mti_Image(void *value, mtiTypeIdT type)
```

Returns a pointer to a static buffer containing the string image of the value pointed to by the value parameter. The format is determined by the specified type. The image is the same as would be returned by the VHDL 1076-1993 attribute 'IMAGE. The returned string is valid only until the next call to any FLI function. This pointer must not be freed.

```
void *mti_Interp(void)
```

Returns the Tcl_Interp* used in the simulator. This pointer is needed in most Tcl calls and can also be used in conjunction with mti_Cmd() to obtain the command return value (result). For example,

```
{
```

```

    Tcl_Interp *interp = mti_Interp( );
    int stat = mti_Cmd("examine foo");
    if (stat == TCL_OK) {
        printf("Examine foo results = %s\n", interp->result);
    } else {
        printf("Command Error: %s\n", interp->result);
    }
}

```

```
int mti_IsColdRestore(void)
```

Returns 1 when a “cold” restore operation is in progress; otherwise, returns 0. A “cold” restore is when VSIM has been terminated and is re-invoked with the **-restore** command-line option.

```
int mti_IsFirstInit(void)
```

Returns 1 if this is the first call to the initialization function; 0 if the simulation has been restarted.

```
int mti_IsPE(void)
```

Returns 1 if the current simulator is the PE version. Returns 0 for the EE version.

```
int mti_IsRestore(void)
```

Returns 1 when a restore operation is in progress; otherwise, returns 0.

```
void *mti_Malloc(unsigned long size)
```

Allocates a block of memory of the specified size and returns a pointer to it. The memory is initialized to zero. On restore, the memory block is guaranteed to be restored to the same location with the values contained at the time of the checkpoint. This memory can be freed by `mti_Free()`.

```
mtiProcessIdT mti_NextProcess(void)
```

Returns the process ID for the next process in the current region. (See `mti_FirstProcess()` (12-314).) Returns NULL if there are no more processes.

```
mtiRegionIdT mti_NextRegion(mtiRegionIdT reg)
```

Returns the region ID of the next region at the same level as the specified region. Returns NULL if this is the last region. (See `mti_FirstLowerRegion()` (12-314).) The operation is the same for Verilog instances.

```
mtiSignalIdT mti_NextSignal(void)
```

Returns the signal ID of the next signal in the current region. Returns NULL if there are no more signals. (See `mti_FirstSignal()` (12-314).)

```
mtiVariableIdT mti_NextVar(void)
```

Returns the variable ID of the next variable in the process specified in the call to `mti_FirstVar()`. Returns NULL when there are no more variables. This function can be called successfully only after the end of elaboration.

```
mtiInt32T mti_Now(void)
```

Returns the low order 32 bits of the current simulation time. The time units are equivalent to the current simulator time unit setting. (See `mti_GetResolutionLimit()` (12-317).)

```
mtiTime64T *mti_NowIndirect(mtiTime64T *buf)
```

Returns a structure containing the upper and lower 32 bits of the 64-bit current simulation time. The time units are equivalent to the current simulator time unit setting. (See `mti_GetResolutionLimit()`.) If the `buf` parameter is NULL, this function allocates memory for the value and returns a pointer to it. The caller is responsible for freeing this memory with the `free()` C-library function. If the `buf` parameter is not NULL, this function copies the value into `buf` and returns `buf`.

```
mtiInt32T mti_NowUpper(void)
```

Returns the high order 32 bits of the current simulation time. The time units are equivalent to the current simulator time unit setting. (See `mti_GetResolutionLimit()` (12-317).)

```
void mti_PrintMessage(char *msg)
```

Prints a message in the main VSIM window. The newline character (`\n`) can be included in the string to print text on a new line; however, a newline character is provided by default.

```
void mti_Quit(void)
```

Shuts down the simulator immediately.

```
void *mti_Realloc(void *ptr, unsigned long size)
```

Works just like the UNIX `realloc()` function on memory allocated by `mti_Malloc()`. If the specified size is larger than the size of block already allocated to the pointer `ptr`, then new memory of the required size is allocated and initialized to zero, the entire contents of the old block are copied into the new block, and a pointer to the new block is returned. Otherwise, a pointer to the old block is returned. Any memory allocated by `mti_Realloc()` is guaranteed to be restored like `mti_Malloc()`. This memory can be freed by `mti_Free()`.

```
void mti_RemoveEnvCB(mtiVoidFuncPtrT func, void *param)
```

Removes the function from the environment callback list.

```
void mti_RemoveLoadDoneCB(mtiVoidFuncPtrT func, void *param)
```

Removes the function from the elaboration complete callback list.

```
void mti_RemoveQuitCB(mtiVoidFuncPtrT func, void *param)
```

Removes the function from the quit callback list.

```
void mti_RemoveRestartCB(mtiVoidFuncPtrT func, void *param)
```

Removes the function from the restart callback list.

```
void mti_RemoveRestoreCB(mtiVoidFuncPtrT func, void *param)
```

Removes the function from the restore callback list.

```
void mti_RemoveSaveCB(mtiVoidFuncPtrT func, void *param)
```

Removes the function from the save callback list.

```
void mti_RemoveSimStatusCB(mtiVoidFuncPtrT func, void *param)
```

Removes the function from the simulator run status callback list.

```
char *mti_Resolution(void)
```

Returns a string that has the name of the time unit for the simulator resolution. This pointer must not be freed. For backward compatibility with versions prior to 5.0. CAUTION: If this function is called by an FLI application and the resolution is not a power of 1000 (e.g., "-t 10ps" is not a power of 1000, but "-t ps" is), then mti_Resolution() will cause the simulation to terminate.

```
void mti_RestoreBlock(char *ptr)
```

Restores a block of data from the checkpoint file to the address pointed to by the ptr parameter. The size of the data block restored is the same as the size that was saved by the corresponding mti_SaveBlock() call.

```
char mti_RestoreChar(void)
```

Returns one byte of data read from the checkpoint file.

```
long mti_RestoreLong(void)
```

Returns sizeof(long) bytes of data read from the checkpoint file.

```
void mti_RestoreProcess(mtiProcessIdT proc, char *name, mtiVoidFuncPtrT func,  
void *param)
```

Restores a process that was created by mti_CreateProcess(). The first argument is the process ID that was returned from the original mti_CreateProcess() call. The remaining arguments are the same as the original mti_CreateProcess() call.

```
short mti_RestoreShort(void)
```

Returns sizeof(short) bytes of data read from the checkpoint file.

```
char *mti_RestoreString(void)
```

Returns a pointer to a temporary buffer holding a null terminated string read from the checkpoint file. If the size of the string is less than 1024 bytes, the string must be copied if it is to be used later, because it will be overwritten on the next `mti_RestoreString()` call. If the size exceeds the 1024 byte temporary buffer size, memory is allocated automatically by the `mti_RestoreString()` function. The function is designed to handle unlimited size strings. The returned pointer must not be freed.

```
void mti_SaveBlock(char *ptr, unsigned long size)
```

Saves a block of data of the specified size, pointed to by the `ptr` parameter, to the checkpoint file.

```
void mti_SaveChar(char data)
```

Saves one byte of data to the checkpoint file.

```
void mti_SaveLong(long data)
```

Saves `sizeof(long)` bytes of data to the checkpoint file.

```
void mti_SaveShort(short data)
```

Saves `sizeof(short)` bytes of data to the checkpoint file.

```
void mti_SaveString(char *data)
```

Saves a null-terminated string to the checkpoint file. The function is designed to handle unlimited size strings.

```
void mti_ScheduleDriver(mtiDriverIdT driver, long value, mtiDelayT delay,
    mtiDriverModeT mode)
```

Schedules a transaction on the specified driver. If the signal being driven is of an `ARRAY`, `REAL` or `TIME` type, then the value is considered to be “void *” instead of `long`. The mode parameter may be either `MTI_INERTIAL` or `MTI_TRANSPORT`. The delay time units are equivalent to the current simulator time unit setting. (See `mti_GetResolutionLimit()`.)

```
void mti_ScheduleWakeup(mtiProcessIdT process, mtiDelayT delay)
```

Schedules the specified process to be called after the specified delay. A process can have no more than one pending wake-up call. A call to `mti_ScheduleWakeup()` cancels a prior pending wake-up call regardless of the delay values. The delay time units are equivalent to the current simulator time unit setting. (See `mti_GetResolutionLimit()`.)

```
void mti_Sensitize(mtiProcessIdT proc, mtiSignalIdT sig, mtiProcessTriggerT
    when)
```

Causes the specified process to be called when the specified signal is updated. If the *when*

parameter is `MTI_EVENT`, then the process is called when the signal changes value. If the *when* parameter is `MTI_ACTIVE`, then the process is called whenever the signal is active.

```
void mti_SetDriverOwner(mtiDriverIdT driverid, mtiProcessIdT processid)
```

Makes the specified process the owner of the specified driver. Normally `mti_CreateDriver()` makes the `<foreign_architecture>` process the owner of a new driver. See `mti_CreateDriver()` (12-311) for more details.

```
void mti_SetSignalValue(mtiSignalIdT sig, long val)
```

Sets the specified signal to the specified value. The signal may be either an unresolved signal or a resolved signal. Setting the signal makes it “active” in the current delta. If the new value is different than the old value, then an “event” occurs on the signal in the current delta. If the signal being set is of an `ARRAY`, `REAL`, or `TIME` type, then the value is considered to be “void *” instead of long.

`mti_SetSignalValue()` cannot be used to set the value on an entire `RECORD`, but it can be used to set the values on the individual scalar or array subelements.

Setting a resolved signal is not the same as driving it. After a resolved signal is set it may be changed to a new value the next time its resolution function is activated. Use `mti_ScheduleDriver()` to drive a signal.

```
void mti_SetVarValue(mtiVariableIdT var, long val)
```

Sets the specified variable to the specified value. If the variable being set is of an `ARRAY`, `REAL` or `TIME` type, then the value is considered to be “void *” instead of long. This function cannot be used to set the value of a `RECORD` variable, but it can be used to set the values on the individual scalar or array subelements.

```
char *mti_SignalImage(mtiSignalIdT sig)
```

Returns a pointer to a static buffer containing the string image of the value of the specified signal. The image is the same as would be returned by the VHDL 1076-1993 attribute `'IMAGE`. The returned string is valid only until the next call to any FLI function. This pointer must not be freed.

```
mtiInt32T mti_TickDir(mtiTypeIdT type)
```

Returns the index direction of an `ARRAY` type or the range direction for any type that has a range. The possible return values are: +1 for ascending, -1 for descending, and 0 for not-defined (for types that have no direction).

```
mtiInt32T mti_TickHigh(mtiTypeIdT type)
```

Returns the value of type `'HIGH`. For `REAL` and `RECORD` types it returns 0.

```
mtiInt32T mti_TickLeft(mtiTypeIdT type)
```

Returns the value of type'LEFT. For REAL and RECORD types it returns 0.

```
mtiInt32T mti_TickLength(mtiTypeIdT type)
```

Returns the value of type'LENGTH. For RECORD types, returns the number of fields.

```
mtiInt32T mti_TickLow(mtiTypeIdT type)
```

Returns the value of type'LOW. For REAL and RECORD types it returns 0.

```
mtiInt32T mti_TickRight(mtiTypeIdT type)
```

Returns the value of type'RIGHT. For REAL and RECORD types it returns 0.

```
void mti_TraceActivate(void)
```

Turns FLI/PLI tracing back on if tracing was suspended.

```
void mti_TraceOff(void)
```

Turns off FLI/PLI tracing and initiates a dump of callback-related information to the replay files.

```
void mti_TraceOn(int level, char *tag)
```

Optional way to turn on FLI/PLI tracing. NOT RECOMMENDED.

```
void mti_TraceSkipID(int n)
```

Used when replaying an FLI/PLI trace to increment or decrement the symbol generator to keep it in sync with the original trace. The argument **n** is a positive or negative integer indicating how much to increment or decrement the symbol table naming sequence.

```
void mti_TraceSuspend(void)
```

If FLI/PLI tracing is on, turns off trace output.

```
void mti_WriteProjectEntry(char *key, char *val)
```

Writes an entry into the *modelsim.ini* project file, in the form:

```
key = val
```

Mapping to VHDL data types

Many FLI functions have parameters and return values that represent VHDL object values. This section describes how the object values are mapped to the various VHDL data types.

VHDL data types are identified in the C interface by a type ID. A type ID can be obtained for a signal by calling `mti_GetSignalType()` and for a variable by calling `mti_GetVarType()`.

Alternatively, the `mti_CreateScalarType()`, `mti_CreateRealType()`, `mti_CreateEnumType()`, and `mti_CreateArrayType()` functions return type IDs for the data types they create.

Given a type ID handle, the `mti_GetTypeKind()` function returns a C enumeration of `mtiTypeKindT` that describes the data type. The mapping between `mtiTypeKindT` values and VHDL data types is as follows:

mtiTypeKindT value	VHDL data type
MTI_TYPE_ACCESS	Access type (pointer)
MTI_TYPE_ARRAY	Array composite type
MTI_TYPE_ENUM	Enumeration scalar type
MTI_TYPE_FILE	File type
MTI_TYPE_PHYSICAL	Physical scalar type
MTI_TYPE_REAL	Floating point scalar type
MTI_TYPE_RECORD	Record composite type
MTI_TYPE_SCALAR	Integer scalar types
MTI_TYPE_TIME	Time type

Object values for access and file types are not supported by the C interface. Values for record types are supported at the non-record subelement level. Effectively, this leaves scalar types and arrays of scalar types as valid types for C interface object values. In addition, multi-dimensional arrays are accessed in the same manner as arrays of arrays. For example, `toto (x,y,z)` is accessed as `toto (x)(y)(z)`.

Scalar and physical types use 4 bytes of memory, enumeration types use either 1 or 4 bytes, and TIME and REAL types use 8 bytes. The C type “long” is used for scalar and physical object values. An enumeration uses either 1 byte or 4 bytes,

depending on how many values are in the enumeration. If it has 256 or fewer values, then it uses 1 byte; otherwise, it uses 4 bytes. In some cases, all scalar types are cast to “long” before being passed as a non-array scalar object value across the C interface. The `mti_GetSignalValue()` function can be used to get the value of any non-array scalar signal object except `TIME` and `REAL` types, which can be retrieved using `mti_GetSignalValueIndirect()`. Use `mti_GetVarValue()` and `mti_GetVarValueIndirect()` for variables.

Enumeration types

Enumeration object values are equated to the position number of the corresponding identifier or character literal in the VHDL type declaration. For example:

```
-- C interface values
TYPE std_ulogic IS('U',-- 0
                  'X',-- 1
                  '0',-- 2
                  '1',-- 3
                  'Z',-- 4
                  'W',-- 5
                  'L',-- 6
                  'H',-- 7
                  '-' -- 8
                  );
```

Real and time types

Eight bytes are required to store the values of variables and signals of type `REAL` and `TIME`. In C, this corresponds, respectively, to the C “double” data type and the `mtiTime64T` structure defined in *mti.h*. The `mti_GetSignalValueIndirect()` and `mti_GetVarValueIndirect()` functions are used to retrieve these values.

Array types

The C type “void *” is used for array type object values. The pointer points to the first element of an array of C type “char” for enumeration types with 256 or fewer values, “double” for REAL types, “mtiTime64T” for TIME types, and “long” in all other cases. The first element of the array corresponds to the left bound of the array index range.

Multi-dimensional arrays are represented internally as arrays of arrays. For example, toto (x,y,z) is represented as toto (x)(y)(z). In order to get the values of the scalar subelements, you must use mti_GetSignalSubelements() or mti_GetVarSubelements() at each level of the array until you get to an array of scalars.

Note: A STRING data type is represented as an array of enumeration values. The array is not NULL terminated as you would expect for a C string, so you must call mti_TickLength() to get its length.

VHDL FLI examples

Several examples that illustrate how to use the foreign language interface along with an include file are shipped with ModelSim EE.

All example files are located in:

/<install_dir>/modeltech/examples/foreign/

You’ll find the include file at:

/<install_dir>/modeltech/mti.h

Example one uses the following VHDL source and C code files:

example1.vhd
example1.c

Example two uses one VHDL source code file and several C files:

foreign.vhd
dumpdes.c
gates.c
monitor.c

Example three is an entire VHDL testbed:

test_circuit.vhd
tester.vhd
xcvr.vhd
tester.c
vectors

Compiling and linking FLI applications

The following platform-specific instructions show you how to compile and link your FLI applications so they can be loaded by VSIM. Microsoft Visual C/C++ is supported for creating Windows DLLs while gcc and cc compiler instructions are shown for UNIX platforms.

Windows NT/95/98 platforms

Under Windows NT/95/98, VSIM loads a 32-bit dynamically linked library for each FLI or PLI application. The following compile and link steps are used to create the necessary.dll file (and other supporting files) using the Microsoft Visual C/C++ compiler.

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

Where <C_init_function> is the function name specified in the FOREIGN attribute (for FLI).

Note: The FLI interface has been tested with DLLs built using Microsoft Visual C/C++ compiler version 4.1 or greater.

Solaris platform

Under SUN Solaris, VSIM loads shared objects. Use these **gcc** or **cc** compiler commands to create a shared object:

gcc compiler:

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

cc compiler:

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

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

If *app.so* is in your current directory you must force Solaris to search the directory. There are two ways you can do this:

- Add “. / “ before *app.so* in the foreign attribute specification, or
- Load the path as a UNIX shell environment variable:
`LD_LIBRARY_PATH= <library path without filename>`

SunOS 4 platform

Under SUN OS4, VSIM loads shared objects. Use the following commands to create a shared object:

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

HP700 platform

VSIM loads shared libraries on the HP700 workstation. A shared library is created by creating object files that contain position-independent code (use the **+z** compiler option) and by linking as a shared library (use the **-b** linker option). Use these **gcc** or **cc** compiler commands:

gcc compiler:

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

cc compiler:

```
cc -c +z -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.

for HP-UX 11.0 users

If you are building the FLI module under HP-UX 11.0, you should not specify the "-lc" option to the invocation of ld, since this will cause an incorrect version of the standard C library to be loaded with the module.

In other words, build modules like this:

```
cc -c +z -I/<install_dir>/modeltech app.c
ld -b -o app.sl app.o
```

If you receive the error "Exec format error" when the simulator is trying to load an FLI module, then you have most likely built under 11.0 and specified the "-lc" option. Just rebuild without "-lc" (or rebuild on an HP-UX 9.0/10.0 machine).

IBM RISC/6000 platform

VSIM loads shared libraries on the IBM RS/6000 workstation. The shared library must import VSIM's C interface symbols and it must export the C initialization function. VSIM's export file is located in the ModelSim installation directory in *rs6000/mti_exports*.

If your foreign module uses anything from a system library, you'll need to specify that library when you link your foreign module. For example, to use the standard C library, specify '-lc' to the 'ld' command.

The resulting object must be marked as shared reentrant using the compiler option appropriate for your version of AIX:

for AIX 4.1

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

for AIX 4.2 (choose gcc or cc compiler commands)

gcc compiler:

```
gcc -c -I/<install_dir>/modeltech/include app.c
ld -bPT:0x100000000 -bPD:0x20000000 -btextro -bnoelcsect\
    -o app.sl app.o -bI:/<install_dir>/modeltech/rs6000/mti_exports\
    -bnoentry
```

cc compiler:

```
cc -c -I/<install_dir>/modeltech/include app.c
cc -o app.sl app.o -bE:app.exp\
    -bI:/<install_dir>/modeltech/rs6000/mti_exports\
    -Wl-G -bnoentry
```

The *app.exp* file must export the C initialization function named in the FOREIGN attribute.

Note: Although compilation and simulation switches are platform-specific, references to load shared objects are the same for all platforms. For information on loading objects see ["Declaring the FOREIGN attribute"](#) (12-298) and ["Declaring a subprogram in VHDL"](#) (12-300).

FLI tracing

The foreign interface tracing feature is available for tracing user foreign language calls made to the MTI VHDL FLI. 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 to replay what the foreign interface side did.

The purpose of tracing files

The purpose of the logfile is to aid you in debugging FLI code. The primary purpose of the replay facility is to send the replay file to MTI support for debugging co-simulation problems, or debugging FLI problems for which it is impractical to send the FLI code. MTI still would need the customer 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-208) with the **-trace_foreign** option:

Syntax

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

Arguments

<action>
Specifies one of the following actions:

Value	Action	Result
1	create log only	writes a local file called "mti_trace_<tag>"
2	create replay only	writes local files called "mti_data_<tag>.c", "mti_init_<tag>.c", "mti_replay_<tag>.c" and "mti_top_<tag>.c"

Value	Action	Result
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 VHDL foreign process call-backs and Verilog VCL call-backs. The miscellaneous VHDL call-backs (LoadComplete, Restart, Quit, EnvChanged, SimStatus, Save and Restore) are traced during execution but not explicitly identified as being from a callback function in the current product.

Note: Tracing does not work across checkpoint/restore operations.

13 - Value Change Dump (VCD) Files

Chapter contents

ModelSim VCD commands and VCD tasks	338
Resimulating a VHDL design from a VCD file	338
Extracting the proper stimulus for bidirectional ports	338
Specifying a filename and state mappings	339
Creating the VCD file	339
A VCD file from source to output	340
VHDL source code	340
VCD simulator commands	341
VCD output	341
Capturing port driver data with -dumpports	344

This chapter explains Model Technology's Verilog VCD implementation for *ModelSim*.

The VCD file format is specified in the IEEE 1364 standard. It is an ASCII file containing header information, variable definitions, and variable value changes. VCD is in common use for Verilog designs, and is controlled by VCD system task calls in the Verilog source code. *ModelSim* provides simulator command equivalents for these system tasks and extends VCD support to VHDL designs; the *ModelSim* commands can be used on either VHDL or Verilog designs.

VHDL VCD files may be used for resimulation with the ***vsim -vcdread*** command. See "[Resimulating a VHDL design from a VCD file](#)" (13-338).

Note: If you need vendor-specific ASIC design-flow documentation that incorporates VCD, please contact your ASIC vendor.

ModelSim VCD commands and VCD tasks

ModelSim VCD commands map to IEEE 1364 VCD system tasks and appear in the VCD file along with the results of those commands. The table below shows the mapping of the extended VCD commands to the IEEE 1364 keywords.

VCD commands	VCD system tasks
vcd add (CR-160)	\$dumpvars
vcd checkpoint (CR-161)	\$dumpall
vcd file (CR-163)	\$dumpfile
vcd flush (CR-165)	\$dumpflush
vcd limit (CR-166)	\$dumplimit
vcd off (CR-167)	\$dumpoff
vcd on (CR-168)	\$dumpon

In addition to the commands above, the **vcd comment** command (CR-162) can be used to add comments to the VCD file.

Resimulating a VHDL design from a VCD file

A VCD file intended for resimulation is created by capturing the ports of a VHDL design unit instance within a testbench or design. The following discussion shows you how to prepare a VCD file for resimulation. Note that the preparation varies depending on your design.

Extracting the proper stimulus for bidirectional ports

To extract the proper stimulus for bidirectional ports, the **splitio** command (CR-149) must be used before creating the VCD file. This splits bidirectional ports into separate signals that mirror the output driving contributions of their related ports. By recording in the VCD file both the resolved value of a bidirectional port and its output driving contribution, an appropriate stimulus can be derived by **vsim -vcdread**. The **splitio** command (CR-149) operates on a bidirectional port and creates a new signal having the same name as the port suffixed with "__o". This

new signal must be captured in the VCD file along with its related bidirectional port. See the description of the [splitio](#) command (CR-149) for more details.

Specifying a filename and state mappings

After using [splitio](#), the VCD filename and state mapping are specified using the [vcd file](#) command (CR-163) with the **-nomap** **-direction** options.

Note that the **-nomap** option is not necessary if the port types on the top-level design are bit or bit_vector. It is required, however, for std_logic ports because it records the entire std_logic state set. This allows the **-vcdread** option to duplicate the original stimulus on the ports.

The default VCD file is *dump.vcd*, but you can specify a different filename with [vcd file](#). For example,

```
vcd file mydumpfile.vcd -direction
```

Creating the VCD file

After invoking [vcd file](#) you can create the new VCD file by executing [vcd add](#) (CR-160) at the time you wish to begin capturing value changes. To dump everything in a design to a dump file you might use a command like this:

```
vcd add -r /*
```

At a minimum, the VCD file must contain the in and inout ports of the design unit. Value changes on all other signals are ignored by **-vcdread**. This also means that the simulation results are not checked against the VCD file.

After the VCD file is created, it can be input to [vsim](#) (CR-208) with the **-vcdread** option to resimulate the design unit stand-alone.

Example

The following example illustrates a typical sequence of commands to create a VCD file for input to **-vcdread**. Assume that a VHDL testbench named testbench instantiates dut with an instance name of u1, and that you would like to simulate testbench and later be able to resimulate dut stand-alone:

```
vsim -c -t ps testbench
VSIM 1> splitio /u1/*
VSIM 2> vcd file -nomap -direction
VSIM 3> vcd add -ports /u1/*
VSIM 4> run 1000
VSIM 5> quit
```

Now, to resimulate using the VCD file:

```
vsim -c -t ps -vcdread dump.vcd dut
VSIM 1> run 1000
VSIM 2> quit
```

Note: You must manually invoke the **run** command (CR-139) even when using **-vcdread**.

A VCD file from source to output

The following example shows the VHDL source, a set of simulator commands, and the resulting VCD output.

VHDL source code

The design is a simple shifter device represented by the following VHDL source code:

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;

entity SHIFTER_MOD is
    port (CLK, RESET, data_in : IN STD_LOGIC;
          Q : INOUT STD_LOGIC_VECTOR(8 downto 0));
END SHIFTER_MOD ;

architecture RTL of SHIFTER_MOD is
begin
    process (CLK,RESET)
    begin
        if (RESET = '1') then
            Q <= (others => '0') ;
        elsif (CLK'event and CLK = '1') then
            Q <= Q(Q'left - 1 downto 0) & data_in ;
        end if ;
    end process ;
end ;
```

VCD simulator commands

At simulator time zero, the designer executes the following commands and quits the simulator at time 1200:

```
vcd file output.vcd
vcd add -r *
force reset 1 0
force data_in 0 0

force clk 0 0
run 100
force clk 1 0, 0 50 -repeat 100
run 100
vcd off
force reset 0 0
force data_in 1 0
run 100
vcd on
run 850
force reset 1 0
run 50
vcd checkpoint
```

VCD output

The VCD file created as a result of the preceding scenario would be called *output.vcd*. The following pages show how it would look.

VCD output

\$comment	0'
File created using the following	0(
command:	0)
vcd file output.vcd	0*
\$date	0+
Fri Apr 12 09:07:17 1996	0,
\$end	\$end
\$version	#100
ModelSim EE/PLUS 5.1	1!
\$end	#150
\$timescale	0!
1ns	#200
\$end	1!
\$scope module shifter_mod \$end	\$dumpo
\$var wire 1 ! clk \$end	ff
\$var wire 1 " reset \$end	x!
\$var wire 1 # data_in \$end	x"
\$var wire 1 \$ q [8] \$end	x#
\$var wire 1 % q [7] \$end	x\$
\$var wire 1 & q [6] \$end	x%
\$var wire 1 ' q [5] \$end	x&
\$var wire 1 (q [4] \$end	x'
\$var wire 1) q [3] \$end	x(
\$var wire 1 * q [2] \$end	x)
\$var wire 1 + q [1] \$end	x*
\$var wire 1 , q [0] \$end	x+
\$upscope \$end	x,
\$enddefinitions \$end	\$end
#0	#300
\$dumpvars	\$dumpo
0!	n
1"	1!
0#	0"
0\$	1#
0%	0\$
0&	0%

0&	#1000	
0'	1!	
0(1%	
0)	#1050	
0*	0!	
0+	#1100	
1,	1!	
\$end	1\$	
#350	#1150	
0!	0!	
#400	1"	
1!	0\$	
1+	0%	
#450	0&	
0!	0'	
#500	0(
1!	0)	
1*	0*	
#550	0+	
0!	0,	
#600	#1200	
1!	1!	
1)	\$dumpa	
#650	11	
0!	1!	
#700	1"	
1!	1#	
1(0\$	
#750	0%	
0!	0&	
#800	0'	
1!	0(
1'	0)	
#850	0*	
0!	0+	
#900	0,	
1!	\$end	
1&		
#950		
0!		

Capturing port driver data with -dumpports

Some ASIC vendor’s toolkits read a VCD file format that provides details on port’s drivers. This information can be used, for example, to drive a tester. See the ASIC vendor’s documentation for toolkit specific information.

Execute the **vcd file** command (CR-163) with the **-dumpports** option to specify that you are capturing port driver data. Follow the **vcd file** command by **vcd add** (CR-160) on the ports you wish to capture.

Port driver direction information is captured as TSSI states in the VCD file. Each time an external or internal port driver changes values, a new value change is recorded in the VCD file with the following format:

```
p<TSSI state> <0 strength> <1 strength> <identifier_code>
```

Supported TSSI states

The supported <TSSI states> are:

Input (testfixture)
D low
U high
N unknown
Z tri-state

Output (dut)
L low
H high
X unknown
T tri-state

Unknown direction
0 low (both input and output are driving low)
1 high (both input and output are driving high)
? unknown (both input and output are driving unknown)
f tri-state
A unknown (input driving low and output driving high)
a unknown (input driving low and output driving unknown)
C unknown (input driving unknown and output driving low)
b unknown (input driving high and output driving unknown)
B unknown (input driving high and output driving low)
c unknown (input driving unknown and output driving high)

Strength values

The <strength> values are based on Verilog strengths:

Strength	VHDL std_logic mappings
0 highz	'Z'
1 small	
2 medium	
3 weak	
4 large	
5 pull	'W','H','L'
6 strong	'U','X','0','1','-'
7 supply	

Port identifier code

The <identifier_code> is an integer preceded by < that starts at zero and is incremented for each port in the order the ports are specified in the **vcd add** (CR-160). Also, the variable type recorded in the VCD header is "port".

Example VCD output from -dumpports

The following is an example VCD file created with the **-dumpports** option:

<pre> \$comment File created using the following command: vcd file results/dump1 -dumpports \$end \$date Tue Jan 20 13:33:02 1998 \$end \$version ModelSim Version 5.1c \$end \$timescale 1ns \$end \$scope module top1 \$end \$scope module u1 \$end \$var port 1 <0 a \$end \$var port 1 <1 b \$end \$var port 1 <2 c \$end \$upscope \$end \$upscope \$end \$enddefinitions \$end #0 \$dumpvars pN 6 6 <0 pX 6 6 <1 p? 6 6 <2 \$end #10 pX 6 6 <1 pN 6 6 <0 p? 6 6 <2 </pre>	<pre> #20 pL 6 0 <1 pD 6 0 <0 pa 6 6 <2 #30 pH 0 6 <1 pU 0 6 <0 pb 6 6 <2 #40 pT 0 0 <1 pZ 0 0 <0 pX 6 6 <2 #50 pX 5 5 <1 pN 5 5 <0 p? 6 6 <2 #60 pL 5 0 <1 pD 5 0 <0 pa 6 6 <2 #70 pH 0 5 <1 pU 0 5 <0 pb 6 6 <2 #80 pX 6 6 <1 pN 6 6 <0 p? 6 6 <2 </pre>
--	---

14 - Logic Modeling SmartModels

Chapter contents

VHDL SmartModel interface	348
Creating foreign architectures with <code>sm_entity</code>	349
Vector ports	352
Command channel	354
SmartModel Windows	354
Verilog SmartModel interface	357
Linking the LMTV interface to the simulator	357
Compiling Verilog shells	357

The Logic Modeling SWIFT-based SmartModel library may be used with ModelSim VHDL and Verilog. The SmartModel library is a collection of behavioral models supplied in binary form with a procedural interface that is accessed by the simulator. This chapter describes how to use the SmartModel library with ModelSim.

Note: The SmartModel library must be obtained from Logic Modeling along with the [SmartModel library documentation](#) that describes how to use it. This chapter only describes the specifics of using the library with ModelSim EE/SE.

VHDL SmartModel interface

ModelSim VHDL interfaces to a SmartModel through a foreign architecture. The foreign architecture contains a foreign attribute string that associates a specific SmartModel with the architecture. On elaboration of the foreign architecture, the simulator automatically loads the SmartModel library software and establishes communication with the specific SmartModel.

The *ModelSim* software locates the SmartModel interface software based on entries in the *modelsim.ini* initialization file. The simulator and the **sm_entity** tool (for creating foreign architectures) both depend on these entries being set correctly. These entries are found under the [lmc] section of the default *modelsim.ini* file located in the *ModelSim* installation directory. The default settings are as follows:

```
[lmc]

; ModelSim's interface to Logic Modeling's SmartModel SWIFT software
libsm = $MODEL_TECH/libsm.sl
; ModelSim's interface to Logic Modeling's SmartModel SWIFT software (Windows NT)
; libsm = $MODEL_TECH/libsm.dll
; Logic Modeling's SmartModel SWIFT software (HP 9000 Series 700)
; libswift = $LMC_HOME/lib/hp700.lib/libswift.sl
; Logic Modeling's SmartModel SWIFT software (IBM RISC System/6000)
; libswift = $LMC_HOME/lib/ibmrs.lib/swift.o
; Logic Modeling's SmartModel SWIFT software (Sun4 Solaris)
; libswift = $LMC_HOME/lib/sun4Solaris.lib/libswift.so
; Logic Modeling's SmartModel SWIFT software (Sun4 SunOS)
; do setenv LD_LIBRARY_PATH $LMC_HOME/lib/sun4SunOS.lib
; and run "vsim.swift".
; Logic Modeling's SmartModel SWIFT software (Windows NT)
; libswift = $LMC_HOME/lib/pcnt.lib/libswift.dll
```

The **libsm** entry points to the *ModelSim* dynamic link library that interfaces the foreign architecture to the SmartModel software. The **libswift** entry points to the Logic Modeling dynamic link library software that accesses the SmartModels. The simulator automatically loads both the **libsm** and **libswift** libraries when it elaborates a SmartModel foreign architecture.

By default, the **libsm** entry points to the *libsm.sl* supplied in the *ModelSim* installation directory indicated by the **MODEL_TECH** environment variable. *ModelSim* automatically sets the **MODEL_TECH** environment variable to the appropriate directory containing the executables and binaries for the current operating system. If you are running the Windows operating system, then you must comment out the default **libsm** entry (precede the line with the ";" character) and uncomment the **libsm** entry for the Windows operating system.

Uncomment the appropriate **libswift** entry for your operating system. The **LMC_HOME** environment variable must be set to the root of the SmartModel library installation directory. Consult Logic Modeling's [SmartModel library documentation](#) for details.

SPARCstation SunOS 4.x note

With SunOS 4.x, there are additional steps to take in order to simulate with the SmartModel library. First, add the path to the Logic Modeling SWIFT software to the **LD_LIBRARY_PATH** environment variable. For example,

```
setenv LD_LIBRARY_PATH $LMC_HOME/lib/sun4SunOS.lib
```

This is a necessary substitution for the **libswift** entry in the *modelsim.ini* initialization file in order to work around a SunOS 4.x bug. Also, you must invoke the simulator executable **vsim.swift** rather than the normal **vsim**.

Creating foreign architectures with sm_entity

The ModelSim **sm_entity** tool automatically creates entities and foreign architectures for SmartModels. Its usage is as follows:

Syntax

```
sm_entity
    [-] [-xe] [-xa] [-c] [-all] [-v] [-93] [<SmartModelName>...]
```

Arguments

<SmartModelName>
Name of a SmartModel (see the [SmartModel library documentation](#) for details on SmartModel names).

-
Read SmartModel names from standard input.

-xe
Do not generate entity declarations.

-xa
Do not generate architecture bodies.

-c
Generate component declarations.

- all
Select all models installed in the SmartModel library.
- v
Display progress messages.
- 93
Use extended identifiers where needed.

By default, the **sm_entity** tool writes an entity and foreign architecture to stdout for each SmartModel name listed on the command line. Optionally, you can include the component declaration (**-c**), exclude the entity (**-xe**), and exclude the architecture (**-xa**).

The simplest way to prepare SmartModels for use with ModelSim VHDL is to generate the entities and foreign architectures for all installed SmartModels, and compile them into a library named **lmc**. This is easily accomplished with the following commands:

```
% sm_entity -all > sml.vhd
% vlib lmc
% vcom -work lmc sml.vhd
```

To instantiate the SmartModels in your VHDL design, you also need to generate component declarations for the SmartModels. Add these component declarations to a package named **sml** (for example), and compile the package into the **lmc** library:

```
% sm_entity -all -c -xe -xa > smlcomp.vhd
```

Edit the resulting *smlcomp.vhd* file to turn it into a package of SmartModel component declarations as follows:

```
library ieee;
use ieee.std_logic_1164.all;
package sml is
    <component declarations go here>
end sml;
```

Compile the package into the **lmc** library:

```
% vcom -work lmc smlcomp.vhd
```

The SmartModels can now be referenced in your design by adding the following **library** and **use** clauses to your code:

```
library lmc;
use lmc.sml.all;
```

The following is an example of an entity and foreign architecture created by **sm_entity** for the cy7c285 SmartModel.

```

library ieee;
use ieee.std_logic_1164.all;

entity cy7c285 is
  generic (TimingVersion : STRING := "CY7C285-65";
    DelayRange : STRING := "Max";
    MemoryFile : STRING := "memory" );
  port ( A0 : in std_logic;
    A1 : in std_logic;
    A2 : in std_logic;
    A3 : in std_logic;
    A4 : in std_logic;
    A5 : in std_logic;
    A6 : in std_logic;
    A7 : in std_logic;
    A8 : in std_logic;
    A9 : in std_logic;
    A10 : in std_logic;
    A11 : in std_logic;
    A12 : in std_logic;
    A13 : in std_logic;
    A14 : in std_logic;
    A15 : in std_logic;
    CS : in std_logic;
    O0 : out std_logic;
    O1 : out std_logic;
    O2 : out std_logic;
    O3 : out std_logic;
    O4 : out std_logic;
    O5 : out std_logic;
    O6 : out std_logic;
    O7 : out std_logic;
    WAIT_PORT : inout std_logic );
end;

architecture SmartModel of cy7c285 is
  attribute FOREIGN : STRING;
  attribute FOREIGN of SmartModel : architecture is
    "sm_init $MODEL_TECH/libsm.sl ; cy7c285";
begin
end SmartModel;

```

Entity details

- The entity name is the SmartModel name (you may 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, then it 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 **vsim** 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 may also rename and reorder the ports in the component declaration. You can also reorder the ports in the entity declaration, but you can't rename them!

The following is an example component declaration and specification that groups the address and data ports of the CY7C285 SmartModel:

```

component cy7c285
  generic ( TimingVersion : STRING := "CY7C285-65";
            DelayRange : STRING := "Max";
            MemoryFile : STRING := "memory" );
  port ( A : in std_logic_vector (15 downto 0);
        CS : in std_logic;
        O : out std_logic_vector (7 downto 0);
        WAIT_PORT : inout std_logic );
end component;

for all: cy7c285
  use entity work.cy7c285
  port map (A0 => A(0),
            A1 => A(1),
            A2 => A(2),
            A3 => A(3),
            A4 => A(4),
            A5 => A(5),
            A6 => A(6),
            A7 => A(7),
            A8 => A(8),
            A9 => A(9),
            A10 => A(10),
            A11 => A(11),
            A12 => A(12),
            A13 => A(13),
            A14 => A(14),
            A15 => A(15),
            CS => CS,
            O0 => O(0),
            O1 => O(1),
            O2 => O(2),
            O3 => O(3),
            O4 => O(4),
            O5 => O(5),
            O6 => O(6),
            O7 => O(7),
            WAIT_PORT => WAIT_PORT );

```

Command channel

The command channel is a SmartModel feature that lets you invoke SmartModel specific commands. These commands are documented in the [SmartModel library documentation](#). **Vsim** provides access to the Command Channel from the **vsim** 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-80) in this manual). For example, to turn timing checks off for SmartModel */top/u1*:

```
lmc /top/u1 "SetConstraints Off"
```

Use **-all** to apply the command to all SmartModel instances. For example, to turn timing checks off for all SmartModel instances:

```
lmc -all "SetConstraints Off"
```

There are also some SmartModel commands that apply globally to the current simulation session rather than to models. The form of a SmartModel session command is:

```
lmcsession "<SmartModel session command>"
```

Once again, consult your [SmartModel library documentation](#) for details on these commands.

SmartModel Windows

Some models in the SmartModel library provide access to internal registers with a feature called SmartModel Windows. Refer to Logic Modeling's [SmartModel library documentation](#) for details on this feature. The simulator interface to this feature is described below.

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 may 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 may 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 may then be

referenced in other simulator commands just like any other signal (the **add list** command (CR-22) is 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.

LMTV usage documentation

The *SmartModel Library Simulator Interface Manual* is installed with Logic Modeling's software. Look for the file: `<LMC_install_dir>/doc/smartmodel/manuals/slim.pdf`. This document is written with Cadence Verilog in mind, but mostly applies to ModelSim Verilog. **Make sure you follow the instructions below for linking the LMTV interface to the simulator.**

Linking the LMTV interface to the simulator

Model Technology ships a dynamically loadable library that links ModelSim to the LMTV interface. To link to the LMTV all you need to do is add *libswiftpli.sl* to the **Veriuser** line in *modelsim.ini* as in the example below:

```
Veriuser = $MODEL_TECH/libswiftpli.sl
```

Compiling Verilog shells

Once *libswiftpli.sl* is in the *modelsim.ini* file you can compile the Verilog shells provided by Logic Modeling. You compile them just like any other Verilog modules in ModelSim Verilog. Details on the Verilog shells are in the *SmartModel Library Simulator Interface Manual* as well. The command line plus options and LMTV system tasks described in that document also apply to ModelSim.

Note: On sun4 you must run **vsim.swift** instead of **vsim**. Also, be sure to add `$LMC_HOME/lib/sun4SunOS.lib` to your **LD_LIBRARY_PATH** environment variable.

15 - Logic Modeling Hardware Models

Chapter contents

VHDL Hardware Model interface	360
Creating foreign architectures with <code>hm_entity</code>	361
Vector ports	364
Hardware model commands	366

Logic Modeling hardware models may be used with ModelSim VHDL and Verilog. A hardware model allows simulation of a device using the actual silicon installed as a hardware model in one of Logic Modeling's hardware modeling systems. The hardware modeling system is a network resource with a procedural interface that is accessed by the simulator. This chapter describes how to use Logic Modeling hardware models with ModelSim.

Note: Please refer to the [Logic Modeling documentation](#) for details on using the hardware modeler. This chapter only describes the specifics of using hardware models with ModelSim EE/PLUS.

VHDL Hardware Model interface

ModelSim VHDL interfaces to a hardware model through a foreign architecture. The foreign architecture contains a foreign attribute string that associates a specific hardware model with the architecture. On elaboration of the foreign architecture, the simulator automatically loads the hardware modeler software and establishes communication with the specific hardware model .

The *ModelSim* software locates the hardware modeler interface software based on entries in the *modelsim.ini* initialization file. The simulator and the **hm_entity** tool (for creating foreign architectures) both depend on these entries being set correctly. These entries are found under the [lmc] section of the default *modelsim.ini* file located in the *ModelSim* installation directory. The default settings are as follows:

```
[lmc]
; ModelSim's interface to Logic Modeling's hardware modeler SFI software
libhm = $MODEL_TECH/libhm.sl
; ModelSim's interface to Logic Modeling's hardware modeler SFI software (Windows NT)
; libhm = $MODEL_TECH/libhm.dll
; Logic Modeling's hardware modeler SFI software (HP 9000 Series 700)
; libsfi = <sfi_dir>/lib/hp700/libsfi.sl
; Logic Modeling's hardware modeler SFI software (IBM RISC System/6000)
; libsfi = <sfi_dir>/lib/rs6000/libsfi.a
; Logic Modeling's hardware modeler SFI software (Sun4 Solaris)
; libsfi = <sfi_dir>/lib/sun4.solaris/libsfi.so
; Logic Modeling's hardware modeler SFI software (Sun4 SunOS)
; libsfi = <sfi_dir>/lib/sun4.sunos/libsfi.so
; Logic Modeling's hardware modeler SFI software (Window NT)
; libsfi = <sfi_dir>/lib/pcnt/lm_sfi.dll
```

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 the [Logic Modeling documentation](#).

Creating foreign architectures with **hm_entity**

The ModelSim **hm_entity** tool automatically creates entities and foreign architectures for hardware models. Its usage is as follows:

Syntax

```
hm_entity  
    [-xe] [-xa] [-c] [-93] <shell software filename>
```

Arguments

<shell software filename>

Hardware model shell software filename (see [Logic Modeling documentation](#) for details on shell software files)

-xe

Do not generate entity declarations.

-xa

Do not generate architecture bodies.

-c

Generate component declarations.

-93

Use extended identifiers where needed.

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, then you may want to add all of their component declarations to a package so that you can easily reference them in your design. The following command writes the component declaration to stdout for the **LMTEST** hardware model.

```
% hm_entity -c -xe -xa LMTEST.MDL
```

Paste the resulting component declaration into the appropriate place in your design or into a package.

The following is an example of the entity and foreign architecture created by **hm_entity** for the CY7C285 hardware model:

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

entity cy7c285 is
  generic ( DelayRange : STRING := "Max" );
  port ( A0 : in std_logic;
        A1 : in std_logic;
        A2 : in std_logic;
        A3 : in std_logic;
        A4 : in std_logic;
        A5 : in std_logic;
        A6 : in std_logic;
        A7 : in std_logic;
        A8 : in std_logic;
        A9 : in std_logic;
        A10 : in std_logic;
        A11 : in std_logic;
        A12 : in std_logic;
        A13 : in std_logic;
        A14 : in std_logic;
        A15 : in std_logic;
        CS : in std_logic;
        O0 : out std_logic;
        O1 : out std_logic;
        O2 : out std_logic;
        O3 : out std_logic;
        O4 : out std_logic;
        O5 : out std_logic;
        O6 : out std_logic;
        O7 : out std_logic;
        W : inout std_logic );
end;

architecture Hardware of cy7c285 is
  attribute FOREIGN : STRING;
  attribute FOREIGN of Hardware : architecture is
    "hm_init $MODEL_TECH/libhm.sl ; CY7C285.MDL";
begin
end Hardware;
```

Entity details

- The entity name is the hardware model name (you may manually change this name if you like).
- The port names are the same as the hardware model port names (*these names must not be changed*). If the hardware model port name is not a valid VHDL identifier, then **hm_entity** issues an error message. If **hm_entity** is invoked with the **-93** option, then the identifier is converted to an extended identifier, and the resulting entity must also be compiled with the **-93** option. Another option is to create a pin-name mapping file. Consult the [Logic Modeling documentation](#) for details.
- The port types are **std_logic**. This data type supports the full range of hardware model logic states.
- The *DelayRange* generic selects minimum, typical, or maximum delay values. Valid values are "min", "typ", or "max" (the strings are not case-sensitive). The default is "max".

Architecture details

- The first part of the foreign attribute string (hm_init) is the same for all hardware model s.
- 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 may also rename and reorder the ports in the component declaration. You can also reorder the ports in the entity declaration, but you can't rename them!

The following is an example component declaration and specification that groups the address and data ports of the CY7C285 SmartModel:

```

component cy7c285
  generic ( TimingVersion : STRING := "CY7C285-65";
            DelayRange : STRING := "Max";
            MemoryFile : STRING := "memory" );
  port ( A : in std_logic_vector (15 downto 0);
        CS : in std_logic;
        O : out std_logic_vector (7 downto 0);
        WAIT_PORT : inout std_logic );
end component;

for all: cy7c285
  use entity work.cy7c285
  port map (A0 => A(0),
            A1 => A(1),
            A2 => A(2),
            A3 => A(3),
            A4 => A(4),
            A5 => A(5),
            A6 => A(6),
            A7 => A(7),
            A8 => A(8),
            A9 => A(9),
            A10 => A(10),
            A11 => A(11),
            A12 => A(12),
            A13 => A(13),
            A14 => A(14),
            A15 => A(15),
            CS => CS,
            O0 => O(0),
            O1 => O(1),
            O2 => O(2),
            O3 => O(3),
            O4 => O(4),
            O5 => O(5),
            O6 => O(6),
            O7 => O(7),
            WAIT_PORT => WAIT_PORT );

```

Hardware model commands

The following simulator commands are available for hardware models. Refer to the [Logic Modeling documentation](#) for details on these operations.

`lm_vectors on|off <instance_name> [<filename>]`

Enable/disable test vector logging for the specified hardware model.

`lm_measure_timing on|off <instance_name> [<filename>]`

Enable/disable timing measurement for the specified hardware model.

`lm_timing_checks on|off <instance_name>`

Enable/disable timing checks for the specified hardware model.

`lm_loop_patterns on|off <instance_name>`

Enable/disable pattern looping for the specified hardware model.

`lm_unknowns on|off <instance_name>`

Enable/disable unknown propagation for the specified hardware model.

16 - Tcl and ModelSim

Chapter contents

Tcl features within ModelSim	368
Tcl References	368
Tcl commands	369
Tcl command syntax	370
if command syntax	373
set command syntax	374
Command substitution	375
Command separator.	375
Multiple-line commands	375
Evaluation order	375
Tcl relational expression evaluation	375
Variable substitution	376
System commands	377
List processing	377
VSIM Tcl commands	378
ModelSim Tcl time commands	378
Tcl examples - UNIX only	380

This chapter provides an overview of Tcl (tool command language) as used with ModelSim. Additional Tcl and Tk (Tcl's toolkit) can be found through several [Tcl online references](#) (16-368).

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 VSIM. In addition, if VSIM 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)

Tcl References

Tcl print references

Two sources of information 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.

Tcl online references

The following are a few of the many Tcl references available:

- When using ModelSim, select **Help > Tcl Man Pages** from the Main window menus.
- Tcl man pages are also available at: www.elf.org/tcltk-man-html/contents.htm
- Tcl/Tk general information is available from the Tcl/Tk Consortium: www.tclconsortium.org
- The Scriptics Corporation, John Ousterhout's company (the original Tcl developer): www.scriptics.com.

Tcl tutorial

For some hands-on experience using Tcl with ModelSim, see the Lessons chapter of the *ModelSim EE/SE Tutorial*.

Tcl commands

The Tcl commands are listed below. For complete information on Tcl commands use the Main window menu selection: **Help > Tcl Man Pages**, or refer to one of the Tcl/Tk resources noted above. Also see ["Preference variables located in TCL files"](#) (B-406) for information on Tcl variables.

append	array	break	case	catch
cd	close	concat	continue	eof
error	eval	exec	expr	file
flush	for	foreach	format	gets
glob	global	history	if	incr
info	insert	join	lappend	list
llength	lindex	lrange	lreplace	lsearch
lsort	open	pid	proc	puts
pwd	read	regexp	regsub	rename
return	scan	seek	set	split
string	switch	tell	time	trace
source	unset	uplevel	upvar	while

Note: ModelSim command names that conflict with Tcl commands have been renamed or have been replaced by Tcl commands. See the list below:

Previous ModelSim command	Command changed to (or replaced by)
continue	run (CR-139) with the -continue option
format list wave	write format (CR-231) with either list or wave specified

Previous ModelSim command	Command changed to (or replaced by)
if	replaced by the Tcl if command, see "if command syntax" (16-373) for more information
list	add list (CR-22)
nolist nowave	delete (CR-62) with either list or wave specified
set	replaced by the Tcl set command, see "set command syntax" (16-374) for more information
source	vsource (CR-220)
wave	add wave (CR-33)

Tcl command syntax

The former ModelSim commands, **if** and **set** are now Tcl commands. An understanding of Tcl command syntax will help in your future use of these commands. The syntax, especially for the **if** command, may be unfamiliar.

The following rules define the syntax and semantics of the Tcl language. Both the [if command syntax](#) (16-373) and [set command syntax](#) (16-374) follow the general discussion of Tcl command syntax.

- 1 A Tcl script is a string containing one or more commands. Semi-colons and newlines are command separators unless quoted as described below. Close brackets are command terminators during command substitution (see below) unless quoted.
- 2 A command is evaluated in two steps. First, the Tcl interpreter breaks the command into words and performs substitutions as described below. These substitutions are performed in the same way for all commands. The first word is used to locate a command procedure to carry out the command, then all of the words of the command are passed to the command procedure. The command procedure is free to interpret each of its words in any way it likes, such as an integer, variable name, list, or Tcl script. Different commands interpret their words differently.
- 3 Words of a command are separated by white space (except for newlines, which are command separators).
- 4 If the first character of a word is double-quote (""") then the word is terminated by the next double-quote character. If semi-colons, close brackets, or white space characters (including newlines) appear between the quotes then they are treated as ordinary

characters and included in the word. Command substitution, variable substitution, and backslash substitution are performed on the characters between the quotes as described below. The double-quotes are not retained as part of the word.

- 5 If the first character of a word is an open brace ("{") then the word is terminated by the matching close brace ("}"). Braces nest within the word: for each additional open brace there must be an additional close brace (however, if an open brace or close brace within the word is quoted with a backslash then it is not counted in locating the matching close brace). No substitutions are performed on the characters between the braces except for backslash-newline substitutions described below, nor do semi-colons, newlines, close brackets, or white space receive any special interpretation. The word will consist of exactly the characters between the outer braces, not including the braces themselves.
- 6 If a word contains an open bracket ("[") then Tcl performs command substitution. To do this it invokes the Tcl interpreter recursively to process the characters following the open bracket as a Tcl script. The script may contain any number of commands and must be terminated by a close bracket ("]"). The result of the script (i.e. the result of its last command) is substituted into the word in place of the brackets and all of the characters between them. There may be any number of command substitutions in a single word. Command substitution is not performed on words enclosed in braces.
- 7 If a word contains a dollar-sign ("\$") then Tcl performs variable substitution: the dollar-sign and the following characters are replaced in the word by the value of a variable. Variable substitution may take any of the following forms:

\$name

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

\$name (index)

Name gives the name of an array variable and index gives the name of an element within that array. Name must contain only letters, digits, and underscores. Command substitutions, variable substitutions, and backslash substitutions are performed on the characters of index.

\${name}

Name is the name of a scalar variable. It may contain any characters whatsoever except for close braces.

There may be any number of variable substitutions in a single word. Variable substitution is not performed on words enclosed in braces.

- 8 If a backslash ("\") appears within a word then backslash substitution occurs. In all cases but those described below the backslash is dropped and the following character

is treated as an ordinary character and included in the word. This allows characters such as double quotes, close brackets, and dollar signs to be included in words without triggering special processing. The following table lists the backslash sequences that are handled specially, along with the value that replaces each sequence.

<code>\a</code>	Audible alert (bell) (0x7).
<code>\b</code>	Backspace (0x8).
<code>\f</code>	Form feed (0xc).
<code>\n</code>	Newline (0xa).
<code>\r</code>	Carriage-return (0xd).
<code>\t</code>	Tab (0x9).
<code>\v</code>	Vertical tab (0xb).
<code>\<newline>whiteSpace</code>	A single space character replaces the backslash, newline, and all spaces and tabs after the newline. This backslash sequence is unique in that it is replaced in a separate pre-pass before the command is actually parsed. This means that it will be replaced even when it occurs between braces, and the resulting space will be treated as a word separator if it isn't in braces or quotes.
<code>\\</code>	Backslash ("").
<code>\ooo</code>	The digits ooo (one, two, or three of them) give the octal value of the character.
<code>\xhh</code>	The hexadecimal digits hh give the hexadecimal value of the character. Any number of digits may be present.

Backslash substitution is not performed on words enclosed in braces, except for backslash-newline as described above.

- 9 If a hash character ("#") appears at a point where Tcl is expecting the first character of the first word of a command, then the hash character and the characters that follow it, up through the next newline, are treated as a comment and ignored. The comment character only has significance when it appears at the beginning of a command.
- 10 Each character is processed exactly once by the Tcl interpreter as part of creating the words of a command. For example, if variable substitution occurs then no further substitutions are performed on the value of the variable; the value is inserted into the word verbatim. If command substitution occurs then the nested command is processed entirely by the recursive call to the Tcl interpreter; no substitutions are performed before making the recursive call and no additional substitutions are performed on the result of the nested script.
- 11 Substitutions do not affect the word boundaries of a command. For example, during variable substitution the entire value of the variable becomes part of a single word, even if the variable's value contains spaces.

if command syntax

The Tcl **if** command executes scripts conditionally. Note that in the syntax below the "?" indicates an optional argument.

Syntax

```
if expr1 ?then? body1 elseif expr2 ?then? body2 elseif ... ?else? ?bodyN?
```

Description

The **if** command evaluates *expr1* as an expression. The value of the expression must be a boolean (a numeric value, where 0 is false and anything is true, or a string value such as **true** or **yes** for true and **false** or **no** for false); if it is true then *body1* is executed by passing it to the Tcl interpreter. Otherwise *expr2* is evaluated as an expression and if it is true then *body2* is executed, and so on. If none of the expressions evaluates to true then *bodyN* is executed. The **then** and **else** arguments are optional "noise words" to make the command easier to read. There may be any number of **elseif** clauses, including zero. *BodyN* may also be omitted as long as **else** is omitted too. The return value from the command is the result of the body script that was executed, or an empty string if none of the expressions was non-zero and there was no *bodyN*.

set command syntax

The Tcl **set** 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 set the value of *varName* to *value*, creating a new variable if one doesn't already exist, and return 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.

More Tcl commands

All Tcl commands are documented from within *ModelSim*. Select **Help > Tcl Help** from the **Main** window menus.

Command substitution

Placing a command in square brackets [] will cause that command to be evaluated first and its results returned in place of the command. An example is:

```
set a 25
set b 11
set c 3
echo "the result is [expr ($a + $b)/$c]"
```

will output:

```
"the result is 12"
```

This feature allows VHDL variables and signals, and Verilog nets and registers to be accessed using:

```
[examine -<radix> name]
```

The %name substitution is no longer supported. Everywhere %name could be used, you now can use [examine -value -<radix> name] which allows the flexibility of specifying command options. The radix specification is optional.

Command separator

A semicolon character (;) works as a separator for multiple commands on the same line. It is not required at the end of a line in a command sequence.

Multiple-line commands

With Tcl, multiple-line commands can be used within macros and on the command line. The command line prompt will change (as in a C shell) until the multiple-line command is complete.

In the example below, note the way the opening brace { is at the end of the if and else lines. This is important because otherwise the Tcl scanner won't know that there is more coming in the command and will try to execute what it has up to that point, which won't be what you intend.

```
if { [exa sig_a] == "0011ZZ" } {  
    echo "Signal value matches"  
    do macro_1.do  
}  
else {  
    echo "Signal value fails"  
    do macro_2.do  
}
```

Evaluation order

An important thing to remember when using Tcl is that anything put in curly brackets {} is not evaluated immediately. This is important for if-then-else, procedures, loops, and so forth.

Tcl relational expression evaluation

When you are comparing values, the following hints may be useful:

- Tcl stores all values as strings, and will convert certain strings to numeric values when appropriate. If you want a literal to be treated as a numeric value, don't quote it.

```
if {[exa var_1] == 345}...
```

The following will also work:

```
if {[exa var_1] == "345"}...
```

- However, if a literal cannot be represented as a number, you *must* quote it, or Tcl will give you an error. For instance:

```
if {[exa var_2] == 001Z}...
```

will give an error.

```
if {[exa var_2] == "001Z"}...
```

will work okay.

- Don't quote single characters in single quotes:

```
if {[exa var_3] == 'X'}...
```

will give an error

```
if {[exa var_3] == "X"}...
```

will work okay.

- For the equal operator, you must use the C operator "==" . For not-equal, you must use the C operator "!=".

Variable substitution

When a \$<var_name> is encountered, the Tcl parser will look for variables that have been defined either by VSIM or by you, and substitute the value of the variable.

Note: Tcl is case sensitive for variable names.

To access environment variables, use the construct:

```
$env(<var_name>)  
echo My user name is $env(USER)
```

Environment variables can also be set using the env array:

```
set env(SHELL) /bin/csh
```

See "[Simulator state variables](#)" (B-420) for more information about VSIM-defined variables.

System commands

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

```
echo The date is [exec date]
```

List processing

In Tcl a "list" is a set of strings in curly braces separated by spaces. Several Tcl commands are available for creating lists, indexing into lists, appending to lists, getting the length of lists and shifting lists. These commands are:

Command syntax	Description
lappend var_name val1 val2 ...	appends val1, val2, etc. to list var_name
lindex list_name index	return the index-th element of list_name; the first element is 0
linsert list_name index val1 val2 ...	inserts val1, val2, etc. just before the index-th element of list_name
list val1, val2 ...	returns a Tcl list consisting of val1, val2, etc.
llength list_name	returns the number of elements in list_name
lrange list_name first last	returns a sublist of list_name, from index first to index last; first or last may be "end", which refers to the last element in the list
lreplace list_name first last val1, val2, ...	replaces elements first through last with val1, val2, etc.

Two other commands, **lsearch** and **lsort**, are also available for list manipulation. See the Tcl man pages (Main window: **Help > Tcl Man Pages**) for more information on these commands.

See also the ModelSim Tcl command: [lecho](#) (CR-92)

VSIM Tcl commands

These additional VSIM commands enhance the interface between Tcl and ModelSim. Only brief descriptions are provided here; for more information and command syntax see the "[ModelSim Commands](#)" (CR-9).

Command	Description
alias (CR-37)	creates a new Tcl procedure that evaluates the specified commands; used to create a user-defined alias
lecho (CR-92)	takes one or more Tcl lists as arguments and pretty-prints them to the VSIM Main window
lshift (CR-95)	takes a Tcl list as argument and shifts it in-place one place to the left, eliminating the 0th element
lsublist (CR-96)	returns a sublist of the specified Tcl list that matches the specified Tcl glob pattern
printenv (CR-114)	echoes to the VSIM Main window the current names and values of all environment variables

ModelSim Tcl time commands

ModelSim Tcl time commands make simulator-time-based values available for use within other Tcl procedures.

Time values may optionally contain a units specifier where the intervening space is also optional. If the space is present, the value must be quoted (e.g. 10ns, "10 ns"). Time values without units are taken to be in the UserTimeScale. Return values are always in the current Time Scale Units. All time values are converted to a 64-bit integer value in the current Time Scale. This means that values smaller than the current Time Scale will be truncated to 0.

Conversions

Command	Description
intToTime <intHi32> <intLo32>	converts two 32-bit pieces (high and low order) into a 64-bit quantity (Time in ModelSim is a 64-bit integer)

Command	Description
RealToTime <real>	converts a <real> number to a 64-bit integer in the current Time Scale
scaleTime <time> <scaleFactor>	returns the value of <time> multiplied by the <scaleFactor> integer

Relations

Command	Description
eqTime <time> <time>	evaluates for equal
neqTime <time> <time>	evaluates for not equal
gtTime <time> <time>	evaluates for greater than
gteTime <time> <time>	evaluates for greater than or equal
ltTime <time> <time>	evaluates for less than
lteTime <time> <time>	evaluates for less than or equal

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

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

Arithmetic

Command	Description
addTime <time> <time>	add time
divTime <time> <time>	64-bit integer divide
mulTime <time> <time>	64-bit integer multiply
subTime <time> <time>	subtract time

Tcl examples - UNIX only

The following Tcl/ModelSim example for UNIX shows how you can access system information and transfer it into VHDL variables or signals and Verilog nets or registers. When a particular HDL source breakpoint occurs, a Tcl function is called that gets the date and time and deposits it into a VHDL signal of type STRING. If a particular environment variable (DO_ECHO) is set, the function also echoes the new date and time to the transcript file by examining the VHDL variable.

Note: In a Windows environment, the Tcl **exec** command shown below will execute compiled files only, not system commands.

(in VHDL source):

```
signal datetime : string(1 to 28) := "                                ";# 28 spaces
```

(on VSIM command line or in macro):

```
proc set_date {} {
    global env
    set do_the_echo [set env(DO_ECHO)]
    set s [exec date]
    force -deposit datetime $s
    if {do_the_echo} {
        echo "New time is [examine -value datetime]"
    }
}
bp src/waveadd.vhd 133 {set_date; continue}
--sets the breakpoint to call set_date
```

This is an example of using the Tcl **while** loop to copy a list from variable a to variable b, reversing the order of the elements along the way:

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

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

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

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

```
set b ""
foreach i $a {
    set b [linsert $b 0 $i]
}
```

This example shows a list reversal as above, this time aborting on a particular element using the Tcl **break** command:

```
set b ""
foreach i $a {
    if {$i = "ZZZ"} break
    set b [linsert $b 0 $i]
}
```

This example is a list reversal that skips a particular element by using the Tcl **continue** command:

```
set b ""
foreach i $a {
    if {$i = "ZZZ"} continue
    set b [linsert $b 0 $i]
}
```

The last example is of the Tcl **switch** command:

```
switch $x {
    a {incr t1}
    b {incr t2}
    c {incr t3}
}
```


A - Technical Support, Updates, and Licensing

Appendix contents

ModelSim worldwide contacts	383
Technical support - by telephone	384
Technical support - electronic support services	384
Technical support - other channels	386
Updates	386
FLEXlm Licenses	386

Model*Sim* worldwide contacts

If you are viewing this document online, select a link for Model*Sim* sales and support from the table below, or see the sections that follow for additional detail.

Model Technology (www.model.com)		
Americas	Europe	Asia

Mentor Graphics (www.mentorg.com)	
North America 1-800-547-4303	Outside North America supportnetweb.mentorg.com

Annapolis Microsystems	www.annapmicro.com
Hewlett Packard EEsof	www.hp.com/go/hpeesof
Synplicity	www.synplicity.com

Technical support - by telephone

Mentor Graphics customers In North America

For customers who purchased products from Mentor Graphics in North America, and are under a current support contract, technical telephone support is available from the central SupportCenter by calling toll-free 1-800-547-4303. The coverage window is from 6:00am to 5:30pm Pacific Time, Monday through Friday, excluding Mentor Graphics holidays.

The more preliminary information customers can supply about a problem or issue at the beginning of the reporting process, the sooner the Software Support Engineer (SSE) can supply a solution or workaround. Information of most help to the SSE includes accurate operating system and software version numbers, the steps leading to the problem or crash, the first few lines of a traceback, and problem sections of the Procedural Interface code.

Mentor Graphics customers outside North America

For customers who purchased products from Mentor Graphics outside of North America, should contact their local support organization. A list of local Mentor Graphics SupportCenters outside North America can be found at supportnetweb.mentorg.com under "Connections", then "International Directory".

Model Technology customers worldwide

For customers who purchased from Model Technology, please contact Model Technology via the support line at 1-503-641-1340 from 8:00 AM to 5:00 PM Pacific Time. Be sure to have your server hostID or hardware security key authorization number handy.

Technical support - electronic support services

Mentor Graphics customers

Mentor Graphics Customer Support offers a SupportNet-Email server for North American and European companies that lets customers find product information or submit service requests (call logs) to the SupportCenter 24 hours a day, 365 days a year. The server will return a call log number within minutes. SSEs follow up on the call logs submitted through SupportNet-Email using the same process

as if a customer had phoned the SupportCenter. For more information about using the SupportNet-Email server, send a blank e-mail message to the following address: support_net@mentorg.com.

Additionally, customers can open call logs or search the Mentor TechNote database of solutions to try to find the answers to their questions by logging onto Mentor Graphics' Customer Support web home page at <http://supportnetweb.mentorg.com>.

To establish a SupportNet account for your site (both a site-based SupportNet-Web account and a user-based SupportNet Email account), please submit the following information: name, phone number, fax number, email address, company name, site ID. Within one business day, you will be provided with the account information for new registrations.

While all customers worldwide are invited to obtain a SupportNet-Web site login, the SupportNet-Email services are currently limited to customers who receive support from Mentor support offices in North America or Europe. If you receive support from Mentor offices outside of North America or Europe, please contact your local field office to obtain assistance for a technical-support issue.

Model Technology customers

Support questions may be submitted through the Model Technology online support form at: www.model.com. Model Technology customers may also email test cases to support@model.com; please provide the following information, in this format, in the body of your email message:

- Your name:
Company:
Email address (if different from message address):
Telephone:
FAX (optional):
- ModelSim product (EE or PE, and VHDL, VLOG, or PLUS):
- ModelSim Version:
(Use the Help About dialog box with Windows; type **vcom** for UNIX workstations.)
- Host operating system version:
- PC hardware security key authorization number:
- Host ID of license server for workstations:
- Description of the problem (please include the exact wording of any error messages):

Technical support - other channels

For customers who purchased *ModelSim* as part of a bundled product from an OEM, or VAR, support is available at the following:

- **Annapolis Microsystems**
email: wftech@annapmicro.com
telephone: 1-410-841-2514
web site: <http://www.annapmicro.com>
- **Hewlett Packard EEsof**
email: hpeesof_support@hp.com
telephone: 1-800-HPEESOF (1-800-473-3763)
web site: <http://www.hp.com/go/hpeesof>
- **Synplicity**
email: support@synplicity.com
telephone: 1-408-617-6000
web site: <http://www.synplicity.com>

Updates

Mentor customers: getting the latest version via FTP

You can ftp the latest EE or PE version of the software from the SupportNet site at <ftp://supportnet.mentorg.com/pub/mentortech/modeltech/>, instructions are there as well. A valid license file from Mentor Graphics is needed to uncompress the *ModelSim* EE files.

Model Technology customers: getting the latest version via FTP

You can ftp the latest version of the software from the web site at <ftp://ftp.model.com>. Instructions are there as well.

FLEXIm Licenses

Where to obtain your license

Mentor Graphics customers must contact their Mentor Graphics salesperson for *ModelSim* EE licensing. All other customers may obtain *ModelSim* licenses from Model Technology. Please contact Model Technology at license@model.com.

If you have trouble with licensing

Contact your normal technical support channel:

- [Technical support - by telephone](#) (A-384)
- [Technical support - electronic support services](#) (A-384)
- [Technical support - other channels](#) (A-386)

All customers: ModelSim licensing

ModelSim uses Globetrotter's FLEXIm license manager and files. Globetrotter FLEXIm license files contain lines that can be referred to by the word that appears first on the line. Each kind of line has a specific purpose and there are many more kinds of lines that MTI does not use.

If you are using Mentor Graphics licensing

Mentor Graphics provides licensing information in their online (Inform) documentation. See the *Mentor Graphics Licensing* chapter in the *Managing Mentor Graphics Software* document. In addition, Model Technology provides some basic Mentor Graphics licensing files. See the readme file in the MGLS-related directory at <ftp.model.com/pub/ee> for more information.

All customers: maintenance renewals and licenses

When maintenance is renewed, a new license file that incorporates the new maintenance expiration date will be automatically sent to you. If maintenance is not renewed, the current license file will still permit the use of any version of the software built before the maintenance expired until the stop date is reached.

All customers: license transfers and server changes

Model Technology and Mentor Graphics both charge a fee for server changes or license transfers. Contact sales@model.com for more information from Model Technology, or contact your local Mentor Graphics sales office for Mentor Graphics purchases.

Additional licensing details

A complete discussion of licensing is located in the *Start Here for ModelSim* guide. For an online version of *Start Here* check the ModelSim Main window Help menu for EE Documentation > Licensing & Support.

B - ModelSim Variables

Appendix contents

Variable settings report	390
Personal preferences	390
Returning to the original ModelSim defaults	390
Environment variables	391
Preference variables located in INI and MPF files	394
[Library] library path variables	395
[vcom] VHDL compiler control variables	395
[vlog] Verilog compiler control variables	397
[vsim] simulator control variables	398
[Imc] Logic Modeling variables	401
[Project] project file section (MPF files only)	402
Setting variables in INI / MPF files.	402
Reading variable values from the INI file	403
Preference variables located in TCL files	406
Viewing the default preference file (pref.tcl)	407
Preference variable arrays	407
Main window preference variables	411
Individual preference variables	411
The addons variable.	412
Setting Tcl preference variables	412
Simulator state variables.	420

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 (usually in .tcl files) and modify the appearance of the ModelSim GUI (usually in INI and MPF files).
- **simulator state variables**
Variables that provide feedback on the state of the current simulation.

Variable settings report

The **report** command (CR-131) returns a list of current settings for either the simulator state, or simulator control variables. Use the following commands at either the ModelSim or VSIM prompt:

```
report simulator state
report simulator control
```

Personal preferences

There are several preferences stored by ModelSim on a personal bases, independent of *modelsim.ini* or *modelsim.tcl* files. These preferences are stored in \$(HOME)/.modelsim on UNIX and in the Windows Registry under HKEY_CURRENT_USER\Software\Model Technology Incorporated\ModelSim.

- **cwd**
History of the last five working directories (pwd). This history appears in the Main window File menu.
- **phst**
Project History
- **pinit**
Project Initialization state (one of: Welcome | OpenLast | NoWelcome)
- **printersetup**
All setup parameters related to Printing (i.e., current printer, etc.)

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 VSIM 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 95/98 machines, and set through the System control panel on NT machines. For UNIX, the variables are typically found in the *.login* script. The LM_LICENSE_FILE variable is required, all others are optional.

ModelSim Environment Variables

Variable	Description
DOPATH	used by VSIM to search for simulator command files (do files); consists of a colon-separated (semi-colon for Windows) list of paths to directories; optional; this variable can be overridden by the DOPATH (B-412) .tcl file variable
EDITOR	specifies the editor to invoke with the edit command (CR-76)
HOME	used by VSIM to look for an optional graphical preference file and optional location map file; see: "Preference variables located in INI and MPF files" (B-394) and "Using location mapping" (E-449)
LM_LICENSE_FILE	used by the ModelSim license file manager to find the location of the license file; may be a colon-separated (semi-colon for Windows) set of paths, including paths to other vendor license files; REQUIRED; see: "Using the FLEXlm License Manager" (D-429)
MODEL_TECH	set by all ModelSim tools to the directory in which the binary executables reside; YOU SHOULD NOT SET THIS VARIABLE
MODEL_TECH_FPGA	used by the The FPGA Library Manager (10-274) to locate vendor files
MODEL_TECH_TCL	used by VSIM to find Tcl libraries for: Tcl/Tk 8.0, Tix, and VSIM; defaults to <install_dir>/.tcl; may be set to an alternate path
MGC_LOCATION_MAP	used by ModelSim tools to find source files based on easily reallocated "soft" paths; optional; see: "Using location mapping" (E-449); also see the Tcl variables: SourceDir (B-412), and SourceMap (B-412)
MTI_TF_LIMIT	limits the size of the VSOUT temp file (generated by the VSIM kernel); the value of the variable is the size of k-bytes; TMPDIR (below) controls the location of this file, STDOUT controls the name; default = 10, 0 = no limit

Variable	Description
PLIOBJS	used by VSIM to search for PLI object files for loading; consists of a space-separated list of file or path names; optional
STDOUT	the VSOUT temp file (generated by the simulator kernel) is deleted when the simulator exits; the file is not deleted if you specify a filename for VSOUT with STDOUT; specifying a name and location (use TMPDIR) for the VSOUT file will also help you locate and delete the file in event of a crash (an unnamed VSOUT file is not deleted after a crash either)
TMPDIR	specifies the path to a tempnam() generated file (VSOUT) containing all stdout from the simulation kernel; optional
MODELSIM	used by all ModelSim tools to find the <i>modelsim.ini</i> file; consists of a path including the file name; optional
MODELSIM_TCL	used by VSIM to look for an optional graphical preference file

Setting 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 95/98

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

Right-click the My Computer icon and select Properties, then select the Environment tab of the System Properties control panel. Add the new variable to these fields: Variable:MY_PATH and Value:\temp\work.

Click Set and Apply to initialize the variable (you don't need to restart NT).

Library mapping with environment variables

Once the **MY_PATH** variable is set, you can use it with the **vmap** command (CR-207) to add library mappings to the current *modelsim.ini* file.

If you're using the **vmap** command from DOS prompt type:

```
vmap MY_VITAL %MY_PATH%
```

If you're using **vmap** from 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 ModelSim/VSIM prompt, the *modelsim.ini* 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
```

Note: 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,

```
entity test is end;  
use std.textio.all;  
  
architecture only of test is  
  
begin  
  process  
    FILE in_file : text is in "$ENV_VAR_NAME";  
  begin  
    wait;  
  end process;  
end;
```

Environment variables may also be referenced from the ModelSim command line or in macros using the Tcl **env** array mechanism:

```
echo "$env(ENV_VAR_NAME)"
```

Removing temp files (VSOUT)

The *VSOUT* temp file is the communication mechanism between the simulator kernel and the ModelSim GUI. In normal circumstances the file is deleted when the simulator exits. If ModelSim crashes, however, the temp file must be deleted manually. Specifying the location of the temp file with **TMPDIR** (above) will help you locate and remove the file.

Note: 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** compiler or **vsim** simulator was invoked. **MODEL_Tech** is used by the other Model Technology tools to find the libraries.

Preference variables located in INI and MPF files

ModelSim initialization (INI) and project (MPF) files contain control variables that specify reference library paths, and compiler and simulator settings. When first created, the MPF project file includes all of the variables from the current *modelsim.ini* file plus an additional [Project] section for project-specific variables.

For information on creating ModelSim projects see *Chapter 3 - Projects and system initialization*.

The following tables list the variables by section, and in order of their appearance within the INI/MPF file:

INI and MPF file sections
[Library] library path variables (B-395)
[vcom] VHDL compiler control variables (B-395)
[vlog] Verilog compiler control variables (B-397)
[vsim] simulator control variables (B-398)
[lmc] Logic Modeling variables (B-401)
[Project] project file section (MPF files only) (B-402)

[Library] library path variables

Variable name	Value range	Purpose
std	any valid path; may include environment variables	sets path to the VHDL STD library; default is <install_dir>/../std
ieee	any valid path; may include environment variables	sets path to the library containing IEEE and Synopsys arithmetic packages; default is <install_dir>/../ieee
verilog	any valid path; may include environment variables	sets path to the library containing VHDL/Verilog type mappings; default is <install_dir>/../verilog
std_developerskit	any valid path; may include environment variables	sets path to the libraries for MGC standard developer's kit
synopsys	any valid path; may include environment variables	sets path to the accelerated arithmetic packages

[vcom] VHDL compiler control variables

Variable name	Value range	Purpose	Default
VHDL93	0, 1	if 1, turns on VHDL-1993	off (0)
Show_source	0, 1	if 1, shows source line containing error	off (0)
Show_VitalChecksWarnings	0, 1	if 0, turns off VITAL compliance-check warnings	on (1)
Show_Warning1	0, 1	if 0, turns off unbound-component warnings	on (1)
Show_Warning2	0, 1	if 0, turns off process-without-a-wait-statement warnings; default is on	on (1)

[vcom] VHDL compiler control variables

Variable name	Value range	Purpose	Default
Show_Warning3	0, 1	if 0, turns off null-range warnings	on (1)
Show_Warning4	0, 1	if 0, turns off no-space-in-time-literal warnings	on (1)
Show_Warning5	0, 1	if 0, turns off multiple-drivers-on-unresolved-signal warnings	on (1)
Optimize_1164	0, 1	if 0, turns off optimization for IEEE std_logic_1164 package	on (1)
Explicit	0, 1	if 1, turns on resolving of ambiguous function overloading in favor of the "explicit" function declaration (not the one automatically created by the compiler for each type declaration)	off (0)
NoVitalCheck	0, 1	if 1, turns off VITAL compliance checking	off (0)
IgnoreVitalErrors	0, 1	if 1, ignores VITAL compliance checking errors	off (0)
NoDebug	0, 1	if 1, turns off inclusion of debugging info within design units	off (0)
NoVital	0, 1	if 1, turns off acceleration of the VITAL packages	off (0)
Hazard	0, 1	if 1, turns on Verilog hazard checking (order-dependent accessing of global vars)	off (0)
Quiet	0, 1	if 1, turns off "loading..." messages	off (0)
CheckSynthesis	0, 1	if 1, turns on limited synthesis rule compliance checking; checks only signals used (read) by a process	off (0)

[vcom] VHDL compiler control variables

Variable name	Value range	Purpose	Default
ScalarOpts	0, 1	if 1, activate optimizations on expressions that don't involve signals, waits or function/procedure/task invocations	off (0)

[vlog] Verilog compiler control variables

Variable name	Value range	Purpose	Default
Hazard	0, 1	if 1, turns on Verilog hazard checking (order-dependent accessing of global vars)	off (0)
NoDebug	0, 1	if 1, turns off inclusion of debugging info within design units	off (0)
Quiet	0, 1	if 1, turns off "loading..." messages	off (0)
ScalarOpts	0, 1	if 1, activate optimizations on expressions that don't involve signals, waits or function/procedure/task invocations	off (0)
Show_source	0, 1	if 1, shows source line containing error	off (0)
UpCase	0, 1	if 1, turns on converting regular Verilog identifiers to uppercase. Allows case insensitivity for module names; see also "Verilog-XL compatible compiler options" (5-70)	off (0)

[vsim] simulator control variables

Variable name	Value range	Purpose	Default
AssertFile	any valid filename	alternative file for storing assertion messages	commented out (;)
AssertionFormat	see purpose	sets the message to display after a break on assertion; message formats include: %S - severity level %R - report message %T - time of assertion %D - delta %I - instance or region pathname (if available) %% - print '%' character	*** %S: %R\n Time: %T Iteration: %D%I\n"
BreakOnAssertion	0-4	defines severity of assertion that causes a simulation break (0 = note, 1 = warning, 2 = error, 3 = failure, 4 = fatal)	3
CheckpointCompressMode	0, 1	if 1, checkpoint files are written in compressed format	on (1)
CommandHistory	any valid filename	set the name of a file to store the Main window command history	commented out (;)
ConcurrentFileLimit	any positive integer	controls the number of VHDL files open concurrently; this number should be less than the current ulimit setting for max file descriptors; 0 = unlimited	40
DatasetSeparator	any single character	the dataset separator for fully-rooted contexts, for example sim:/top; must not be the same character as PathSeparator	:
DefaultForceKind	freeze, drive, or deposit	defines the kind of force used when not otherwise specified	commented out (;)

[vsim] simulator control variables

Variable name	Value range	Purpose	Default
DefaultRadix	symbolic, binary, octal, decimal, unsigned, hexadecimal, ascii	any radix may be specified as a number or name, i.e., binary can be specified as binary or 2	symbolic
DelayFileOpen	0, 1	if 1, open VHDL87 files on first read or write, else open files when elaborated	off (0)
GenerateFormat	%s__%d	control the format of a generate statement label (don't quote it)	commented out (;)
IgnoreError	0,1	if 1, ignore assertion errors	off (0)
IgnoreFailure	0,1	if 1, ignore assertion failures	off (0)
IgnoreNote	0,1	if 1, ignore assertion notes	off (0)
IgnoreWarning	0,1	if 1, ignore assertion warnings	off (0)
IterationLimit	positive integer	limit on simulation kernel iterations during one time delta	5000
License	any single <license_option>	if set, controls ModelSim license file search; license options include: nomgc - excludes MGC licenses nomti - excludes MTI licenses vlog - only use VLOG license vhdl - only use VHDL license plus - only use PLUS license noqueue - do not wait in license queue if no license see also the vsim command (CR-208) <license_option>	search all licenses

[vsim] simulator control variables

Variable name	Value range	Purpose	Default
LockedMemory	positive integer; mb of memory to lock	for HP 10.2 UX use only; enables memory locking to speed up large designs (> 500mb memory footprint); see " Accelerate simulation by locking memory under HP-UX 10.2 " (E-451)	commented out (;)
NumericStdNoWarnings	0, 1	if 1, warnings generated within the accelerated numeric_std and numeric_bit packages are suppressed	off (0)
PathSeparator	any single character	used for hierarchical path names	/
Resolution	fs, ps, ns, us, ms, sec - also 10x and 100x	simulator resolution; default is ns; this value must be less than or equal to the UserTimeUnit specified below; NOTE - if your delays are truncated, set the resolution smaller	ns
RunLength	positive integer	default simulation length in units specified by the UserTimeUnit variable	100
Start up	= do <DO filename>; any valid macro (do) file	specifies the VSIM startup macro; default is commented out; see the do command (CR-67)	commented out (;)
StdArithNoWarnings	0, 1	if 1, warnings generated within the accelerated Synopsys std_arith packages are suppressed	off (0)
TranscriptFile	any valid filename	file for saving command transcript; environment variables may be included in the path name; default is "transcript"; the size of this file can be controlled with the MTL_TF_LIMIT (B-391)	transcript

[vsim] simulator control variables

Variable name	Value range	Purpose	Default
UnbufferedOutput	0, 1	controls VHDL files open for write; 0 = Buffered, 1 = Unbuffered	0
UserTimeUnit	fs, ps, ns, us, ms, sec, min, hr	specifies the default units to use for the "<timesteps> [<time_units>]" argument to the run command (CR-139); NOTE - the value of this variable must be set equal to, or larger than, the current simulator resolution specified by the Resolution variable shown above	ns
Veriuser	one or more valid shared object	list of dynamically loaded objects for Verilog PLI applications; see " Using the Verilog PLI " (5-90)	commented out (;)
WaveSignalNameWidth	0, positive or negative integer	controls the number of visible hierarchial regions of a signal name shown in the Wave window (10-212); the default value of zero displays the full name, a setting of one or above displays the corresponding level(s) of hierarchy	0

[lmc] Logic Modeling variables

Logic Modeling SmartModels and hardware modeler interface

ModelSim's interface with Logic Modeling's SmartModels and hardware modeler are specified in the **[lmc]** section of the INI/MPF file; for more information see "[VHDL SmartModel interface](#)" (14-348) and "" (14-357) respectively.

[Project] project file section (MPF files only)

Variable name	Value range	Purpose
Src_Files	a bracketed, space-separated list of source file paths in the form of: {<pathname>} {<path with spaces>}	list all source files that have not yet been compiled into a library; the bracketed list allows spaces within paths (as does Windows); see the example MPF file below
Cur_Top_DUs	*NONE* or list of one or more top-level design units (DUs) consisting of a space-separated list in the form of: <library>.<design_unit>	lists current or last top-level design unit(s) loaded; signifies that there is loadable design; *NONE* is the initial state and indicates that no auto design load should occur; a subset of Top_DUs
Top_DUs	one or more top level design units consisting of a space-separated list in the form of: <library>.<design_unit>	lists all top-level design units that could be loaded for simulation
Work_Libs	a bracketed, space-separated list of library paths in the form of: {<pathname>} {<path with spaces>}	complete list of all libraries local to this project; libraries are listed in compile order; the bracketed list allows spaces within paths (as does Windows); see the example MPF file below
<Lib>_script	path to a do script containing compiler instructions for recompiling a library <Lib>	the specified script is invoked to build the corresponding library <Lib>

Spaces in path names

For the Src_Files and Work_Libs variables, each element in the list is enclosed within curly braces ({}). This allows spaces inside elements (since Windows allows spaces inside path names). For example a source file list might look like:

```
Src_Files = {$MODELSIM_PROJECT/counter.v} {$MODELSIM_PROJECT/tb counter.v}
```

Where the file *tb counter.v* contains a space character between the "b" and "c".

Setting variables in INI / MPF files

Edit the initialization or project file directly with any text editor to change or add a variable. The syntax for variables in the file is:

```
<variable> = <value>
```

Comments within the file are preceded with a semicolon (;).

Note: The **vmap** command (CR-207) automatically modifies library mapping in the current INI / MPF file.

Reading variable values from the INI file

These Tcl functions allow you to read values from the *modelsim.ini* file.

```
GetIniInt <var_name> <default_value>
```

Reads the integer value for the specified variable.

```
GetIniReal <var_name> <default_value>
```

Reads the real value for the specified variable.

```
GetProfileString <section> <var_name> [<default>]
```

Reads the string value for the specified variable in the specified section. Optionally provides a default value if no value is present.

Setting Tcl variables with values from the *modelsim.ini* file is one use of these Tcl functions. For example,

```
set MyCheckpointCompressMode [GetIniInt "CheckpointCompressMode" 1]
```

```
set PrefMain(file) [GetProfileString vsim TranscriptFile ""]
```

Variable functions

Several, though not all, of the *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.

FuserExamples

```
[Library]
work = $HOME/work_lib
test_lib = ./$TESTNUM/work
...
[vsim]
IgnoreNote = $IGNORE_ASSERTS
IgnoreWarning = $IGNORE_ASSERTS
```

```
IgnoreError = 0
IgnoreFailure = 0
```

Tip:

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 compiler 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 the library section of the initialization file specified by the "others" clause.

Examples

```
[Library]
asic_lib = /cae/asic_lib
work = my_work
others = /install_dir/modeltech/modelsim.ini
```

Tip:

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.

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 VSIM history. The size of this file can be controlled with the [MTL_TF_LIMIT](#) (B-391).

```
; Save the command window contents to this file
TranscriptFile = trnsrpt
```

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 VSIM 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-67) for additional information on creating do files.

Turning off assertion messages

You can turn off assertion messages from your VHDL code by setting a switch in the *modelsim.ini* file. This option was added because some utility packages print a huge number of warnings.

```
[vsim]
IgnoreNote = 1
IgnoreWarning = 1
IgnoreError = 1
IgnoreFailure = 1
```

Messages may also be turned off with Tcl variables; see ["Preference variables located in TCL files"](#) (B-406).

Turning off warnings from arithmetic packages

You can disable warnings from the synopsys and numeric standard packages by adding the following lines to the [vsim] section of the *modelsim.ini* file.

```
[vsim]
NumericStdNoWarnings = 1
StdArithNoWarnings = 1
```

Warnings may also be turned off with Tcl variables; see ["Preference variables located in TCL files"](#) (B-406).

Force command defaults

The VSIM **force** command has **-freeze**, **-driver**, and **-deposit** options. When none of these is specified, then **-freeze** is assumed for unresolved signals and **-drive** is assumed for resolved signals. This is designed to provide compatibility with version 4.1 and earlier force files. But if you prefer **-freeze** as the default for both resolved and unresolved signals, you can change the defaults in the *modelsim.ini* file.

```
[vsim]
; Default Force Kind
; The choices are freeze, drive, or deposit
```

```
DefaultForceKind = freeze
```

VHDL93

You can make the VHDL93 standard the default by including the following line in the *INI* file:

```
[vcom]
; Turn on VHDL1993 as the default (default is 0)
VHDL93 = 1
```

Opening VHDL files

You can delay the opening of VHDL files with a entry in the *INI* file if you wish. Normally VHDL files are opened when the file declaration is elaborated. If the DelayFileOpen option is enabled, then the file is not opened until the first read or write to that file.

```
[vsim]
DelayFileOpen = 1
```

More preferences

Additional compiler and simulator preferences may be set in TCL files, see ["Preference variables located in TCL files"](#) (B-406).

Preference variables located in TCL files

ModelSim TCL preference variables give you control over fonts, colors, prompts, window positions and other simulator window characteristics. Preference files, which contain Tcl commands that set preference variables, are loaded before any windows are created, and so will affect all windows.

When ModelSim is invoked for the first time, default preferences are loaded from the *pref.tcl* file. Customized variable settings may be set from within the ModelSim GUI, on the ModelSim command line, or by directly editing the preference file.

The default file for customized preferences is *modelsim.tcl*. If your preference file is not named *modelsim.tcl*, you must refer to it with the [MODELSIM_TCL](#) (B-392) environment variable.

User-defined variables

Temporary user-defined variables can be created with the Tcl **set** command. Like simulator variables, user-defined variables are preceded by a dollar sign when referenced. To create a variable with the **set** command:

```
set user1 7
```

You can use the variable in a command like:

```
echo "user1 = $user1"
```

Viewing the default preference file (pref.tcl)

This documentation covers the categories of preference variables found in the *pref.tcl* file. You can open the file in any text editor to see a complete listing of the default preferences. See ["Setting Tcl preference variables"](#) (B-412) before you change any of the preferences.

Note: If you do open and edit a TCL file, make sure you save it as plain text, otherwise it will not be properly interpreted by ModelSim.

Preference variable arrays

Most preference variables are Tcl procedure lists (arrays), grouped by name within the TCL file. A unique array is defined for:

- all GUI defaults
- each VSIM window type
- the library browser
- the Code Coverage and Performance Analyzer windows
- logic value translations used in the [List window](#) (10-174) and [Wave window](#) (10-212)
- the **force** command (CR-87)

The most common variable array types are listed in the table below.

Variable array type	Description
PrefDefault(<argument>)	handles all default GUI preferences such as font attributes, menu features, and colors

Variable array type	Description
Pref<WindowName>(<argument>)	an array exists for each ModelSim window covering fonts, colors, and window-specific variables such as the (isTrigger) List window variable
Pref<WindowName>(geometry)	(geometry) variables are used for initial window positions with a new design; one variable for each window
PrefGeometry(<WindowName>)	<p>if you change the window positions and invoke this command:</p> <pre>write pref ./modelsim.tcl</pre> <p>this additional set of geometry preference variables is written to <i>modelsim.tcl</i> with a higher priority; on the next invocation of the vsim (CR-208) command you will get the newly-saved positions; (write pref saves all current preference setting)</p>
PrefLibrary(<argument>)	library preferences allow you to add color to different design units within the library browser

Variable array type	Description
<p>user_hook variables</p> <p>Pref<WindowName>(user_hook)</p> <p>There is also a user_hook variable defined for use in batch-mode:</p> <p>PrefBatch(user_hook)</p>	<p>The (user_hook) preference variable allows you to specify Tcl procedures to be called when a new window is created, or when the simulator is used in batch mode. Multiple procedures may be separated with a space.</p> <p>For window-specific user_hooks, each procedure added will be called after the window is created, with a single argument: the full Tk path name of the window that was just created. For example, if you invoke this command:</p> <pre>lappend PrefSource(user_hook) AddMyMenus</pre> <p>and create a new Source window, the procedure "AddMyMenus" will be called with argument ".source1" (the new Source window name). See the view (CR-180) command for information on creating a new window.</p> <p>The Tcl procedures listed in PrefSource(user_hook) will be executed once the Source window is open and completely initialized. There is a user_hook setting for each window type (i.e. Structure, Wave, List, etc.) It's important to note that the user_hook variable is a list of Tcl procedure names, so it is necessary to use the lappend command instead of the set command so that one does not inadvertently overwrite other extensions.</p> <p>User_hooks allow you to customize the ModelSim interface by adding or changing menus, menu options and buttons. See the example with the add_menu (CR-26) command for an example of a Tcl procedure that customizes the menus of a new window.</p>
ListTranslateTable(<argument>)	<p>ListTranslateTable specifies how various enumerations of various types map into the nine logic types that the List and Wave window know how to display. This mapping is used for vectors only; scalars are displayed with the original enum value. The following example values show that the std_logic_1164 types map in a one-to-one manner, and also shows mappings for boolean and Verilog types. You can add additional translations for your own user-defined types.</p>

Variable array type	Description
LogicStyleTable(<argument>)	LogicStyleTable variables allow you to control how each of the nine internal logic types are graphically displayed in the Wave window. For each of the nine internal logic types, a three-element Tcl list specifies: the line type, the line color, and the line vertical location; the line type may be Solid, OnOffDash, or DoubleDash. For vertical location, 0 is at the bottom of the waveform, 1 is at the middle, and 2 is at the top.
ForceTranslateTable(<argument>)	ForceTranslateTable is used only for vectors, and maps how a string of digits are mapped into enumerations. First, digits 0, 1, X and Z are mapped in the obvious way into LOGIC_0, LOGIC_1, LOGIC_X and LOGIC_Z. Then the ForceTranslateTable is used to map from there to the enumeration appropriate for the type of signal being forced.
PrefProfile(<argument>)	PrefProfile controls colors and display criteria for elements in the Performance Analyzer windows. For additional information on the ModelSim Performance Analyzer see <i>Chapter 7 - ModelSim SE Performance Analyzer</i>
PrefCoverage(<argument>)	PrefCoverage controls colors and display criteria for elements in the Code Coverage windows. For additional information on ModelSim Code Coverage see: <i>Chapter 8 - ModelSim SE Code Coverage</i>

Main window preference variables

The Main window uses preference variables similar to other *ModelSim* window to control colors and fonts. The variables below control some additional functions.

Individual preference variables

Variable	Description	Default
PrefMain(cmdHistory)	set the name of a file to store the Main window command history	history
PrefMain(DisplayDatasetPrefix)	turns dataset prefix viewing on or off, 1 displays the prefix and 0 turns the prefix off	1, or dataset prefix is displayed
PrefMain(file)	name of the file for saving transcript; an environment variable may be used	transcript
PrefMain(htmlBrowser)	UNIX only - set the path for the default html browser	/usr/local/netcape
PrefMain(linkWindows)	controls whether all <i>ModelSim</i> windows minimize with the Main window: 1 indicates windows are linked; 0 indicates windows are not linked (see note below this table). When linking is disabled, select Windows->Icon All to iconize all windows.	1 for Windows NT/98 0 for UNIX
PrefMain(prompt1)	used as primary prompt	{VSIM [history nextid]>}
PrefMain(prompt2)	prompt used when no design is loaded	"ModelSim> "
PrefMain(prompt3)	prompt used when macro is interrupted	"VSIM(paused)> "

Though most preference variables occur in arrays, the following individual variables are also found in TCL files.

Variable name	Value range	Purpose
DOPATH	a colon-separated list of paths to directories	used by VSIM to search for simulator command files (DO files); overrides the DOPATH (B-391) environment variable
PlotFilterResolution	0.1 +	specifies the output resolution for the waveform postscript file; default is 0.2, which equals 600dpi resolution; 0.1 equals 1200dpi; 0.4 equals 300dpi, etc.
SourceDir	any valid path	a list of alternate directories to search for source files; separate multiple paths with a colon
SourceMap	any valid path	a Tcl associative array for mapping a particular source file path (index) to another source file path (value)

The addons variable

In addition to window and batch user_hooks, the PrefVsim(addOns) variable provides a user_hook that allows you to integrate add-ons with VSIM. Refer to any vendor-supplied instructions for specifics about connecting third-party add-ons to VSIM.

Variable	Description	Example value
PrefVsim(addOns)	implicitly adds switches to VSIM invocation; useful for loading foreign libraries (see example value)	{-f "power_init \$MGC_HOME/lib/libpwr.dll" }

Setting Tcl preference variables

Preference variable within TCL file may be set in one of three ways.

- [Setting variables with the GUI](#) (B-413)
- [Setting preferences from the ModelSim command line](#) (B-418)
- [Directly editing preference files](#) (B-419)

Setting variables with the GUI

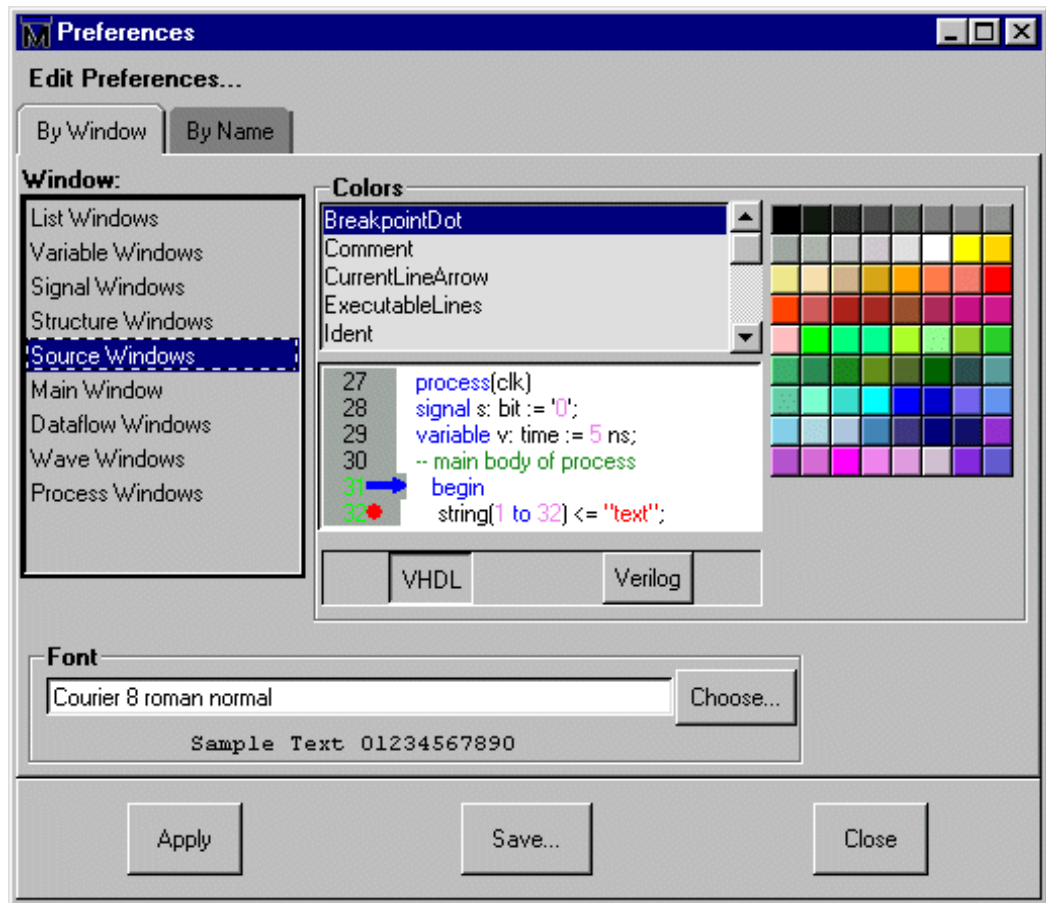
Use the **Main > Options > Edit Preferences** menu selection to open the Preferences dialog box shown below. Preference variable options include: window fonts, colors, window size and location (window geometry), and the user_hook variable. ModelSim "user_hook variables" (B-409) allow you to specify Tcl procedures to be called when a new window is created.

Use the Apply button to set temporary defaults for the current simulation. Use the Save button to write the preferences as permanent defaults to *modelsim.tcl*. Use the Close button to close the dialog box and return to the [Main window](#) (10-158) without making changes.

You can also use the **Main > Options > Save Preferences** menu selection to save current window settings to a tcl preference file.

The Preferences dialog box allows you to make preference changes **By Window** or **By Name** as shown below.

By Window page

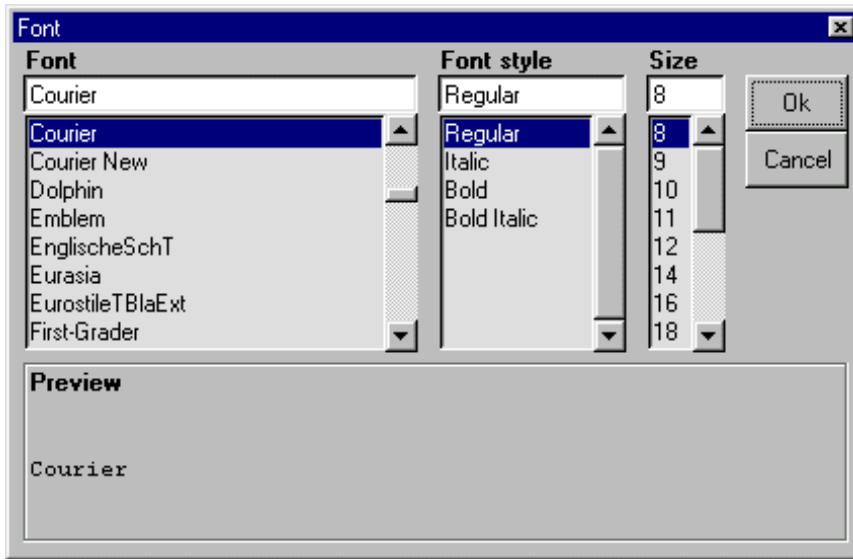


The **By Window** page includes these options:

- **Window**
Select the window type to modify; the color and font changes you make apply to all windows of this type. The Source window view allows you to preview source examples for either VHDL or Verilog. When you select the Source window, you can change preferences based on VHDL or Verilog source.
- **Colors**
Select the color element to change, and choose a color from the palette; view your changes in the sample graphic at the center of the window.

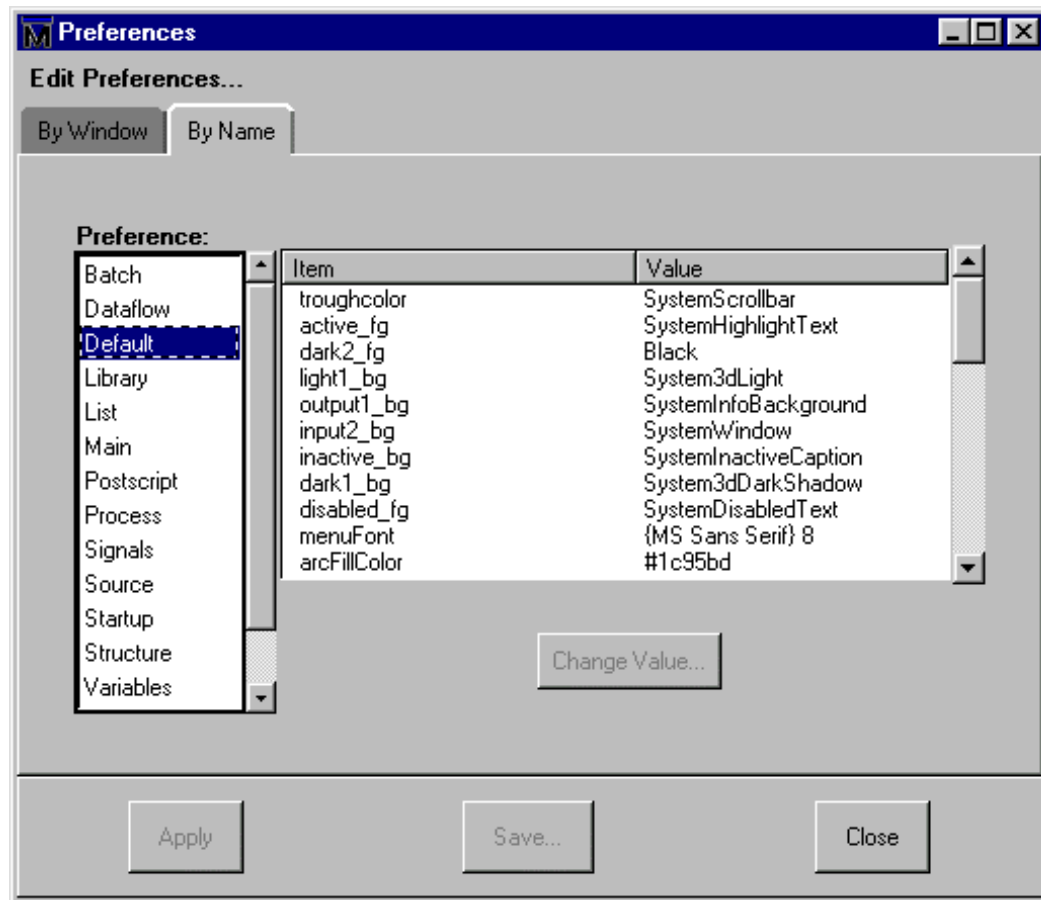
- **Font**

Select the Choose button; the Font Selection dialog box opens for your selection.



In the Font Selection dialog box, any selection you make automatically updates the By Window page; the **Reset** button scans your system for fonts and **Quit** closes the selection box.

By Name page



The **By Name** page includes these options:

- **Preference**

Select the preference to change and click the **Change Value** button. Enter the new value into the field provided in the resulting dialog box.

In addition to window preferences (listed by window name in the Preference list), you may also set:

- **Batch**

Specify the ["user_hook variables"](#) (B-409) to use in batch-mode simulation,

multiple procedures may be specified when separated by a space. See also, ["Batch mode"](#) (E-441).

- **Default**
Set default colors and fonts for menus and tree windows, also fill colors for VHDL (box) and Verilog (arc) structure symbols; these may be changed for individual windows.
- **Geometry**
Set the default size and position for the selected window type; used for the geometry of any newly-created window. This option will only exist after preferences have been saved for the first time (using the Save button).
- **ForceTranslateTable**
Specify how a string of digits are mapped into enumerations. This mapping is used for vectors only.
- **Library**
Set colors for design libraries and library elements: architectures, configurations, entities, modules, and packages.
- **ListTranslateTable**
Specify how various enumeration types map to the ModelSim's internal logic types. This mapping is used for vectors only.
- **LogicStyleTable**
Set the line type, color, and vertical location for internal logic types.
- **Postscript**
Specifies postscript font mapping. The fonts you specify for the Dataflow and Wave windows on the By Window page are mapped to these fonts when the Dataflow/Wave window is output to postscript. See ["Saving the Dataflow window as a Postscript file"](#) (10-173) or ["Printing and saving waveforms"](#) (10-238) for information on postscript output.
- **Startup**
Set the location (geometry) of the Load a Design dialog box.
- **Vsim**
This preference provides a user_hook variable to attach add-ons to ModelSim. See ["user_hook variables"](#) (B-409) for more information. This variable must be set prior to simulation. If it is set from this dialog box it will take effect the next time VSIM is invoked from the command line.

The By Name page is a graphic representation of the ["Preference variable arrays"](#) (B-407) located in the *modelsim.tcl* preference file. Setting a value in the Change a Preference Value dialog box invokes the Tcl **set** command as shown below in ["Setting preferences from the ModelSim command line"](#) (B-418).

The changes you make in the Preferences dialog box are temporary for the current simulation. Your changes can be saved as permanent defaults by using Main > Options > Save Preferences. The new settings are saved to the current directory in the *modelsim.tcl* file by default. You may choose a different name for the TCL file if you wish.

See "[Preference variable arrays](#)" (B-407) for a list of the standard preference variables to be found in the *modelsim.tcl* file.

Setting preferences from the ModelSim command line

In addition to the GUI, all preferences can be set from the ModelSim command line in the [Main window](#) (10-158). Note that if you save to a preference file other than *modelsim.tcl* you must refer to it with the [MODELSIM_TCL](#) (B-392) environment variable.

Set variables temporarily in the current environment with the **set** command:

```
set <variable name> <variable value>
```

User_hook variables can include multiple procedures; you can append additional procedures to a user_hook with the **lappend** command:

```
lappend <variable name>(user_hook) <Tcl procedure> ...
```

Note: Since the user_hook variable is a list of Tcl procedure names, it is important to use the **lappend** command instead of the **set** command so that one does not inadvertently overwrite other preferences.

Save all current preference settings as permanent defaults with the **write preferences** command:

```
write preferences <preference file name>
```

You can also modify variables by editing the preference file with the ModelSim [notepad](#) (CR-104):

```
notepad <preference file name>See also
```

See the Tcl man pages (Main window: Help > Tcl Man Pages) for more information on using the **set** command, **lappend** command and Tcl variables. For a complete list of preference variables see: "[Preference variable arrays](#)" (B-407). And for more information on user_hook variables see: "[user_hook variables](#)" (B-409).

Directly editing preference files

You can also modify variables by editing a preference file with any text editor. Make sure the file is saved as plain text. The ModelSim **notepad** (CR-104) provides a convenient way to edit a text file. At the ModelSim prompt type:

```
notepad <preference file path>
```

To edit the file once opened, change the read-only status with the Edit > read only menu selection.

Note: Case is significant in variable names, be careful when you edit variables!

More preferences

Additional compiler and simulator preferences may be set in the *modelsim.ini* and MPF files, see "[Preference variables located in INI and MPF files](#)" (B-394).

Preference variable loading order

ModelSim *.tcl*, INI, and MPF files all contain variables that are loaded when you start ModelSim. The files are evaluated for variable settings in the order below.

.tcl file variables are evaluated before the design is loaded

ModelSim evaluates *.tcl* files prior to loading a design for simulation. Any window "[user_hook variables](#)" (B-409) are evaluated after the associated window type is created.

- 1 The *<install_dir>/modeltech/tcl/vsim/pref.tcl* file is always loaded.
- 2 The file specified by the [MODELSIM_TCL](#) (B-392) environment variable is loaded next,
- 3 if MODELSIM_TCL does not exist, the *modelsim.tcl* in the current directory is evaluated,
- 4 or if MODELSIM_TCL and *./modelsim.tcl* do not exist, the file specified by the [HOME](#) (B-391) environment variable used.

INI and MPF file variables are evaluated after the design is loaded

After the design is loaded, *INI* or MPF file variables are found in these locations:

- 1 First the location specified by the **MODELSIM** (B-392) environment variable,
If no **MODELSIM** variable exists, *ModelSim* looks for MPF and INI files in the locations shown below. Project files (MPF) are evaluated first, if no project file is found, *ModelSim* looks for a INI file in the same location.
- 2 next in the current directory if no **MODELSIM** variable exists,
- 3 then in the directory where the executable exists (*/install_dir/modeltech/<platform>*),
- 4 finally in the parent of the directory where the executable is (*/install_dir/modeltech*).

Note: The **MODELSIM** variable is generally set to an INI file. Setting the variable to a MPF file is not recommended since the file would contain project-specific information. Setting the **MODELSIM** variable to a MPF file is only recommended for batch-mode usage.

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 a *ModelSim* DO files (macros).

Variable	Result
argc	returns the total number of parameters passed to the current macro
architecture	returns the name of the top-level architecture currently being simulated; for a configuration or Verilog module, this variable returns an empty string
configuration	returns the name of the top-level configuration currently being simulated; returns an empty string if no configuration
delta	returns the number of the current simulator iteration
entity	returns the name of the top-level VHDL entity or Verilog module currently being simulated
library	returns the library name for the current region
MacroNestingLevel	returns the current depth of macro call nesting
n	represents a macro parameter, where n can be an integer in the range 1-9

Variable	Result
Now	returns the current simulation time expressed in the current time resolution, i.e., 1000 ns
now	returns the current simulation time as an absolute number of time steps, i.e., 1000
resolution	returns the current simulation time resolution

Referencing simulator state variables

Variable values may be referenced in simulator commands by preceding the variable name with a \$ sign. For example, to use the **now** and **resolution** variables in an **echo** command type:

```
echo "The time is $now $resolution."
```

Depending on the current simulator state, this command could result in:

```
The time is 12390 10ps.
```

If you do not want the dollar sign to denote a simulator variable, precede it with a "\". For example, \\$now will not be interpreted as the current simulator time.

C - ModelSim Shortcuts

Appendix contents

Wave window keyboard shortcuts	423
List window keyboard shortcuts	424
Command shortcuts	425
Command history shortcuts	425
Editing the command line, the current source file, and notepads	426
Right mouse button	427

This appendix is a collection of the keyboard and command shortcuts available in the ModelSim GUI.

Wave window keyboard shortcuts

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

Key	Action
i I or +	zoom in
o O or -	zoom out
f or F	zoom full
l or L	zoom last
r or R	zoom range
<arrow up>	scroll waveform display up
<arrow down>	scroll waveform display down
<arrow left>	scroll waveform display left
<arrow right>	scroll waveform display right

Key	Action
<page up>	scroll waveform display up by page
<page down>	scroll waveform display down by page
<tab>	searches forward (right) to the next transition on the selected signal - finds the next edge
<shift-tab>	searches backward (left) to the previous transition on the selected signal - finds the previous edge
<control-f>	opens the find dialog box; search within the specified field in the wave-name pane for text strings

List window keyboard shortcuts

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

Key	Action
<arrow up>	scroll listing up
<arrow down>	scroll listing down
<arrow left>	scroll listing left
<arrow right>	scroll listing right
<page up>	scroll listing up by page
<page down>	scroll listing down by page
<tab>	searches forward (down) to the next transition on the selected signal
<shift-tab>	searches backward (up) to the previous transition on the selected signal (does not function on HP workstations)
<control-f>	opens the find dialog box; find the specified item label within the list display

Command shortcuts

You may abbreviate command syntax, but there's a catch. The minimum 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 maintain macro files as new commands are added.

Command history shortcuts

The simulator command history may be reviewed, or commands may be reused, with these shortcuts at the *ModelSim*/*VSIM* prompt:

Shortcut	Description
!!	repeats the last command
!n	repeats command number n; n is the VSIM prompt number, i.e., for this prompt: VSIM 12>, n =12
!abc	repeats the most recent command starting with "abc"
^xyz^ab^	replaces "xyz" in the last command with "ab"
up and down arrows	scroll through the command history with the keyboard arrows
click on prompt	left-click once on a previous ModelSim or VSIM prompt in the transcript to copy the command typed at that prompt to the active cursor
his or history	shows the last few commands (up to 50 are kept)

Editing the command line, the current source file, and notepads

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

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

Keystrokes - UNIX	Keystrokes - Windows	Result
< left right - arrow >		move the insertion cursor
< up down - arrow >		scroll through command history
< control - p >		move insertion cursor to previous line
< control - n >		move insertion cursor to next line
< control - f >		move insertion cursor forward
< control - b >		move insertion cursor backward
< backspace >		delete character to the left

Keystrokes - UNIX	Keystrokes - Windows	Result
< control - d >		delete character to the right
< control - k >		delete to the end of line
< control - a >	< control - a >, <Home>	move insertion cursor to beginning of line
< control - e >	< control - e >, <End>	move insertion cursor to end of line
< * meta - "<" >	none	move insertion cursor to beginning of file
< * meta - ">" >	none	move insertion cursor to end of file
< control - x >		cut selection
< control - c >		copy selection
< control - v >		insert clipboard

The Main window allows insertions or pastes only after the prompt, therefore, you don't need to set the cursor when copying strings to the command line.

Right mouse button

The right mouse button provides shortcut menus in the Main and Wave windows. In the Source window, the button gives you feedback on any HDL item under the cursor.

D - Using the FLEXlm License Manager

Appendix contents

Starting the license server daemon	430
Locating the license file.	430
Manual start.	430
Automatic start at boot time	431
What to do if another application uses FLEXlm	431
Format of the license file	432
Format of the daemon options file	432
License administration tools.	434
lmstat	434
lmdown	435
lmremove	435
lmreread.	436

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

Globetrotter Software’s Flexible License Manager (FLEXlm) is a network floating licensing package that allows the application to be licensed on a concurrent usage basis, as well as on a per-computer basis.

FLEXlm user’s manual

The content of this appendix is limited to the use of FLEXlm with Model Technology’s software. If you need additional information a complete user’s manual for FLEXlm is available at Globetrotter Software’s home page:

<http://www.globetrotter.com/manual.htm>

Starting the license server daemon

Locating the license file

When the license manager daemon is started, it must be able to find the license file. The default location is `/usr/local/flexlm/licenses/license.dat`. You can change where the daemon looks for the license file by one of two methods:

- By starting the license manager using the **-c <pathname>** option.
- By setting the [LM_LICENSE_FILE](#) (B-391) environment variable to the path of the file.

More information about installing ModelSim and using a license file is available in Model Technology's *Start Here for ModelSim* guide, see "[Where to find our documentation](#)" (1-27), or email us at license@model.com.

Controlling the license file search

By default, VSIM checks for the existence of both Model Technology and Mentor Graphics generated licenses. When VSIM is invoked it will first locate and use any available MTI licenses, then search for MGC licenses as needed. You can set one of the following **vsim** command (CR-208) switches to narrow the search to exclude or include specific licenses.

license option	Description
-lic_nomgc	excludes any MGC licenses from the search
-lic_nomti	excludes any MTI licenses from the search
-lic_vlog	searches only for ModelSim EE/VLOG licenses
-lic_vhdl	searches only for ModelSim EE/VHDL licenses
-lic_plus	searches only for ModelSim EE/PLUS licenses

The options may be specified with the [License](#) (B-399) variable in the *modelsim.ini* file; see the [\[vsim\] simulator control variables](#) (B-398).

Manual start

To start the license manager daemon, place the license file in the *modeltech* installation directory and enter the following commands:

```
cd /<install_dir>/modeltech/<platform>
lmgrd -c license.dat >& report.log
```

where `./<platform>` can be `sun4`, `sunos5`, `hp700`, or `rs6000`.

This can be done by an ordinary user; you do not need to be logged in as root.

Automatic start at boot time

You can cause the license manager daemon to start automatically at boot time by adding the following line to the file `/etc/rc.boot` or to `/etc/rc.local`:

```
/<install_dir>/modeltech/<platform>/lmgrd -c /<install_dir>/license.dat &
```

What to do if another application uses FLEXlm

If you have other applications that use FLEXlm, you can handle any conflict in one of the following ways:

Case 1: All the license files use the same license server nodes

You can combine the license files by taking the set of `SERVER` lines from one license file, and add all the `DAEMON`, `FEATURE`, and `FEATURESET` lines from all the license files. This combined file can be copied to `/<install_dir>/license/license.dat` and to any location required by the other applications.

Case 2: The applications use different license server nodes

You cannot combine the license files if the applications use different servers. Instead, set the `LM_LICENSE_FILE` (B-391) environment variable to be a list of files, as follows:

```
setenv LM_LICENSE_FILE \
  lic_file1:lic_file2:/<install_dir>/license.dat
```

Do not use the `-c` option when you start the license manager daemon. For example:

```
lmgrd > report.log
```

Format of the license file

The Model*Sim* EE license files contain three types of lines: SERVER lines, DAEMON lines, and FEATURE lines. For example:

```
SERVER hostname hostid [TCP_portnumber]
DAEMON daemon-name path-to-daemon [path-to-options-file]
FEATURE name daemon-name version exp_date #users_code \
    "description" [hostid]
```

Only the following items may be modified:

- the hostname on SERVER lines
- the TCP_portnumber on SERVER lines
- the path-to-daemon on DAEMON lines
- the path-to-options file on DAEMON lines
- anything in the daemon options file (described in the following section)

Format of the daemon options file

You can customize your use of Model*Sim* EE by using the daemon options file. This options file allows you to reserve licenses for specified users or groups of users, to allow or disallow access to Model*Sim* EE software to certain users, to set software time-outs, and to log activity to an optional report writer.

RESERVE

Ensures that Model*Sim* EE will always be available to one or more users on one or more host computers.

INCLUDE

Allows you to specify a list of users who are allowed access to the Model*Sim* EE software.

EXCLUDE

Allows you to disallow certain people to use Model*Sim* EE.

GROUP

Allows you to define a group of users for use in the other commands.

NOLOG

Causes messages of the specified type to be filtered out of the daemon's log output.

To use the daemon options capability, you must create a daemon options file and list its pathname as the fourth field on the line that begins with DAEMON modeltech.

A daemon options file consists of lines in the following format:

```
RESERVE number feature {USER | HOST | DISPLAY | GROUP} name
INCLUDE feature {USER | HOST | DISPLAY | GROUP} name
EXCLUDE feature {USER | HOST | DISPLAY | GROUP} name
GROUP name <list_of_users>
NOLOG {IN | OUT | DENIED | QUEUED}
REPORTLOG file
```

Lines beginning with the number character (#) are treated as comments. If the filename in the REPORTLOG line starts with a plus (+) character, the old report logfile will be opened for appending.

For example, the following options file would reserve a copy of the feature **vsim** for the user walter, three copies for the user john, a copy for anyone on a computer with the hostname of bob, and would cause QUEUED messages to be omitted from the logfile. The user rita would not be allowed to use the vsim feature.

```
RESERVE 1 vsim USER walter
RESERVE 3 vsim USER john
RESERVE 1 vsim HOST bob
EXCLUDE vsim USER rita
NOLOG QUEUED
```

If this data were in the file named:

```
/usr/local/options
```

modify the license file DAEMON line as follows:

```
DAEMON modeltech /<install_dir>/<platform>/modeltech \
/usr/local/options
```

License administration tools

lmstat

License administration is simplified by the **lmstat** utility. **lmstat** allows a user of FLEXlm to instantly monitor the status of all network licensing activities. **lmstat** allows a system administrator at a user site to monitor license management operations, including:

- which daemons are running;
- which users are using individual features; and
- which users are using features served by a specific DAEMON.

The case-sensitive syntax is shown below:

Syntax

```
lmstat
-a -A
-S <daemon>
-c <license_file>
-f <feature_name>
-s <server_name>
-t <value>
```

Arguments

- a
Displays everything.
- A
Lists all active licenses.
- c <license_file>
Specifies that the specified license file is to be used.
- S <daemon>
Lists all users of the specified daemon's features.
- f <feature_name>
Lists users of the specified feature(s).

`-s <server_name>`
Displays the status of the specified server node(s).

`-t <value>`
Sets the lmstat time-out to the specified value.

lmdown

The **lmdown** utility allows for the graceful shutdown of all license daemons (both **lmgrd** and all vendor daemons) on all nodes.

Syntax

```
lmdown  
-c [<license_file_path>]
```

If not supplied here, the license file used is in either `/user/local/flexlm/licenses/license.dat`, or the license file pathname in the environment variable [LM_LICENSE_FILE](#) (B-391).

The system administrator should protect the execution of **lmdown**, since shutting down the servers will cause loss of licenses.

lmremove

The **lmremove** utility allows the system administrator to remove a single user's license for a specified feature. This could be required in the case where the licensed user was running the software on a node that subsequently crashed. This situation will sometimes cause the license to remain unusable. **lmremove** will allow the license to return to the pool of available licenses.

Syntax

```
lmremove  
-c <file> <feature> <user> <host> <display>
```

lmremove removes all instances of user on the node host (on the display, if specified) from usage of feature. If the optional **-c <file>** switch is specified, the indicated file will be used as the license file. The system administrator should protect the execution of **lmremove**, since removing a user's license can be disruptive.

lmreread

The **lmreread** utility causes the license daemon to reread the license file and start any new vendor daemons that have been added. In addition, all preexisting daemons will be signaled to reread the license file for changes in feature licensing information.

Syntax

```
lmreread [daemon]  
        [-c <license_file>]
```

Note: If the **-c** option is used, the license file specified will be read by the daemon, not by **lmgrd**. **lmgrd** rereads the file it read originally. Also, **lmreread** cannot be used to change server node names or port numbers. Vendor daemons will not reread their option files as a result of **lmreread**.

E - Tips and Techniques

Appendix contents

How to use checkpoint/restore	438
How to use checkpoint/restore	438
Passing parameters to macros	442
Source code security and -nodebug	443
Saving and viewing waveforms	445
Setting up libraries for group use	445
Bus contention checking.	446
Bus float checking	447
Design stability checking	447
Toggle checking	447
Detecting infinite zero-delay loops	448
Referencing source files with location maps	449
Accelerate simulation by locking memory under HP-UX 10.2	451
Modeling memory in VHDL	452
Setting up a List trigger with Expression Builder	456

This appendix is an effort to organize information to make it more accessible. We've collected documentation from several parts of the manual; some examples have evolved from answers to questions received by tech support. Your suggestions, tips, and techniques for this section would be appreciated.

How to use checkpoint/restore

The **checkpoint** (CR-53) and **restore** (CR-134) commands will save and restore the simulator state within the same invocation of **vsim** or between **vsim** sessions.

If you want to **restore** while running **vsim**, use the **restore** command (CR-134), this we call a "warm restore". If you want to start up **vsim** and restore a previously-saved checkpoint, use the **-restore** switch with the **vsim** command (CR-208), this we call a "cold restore".

Note: Checkpoint/restore allows a cold restore, followed by simulation activity, followed by a warm restore back to the original cold-restore checkpoint file. Warm restores to checkpoint files that were not created in the current run are not allowed except for this special case of an original cold restore file.

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

- simulation kernel state
- *vsim.wav* 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

Things that are NOT restored are:

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

In order to save the simulator state, use the **vsim** command

```
checkpoint <filename>
```

To restore the simulator state during the same session as when the state was saved, use the **vsim** command:

```
restore <filename>
```

To restore the state after quitting ModelSim, invoke **vsim** as follows:

```
vsim -restore <filename> [-nocompress]
```

The checkpoint file is normally compressed. If there is a need to turn off the compression, you can do so by setting a special Tcl variable. Use:

```
set CheckpointCompressMode 0
```

to turn compression off, and turn compression back on with:

```
set CheckpointCompressMode 1
```

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

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

If you use the foreign interface, you will need to add additional function calls in order to use **checkpoint/restore**. See ["Using checkpoint/restore with the FLI"](#) (12-305) for more information.

The difference between checkpoint/restore and restarting

The **restart** (CR-133) command resets the simulator to time zero, clears out any logged waveforms, and closes any files opened under VHDL and the Verilog \$fopen system task. You can get the same effect by first doing a checkpoint at time zero and later doing a restore. But with **restart** you don't have to save the checkpoint and the **restart** is likely to be faster. But when you need to set the state to anything other than time zero, you will need to use **checkpoint/restore**.

Using macros with restart and checkpoint/restore

The **restart** (CR-133) 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-53) and later in the same session doing a **restore** (CR-134) of the earlier checkpoint. The **restore** does not touch the state of the macro interpreter so you may also do **checkpoint** and **restore** commands within macros.

Running command-line and batch-mode simulations

The typical method of running ModelSim is interactive: you push buttons and/or pull down menus in a series of windows in the GUI (graphic user interface). But there are really three specific modes of **vsim** operation: GUI, command line, and batch. Here are their characteristics:

- **GUI mode**

This is the usual interactive mode; it has graphical windows, push-button menus, and a command line in the text window. This is the default mode when **vsim** is invoked from within ModelSim.

- **Command-line mode**

This is an operational mode that has only an interactive command line; no interactive windows are opened. To run in this manner, invoke it with the **-c** option as the first argument.

- **Batch mode**

Batch mode is an operational mode that has neither an interactive command line nor interactive windows. **vsim** can be invoked in this mode by redirecting standard input using the UNIX “here-document” technique. Batch mode does not require the **-c** option. An example is:

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

Here is another example of batch mode, this time using a file as input:

```
vsim counter < yourfile
```

From a user's point of view, command-line mode can look like batch mode if you use the **vsim** command (CR-208) with the **-do** option to execute a macro that does a **quit -f** (CR-128) before returning, or if the startup.do macro does a **quit -f** before returning. But technically, that mode of operation is still command-line mode because stdin is still operating from the terminal.

The following paragraphs describe the behavior defined for the batch and command-line modes.

Command-line mode

In command-line mode *ModelSim* executes any startup command specified by the [Start up](#) (B-400) in the *modelsim.ini* file. If **vsim** (CR-208) 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-158) after the design loads (see the example below). The **transcript on** command will write all of the commands you invoke to the transcript file. For example, the following series of commands will result in a transcript file that can be used for command input if *top* is resimulated (remove the **quit -f** command from the transcript file if you want to remain in the simulator).

```
vsim -c top
```

library and design loading messages... then execute:

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

Note: 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 "#".

Batch mode

In batch mode *ModelSim* behaves much as in command-line mode except that there are no prompts, and commands from re-directed stdin are not echoed to stdout. Do not use the **-c** argument with **vsim** for batch mode simulations because **-c** invokes the command-line mode, which supplies the prompts and echoes the commands.

Tcl "[user_hook variables](#)" (B-409) may also be used for Tcl customization during batch-mode simulation; see also, "[Setting variables with the GUI](#)" (B-413).

Passing parameters to macros

In *ModelSim*, you invoke macros with the `do` command:

Syntax

```
do
    <filename> [ <parameter_value> ... ]
```

Arguments

`<filename>`

Specifies the name of the macro file to be executed.

`<parameter_value>`

Specifies values that are to be passed to the corresponding parameters \$1 through \$9 in the macro file. Multiple parameter values must be separated by spaces. If you specify fewer parameter values than the number of parameters used in the macro, the unspecified values are treated as empty strings in the macro.

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-147) to see the other parameters.

If you do not know how many parameters have been passed, you can use the **argc** variable; it returns the total number of currently-active parameters (i.e. the number of parameters passed less the number of [shift](#) (CR-147) executed).

See also

The [do](#) command (CR-67) for more information on do files. Also see the [DOPATH](#) (B-391) for adding a do file path to your environment.

Source code security and -nodebug

The **-nodebug** option on both **vdel** (CR-175) and **vlog** (CR-199) hides internal model data. This allows a model supplier to provide pre-compiled libraries without providing source code and without revealing internal model variables and structure.

Note: ModelSim's **-nodebug** compiler option provides protection for proprietary model information. The Verilog `'protect` compiler directive provides similar protection, but uses a Cadence encryption algorithm that is unavailable to Model Technology.

If a design unit is compiled with **-nodebug** the Source window will not display the design unit's source code, the Structure window will not display the internal structure, the Signals window will not display internal signals (it still displays ports), the Process window will not display internal processes, and the Variables window will not display internal variables. In addition, none of the hidden objects may be accessed through the Dataflow window or with ModelSim commands.

Even with the data hiding of **-nodebug**, there remains some visibility into models compiled with **-nodebug**. The names of all design units comprising your model are visible in the library, and you may invoke **vsim** (CR-208) directly on any of these design units and see the ports. Design units or modules compiled with **-nodebug** can only instantiate design units or modules that are also compiled **-nodebug**.

To restrict visibility into the lower levels of your design you can use the following **-nodebug** switches when you compile.

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

Note: Don't use the **=ports** switch on a design without hierarchy, or on the top level of a hierarchical design, if you do no ports will be visible for simulation. To properly use the switch, compile all lower portions of the design with -nodebug=ports first, then compile the top level with -nodebug alone.

Also note the =pli switch will not work with vcom (the VHDL compiler). PLI functions are valid only for Verilog design units.

Saving and viewing waveforms

You can run **vsim** as a batch job, but view the resulting waveforms later.

- 1 When you invoke **vsim** the first time, use the **-wav** option to rename the logfile, and redirect stdin to invoke the batch mode. The command should look like this:

```
vsim -wav wavesavl.wav counter < command.do
```

Within your *command.do* file, use the **log** command (CR-93) to save the waveforms you want to look at later, run the simulation, and quit.

When **vsim** runs in batch mode, it does not write to the screen, and can be run in the background.

- 2 When you return to work the next day after running several batch jobs, you can start up **vsim** in its viewing mode with this command and the appropriate *.wav* files:

```
vsim -view wavesavl.wav
```

Now you will be able to use the Waveform and List windows normally.

Setting up libraries for group use

By adding an “others” clause to your *modelsim.ini* file, you can have a hierarchy of library mappings. If the ModelSim tools don’t find a mapping in the *modelsim.ini* file, then they will search the library section of the initialization file specified by the “others” clause. For example:

```
[library]
asic_lib = /cae/asic_lib
work = my_work
others = /usr/modeltech/modelsim.ini
```

Bus contention checking

Bus contention checking detects bus fights on nodes that have multiple drivers. A bus fight occurs when two or more drivers drive a node with the same strength and that strength is the strongest of all drivers currently driving the node. The following table provides some examples for two drivers driving a std_logic signal:

driver 1	driver 2	fight
Z	Z	no
0	0	yes
1	Z	no
0	1	yes
L	1	no
L	H	yes

Detection of a bus fight results in an error message specifying the node and its drivers current driving values. If a node's drivers later change value and the node is still in contention, a message is issued giving the new values of the drivers. A message is also issued when the contention ends. The bus contention checking commands can be used on VHDL and Verilog designs.

These bus checking commands are in the ["ModelSim Commands"](#) (CR-9):

- **check contention add** (CR-45)
- **check contention config** (CR-46)
- **check contention off** (CR-47)

Bus float checking

Bus float checking detects nodes that are in the high impedance state for a time equal to or exceeding a user-defined limit. This is an error in some technologies. Detection of a float violation results in an error message identifying the node. A message is also issued when the float violation ends. The bus float checking commands can be used on VHDL and Verilog designs.

These bus float checking commands are in ["ModelSim Commands"](#) (CR-9):

- **check float add** (CR-48)
- **check float config** (CR-49)
- **check float off** (CR-50)

Design stability checking

Design stability checking detects when circuit activity has not settled within a period you define for synchronous designs. You specify the clock period for the design and the strobe time within the period during which the circuit must be stable. A violation is detected and an error message is issued if there are pending driver events at the strobe time. The message identifies the driver that has a pending event, the node that it drives, and the cycle number. The design stability checking commands can be used on VHDL and Verilog designs.

These design stability checking commands are in ["ModelSim Commands"](#) (CR-9):

- **check stable on** (CR-51)
- **check stable off** (CR-52)

Toggle checking

Toggle checking counts the number of transitions to 0 and 1 on specified nodes. Once the nodes have been selected, a toggle report may be requested at any time during the simulation. The toggle commands can be used on VHDL and Verilog designs.

These toggle checking commands are in ["ModelSim Commands"](#) (CR-9):

- **toggle add** (CR-154)
- **toggle reset** (CR-155)
- **toggle report** (CR-156)

Detecting infinite zero-delay loops

VHDL simulation uses steps that advance simulated time, and steps that do not advance simulated time. Steps that do not advance simulated time are called "delta cycles". Delta cycles are used when signal assignments are made with zero time delay.

If a large number of delta cycles 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 delta cycles that can occur. When the iteration_limit is exceeded, **vsim** stops the simulation and gives a warning message.

You can set the iteration_limit from the **Simulation > Properties** menu, by modifying the *modelsim.ini* file or by setting a Tcl variable called [IterationLimit](#) (B-399).

The iteration_limit default value is 5000.

When you get an iteration_limit warning, first increase the iteration limit and try to continue simulation. If the problem persists, look for zero-delay loops.

One approach to finding zero-delay loops is to increase the iteration limit again and start single stepping. You should be able to see the assignment statements or processes that are looping. Looking at the Process window will also help you to see the active looping processes.

When the loop is found, you will need to change the design to eliminate the unstable loop.

See ["Projects and system initialization"](#) (3-43) for more information on modifying the *modelsim.ini* file. And see ["Preference variables located in TCL files"](#) (B-406) for more information on Tcl variables. Also see the Main window Help menu for Tcl Help and man pages.

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` (B-391) 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; see, [SourceDir](#) (B-412), and [SourceMap](#) (B-412).

Accelerate simulation by locking memory under HP-UX 10.2

ModelSim 5.3 contains a feature to allow HP-UX 10.2 to use locked memory when using **vsim**. This feature provides significant acceleration of simulation time for large designs – i.e. with a memory footprint > 500Mb. (Test cases showed 2x acceleration of large simulations.) The following steps show how to set up the HP-UX 10.2 so memory can be locked.

- 1 Allow the average-user to lock memory. By default, this privilege is not allowed, so it has to be enabled. To allow everyone MLOCK privileges, the administrator needs to execute this command on the machine that will be running vsim:

```
/usr/sbin/setprivgrp -g MLOCK
```

To only allow a particular group MLOCK privileges, use the command:

```
/usr/sbin/setprivgrp <group-name> MLOCK
```

Note: This allows you to lock memory. No other privileges are enabled.

- 2 Once the MLOCK privilege is enabled, you merely have to modify the modelsim.ini file, and add the following entry to the [vsim] section:

```
LockedMemory = <some-value>
```

Where <some-value> is an integer representing the number of megabytes of memory to be locked. Once this is done, the memory will be locked when vsim invokes (using this .ini file).

Note: **vsim** will not lock more memory than is available in the system. The maximum memory that can be locked is: system physical memory (RAM) - 100 Mb = locked memory

Note: When vsim locks memory other processes will not have access to it. Therefore, you should consider how much memory is locked on a per-design basis to avoid locking more than is needed.

System parameters used for shared/locked memory may not be set (by default) high enough to take full advantage of this feature in later generations of HP-UX. Using the "sam" program, go to the "Configurable Parameters" window (under "Kernel Configuration"). There are several values that may need to be increased.

First, enable shared memory. The value for "shmem" should be equal to 1. Set the value for "shmmax" as large as possible (I set my value to over 1 Gb). The defaults for the values of "shmmni" and "shmseg" should be ok.

Note: To change these parameters, you have to rebuild the kernel and reboot.

Modeling memory in VHDL

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

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

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

A simple alternative implementation provides some excellent performance benefits:

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

The trick is to model memory using variables instead of signals.

In the example below, we illustrate three alternative architectures for entity "memory". Architecture "style_87_bad" uses a vhdl signal to store the ram data. Architecture "style_87" uses variables in the "memory" process, and architecture "style_93" uses variables in the architecture.

For large memories, architecture "style_87_bad" runs many times longer than the other two, and uses much more memory. This style should be avoided.

Both architectures "style_87" and "style_93" work with equal efficiency. You'll find some additional flexibility with the VHDL 1993 style, however, because the ram storage can be shared between multiple processes. For example, a second process is shown that initializes the memory; you could add other processes to create a multi-ported memory.

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

Example functions are provided below in package "conversions".

```
-----
-----
use std.standard.all;
library ieee;
use ieee.std_logic_1164.all;
use work.conversions.all;

entity memory is
    generic(add_bits : integer := 12;
            data_bits : integer := 32);
    port(add_in : in std_ulogic_vector(add_bits-1 downto 0);
          data_in : in std_ulogic_vector(data_bits-1 downto 0);
          data_out : out std_ulogic_vector
                        (data_bits-1 downto 0);
          cs, mwrite : in std_ulogic;
          do_init : in std_ulogic);
    subtype word is std_ulogic_vector(data_bits-1 downto 0);
    constant nwords : integer := 2 ** add_bits;
    type ram_type is array(0 to nwords-1) of word;

end;
```

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

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

```

        data_out <= ram(address);
    else
        data_out <= ram(address);
    end if;
end if;
end process;
end style_87;

architecture bad_style_87 of memory is
    -----
    signal ram : ram_type;
    -----
begin
memory:
process (cs)
    variable address : natural := 0;
    begin
        if rising_edge(cs) then
            address := sylv_to_natural(add_in);
            if (mwrite = '1') then
                ram(address) <= data_in;
                data_out <= data_in;
            else
                data_out <= ram(address);
            end if;
        end if;
    end process;
end bad_style_87;

-----
-----
use std.standard.all;
library ieee;
use ieee.std_logic_1164.all;

package conversions is
    function sylv_to_natural(x : std_ulogic_vector) return
        natural;
    function natural_to_sylv(n, bits : natural) return
        std_ulogic_vector;
end conversions;

package body conversions is

    function sylv_to_natural(x : std_ulogic_vector) return
        natural is

```

```
variable n : natural := 0;
variable failure : boolean := false;
begin
  assert (x'high - x'low + 1) <= 31
    report "Range of sulv_to_natural argument exceeds
      natural range"
    severity error;
  for i in x'range loop
    n := n * 2;
    case x(i) is
      when '1' | 'H' => n := n + 1;
      when '0' | 'L' => null;
      when others => failure := true;
    end case;
  end loop;
  assert not failure
    report "sulv_to_natural cannot convert indefinite
      std_ulogic_vector"
    severity error;

  if failure then
    return 0;
  else
    return n;
  end if;
end sulv_to_natural;

function natural_to_sulv(n, bits : natural) return
  std_ulogic_vector is
  variable x : std_ulogic_vector(bits-1 downto 0) :=
    (others => '0');
  variable tempn : natural := n;
begin
  for i in x'reverse_range loop
    if (tempn mod 2) = 1 then
      x(i) := '1';
    end if;
    tempn := tempn / 2;
  end loop;
  return x;
end natural_to_sulv;

end conversions;
```

Setting up a List trigger with Expression Builder

This example shows you how to set a List window trigger based on a gating expression created with the *ModelSim* Expression Builder.

If you want to look at a set of signal values **ONLY** during the simulation cycles during which an enable signal rises, you would need to use the List window Trigger Gating feature. The gating feature suppresses all display lines except those for which a specified gating function evaluates to true.

The easiest way to set it up is to open the **List > Prop > Display Props** dialog box.

Select on the **Trigger Gating: Expression** check box. Then click on **Use Expression Builder**. Select the signal in the List window that you want to be the enable signal by clicking on its name in the header area of the List window. Then click on **Add Selected_Signal(s)'rising** in the Expression Builder.

You should see the name of that signal plus "rising" added to the Expression lines in the Expression Builder and in the Expression entry box of the Modify Display Properties dialog box. (Leave the **On Duration** field zero for now.) Click on the Modify Display Properties **Apply** button.

If you already have simulation data in the List window, the display should immediately switch to showing only those cycles for which the gating signal is rising.

If that isn't quite what you want, you can go back to the expression builder and play with it until you get it right.

If you want the enable signal to work like a "One-Shot" that would display all values for the next, say 10 ns, after the rising edge of enable, then set the **On Duration** value to **10 ns**. Otherwise, leave it at zero, and select **Apply** again. When everything is correct, push **CLOSE** on the Expression Builder and **OK** on the Display Properties dialog box.

When you save the List window configuration, the list gating parameters will be saved as well, and can be set up again by reading in that macro. You can take a look at the macro to see how the gating can be set up using macro commands.

F - What's new in ModelSim 5.3

Appendix contents

New features	459
GUI changes and new features	461
New file types	461
Command and variable changes	462
Documentation changes	463
Find 5.2 features in the 5.3 interface	465

ModelSim 5.3 incorporates change on many levels, including a new, faster Wave window, performance analysis tools, new project and printing utilities, command enhancements, and reorganized documentation.

This reference organizes descriptions of these changes into six groups listed above. Links within the groups will connect you to more detail in the *ModelSim User's Manual*.

You will find references with the "CR" prefix in the new *ModelSim Command Reference*, i.e., [ModelSim Commands](#) (CR-9).

New features

What	Description	Where (select a link)	ModelSim Version
Performance Analyzer	Performance Analyzer is used to easily identify areas in your simulation where performance can be improved	ModelSim SE Performance Analyzer (7-119) The Performance Analyzer is a feature of the ModelSim Special Edition.	EE/SE

New features

What	Description	Where (select a link)	ModelSim Version
Code Coverage	Code Coverage gives you graphical and report file feedback on how the source code is being executed	ModelSim SE Code Coverage (8-133) The Performance Analyzer is a feature of the ModelSim Special Edition.	EE/SE
Wave window	new multi-pane window allows creation of signal groups and dividers, plus the addition of new panes for comparing previous simulations	Wave window (10-212)	ALL
virtual objects	expands the idea of a user-defined bus; allows creation of virtual signals, functions, regions, and types	Virtual signals (9-145), Virtual functions (9-146), Virtual regions (9-146), Virtual types (9-147)	ALL
ModelSim projects	an easy way to manage your simulation projects; what a ModelSim project is and how to create one	Projects and system initialization (3-43)	ALL
direct waveform printing	print to a file or to printer directly from the Wave window	Printing and saving waveforms (10-238)	ALL

GUI changes and new features

What	Description	Where (select a link)	ModelSim Version
Wave window (this is such an important feature we've listed it twice - this is the same entry as above)	new multi-pane window allows creation of signal groups and dividers, plus the addition of new panes for comparing previous simulations	Wave window (10-212)	ALL
View multiple logfiles	allows comparison of current and several previous simulations within the same Wave window	Multiple logfiles and datasets (9-139)	ALL
Quick Start utility	available when ModelSim is invoked; tips and information for new users, plus step-by-step examples for using <i>ModelSim</i>	Quick Start (10-264)	ALL
Dataset Browser	helps locate and load datasets for simulation comparison	Opening and viewing datasets (9-140)	ALL

New file types

What	Description	Where (select a link)	ModelSim Version
MPF files	new ModelSim Project File format	INI and MPF file comparison (3-44)	ALL
WLF files	new Wave Log File format	Multiple logfiles, datasets and virtuals (9-139)	ALL

Command and variable changes

What	Description	Where (select a link)	ModelSim Version
vsim command arguments	new -wavslim , and -wavtlim arguments	vsim command (CR-208)	ALL
environment command arguments	new -dataset and -nodataset arguments	environment command (CR-80)	ALL
force command arguments	-repeat argument clarification	force command (CR-87)	ALL
GUI expression format enhancements	new commands supported, naming conventions, and concatenation of signals	GUI_expression_format (CR-250)	EE
NEW - LockedMemory	for HP 10.2 UX use only; enables memory locking to speed up large designs (> 500mb memory footprint)	Accelerate simulation by locking memory under HP-UX 10.2 (E-451)	EE
NEW - ModelSim Tcl time commands	use in when statements	ModelSim Tcl time commands (16-378)	ALL
NEW - PrefMain(linkWindow)	controls whether all ModelSim windows minimize with the Main window (true) or work independently (false); default is true	Preference variables located in INI and MPF files (B-394)	ALL
NEW - project command	used to perform common operations on new ModelSim projects; also see " What is a project? " (3-44)	project command (CR-121)	ALL
NEW - searchLog command	searches one or more of the currently open logfiles for a specified condition	searchLog command (CR-144)	ALL

Command and variable changes

What	Description	Where (select a link)	ModelSim Version
NEW - time-based breakpoints	new arguments for the when command	when command (CR-226)	ALL
NEW - virtual commands	a set of commands to operate on ModelSim virtuals, also see " Virtual Objects (User-defined buses, and more) " (9-144)	nine new command beginning with the virtual delete describe define command (CR-182)	ALL
vlog command arguments	new arguments: +delay_mode_zero, +delay_mode_unit, +delay_mode_path, +delay_mode_distributed -incr, -noincr, -nologo, +mindelays, +maxdelays, +typdelays	vlog command (CR-199)	ALL
WaveSignalNameWidth	change function to show hierarchy levels	Preference variables located in INI and MPF files (B-394)	ALL

Documentation changes

What	Description	Where (select a link)	ModelSim Version
New <i>ModelSim Command Reference</i>	all ModelSim command and syntax documentation is now in a separate book, the <i>ModelSim Command Reference</i> ; references to commands are prefixed with CR, i.e., vsim command (CR-208)	"ModelSim Commands" (CR-9)	ALL

Documentation changes

What	Description	Where (select a link)	ModelSim Version
New Logic Hardware Modeler chapter	expanded to separate chapter with new material	Logic Modeling Hardware Models (15-359)	EE
New TNT example	Trigger Gating with Expression Builder	Setting up a List trigger with Expression Builder (E-456)	EE
New Variables chapter	Variables consolidated: environment, simulator state, simulator control and GUI preference variables	ModelSim Variables (B-389)	ALL
New VHDL and Verilog chapters	separate chapters for each language	VHDL Simulation (4-51) and Verilog Simulation (5-63)	ALL
Updated FLI chapter	completely updated descriptions	VHDL Foreign Language Interface (12-297)	EE
Updated Logic SmartModels chapter	expanded chapter with new material	Logic Modeling SmartModels (14-347)	EE

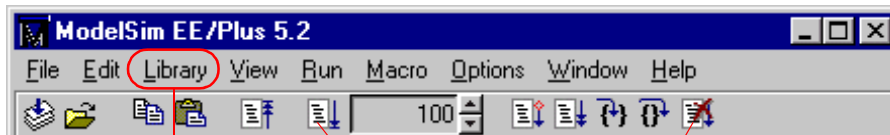
Find 5.2 features in the 5.3 interface

Locate differences between the ModelSim 5.2 and 5.3 interface with the feature list below. Follow links to more detail on the following pages and throughout the *ModelSim Reference Manual*.

Main window toolbar buttons and menu selections	466
Main window menu changes	467
File menu selections	467
View menu selections	469
Library menu selections	470
Run menu selections	471
Options menu selections	472
Window menu selections	473
Help menu selections	474
List window menu changes	475
Prop menu selections	475
Process window menu changes	476
File menu selections	476
Signals window menu changes	477
File menu selections	478
Variables window menu changes	480
Edit menu selections	480
Wave window menu changes	481
File menu selections	481
Edit menu selections	482
Prop menu selections	483

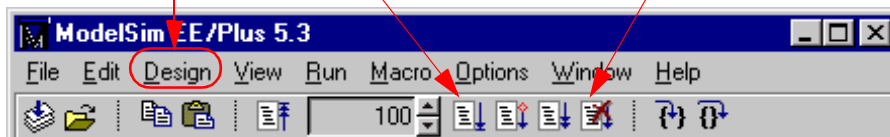
Main window toolbar buttons and menu selections

All 5.2 toolbar buttons can be found on the 5.3 Main window toolbar, but in different locations. In addition, the 5.2 Library menu has been changed in the 5.3 version to the Design menu (F-470).



5.2

ModelSim 5.2
Main window



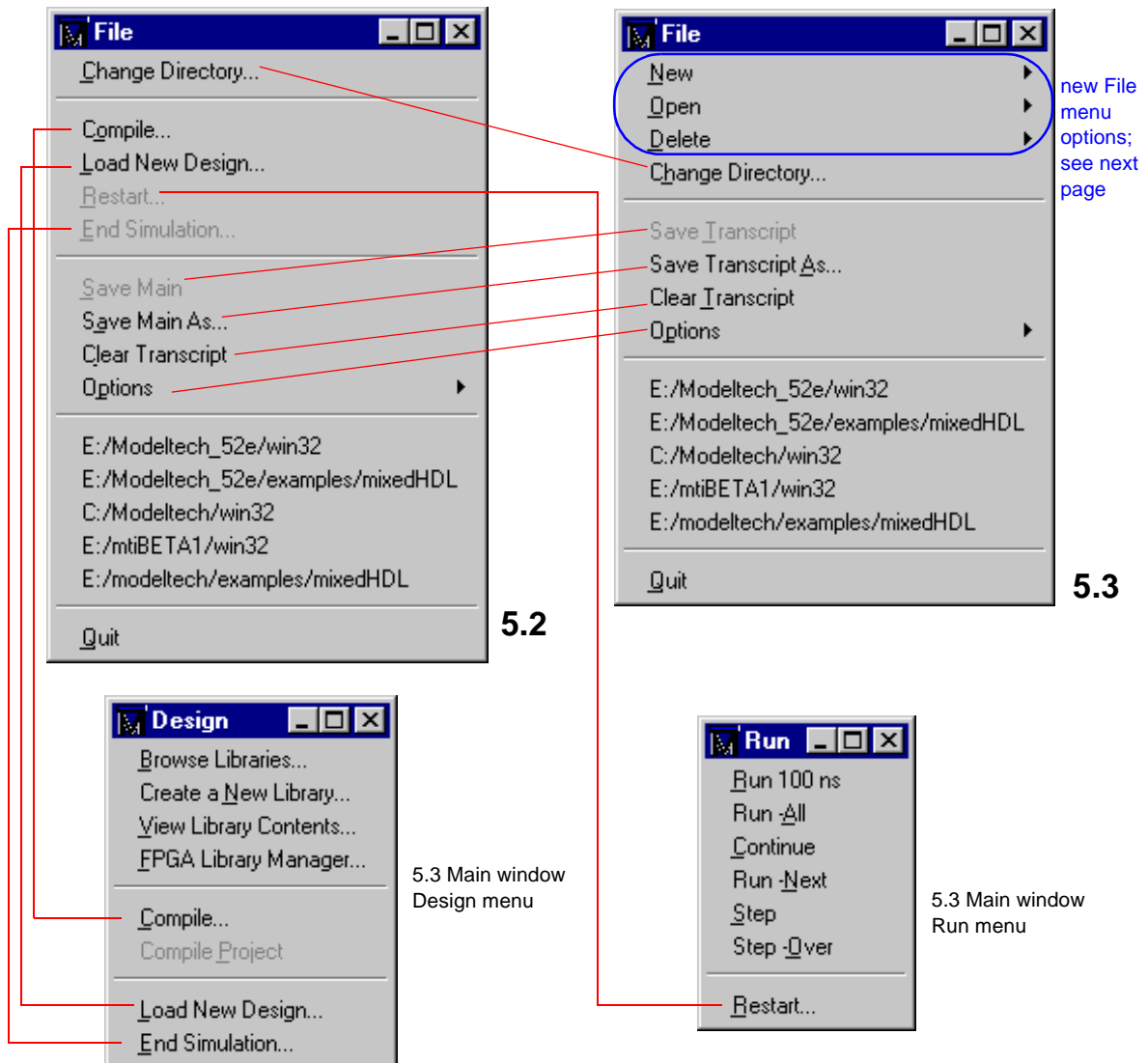
5.3

ModelSim 5.3
Main window

Main window menu changes

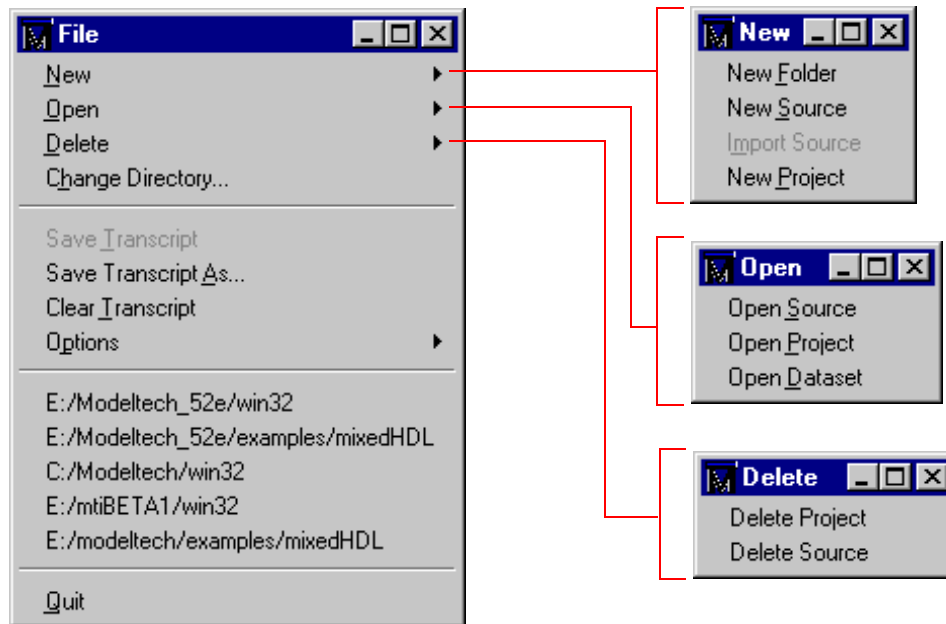
File menu selections

See ["The Main window menu bar"](#) (10-159) for complete menu option details.



New 5.3 version drop down menus available from the Main window File menu

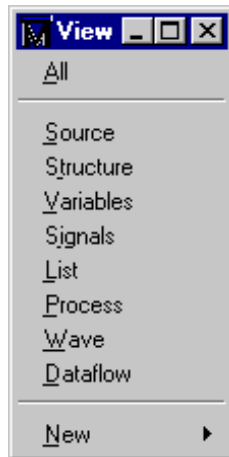
5.3



View menu selections

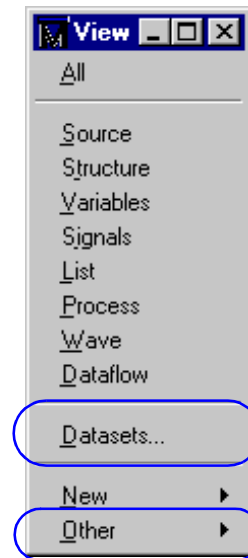
See ["The Main window menu bar"](#) (10-159) for complete menu option details.

5.2



5.2 Main window
View menu

5.3



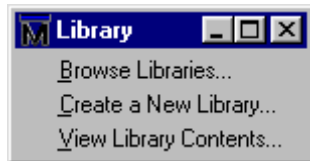
new selections

5.3 Main window
View menu

Library menu selections

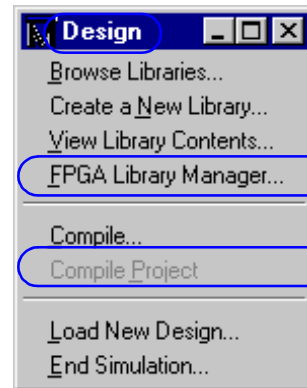
The 5.2 Library menu has been changed in the 5.3 version to the Design menu.
See "[The Main window menu bar](#)" (10-159) for complete menu option details.

5.2



5.2 Main window
Library menu

5.3



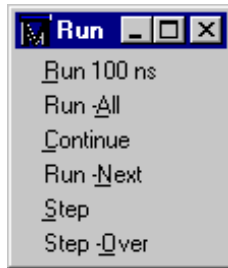
new selections

5.3 Main window
Design menu

Run menu selections

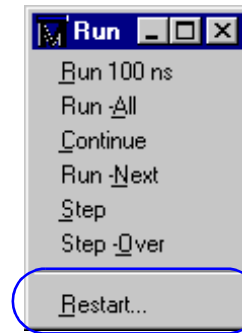
See ["The Main window menu bar"](#) (10-159) for complete menu option details.

5.2



5.2 Main window
Run menu

5.3



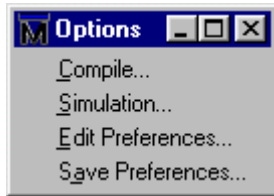
new selection

5.3 Main window
Run menu

Options menu selections

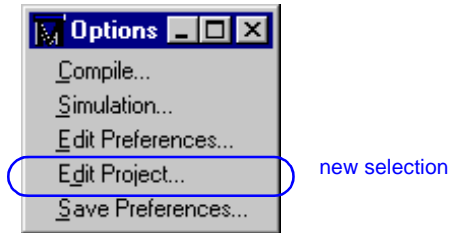
See ["The Main window menu bar"](#) (10-159) for complete menu option details.

5.2



5.2 Main window
Options menu

5.2



5.3 Main window
Options menu

Window menu selections

See "The Main window menu bar" (10-159) for complete menu option details.

5.2



5.2 Main window Window menu

5.3

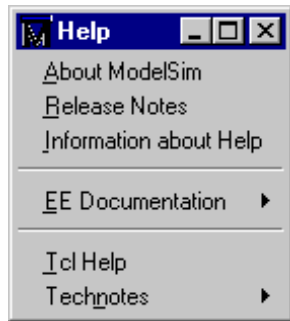


5.3 Main window Window menu

Help menu selections

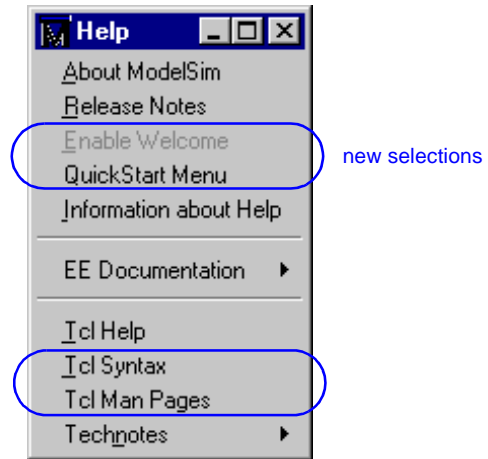
See ["The Main window menu bar"](#) (10-159) for complete menu option details.

5.2



5.2 Main window
Help menu

5.3



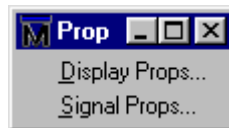
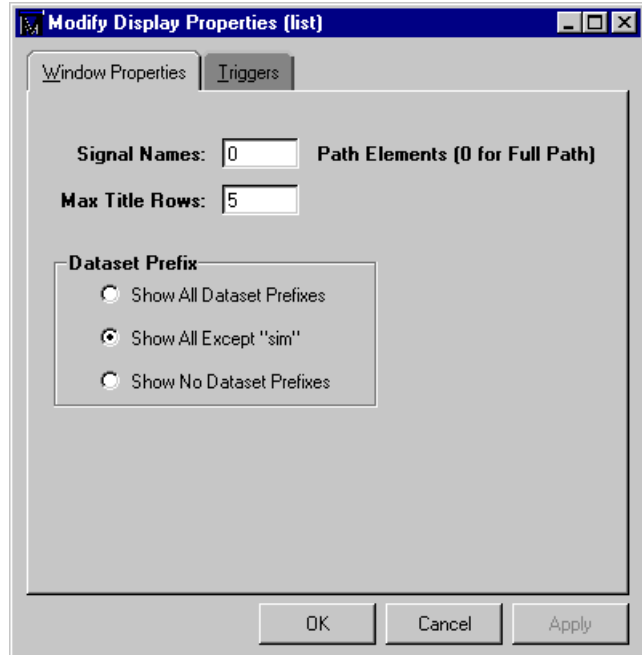
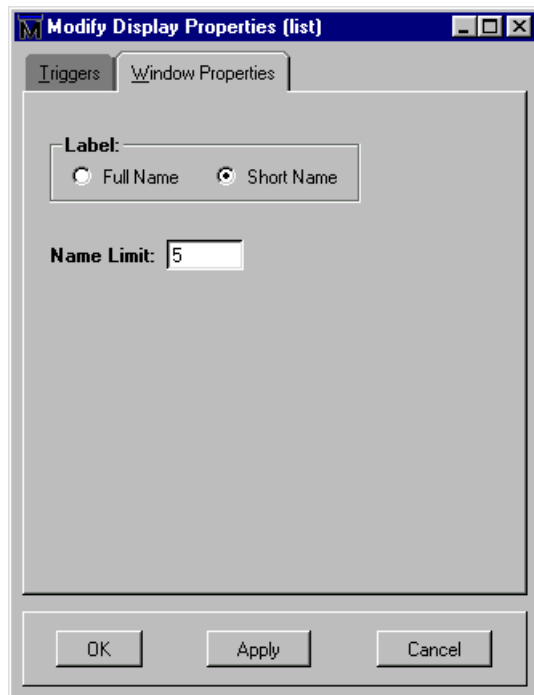
5.3 Main window
Help menu

List window menu changes

Prop menu selections

Changes were made to the Window Properties tab of the Display Props menu selection. See "[The List window menu bar](#)" (10-175) for complete menu option details.

5.2 & 5.3

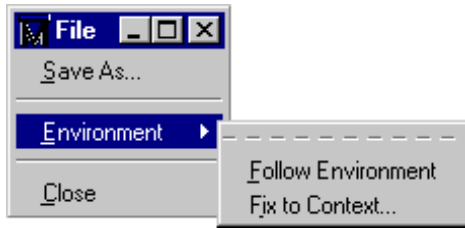
**5.2****5.3**

Process window menu changes

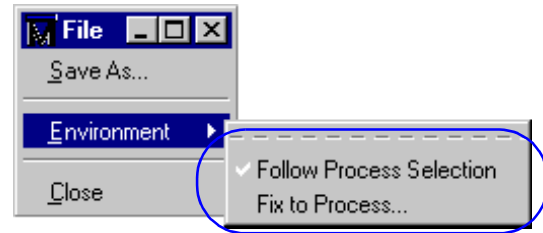
File menu selections

See "[The Process window menu bar](#)" (10-190) for complete menu option details.

5.2



5.3



Signals window menu changes

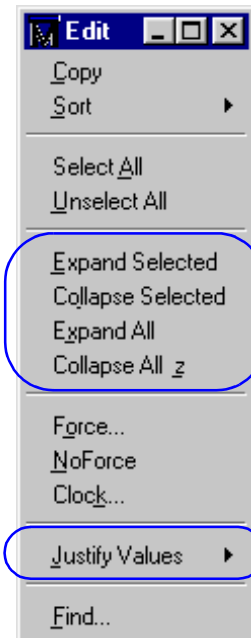
Edit menu selections

See ["The Signals window menu bar"](#) (10-193) for complete menu option details.

5.2



5.3



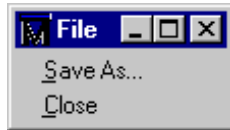
new selections

Structure window menu changes

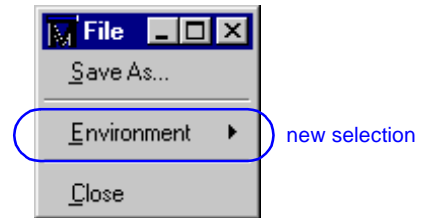
File menu selections

See "[The Structure window menu bar](#)" (10-207) for complete menu option details.

5.2



5.3



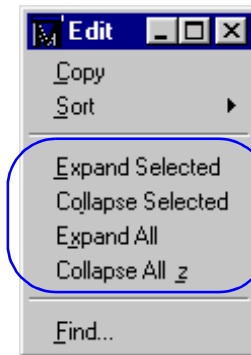
Edit menu selections

See ["The Structure window menu bar" \(10-207\)](#) for complete menu option details.

5.2



5.3



new selections

Variables window menu changes

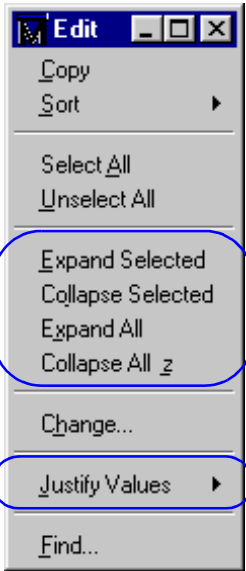
Edit menu selections

See "The Variables window menu bar" (10-210) for complete menu option details.

5.2



5.3

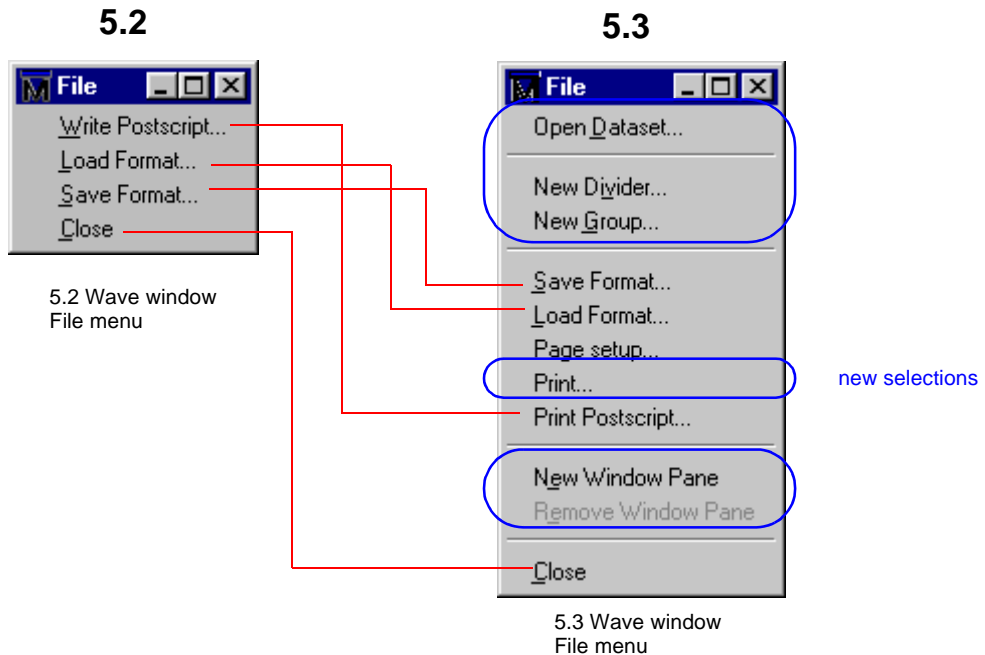


new selections

Wave window menu changes

File menu selections

See "The Wave window menu bar" (10-217) for complete menu option details.



Edit menu selections

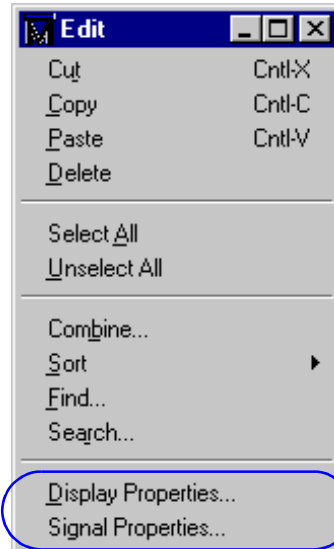
See ["The Wave window menu bar" \(10-217\)](#) for complete menu option details.

5.2



5.2 Wave window
Edit menu

5.3

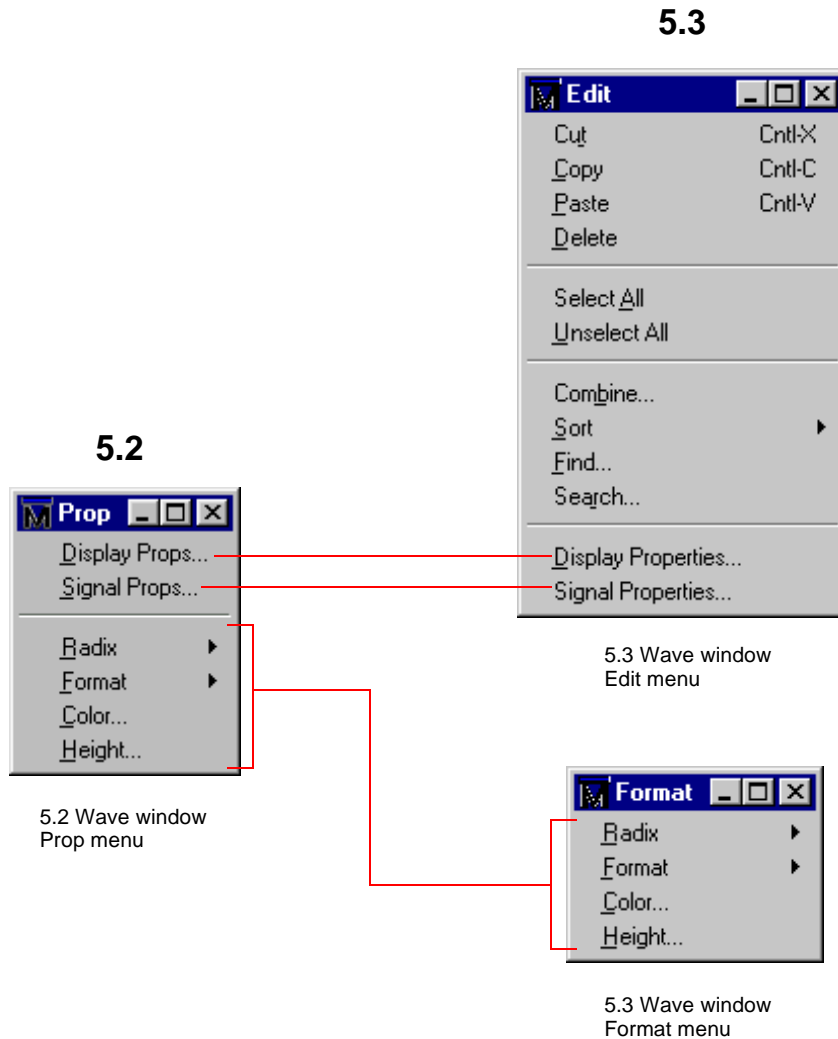


new selections

5.3 Wave window
Edit menu

Prop menu selections

The Prop menu selections in the 5.2 version have been moved to the Edit and Format menus of the Wave window in the 5.3 version. See ["The Wave window menu bar"](#) (10-217) for complete menu option details.



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

A

architecture simulator state variable [420](#)
argc simulator state variable [420](#)
AssertFile .ini file variable [398](#)
AssertionFormat .ini file variable [398](#)
Assertions
 selecting severity that stops simulation [262](#)

B

Batch mode [440](#)
Break
 on assertion [262](#)
BreakOnAssertion .ini file variable [398](#)
Breakpoints
 deleting [200](#)
 setting [200](#)
 viewing [200](#)
Button Adder (add buttons to windows) [265](#)

C

C interface, see Foreign language interface [297](#)
Cell libraries [80](#)
Checkpoint/restore [438](#)
CheckpointCompressMode .ini file variable [398](#)
CheckSynthesis .ini file variable [396](#)
Code Coverage [133](#)
 coverage clear command [134](#)
 coverage reload command [135](#)
 coverage report command [135](#), [138](#)
 coverage_summary window [136](#)
 enabling code coverage [134](#), [138](#)
 invoking code coverage with vsim [54](#)
 Tcl control variables [137](#)
Colors

 changing window colors with the GUI [414](#)
Command reference [26](#)
CommandHistory .ini file variable [398](#)
Command-line mode [440](#), [441](#)
Commands
 graphic interface commands [277](#)
 library management commands [33](#)
 VSIM Tcl commands [378](#)
Compilation and Simulation
 Verilog [64–106](#)
 VHDL [51–61](#)
Compiler directives [87](#)
 IEEE Std 1364-1995 [88](#)
 XL compatible compiler directives [89](#)
Compiling
 locating source errors [245](#)
 projects [49](#)
 Verilog
 XL 'uselib compiler directive [72](#)
 XL compatible options [70](#)
 Verilog designs [65](#)
 incremental compilation [66](#)
 VHDL designs [52](#)
Component declaration
 generating VHDL from Verilog [114](#)
 with vgencomp [114](#)
ConcurrentFileLimit .ini file variable [398](#)
configuration simulator state variable [420](#)
Conventions
 text and command syntax [26](#)
coverage clear command [134](#)
coverage reload command [135](#)
coverage report command [135](#), [138](#)
coverage_summary window [136](#)
Cur_Top_DUs .mpf file variable [402](#)

D

Dataflow window (see also, Windows) [170](#)

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

Dataset Browser [141](#)

Datasets

- restrict dataset prefix display [143](#)

- the Dataset Browser [141](#)

DefaultForceKind .ini file variable [398](#)

DefaultRadix .ini file variable [399](#)

Delay

- specifying stimulus delay [196](#)

DelayFileOpen .ini file variable [399](#)

Delta

- collapse deltas in the List window [178](#)

- referencing simulator iteration
as a simulator state variable [420](#)

delta simulator state variable [420](#)

Descriptions of HDL items [204](#)

Design hierarchy

- viewing in Structure window [206](#)

Design library

- assigning a logical name [37](#)

- creating [33](#)

- for VHDL design units [52](#)

- mapping search rules [39](#)

- resource type [32](#)

- working type [32](#)

Design units [32](#)

- viewing hierarchy [155](#)

Directories

- moving libraries [39](#)

- See also, Library

DOPATH environment variable [391](#)

DOPATH simulator control variable [412](#)

E

Editing

- in notepad windows [167](#), [426](#)

- in the Main window [167](#), [426](#)

- in the Source window [167](#), [426](#)

EDITOR environment variable [391](#)

Email

- Model Technology's email address [30](#)

entity simulator state variable [420](#)

Environment variables [391](#)

- expanding with FindProjectEntry FLI function [313](#)

- for locating license file [430](#)

- in PrefMain(file) variable [411](#)

- loading PLI files with PLIOBJS [96](#)

- location of modelsim.ini file [420](#)

- overriding with DOPATH simulator control variable [412](#)

- referencing from ModelSim command line [393](#)

- referencing with VHDL FILE variable [393](#)

- setting before compiling or simulating [391](#)

- setting in Windows [392](#)

- specify transcript file location with TranscriptFile
[400](#)

- specifying library locations in modelsim.ini file [395](#)

- used in Solaris linking for FLI and PLI [94](#), [332](#)

- using with location mapping [449](#)

- variable substitution using Tcl [376](#)

- within FOREIGN attribute string [298](#)

Errors during compilation, locating [245](#)

Explicit .ini file variable [396](#)

Expression Builder, see GUI expression builder

F

Finding

- a cursor in the Wave window [234](#)

- a marker in the List window [187](#)

Finding names, and searching for values in windows [154](#)

FLEXlm

- lmdown license server utility [435](#)

- lmgrd license server utility [435](#)

- lmremove license server utility [435](#)

- lmreread license server utility [436](#)

- lmstat license server utility [434](#)

FLEXlm license manager [429–436](#)

FLI, see Foreign language interface [297](#)

Fonts

- changing fonts with the GUI [415](#)

Foreign language interface

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

- declaring FOREIGN attribute [298](#)
- enumeration object values [329](#)
- examples [330](#)
- function descriptions, see also mti_ functions [309](#)
- mapping to VHDL data types [328](#)
- restrictions on ports and generics [300](#)
- tracing [105](#), [334](#)
- using checkpoint/restore with the FLI [305](#)
- using with foriegn architectures [298](#)
- using with foriegn subprograms [300](#)

FPGA Library Manager [274](#)

G

GenerateFormat .ini file variable [399](#)

Generics

- VHDL [108](#)

Graphic interface [149–273](#)

GUI_expression_format

- GUI expression builder [272](#)

H

Hazard .ini file variable (VCOM) [396](#)

Hazard .ini file variable (VLOG) [397](#)

Hierarchical profile [123](#)

history shortcuts [425](#)

HOME environment variable [391](#)

Home page

- Model Technology's home-page URL [30](#)

how_Warning5 .ini file variable [396](#)

I

ieee .ini file variable [395](#)

IgnoreError .ini file variable [399](#)

IgnoreFailure .ini file variable [399](#)

IgnoreNote .ini file variable [399](#)

IgnoreVitalErrors .ini file variable [396](#)

IgnoreWarning .ini file variable [399](#)

Initialization file, see Project files

Installation

- locating the license file [430](#)

Instantiation label [207](#)

Iteration_limit

- detecting infinite zero-delay loops [448](#)

IterationLimit .ini file variable [399](#)

K

Keyboard shortcuts, List window [187](#), [424](#)

Keyboard shortcuts, Wave window [237](#), [423](#)

L

<Lib>_script .mpf file variable [402](#)

Libraries

- alternate IEEE libraries [41](#)

- creating design libraries [33](#)

- deleting library contents [35](#)

- design units [32](#)

- FPGA Library Manager [274](#)

- ieee_numeric [41](#)

- ieee_synopsis [41](#)

- library management commands [33](#)

- mapping [37](#)

- mapping from the command line [38](#)

- mapping hierarchy [404](#)

- mapping search rules [39](#)

- moving [39](#)

- naming [37](#)

- predefined [40](#)

- rebuilding ieee_numeric [41](#)

- rebuilding ieee_synopsis [41](#)

- refreshing library images [41](#)

- resource libraries [32](#)

- setting up for groups [445](#)

- std [40](#)

- verilog [68](#), [110](#)

- VHDL library clause [40](#)

- viewing library contents [35](#)

- work library [32](#)
- working libraries [32](#)
- library simulator state variable [420](#)
- License
 - locating the license file [430](#)
- License .ini file variable [399](#)
- List window
 - setting triggers [177](#), [456](#)
- List window (see also, Windows) [174](#)
- LM_LICENSE_FILE environment variable [391](#)
- Locating source errors during compilation [245](#)
- Location maps
 - referencing source files [449](#)
- LockedMemory .ini file variable [400](#), [462](#)
- Logic Modeling
 - SmartModel
 - command channel [354](#)
 - compiling Verilog shells [357](#)
 - SmartModel Windows [355](#)
 - lmcwin commands [355](#)
 - memory arrays [356](#)

M

- MacroNestingLevel simulator state variable [420](#)
- Macros (do files)
 - depth of nesting, simulator state variable [420](#)
 - parameter as a simulator state variable (n) [420](#)
 - parameter total as a simulator state variable [420](#)
 - passing parameters to [442](#)
 - startup macros [404](#)
- Main window (see also, Windows) [158](#)
- Memory
 - locked memory [451](#)
 - modeling in VHDL [452](#)
- Menus
 - customizing menus and buttons [155](#)
 - Dataflow window [171](#)
 - List window [175](#)
 - Main window [159](#)
 - Process window [190](#)

- Signals window [193](#)
- Source window [201](#)
- Structure window [207](#)
- tearing off or pinning menus [154](#)
- Variables window [210](#)
- Wave window [217](#)
- Messages
 - turning off assertion messages [405](#)
 - turning off warnings from arithmetic packages [405](#)
- MGC_LOCATION_MAP environment variable [391](#)
- MODEL_TECH environment variable [391](#)
- MODEL_TECH_FPGA environment variable [391](#)
- MODEL_TECH_TCL environment variable [391](#)
- Modeling memory in VHDL [452](#)
- ModelSim
 - custom setup with daemon options [432](#)
 - license file [430](#)
- MODELSIM environment variable [392](#)
- modelsim.ini, see Project files
- MODELSIM_TCL environment variable [392](#)
- mti_AddCommand [309](#)
- mti_AddEnvCB [309](#)
- mti_AddInputReadyCB [309](#)
- mti_AddLoadDoneCB [309](#)
- mti_AddOutputReadyCB [309](#)
- mti_AddQuitCB [309](#)
- mti_AddRestartCB [310](#)
- mti_AddRestoreCB [310](#)
- mti_AddSaveCB [310](#)
- mti_AddSimStatusCB [310](#)
- mti_AddSocketInputReadyCB [310](#)
- mti_AddSocketOutputReadyCB [310](#)
- mti_AddTclCommand [310](#)
- mti_AskStdin [310](#)
- mti_Break [311](#)
- mti_Cmd [311](#)
- mti_Command [311](#)
- mti_CreateArrayType [311](#)
- mti_CreateDriver [311](#)
- mti_CreateEnumType [312](#)
- mti_CreateProcess [312](#)
- mti_CreateRealType [312](#)

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_CreateRegion 312	mti_GetSignalNameIndirect 318
mti_CreateScalarType 312	mti_GetSignalRegion 318
mti_CreateSignal 312	mti_GetSignalSubelements 318
mti_Delta 312	mti_GetSignalType 319
mti_Desensitize 312	mti_GetSignalValue 319
mti_ElementType 313	mti_GetSignalValueIndirect 319
mti_FatalError 313	mti_GetTopRegion 319
mti_FindDriver 313	mti_GetTraceLevel 319
mti_FindPort 313	mti_GetTypeKind 319
mti_FindProjectEntry 313	mti_GetVarAddr 319
mti_FindRegion 313	mti_GetVarImage 320
mti_FindSignal 313	mti_GetVarImageById 320
mti_FindVar 314	mti_GetVarName 320
mti_FirstLowerRegion 314	mti_GetVarSubelements 320
mti_FirstProcess 314	mti_GetVarType 321
mti_FirstSignal 314	mti_GetVarValue 321
mti_FirstVar 314	mti_GetVarValueIndirect 321
mti_Free 314	mti_HigherRegion 321
mti_GetArrayElementType 315	mti_Image 321
mti_GetArraySignalValue 315	mti_Interp 321
mti_GetArrayVarValue 315	mti_IsColdRestore 322
mti_GetCallingRegion 315	mti_IsFirstInit 322
mti_GetCheckpointFilename 315	mti_IsPE 322
mti_GetCurrentRegion 315	mti_IsRestore 322
mti_GetDriverSubelements 315	mti_Malloc 322
mti_GetEnumValues 316	mti_NextProcess 322
mti_GetGenericList 316	mti_NextRegion 322
mti_GetLibraryName 316	mti_NextSignal 322
mti_GetNextEventTime 316	mti_NextVar 323
mti_GetNextNextEventTime 316	mti_Now 323
mti_GetNumRecordElements 316	mti_NowIndirect 323
mti_GetPrimaryName 317	mti_NowUpper 323
mti_GetProcessName 317	mti_PrintMessage 323
mti_GetProductVersion 317	mti_Quit 323
mti_GetRegionFullName 317	mti_Realloc 323
mti_GetRegionKind 317	mti_RemoveEnvCB 323
mti_GetRegionName 317	mti_RemoveLoadDoneCB 323
mti_GetRegionSourceName 317	mti_RemoveQuitCB 324
mti_GetResolutionLimit 317	mti_RemoveRestartCB 324
mti_GetSecondaryName 317	mti_RemoveRestoreCB 324
mti_GetSignalMode 318	mti_RemoveSaveCB 324
mti_GetSignalName 318	mti_RemoveSimStatusCB 324

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_Resolution [324](#)
mti_RestoreBlock [324](#)
mti_RestoreChar [324](#)
mti_RestoreLong [324](#)
mti_RestoreProces [324](#)
mti_RestoreShort [324](#)
mti_RestoreString [325](#)
mti_SaveBlock [325](#)
mti_SaveChar [325](#)
mti_SaveLong [325](#)
mti_SaveShort [325](#)
mti_SaveString [325](#)
mti_ScheduleDriver [325](#)
mti_ScheduleWakeup [325](#)
mti_Sensitize [325](#)
mti_SetDriverOwner [326](#)
mti_SetSignalValue [326](#)
mti_SetVarValue [326](#)
mti_SignalImage [326](#)
MTI_TF_LIMIT environment variable [391](#)
mti_TickDir [326](#)
mti_TickHigh [326](#)
mti_TickLeft [327](#)
mti_TickLength [327](#)
mti_TickLow [327](#)
mti_TickRight [327](#)
mti_TraceActivate [327](#)
mti_TraceOff [327](#)
mti_TraceOn [327](#)
mti_TraceSkipID [327](#)
mti_TraceSuspend [327](#)
mti_WriteProjectEntry [327](#)
Multiple drivers on unresolved signal [248](#)

N

n simulator state variable [420](#)
Nets
 adding to the Wave and List windows [197](#)
 displaying in Dataflow window [170](#)
 displaying values in Signals window [192](#)

 forcing signal and net values [195](#)
 saving values as binary log file [197](#)
 viewing waveforms [212](#)
Next and previous edges, finding [237](#), [423](#)
No space in time literal [247](#)
NoDebug .ini file variable (VCOM) [396](#)
NoDebug .ini file variable (VLOG) [397](#)
Notepad windows, text editing [167](#), [426](#)
NoVital .ini file variable [396](#)
NoVitalCheck .ini file variable [396](#)
Now simulator state variable [421](#)
now simulator state variable [421](#)
NumericStdNoWarnings .ini file variable [400](#)

O

Operating systems supported [22](#)
Optimize for std_logic_1164 [248](#)
Optimize_1164 .ini file variable [396](#)

P

Packages
 standard [40](#)
 textio [40](#)
Parameters, for macros [442](#)
PathSeparator .ini file variable [400](#)
Performance Analyzer [119](#)
 %in field [125](#)
 %parent field [126](#)
 %under field [125](#)
 commands [131](#)
 getting started [121](#)
 hierarchical profile [123](#)
 interpreting data [122](#)
 name field [125](#)
 profile report command [129](#)
 ranked profile [126](#)
 report option [129](#)
 setting preferences [130](#)
 statistical sampling [120](#)

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

- view_profile command [123](#)
- view_profile_ranked command [124](#)
- viewing results [123](#)

PLI see Verilog PLI

PLIOBJS environment variable [392](#)

PlotFilterResolution simulator control variable [412](#)

Ports

- VHDL and Verilog [109](#)

Preferences

- setting preferences from the command line [418](#)

- performance analyzer preferences [130](#)

- setting preferences with the GUI

 - window fonts and colors [412](#), [418](#)

 - window sizes and locations [412](#), [418](#)

Process window (see also, Windows) [189](#)

Process without a wait statement [247](#)

Processes

- displayed in Dataflow window [170](#)

- values and pathnames in Variables window [209](#)

profile report command [129](#)

Project files

- compile a project [49](#)

- creating a project [46](#)

- modelsim.ini

 - default to VHDL93 [406](#)

 - environment variables [403](#)

 - hierarchical library mapping [404](#)

 - opening VHDL files [406](#)

 - to specify a startup file [404](#)

 - turning off arithmetic warnings [405](#)

 - turning off assertion messages [405](#)

 - using to create a transcript file [404](#)

 - using to define force command default [405](#)

 - using to delay file opening [406](#)

- modelsim.mpf [43](#)

 - project definition [44](#)

- modifying a project [49](#)

- open a project [49](#)

- project operations [45](#)

- simulating a project [49](#)

'protect compiler directive [443](#)

Q

Quick Start [264](#)

Quick Start guide [264](#)

Quiet .ini file variable (VCOM) [396](#)

Quiet .ini file variable (VLOG) [397](#)

R

Radix

- specifying in List window [182](#)

- specifying in Signals window [195](#)

Ranked profile [126](#)

Rebuilding supplied libraries [41](#)

reconstruct RTL-level design busses [145](#)

Records

- changing values of [209](#)

Refreshing library images [41](#)

Register variables

- adding to the Wave and List windows [197](#)

- displaying values in Signals window [192](#)

- saving values as binary log file [197](#)

- viewing waveforms [212](#)

Resolution .ini file variable [400](#)

resolution simulator state variable [421](#)

RunLength .ini file variable [400](#)

S

ScalarOpts .ini file variable [397](#)

SDF

- Errors and warnings [283](#)

- Instance specification [282](#)

- interconnect delays [293](#)

- mixed VHDL and Verilog designs [293](#)

- troubleshooting [294](#)

- Verilog

 - rounded timing values [292](#)

- Verilog

 - \$sdf_annotate system task [286](#)

- optional conditions [292](#)
 - optional edge specifications [291](#)
 - SDF to Verilog construct matching [287](#)
- Verilog SDF annotation [286](#)
- VHDL
 - Resolving errors [285](#)
 - SDF to VHDL generic matching [284](#)
- Searching
 - for values and finding names in windows [154](#)
 - List window
 - signal values, transitions, and names [184](#)
 - text strings in the List window [184](#)
 - text strings in the Wave window [231](#)
 - waveform
 - signal values, edges and names [231](#)
- searchLog simulator command [147](#)
- Shortcuts
 - command history [425](#)
 - command line caveat [425](#)
 - List window [187](#), [424](#)
 - text editing [167](#), [426](#)
 - Wave window [237](#), [423](#)
- Show source lines with errors [247](#)
- Show_source .ini file variable (VCOM) [395](#)
- Show_source .ini file variable (VLOG) [397](#)
- Show_VitalChecksWarning .ini file variable [395](#)
- Show_Warning1 .ini file variable [395](#)
- Show_Warning2 .ini file variable [395](#)
- Show_Warning3 .ini file variable [396](#)
- Show_Warning4 .ini file variable [396](#)
- Signal transitions
 - searching for [235](#)
- Signals
 - adding to a log file [197](#)
 - adding to the Wave and List windows [197](#)
 - applying stimulus to [195](#)
 - combining into a user-defined bus [155](#)
 - displaying in Dataflow window [170](#)
 - displaying values in Signals window [192](#)
 - forcing signal and net values [195](#)
 - saving values as binary log file [197](#)
 - selecting signal types to view [195](#)
 - viewing waveforms [212](#)
- Signals window (see also, Windows) [192](#)
- Simulating
 - applying stimulus
 - see also VSIM command, force
 - applying stimulus to signals and nets [195](#)
 - applying stimulus with textio [58](#)
 - batch mode [440](#)
 - command-line mode [440](#)
 - Mixed Verilog and VHDL Designs
 - compilers [108](#)
 - libraries [108](#)
 - Verilog parameters [109](#)
 - Verilog state mapping [110](#)
 - VHDL and Verilog ports [109](#)
 - VHDL generics [108](#)
 - projects [49](#)
 - saving waveform as a Postscript file [238](#)
 - setting default run length [261](#)
 - setting iteration limit [261](#)
 - setting time resolution [252](#)
- Verilog
 - delay modes [81](#)
 - even order issues [76](#)
 - hazard detection [77](#)
 - resolution limit [75](#)
 - XL compatible simulator options [78](#)
- Verilog designs [74](#)
- VHDL designs [53](#)
 - invoking code coverage [54](#)
- viewing results in List window [174](#)
- Simulation and Compilation
 - Verilog [64–106](#)
 - VHDL [51–61](#)
- Sorting
 - sorting HDL items in VSIM windows [154](#)
- Source code
 - source code security [443](#)
 - viewing [200](#)
- Source directory, setting from source window [201](#)
- Source files
 - referencing with location maps [449](#)

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

- Source window (see also, Windows) 200
- SourceDir simulator control variable 412
- SourceMap simulator control variable 412
- Src_Files .mpf file variable 402
- Startup
 - macro in the modelsim.ini file 400
 - startup macro in command-line mode 441
 - using a startup file 404
- Startup .ini file variable 400
- Startup macros 404
- Status bar
 - Main window 167
- std .ini file variable 395
- std_developerskit .ini file variable 395
- StdArithNoWarnings .ini file variable 400
- STDOUT environment variable 392
- Structure window (see also, Windows) 206
- Symbolic link to design libraries (UNIX) 39
- synopsys .ini file variable 395
- System initialization 43

T

- Tcl 367–381, ??–381
 - command separator 375
 - command substitution 374
 - evaluation order 375
 - history shortcuts 425
 - Man Pages in Help menu 164
 - relational expression evaluation 375
 - variable substitution 338, 376
- Tcl command syntax 370
- Text editing, see Editing
- Text strings
 - finding in the List window 184
 - finding in the Wave window 231
- TextIO package 51
 - alternative I/O files 58
 - containing hexadecimal numbers 57
 - dangling pointers 57
 - ENDFILE function 58

- ENDLINE function 58
- file declaration 55
- implementation issues 56
- providing stimulus 58
- standard input 56
- standard output 56
- WRITE procedure 56
- WRITE_STRING procedure 57

- Time
 - time as a simulator state variable 420
 - time resolution as a simulator state variable 421
- TMPDIR environment variable 392
- Tool bar
 - Main window 165
 - Wave window 221
- Top_DUs .mpf file variable 402
- Tracing HDL items with the Dataflow window 172
- TranscriptFile .ini file variable 400
- Tree windows
 - VHDL and Verilog items in 155
 - viewing the design hierarchy 157
- Triggers, setting in the List window 177, 456

U

- Unbound Component 247
- UnbufferedOutput .ini file variable 401
- UpCase .ini file variable 397
- Use 1076-1993 language standard 246
- Use clause
 - specifying a library 40
- Use explicit declarations only 246
- UserTimeUnit .ini file variable 401

V

- Values of HDL items 204
- Variable settings report 403
- Variables 64
- Variables window (see also, Windows) 209
- Variables, HDL

- changing value of with the GUI [209](#)
- Variables, referencing
 - loading order at ModelSim startup [419](#)
 - reading variable values from the .ini file [403](#)
 - shared objects
 - LD_LIBRARY_PATH, and SHLIB_PATH [299](#)
 - MGC_HOME, and MGC_WD [299](#)
 - simulator state variables
 - iteration number [420](#)
 - name of entity or module as a variable [420](#)
 - resolution [420](#)
 - simulation time [420](#)
- Variables, setting
 - environment variables [391](#)
- VCD files
 - extracting the proper stimulus [338](#)
 - from VHDL source to VCD output [340](#)
 - VCD system tasks [338](#)
- Verilog
 - capturing port driver data with -dumpports [344](#)
 - cell libraries [80](#)
 - compiler directives [87](#)
 - compiling design units [65](#)
 - compiling with XL 'uselib compiler directive [72](#)
 - component declaration [114](#)
 - creating a design library [65](#)
 - instantiation criteria [112](#)
 - instantiation of VHDL design units [116](#)
 - library usage [68](#)
 - mapping states in mixed designs [110](#)
 - mixed designs with VHDL [107](#)
 - parameters [109](#)
 - SDF annotation [286](#)
 - sdf_annotate system task [286](#)
 - simulating [74](#)
 - delay modes [81](#)
 - event order issues [76](#)
 - XL compatible options [78](#)
 - simulation hazard detection [77](#)
 - simulation resolution limit [75](#)
 - SmartModel interface [357](#)
 - source code viewing [200](#)
 - system tasks [82](#)
 - XL compatible compiler options [70](#)
 - XL compatible routines [104](#)
 - XL compatible system tasks [85](#)
- verilog .ini file variable [395](#)
- Verilog PLI [90–106](#)
 - specifying the PLI file to load [95](#)
 - support for VHDL objects [100](#)
- Veriuser .ini file variable [401](#)
- VHDL
 - compiling design units [52](#)
 - creating a design library [52](#)
 - delay file opening [406](#)
 - Dependency checking [53](#)
 - file opening delay [406](#)
 - Hardware Model interface [360](#)
 - instantiation from Verilog [116](#)
 - instantiation of Verilog [108](#)
 - library clause [40](#)
 - mixed designs with Verilog [107](#)
 - object support in PLI [100](#)
 - simulating [53](#)
 - SmartModel interface [348](#)
 - source code viewing [200](#)
 - timing check disabling [54](#)
- VHDL93 .ini file variable [395](#)
- view_profile command [123](#)
- view_profile_ranked command [124](#)
- Viewing design hierarchy [155](#)
- virtual hide command [145](#)
- Virtual objects [144](#)
 - virtual functions [146](#)
 - virtual regions [146](#)
 - virtual signals [145](#)
 - virtual types [147](#)
- virtual region command [146](#)
- Virtual regions
 - reconstruct the RTL Hierarchy in gate level design [146](#)
- virtual save command [145](#)
- virtual signal command [145](#)

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 signals

- reconstruct RTL-level design busses [145](#)
- reconstruct the original RTL hierarchy [145](#)
- virtual hide command [145](#)

VITAL

- compiling and simulating with accelerated VITAL packages [61](#)
- compliance warnings [60](#)
- obtaining the specification [59](#)
- VITAL packages [59](#)

VSIM commands

- searchLog [147](#)

W

Warnings

- turning off warnings from arithmetic packages [405](#)

Wave window (see also, Windows) [212](#)

WaveSignalNameWidth .ini file variable [401](#)

Windows

- change window fonts and colors [413](#)
- finding HDL item names [154](#)
- opening multiple copies [154](#)
- opening with the GUI [161](#)
- save window sizes and locations [413](#)
- searching for HDL item values [154](#)
- adding buttons [265](#)

Dataflow window [170](#)

- tracing signals and nets [172](#)

List window [174](#)

- adding HDL items [180](#)
- adding signals with a log file [197](#)
- examining simulation results [183](#)
- formatting HDL items [181](#)
- locating time markers [154](#)
- saving to a file [188](#)
- setting display properties [177](#)

Main window [158](#)

- status bar [167](#)
- text editing [167](#), [426](#)
- time and delta display [167](#)

tool bar [165](#)

Process window [189](#)

- displaying active processes [189](#)
- specifying next process to be executed [189](#)
- viewing processing in the region [189](#)

Signals window [192](#)

- VHDL and Verilog items viewed in [192](#)

Source window [200](#)

- text editing [167](#), [426](#)
- viewing HDL source code [200](#)

Structure window [206](#)

- HDL items viewed in [206](#)
- instance names [207](#)
- selecting items to view in Signals window [192](#)
- VHDL and Verilog items viewed in [206](#)
- viewing design hierarchy [206](#)

Variables window [209](#)

- displaying values [209](#)
- VHDL and Verilog items viewed in [209](#)

Wave window [212](#)

- adding HDL items [223](#)
- adding signals with a log file [197](#)
- changing display range (zoom) [235](#)
- cursor measurements [234](#)
- locating time cursors [154](#)
- searching for HDL item values [231](#)
- setting display properties [230](#)
- using time cursors [233](#)
- zoom options [235](#)
- zooming [235](#)

Work_Libs .mpf file variable [402](#)

Z

Zero-delay loop, detecting infinite [448](#)

Zoom

- from Wave toolbar buttons [235](#)
- from Zoom menu [235](#)
- options [235](#)
- with the mouse [236](#)
- with VSIM commands [236](#)

Thank you for purchasing ModelSim!

Model Technology
A MENTOR GRAPHICS COMPANY

www.model.com