ModelSim® User's Manual

Software Version 6.2g

February 2007

Table of Contents

List of Examples

List of Figures

List of Tables

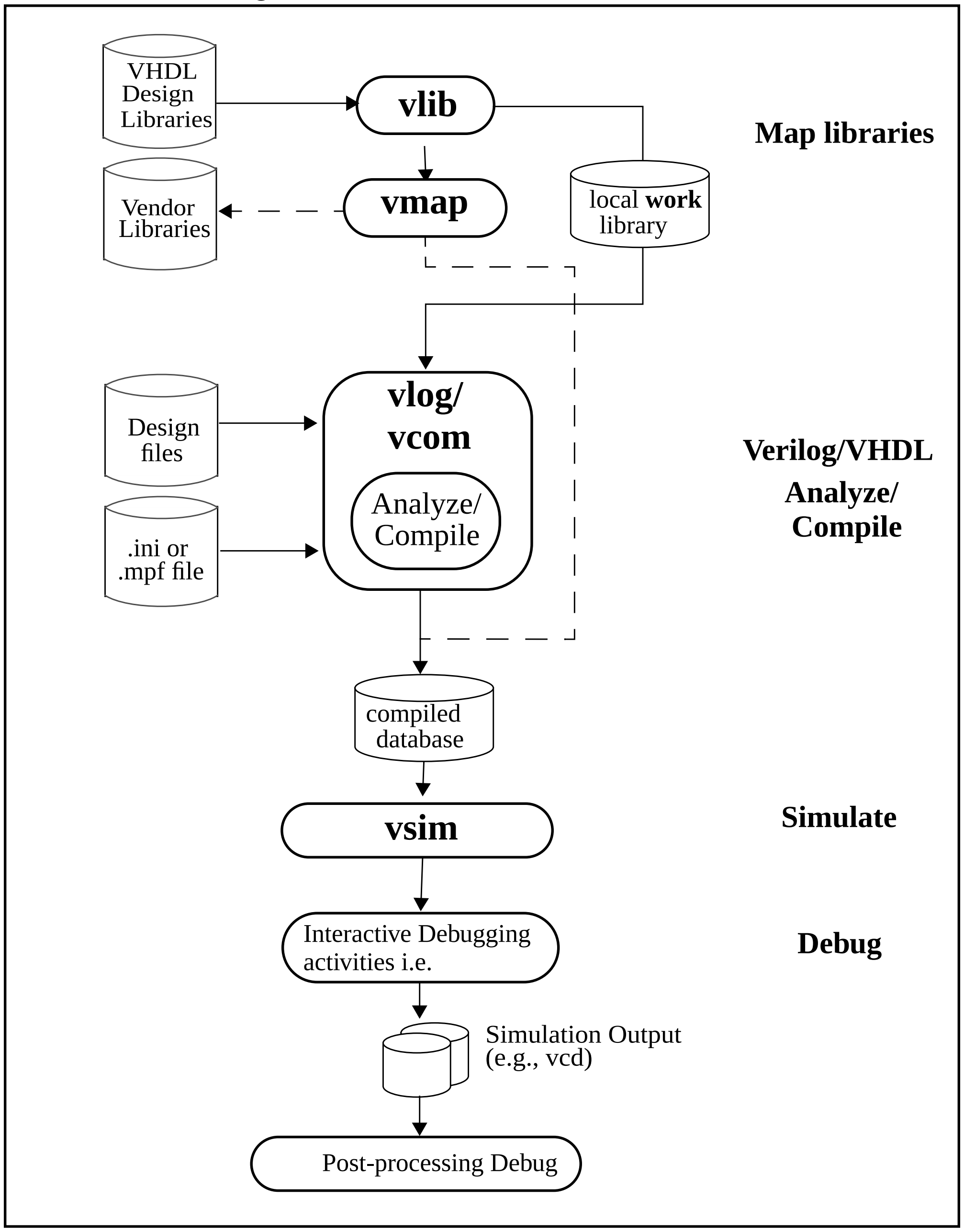
Chapter 1
Introduction

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

Tool Structure and Flow

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

Figure 1-1. Tool Structure and Flow



Simulation Task Overview

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

Table 1-1. Simulation Tasks

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

Basic Steps for Simulation

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

Step 1 — Collecting Files and Mapping Libraries

Files needed to run ModelSim on your design:

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

Providing Stimulus to the Design

You can provide stimulus to your design in several ways:

- Language based testbench
- Tcl-based ModelSim interactive command, force
- VCD files / commands

  See Creating a VCD File and Using Extended VCD as Stimulus
- 3rd party testbench generation tools

What is a Library?

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

Working and Resource Libraries

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

For more information on resource libraries and working libraries, see Working Library Versus Resource Libraries, Managing Library Contents, Working with Design Libraries, and Specifying the Resource Libraries.

Creating the Logical Library (vlib)

Before you can compile your source files, you must create a library in which to store the compilation results. You can create the logical library using the GUI, using **File > New > Library** (see Creating a Library), or you can use the vlib command. For example, the command:

```
vlib work
```

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

Mapping the Logical Work to the Physical Work Directory (vmap)

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

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

You can use the GUI (Library Mappings with the GUI, a command (Library Mapping from the Command Line), or a project (Getting Started with Projects to assign a logical name to a design library.

The format for command line entry is:

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

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

Step 2 — Compiling the Design (vlog, vcom, sccom)

Designs are compiled with one of the three language compilers.

Compiling Verilog (vlog)

ModelSim’s compiler for the Verilog modules in your design is vlog. Verilog files may be compiled in any order, as they are not order dependent. See Compiling Verilog Files for details.

Compiling VHDL (vcom)

ModelSim’s compiler for VHDL design units is vcom. VHDL files must be compiled according to the design requirements of the design. Projects may assist you in determining the compile order: for more information, see Auto-Generating Compile Order*.* See Compiling VHDL Files for details. on VHDL compilation.

Step 3 — Loading the Design for Simulation

vsim topLevelModule

Your design is ready for simulation after it has been compiled. You may then invoke vsim with the names of the top-level modules (many designs contain only one top-level module). For example, if your top-level modules are "testbench" and "globals", then invoke the simulator as follows:

```
vsim testbench globals
```

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

Using SDF

You can incorporate actual delay values to the simulation by applying SDF back-annotation files to the design. For more information on how SDF is used in the design, see Specifying SDF Files for Simulation.

Step 4 — Simulating the Design

Once the design has been successfully loaded, the simulation time is set to zero, and you must enter a **run** command to begin simulation. For more information, see Verilog and SystemVerilog Simulation, and VHDL Simulation.

The basic simulator commands are:

- add wave
- force
- bp
- run
- step

Step 5 — Debugging the Design

Numerous tools and windows useful in debugging your design are available from the ModelSim GUI. In addition, several basic simulation commands are available from the command line to assist you in debugging your design:

- describe
- drivers

- examine
- force
- log
- show

Modes of Operation

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

Table 1-2. Use Modes

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

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

Command Line Mode

In command line mode ModelSim executes any startup command specified by the Startup variable in the *modelsim.ini* file. If vsim is invoked with the **-do "command_string"** option, a DO file (macro) is called. A DO file executed in this manner will override any startup command in the *modelsim.ini* file.

During simulation a transcript file is created containing any messages to stdout. A transcript file created in command line mode may be used as a DO file if you invoke the transcript **on** command after the design loads (see the example below). The transcript **on** command writes all of the commands you invoke to the transcript file. For example, the following series of commands results in a transcript file that can be used for command input if *top* is re-simulated (remove the **quit -f** command from the transcript file if you want to remain in the simulator).

```
vsim -c top
```

library and design loading messages… then execute:

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

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

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

Batch Mode

Batch mode is an operational mode that provides neither an interactive command line nor interactive windows. In a Windows environment, **vsim** is run from a Windows command prompt and standard input and output are redirected from and to files.

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

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

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

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

Standards Supported

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

ModelSim Verilog implements the Verilog language as defined by the IEEE Std 1364-1995 and 1364-2005. ModelSim Verilog also supports a partial implementation of SystemVerilog P1800-2005 (see */<install_dir>/modeltech/docs/technotes/sysvlog.note* for implementation details).

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

In addition, all products support SDF 1.0 through 4.0 (except the NETDELAY statement), VITAL 2.2b, VITAL'95 – IEEE 1076.4-1995, and VITAL 2000 – IEEE 1076.4-2000.

Assumptions

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

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

Finally, we assume that you have worked the appropriate lessons in the *ModelSim Tutorial* and are familiar with the basic functionality of ModelSim. The *ModelSim Tutorial* is available from the ModelSim **Help** menu.

Sections In This Document

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

Chapter 3, Projects — This chapter discusses ModelSim "projects", a container for design files and their associated simulation properties.

Chapter 4, Design Libraries — 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.

Chapter 5, VHDL Simulation — This chapter is an overview of compilation and simulation for VHDL within the ModelSim environment.

Chapter 6, Verilog and SystemVerilog Simulation — This chapter is an overview of compilation and simulation for Verilog and SystemVerilog within the ModelSim environment.

Chapter 7, WLF Files (Datasets) and Virtuals — This chapter describes datasets and virtuals - both methods for viewing and organizing simulation data in ModelSim.

Chapter 8, Waveform Analysis — This chapter describes how to perform waveform analysis with the ModelSim Wave and List windows.

Chapter 9, Tracing Signals with the Dataflow Window — This chapter describes how to trace signals and assess causality using the ModelSim Dataflow window.

Chapter 10, Signal Spy — This chapter describes Signal Spy, a set of VHDL procedures and Verilog system tasks that let you monitor, drive, force, or release a design object from anywhere in the hierarchy of a VHDL or mixed design.

Chapter 11, Standard Delay Format (SDF) Timing Annotation — This chapter discusses ModelSim's implementation of SDF (Standard Delay Format) timing annotation. Included are sections on VITAL SDF and Verilog SDF, plus troubleshooting.

Chapter 12, Value Change Dump (VCD) Files — This chapter explains Model Technology's Verilog VCD implementation for ModelSim. The VCD usage is extended to include VHDL designs.

Chapter 13, Tcl and Macros (DO Files) — This chapter provides an overview of Tcl (tool command language) as used with ModelSim.

Appendix A, Simulator Variables — This appendix describes environment, system, and preference variables used in ModelSim.

Appendix C, Error and Warning Messages — This appendix describes ModelSim error and warning messages.

Appendix D, Verilog PLI/VPI/DPI — This appendix describes the ModelSim implementation of the Verilog PLI and VPI.

Appendix E, Command and Keyboard Shortcuts — This appendix describes ModelSim keyboard and mouse shortcuts.

Appendix G, System Initialization — This appendix describes what happens during ModelSim startup.

What is an "Object"

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

Table 1-3. Definition of Object by Language

Language	An object can be
VHDL	block statement, component instantiation, constant, generate statement, generic, package, signal, alias, or variable
Verilog	function, module instantiation, named fork, named begin, net, task, register, or variable
SystemVerilog	In addition to those listed above for Verilog: class, package, program, interface, array, directive, property, or sequence

Table 1-3. Definition of Object by Language

Language	An object can be
PSL	property, sequence, directive, or endpoint

Text Conventions

Text conventions used in this manual include:

Table 1-4. Text Conventions

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

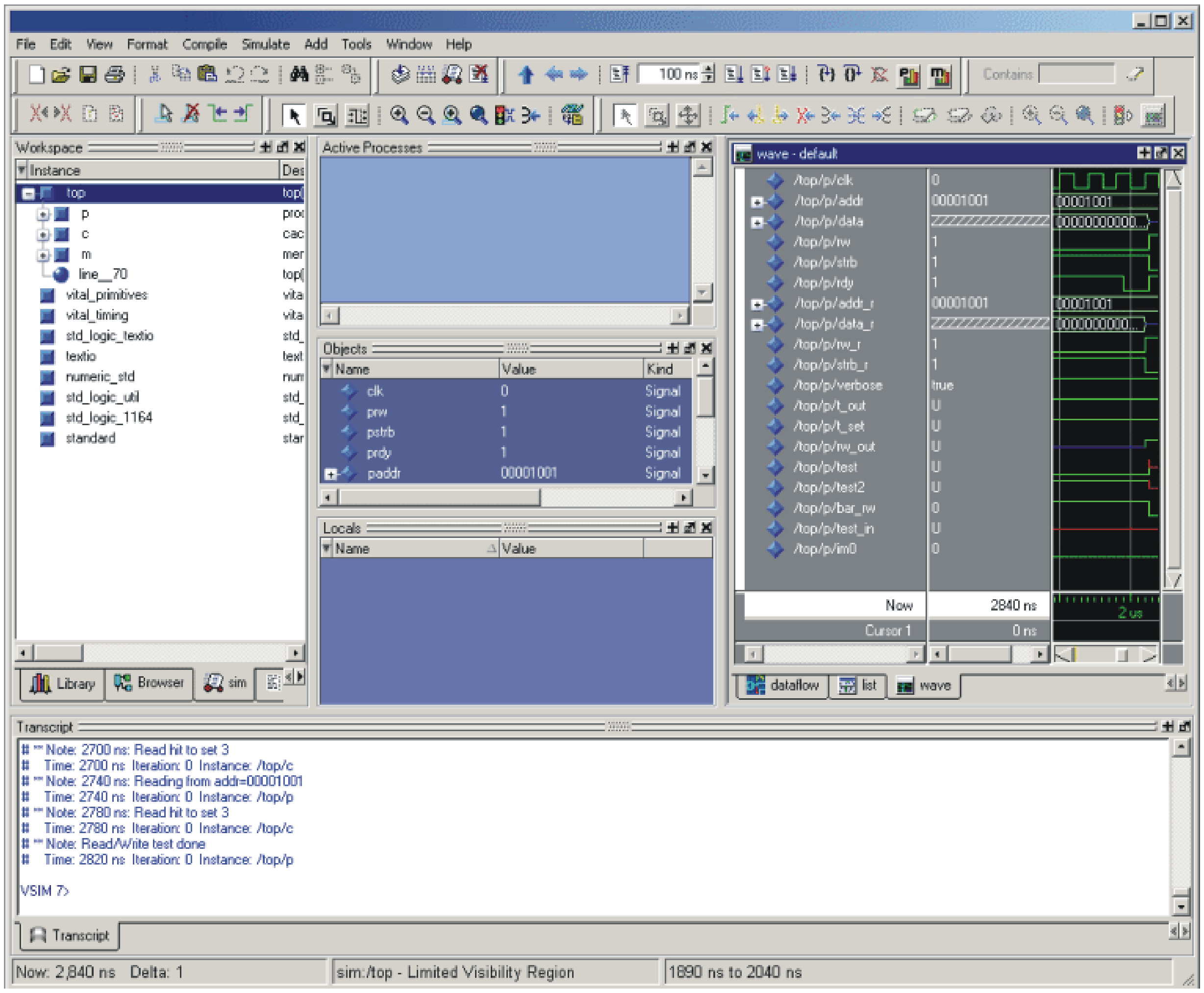
Installation Directory Pathnames

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

Chapter 2
Simulator Windows

ModelSim's graphical user interface (GUI) consists of various windows that give access to parts of your design and numerous debugging tools. Some of the windows display as panes within the ModelSim Main window and some display as windows in the Multiple Document Interface (MDI) frame.

Figure 2-1. Graphical User Interface



```
# ** Note: 2700 ns: Read hit to set 3
#    Time: 2700 ns  Iteration: 0  Instance: /top/c
# ** Note: 2740 ns: Reading from addr=00001001
#    Time: 2740 ns  Iteration: 0  Instance: /top/p
# ** Note: 2780 ns: Read hit to set 3
#    Time: 2780 ns  Iteration: 0  Instance: /top/c
# ** Note: Read/Write test done
#    Time: 2820 ns  Iteration: 0  Instance: /top/p

VSIM 7>
```

Now: 2,840 ns Delta: 1 | sim:/top - Limited Visibility Region | 1890 ns to 2040 ns

The following table summarizes all of the available windows and panes.

Table 2-1. GUI Windows and Panes

Window/pane name	Description	More details
Main	central GUI access point	Main Window
Active Processes	displays all processes that are scheduled to run during the current simulation cycle	Active Processes Pane
Dataflow	displays "physical" connectivity and lets you trace events (causality)	Dataflow Window
List	shows waveform data in a tabular format	List Window
Locals	displays data objects that are immediately visible at the current PC of the selected process	Locals Pane
Memory	a Workspace tab and MDI windows that show memories and their contents	Memory Panes
Watch	displays signal or variable values at the current simulation time	Watch Pane
Objects	displays all declared data objects in the current scope	Objects Pane
Source	a text editor for viewing and editing HDL, DO, etc. files	Source Window
Transcript	keeps a running history of commands and messages and provides a command-line interface	Transcript
Wave	displays waveforms	Wave Window
Workspace	provides easy access to projects, libraries, compiled design units, memories, etc.	Workspace

The windows and panes are customizable in that you can position and size them as you see fit, and ModelSim will remember your settings upon subsequent invocations. See Navigating the Graphic User Interface for more details.

Design Object Icons and Their Meaning

The color and shape of icons convey information about the language and type of a design object. shows the icon colors and the languages they indicate.

Table 2-2. Design Object Icons

Icon color	Design Language
light blue	Verilog or SystemVerilog
dark blue	VHDL
orange	virtual object

Here is a list of icon shapes and the design object types they indicate:

Table 2-3. Icon Shapes and Design Object Types

icon shape	example	design object type
square	🟦	any scope (VHDL block, Verilog named block, SC module, class, interface, task, function, etc.)
circle	🔵	process
diamond	🔷	valued object (signals, nets, registers, etc.)
caution sign	⚠	comparison object
diamond with red dot	🔷	an editable waveform created with the waveform editor
star	✦	transaction; The color of the star for each transaction depends on the language of the region in which the transaction stream occurs: dark blue for VHDL, light blue for Verilog and SystemVerilog, green for SystemC, magenta for PSL.

Setting Fonts

You may need to adjust font settings to accommodate the aspect ratios of wide screen and double screen displays or to handle launching ModelSim from an X-session.

Font Scaling

To change font scaling, select **Tools** > **Options** > **Adjust Font Scaling**. You'll need a ruler to complete the instructions in the lower right corner of the dialog. When you have entered the pixel and inches information, click OK to close the dialog. Then, restart ModelSim to see the change. This is a one time setting; you shouldn't have to set it again unless you change display resolution or the hardware (monitor or video card). The font scaling applies to Windows and UNIX operating systems. On UNIX systems, the font scaling is stored based on the $DISPLAY environment variable.

Controlling Fonts in an X-session

When executed via an X-session (e.g., Exceed, VNC), ModelSim uses font definitions from the .Xdefaults file. To ensure that the fonts look correct, create a .Xdefaults file with the following lines:

```
vsim*Font: -adobe-courier-medium-r-normal--*-120-*-*-*-*-*
vsim*SystemFont: -adobe-courier-medium-r-normal--*-120-*-*-*-*-*
vsim*StandardFont: -adobe-courier-medium-r-normal--*-120-*-*-*-*-*
vsim*MenuFont: -adobe-courier-medium-r-normal--*-120-*-*-*-*-*
```

Alternatively, you can choose a different font. Use the program "xlsfonts" to identify which fonts are available on your system.

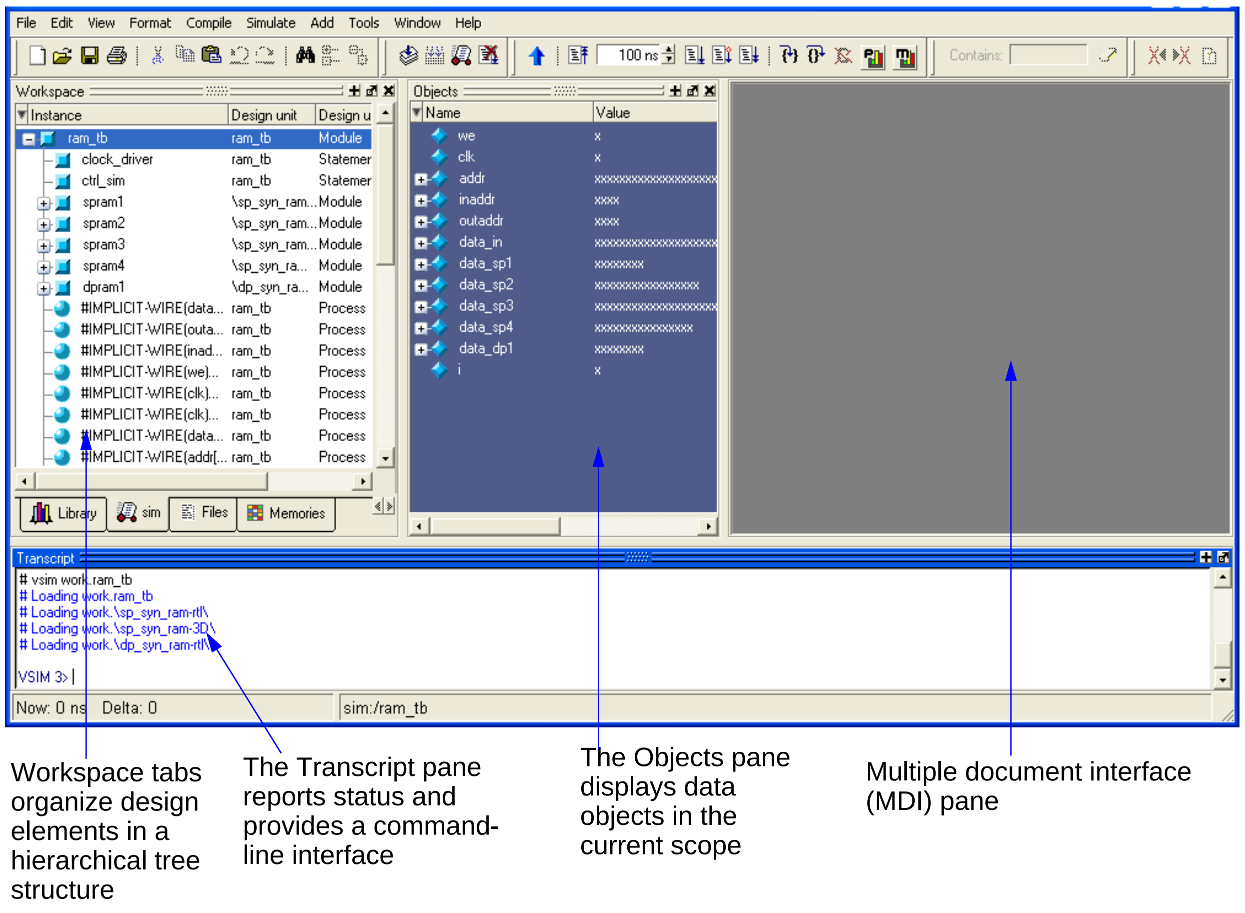
Also, the following command can be used to update the X resources if you make changes to the .Xdefaults and wish to use those changes on a UNIX machine:

```
xrdb -merge .Xdefaults
```

Main Window

The primary access point in the ModelSim GUI is called the Main window. It provides convenient access to design libraries and objects, source files, debugging commands, simulation status messages, etc. When you load a design, or bring up debugging tools, ModelSim adds panes or opens windows appropriate for your debugging environment (Figure 2-2).

Figure 2-2. Main Window



Instance	Design unit	Design u
ram_tb	ram_tb	Module
clock_driver	ram_tb	Statemer
ctrl_sim	ram_tb	Statemer
spram1	\sp_syn_ram...	Module
spram2	\sp_syn_ram...	Module
spram3	\sp_syn_ram...	Module
spram4	\sp_syn_ra...	Module
dpram1	\dp_syn_ra...	Module
#IMPLICIT-WIRE(data...	ram_tb	Process
#IMPLICIT-WIRE(outa...	ram_tb	Process
#IMPLICIT-WIRE(inad...	ram_tb	Process
#IMPLICIT-WIRE(we)...	ram_tb	Process
#IMPLICIT-WIRE(clk)...	ram_tb	Process
#IMPLICIT-WIRE(clk)...	ram_tb	Process
#IMPLICIT-WIRE(data...	ram_tb	Process
#IMPLICIT-WIRE(addr[...	ram_tb	Process

Name	Value
we	x
clk	x
addr	xxxxxxxxxxxxxxxxxxxx
inaddr	xxxx
outaddr	xxxx
data_in	xxxxxxxxxxxxxxxxxxxx
data_sp1	xxxxxxxx
data_sp2	xxxxxxxxxxxxxxxx
data_sp3	xxxxxxxxxxxxxxxxxxxx
data_sp4	xxxxxxxxxxxxxxxx
data_dp1	xxxxxxxx
i	x

```
# vsim work.ram_tb
# Loading work.ram_tb
# Loading work.\sp_syn_ram-rtl\
# Loading work.\sp_syn_ram-3D\
# Loading work.\dp_syn_ram-rtl\

VSIM 3> |
```

Now: 0 ns   Delta: 0   sim:/ram_tb

Workspace tabs organize design elements in a hierarchical tree structure

The Transcript pane reports status and provides a command-line interface

The Objects pane displays data objects in the current scope

Multiple document interface (MDI) pane

Notice some of the elements that appear:

- Workspace tabs organize and display design objects in a hierarchical tree format
- The Transcript pane tracks command history and messages and provides a command-line interface where you can enter ModelSim commands
- The Objects pane displays design objects such as signals, nets, generics, etc. in the current design scope

Workspace

The Workspace provides convenient access to projects, libraries, design files, compiled design units, simulation/dataset structures, and Waveform Comparison objects. It can be hidden or displayed by selecting **View** > **Windows** > **Workspace** (Main window).

The Workspace can display the types of tabs listed below.

- **Project tab** — Shows all files that are included in the open project. Refer to Projects for details.
- **Library tab** — Shows design libraries and compiled design units. To update the current view of the library, select a library, and then Right click > Update. See Managing Library Contents for details on library management.
- **Structure tabs** —Shows a hierarchical view of the active simulation and any open datasets. There is one tab for the current simulation (named "sim") and one tab for each open dataset. See Viewing Dataset Structure for details.

  An entry is created by each object within the design. When you select a region in a structure tab, it becomes the *current region* and is highlighted. The Source Window and Objects Pane change dynamically to reflect the information for the current 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 pane, the Active Processes Pane is updated. The Active Processes window will in turn update the Locals Pane.

  Objects can be dragged from the structure tabs to the Dataflow, List and Wave windows.

  You can toggle the display of processes by clicking in a Structure tab and selecting **View > Filter > Processes**.

  You can also control implicit wire processes using a preference variable. By default Structure tabs suppress the display of implicit wire processes. To enable the display of implicit wire processes, set PrefMain(HideImplicitWires) to 0 (select **Tools > Edit Preferences**, By Name tab, and expand the Main object).
- **Files tab** — Shows the source files for the loaded design.

  You can disable the display of this tab by setting the PrefMain(ShowFilePane) preference variable to 0. See Simulator GUI Preferences for information on setting preference variables.
- **Memories tab** — Shows a hierarchical list of all memories in the design. To display this tab, select **View > Windows > Memory**. When you select a memory on the tab, a memory contents page opens in the MDI frame. See Memory Panes.

Transcript

The Transcript portion of the Main window maintains a running history of commands that are invoked and messages that occur as you work with ModelSim. When a simulation is running, the Transcript displays a VSIM prompt, allowing you to enter command-line commands from within the graphic interface.

You can scroll backward and forward through the current work history by using the vertical scrollbar. You can also use arrow keys to recall previous commands, or copy and paste using the mouse within the window (see Main and Source Window Mouse and Keyboard Shortcuts for details).

Saving the Transcript File

Variable settings determine the filename used for saving the transcript. If either **PrefMain(file)** in the *.modelsim* file or **TranscriptFile** in the *modelsim.ini* file is set, then the transcript output is logged to the specified file. By default the **TranscriptFile** variable in *modelsim.ini* is set to *transcript*. If either variable is set, the transcript contents are always saved and no explicit saving is necessary.

If you would like to save an additional copy of the transcript with a different filename, click in the Transcript pane and then select **File > Save As**, or **File > Save**. The initial save must be made with the **Save As** selection, which stores the filename in the Tcl variable **PrefMain(saveFile)**. Subsequent saves can be made with the **Save** selection. Since no automatic saves are performed for this file, it is written only when you invoke a **Save** command. The file is written to the specified directory and records the contents of the transcript at the time of the save.

Using the Saved Transcript as a Macro (DO file)

Saved transcript files can be used as macros (DO files). Refer to the do command for more information.

Changing the Number of Lines Saved in the Transcript Window

By default, the Transcript window retains the last 5000 lines of output from the transcript. You can change this default by altering the saveLines preference variable. Setting this variable to 0 instructs the tool to retain all lines of the transcript. See Simulator GUI Preferences for details on setting preference variables.

Disabling Creation of the Transcript File

You can disable the creation of the transcript file by using the following ModelSim command immediately after ModelSim starts:

```
transcript file ""
```

Automatic Command Help

When you start typing a command at the Transcript prompt, a dropdown box appears which lists the available commands matching what has been typed so far. You may use the Up and Down arrow keys or the mouse to select the desired command. When a unique command has been entered, the command usage is presented in the drop down box.

You can disable this feature by selecting **Help > Command Completion** or by setting the *PrefMain(EnableCommandHelp)* preference variable to 0. See Simulator GUI Preferences for details on setting preference variables.

Message Viewer

The Message Viewer tab, found in the Transcript pane, allows you to easily access, organize, and analyze any Note, Warning, Error or other elaboration and runtime messages written to the transcript during the simulation run.

By default, the tool writes transcripted messages to both the transcript and the WLF file. By writing to the WLF file, the Message Viewer tab is able to organize the messages for your analysis.

Controlling the Message Viewer Data

- **Command Line** — The -msgmode argument to vsim controls where the simulator outputs the messages.

  ```
  vsim -msgmode {both | tran | wlf}
  ```

  where:

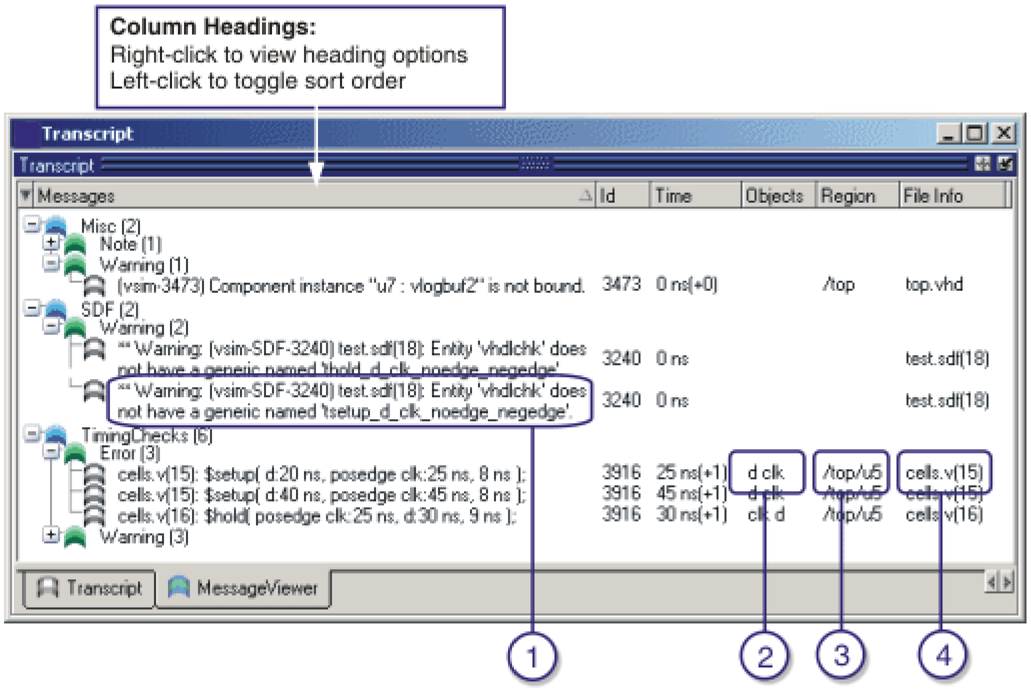
  - both — outputs messages to both the transcript and the WLF file. Default behavior.
  - tran — outputs messages only to the transcript, therefore they are not available in the Message Viewer.
  - wlf — outputs messages only to the WLF file/Message Viewer, therefore they are not available in the transcript.

- **modelsim.ini File** — The msgmode variable in the modelsim.ini file accepts the same values described above for the -msgmode argument.

Message Viewer Interface and Tasks

The Message Viewer tab does not display by default. You can bring it up after a simulation run with the **View > Message Viewer** menu item. The message viewer is also automatically displayed when you perform the dataset open command. Figure 2-3 and Table 2-4 provide an overview of the Message Viewer and several tasks you can perform.

Figure 2-3. Message Viewer Tab


Column Headings:
Right-click to view heading options
Left-click to toggle sort order
Transcript
Transcript
Transcript
MessageViewer
1
2
3
4

```
Messages                                                   Id    Time      Objects  Region   File Info
Misc (2)
  Note (1)
  Warning (1)
    (vsim-3473) Component instance "u7 : vlogbuf2" is not bound.  3473  0 ns(+0)            /top     top.vhd
SDF (2)
  Warning (2)
    ** Warning: (vsim-SDF-3240) test.sdf(18): Entity 'vhdlchk' does
    not have a generic named 'thold_d_clk_noedge_negedge'.        3240  0 ns                         test.sdf(18)
    ** Warning: (vsim-SDF-3240) test.sdf(18): Entity 'vhdlchk' does
    not have a generic named 'tsetup_d_clk_noedge_negedge'.       3240  0 ns                         test.sdf(18)
TimingChecks (6)
  Error (3)
    cells.v(15): $setup( d:20 ns, posedge clk:25 ns, 8 ns );     3916  25 ns(+1)  d clk    /top/u5  cells.v(15)
    cells.v(15): $setup( d:40 ns, posedge clk:45 ns, 8 ns );     3916  45 ns(+1)  d clk    /top/u5  cells.v(15)
    cells.v(16): $hold( posedge clk:25 ns, d:30 ns, 9 ns );      3916  30 ns(+1)  clk d    /top/u5  cells.v(16)
  Warning (3)
```

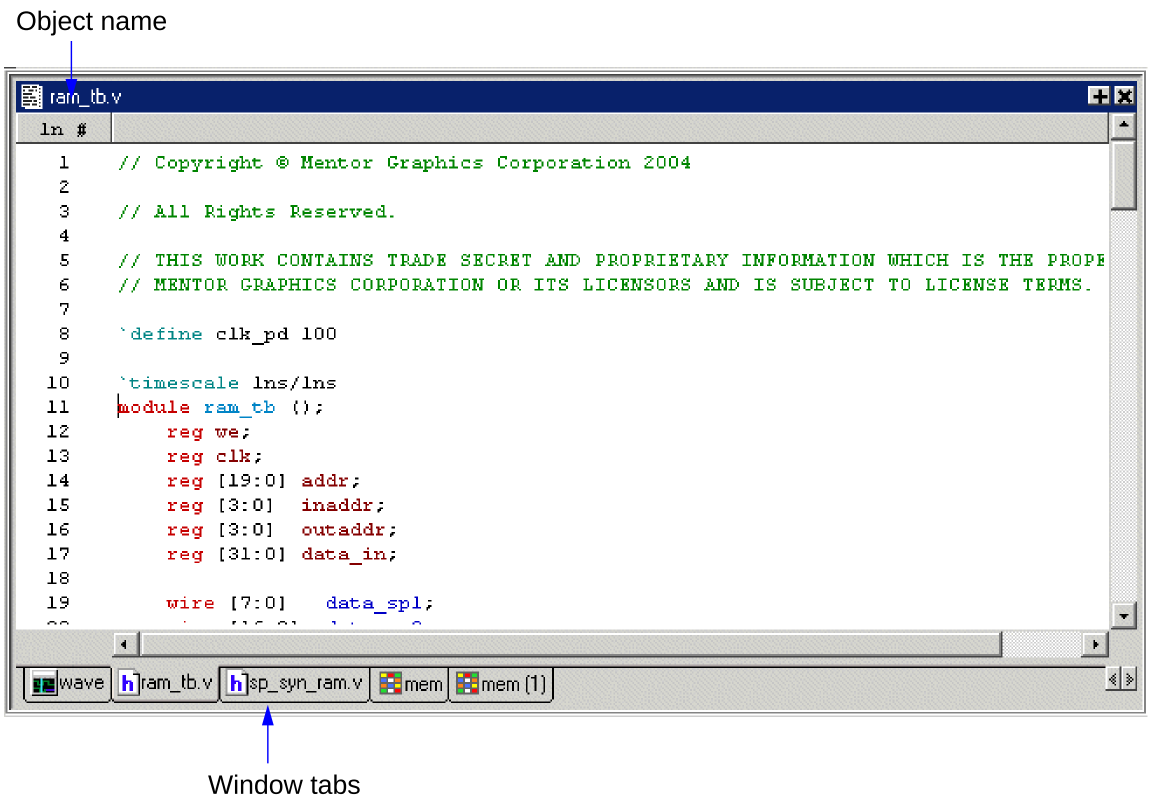
Table 2-4. Message Viewer Tasks

Icon	Task	Action
1	Display a detailed description of the message.	right click the message text then select **View Verbose Message**.
2	Open the source file and add a bookmark to the location of the object(s).	double click the object name(s).
3	Change the focus of the Workspace and Objects panes.	double click the hierarchical reference.
4	Open the source file and set a marker at the line number.	double click the file name.

Multiple Document Interface (MDI) Frame

The MDI frame is an area in the Main window where source editor, memory content, wave, and list windows display. The frame allows multiple windows to be displayed simultaneously, as shown below. A tab appears for each window.

Figure 2-4. Tabs in the MDI Frame



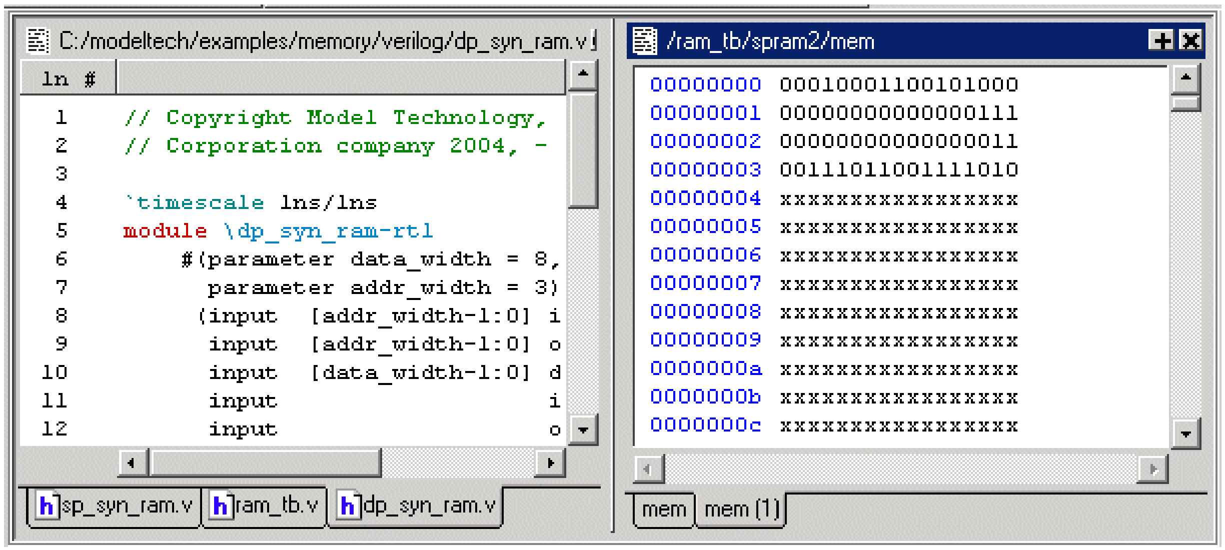
```
// Copyright © Mentor Graphics Corporation 2004

// All Rights Reserved.

// THIS WORK CONTAINS TRADE SECRET AND PROPRIETARY INFORMATION WHICH IS THE PROPE
// MENTOR GRAPHICS CORPORATION OR ITS LICENSORS AND IS SUBJECT TO LICENSE TERMS.

`define clk_pd 100

`timescale 1ns/1ns
module ram_tb ();
    reg we;
    reg clk;
    reg [19:0] addr;
    reg [3:0]  inaddr;
    reg [3:0]  outaddr;
    reg [31:0] data_in;

    wire [7:0]   data_spl;
```

The object name is displayed in the title bar at the top of the window. You can switch between the windows by clicking on a tab.

Organizing Windows with Tab Groups

The MDI can quickly become unwieldy if many windows are open. You can create "tab groups" to help organize the windows. A tab group is a collection of tabs that are separated from other groups of tabs. Figure 2-5 shows how the collection of files in Figure 2-4 could be organized into two tab groups.

Figure 2-5. Organizing Files in Tab Groups



C:/modeltech/examples/memory/verilog/dp_syn_ram.v

```
ln #
     // Copyright Model Technology,
     // Corporation company 2004, -

     `timescale 1ns/1ns
     module \dp_syn_ram-rtl
         #(parameter data_width = 8,
           parameter addr_width = 3)
          (input  [addr_width-1:0] i
           input  [addr_width-1:0] o
           input  [data_width-1:0] d
           input                   i
           input                   o
```

/ram_tb/spram2/mem

```
00000000 00010001100101000
00000001 00000000000000111
00000002 00000000000000011
00000003 00111011001111010
00000004 xxxxxxxxxxxxxxxxx
00000005 xxxxxxxxxxxxxxxxx
00000006 xxxxxxxxxxxxxxxxx
00000007 xxxxxxxxxxxxxxxxx
00000008 xxxxxxxxxxxxxxxxx
00000009 xxxxxxxxxxxxxxxxx
0000000a xxxxxxxxxxxxxxxxx
0000000b xxxxxxxxxxxxxxxxx
0000000c xxxxxxxxxxxxxxxxx
```

The commands for creating and organizing tab groups are accessed by right-clicking on any window tab. The table below describes the commands associated with tab groups:

Table 2-5. Commands for Tab Groups

Command	Description
New Tab Group	Creates a new tab group containing the selected tab
Move Next Group	Moves the selected tab to the next group in the MDI
Move Prev Group	Moves the selected tab to the previous group in the MDI
View > Vertical / Horizontal	Arranges tab groups top-to-bottom (vertical) or right-to-left (horizontal)

Note that you can also move the tabs within a tab group by dragging them with the middle mouse button.

Navigating in the Main Window

The Main window can contain of a number of "panes" and sub-windows that display various types of information about your design, simulation, or debugging session. Here are a few important points to keep in mind about the Main window interface:

- Windows/panes can be resized, moved, zoomed, undocked, etc. and the changes are persistent.

You have a number of options for re-sizing, re-positioning, undocking/redocking, and generally modifying the physical characteristics of windows and panes.

Windows and panes can be undocked from the main window by pressing the Undock button in the header or by using the **view -undock <window_name>** command. For example, **view -undock objects** will undock the Objects window. The default docked or undocked status of each window or pane can be set with the **PrefMain(ViewUnDocked) <window_name>** preference variable.

When you exit ModelSim, the current layout is saved so that it appears the same the next time you invoke the tool.

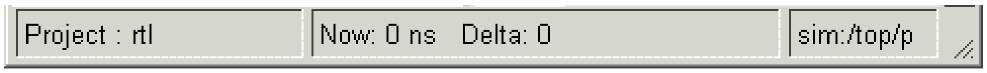
- Menus are context sensitive.

  The menu items that are available and how certain menu items behave depend on which pane or window is active. For example, if the sim tab in the Workspace is active and you choose Edit from the menu bar, the Clear command is disabled. However, if you click in the Transcript pane and choose Edit, the Clear command is enabled. The active pane is denoted by a blue title bar.

For more information, see Navigating the Graphic User Interface.

Main Window Status Bar

Figure 2-6. Main Window Status Bar



Project : rtl | Now: 0 ns Delta: 0 | sim:/top/p

Fields at the bottom of the Main window provide the following information about the current simulation:

Table 2-6. Information Displayed in Status Bar

Field	Description
Project	name of the current project
Now	the current simulation time
Delta	the current simulation iteration number
Profile Samples	the number of profile samples collected during the current simulation
Memory	the total memory used during the current simulation
environment	name of the current context (object selected in the active Structure tab of the Workspace)
line/column	line and column numbers of the cursor in the active Source window

Main Window Toolbar

Buttons on the Main window toolbar give you quick access to various ModelSim commands and functions.

Table 2-7. Main Window Toolbar Buttons

Button	Menu equivalent	Command equivalents
New File create a new source file	File > New > Source	
Open open the Open File dialog	File > Open	
Save save the contents of the active pane	File > Save	
Print open the Print dialog	File > Print	
Cut cut the selected text to the clipboard	Edit > Cut	
Copy copy the selected text to the clipboard	Edit > Copy	
Paste paste the clipboard text	Edit > Paste	
Undo undo the last edit	Edit > Undo	
Redo redo the last undone edit	Edit > Redo	
Find find text in the active window	Edit > Find	
Collapse All collapse all instances in the active window	Edit > Expand > Collapse All	
Expand All expand all instance in the active window	Edit > Expand > Expand All	

Table 2-7. Main Window Toolbar Buttons

Button	Menu equivalent	Command equivalents
Compile open the Compile Source Files dialog to select files for compilation	Compile > Compile	vcom vlog
Compile All compile all files in the open project	Compile > Compile All	vcom vlog
Simulate load the selected design unit or simulation configuration object	Simulate > Start Simulation	vsim
Break stop the current simulation run	Simulate > Break	
Environment up move up one level in the design hierarchy		
Environment back navigate backward to a previously selected context		
Environment forward navigate forward to a previously selected context		
Restart reload the design elements and reset the simulation time to zero, with the option of maintaining various settings and objects	Simulate > Run > Restart	restart
Run Length specify the run length for the current simulation	Simulate > Runtime Options	run
Run run the current simulation for the specified run length	Simulate > Run > Run *default_run_length*	run
Continue Run continue the current simulation run until the end of the specified run length or until it hits a breakpoint or specified break event	Simulate > Run > Continue	run -continue

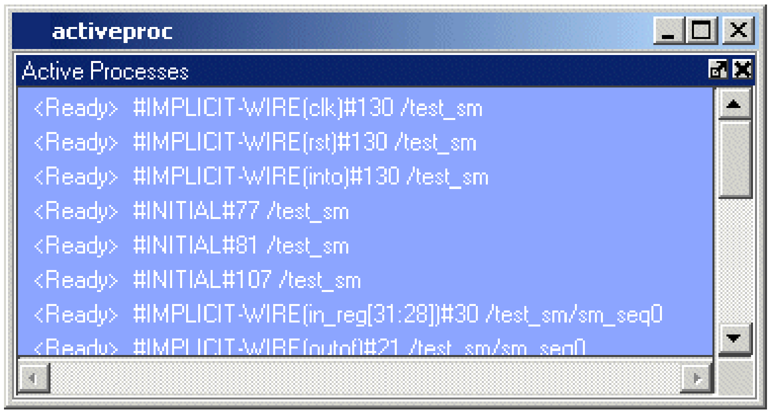
Table 2-7. Main Window Toolbar Buttons

Button	Menu equivalent	Command equivalents
Run -All run the current simulation forever, or until it hits a breakpoint or specified break event	Simulate > Run > Run -All	run -all
Step step the current simulation to the next statement	Simulate > Run > Step	step
Step Over HDL statements are executed but treated as simple statements instead of entered and traced line by line	Simulate > Run > Step -Over	step -over
Contains filter items in Objects pane		
Show Language Templates display language templates	View > Source > Show Language Templates	

Active Processes Pane

The Active Processes pane displays a list of HDL processes. Processes are also displayed in the structure tabs of the Main window Workspace. To filter displayed processes in the structure tabs, select **View** > **Filter** > **Processes**.

Figure 2-7. Active Processes Pane



```
<Ready>  #IMPLICIT-WIRE(clk)#130 /test_sm
<Ready>  #IMPLICIT-WIRE(rst)#130 /test_sm
<Ready>  #IMPLICIT-WIRE(into)#130 /test_sm
<Ready>  #INITIAL#77 /test_sm
<Ready>  #INITIAL#81 /test_sm
<Ready>  #INITIAL#107 /test_sm
<Ready>  #IMPLICIT-WIRE(in_reg[31:28])#30 /test_sm/sm_seq0
<Ready>  #IMPLICIT-WIRE(outof)#21 /test_sm/sm_seq0
```

Process Status

Each object 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. If you select a "Ready" process, it will be executed next by the simulator.
- **<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.

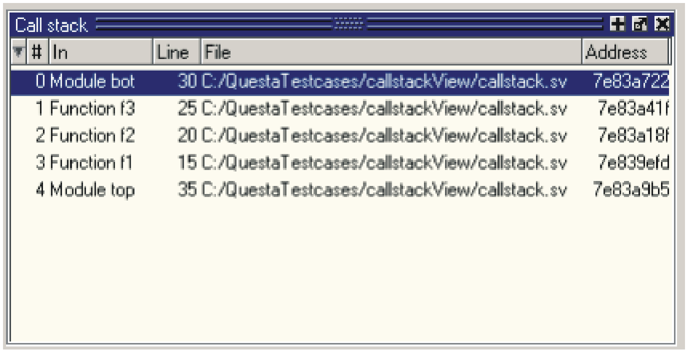
Call Stack Pane

The Call Stack pane displays the current call stack when you single step your simulation or when the simulation has encountered a breakpoint. When debugging your design, you can use the call stack data to analyze the depth of function calls, which include Verilog functions and tasks and VHDL functions and procedures, that led up to the current point of the simulation.

Accessing the Call Stack Pane

View > Call Stack

Figure 2-8. Call Stack Pane



#	In	Line	File	Address
0	Module bot	30	C:/QuestaTestcases/callstackView/callstack.sv	7e83a722
1	Function f3	25	C:/QuestaTestcases/callstackView/callstack.sv	7e83a41f
2	Function f2	20	C:/QuestaTestcases/callstackView/callstack.sv	7e83a18f
3	Function f1	15	C:/QuestaTestcases/callstackView/callstack.sv	7e839efd
4	Module top	35	C:/QuestaTestcases/callstackView/callstack.sv	7e83a9b5

Using the Call Stack Pane

The Call Stack pane contains five columns of information to assist you in debugging your design:

- # — indicates the depth of the function call, with the most recent at the top.
- In — indicates the function.
- Line — indicates the line number containing the function call.
- File — indicates the location of the file containing the function call.

- Address — indicates the address of the execution in a foreign subprogram, such as C.

The Call Stack pane allows you to perform the following actions within the pane:

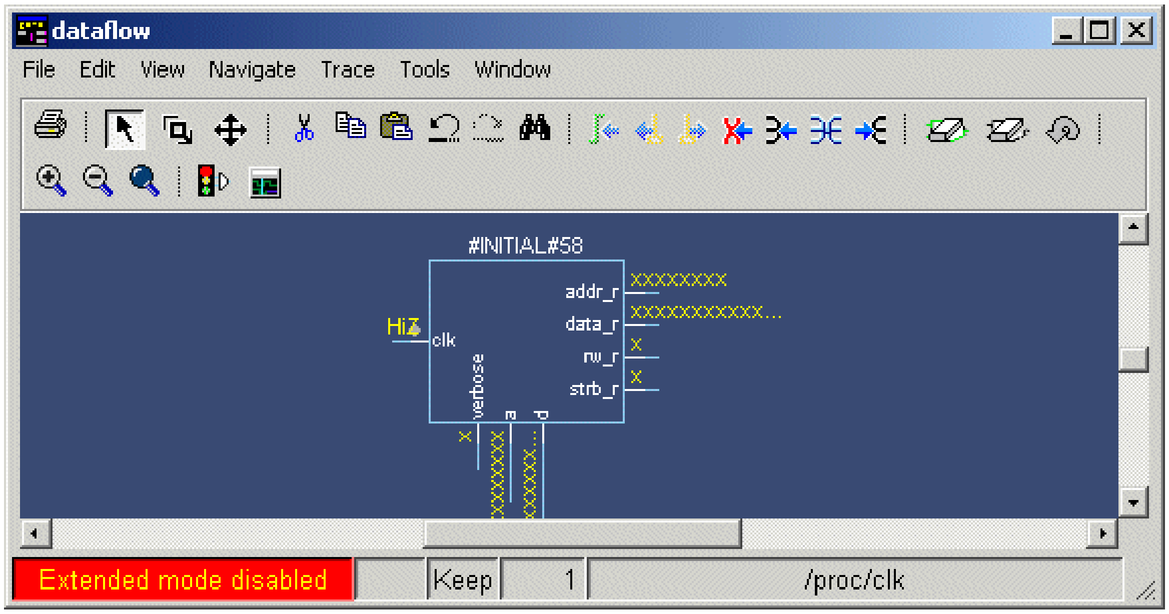
- Double-click on the line of any function call:
  - o Displays the local variables at that level in the Locals Pane.
  - o Displays the corresponding source code in the Source Window.
- Right-click in the column headings
  - o Displays a pop-up window that allows you to show or hide columns.

Dataflow Window

The Dataflow window allows you to explore the "physical" connectivity of your design.

Note
OEM versions of ModelSim have limited Dataflow functionality. Many of the features described below will operate differently. The window will show only one process and its attached signals or one signal and its attached processes, as displayed in Figure 2-9.

Figure 2-9. Dataflow Window



The Dataflow window displays:

- processes

- signals, nets, and registers

The window has built-in mappings for all Verilog primitive gates (i.e., AND, OR, PMOS, NMOS, etc.). For components other than Verilog primitives, you can define a mapping between processes and built-in symbols. See Symbol Mapping for details.

Dataflow Window Toolbar

The buttons on the Dataflow window toolbar are described below.

Table 2-8. Dataflow Window Toolbar

Button	Menu equivalent
🖨 **Print** — print the current view of the Dataflow window	File > Print (Windows) File > Print Postscript (UNIX)
↖ **Select mode** — set left mouse button to select mode and middle mouse button to zoom mode	View > Select
⧉ **Zoom mode** — set left mouse button to zoom mode and middle mouse button to pan mode	View > Zoom
✥ **Pan mode** — set left mouse button to pan mode and middle mouse button to zoom mode	View > Pan
✂ **Cut** — cut the selected object(s)	Edit > Cut
⎘ **Copy** — copy the selected object(s)	Edit > Copy
📋 **Paste** — paste the previously cut or copied object(s)	Edit > Paste
↶ **Undo** — undo the last action	Edit > Undo
↷ **Redo** — redo the last undone action	Edit > Redo
🔍 **Find** — search for an instance or signal	Edit > Find

Table 2-8. Dataflow Window Toolbar

Button	Menu equivalent
Trace input net to event — move the next event cursor to the next input event driving the selected output	Trace > Trace next event
Trace Set — jump to the source of the selected input event	Trace > Trace event set
Trace Reset — return the next event cursor to the selected output	Trace > Trace event reset
Trace net to driver of X — step back to the last driver of an unknown value	Trace > TraceX
Expand net to all drivers — display driver(s) of the selected signal, net, or register	Navigate > Expand net to drivers
Expand net to all drivers and readers — display driver(s) and reader(s) of the selected signal, net, or register	Navigate > Expand net
Expand net to all readers — display reader(s) of the selected signal, net, or register	Navigate > Expand net to readers
Erase highlight — clear the green highlighting which identifies the path you've traversed through the design	Edit > Erase highlight
Erase all — clear the window	Edit > Erase all
Regenerate — clear and redraw the display using an optimal layout	Edit > Regenerate
Zoom In — zoom in by a factor of two from current view	none
Zoom Out — zoom out by a factor of two from current view	none
Zoom Full — zoom out to show all components in the window	none

Table 2-8. Dataflow Window Toolbar

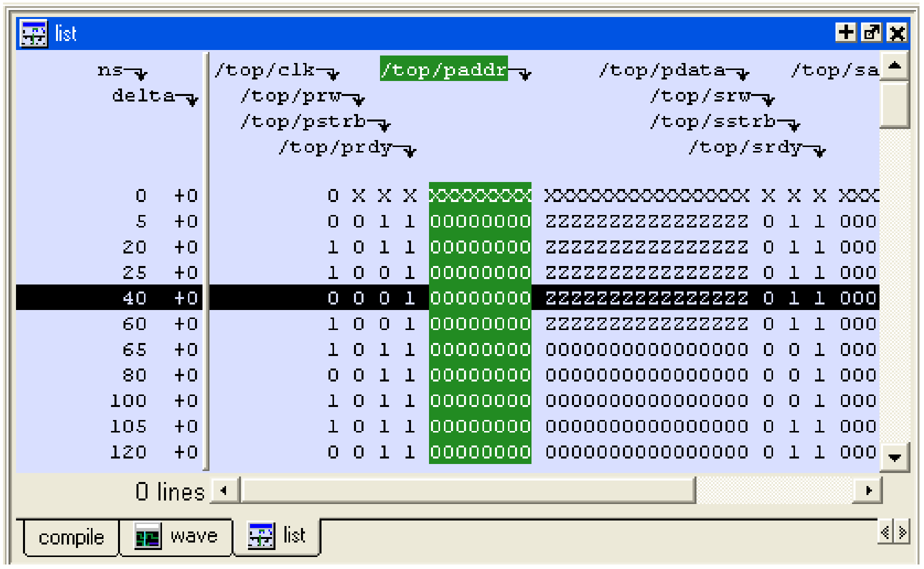
Button	Menu equivalent
Stop Drawing — halt any drawing currently happening in the window	none
Show Wave — display the embedded wave viewer pane	View > Show Wave

List Window

The List window displays the results of your simulation run in tabular format. The window is divided into two adjustable columns, which allow you to scroll horizontally through the listing on the right, while keeping time and delta visible on the left.

The List window opens by default in the MDI frame of the Main window as shown in Figure 2-10.

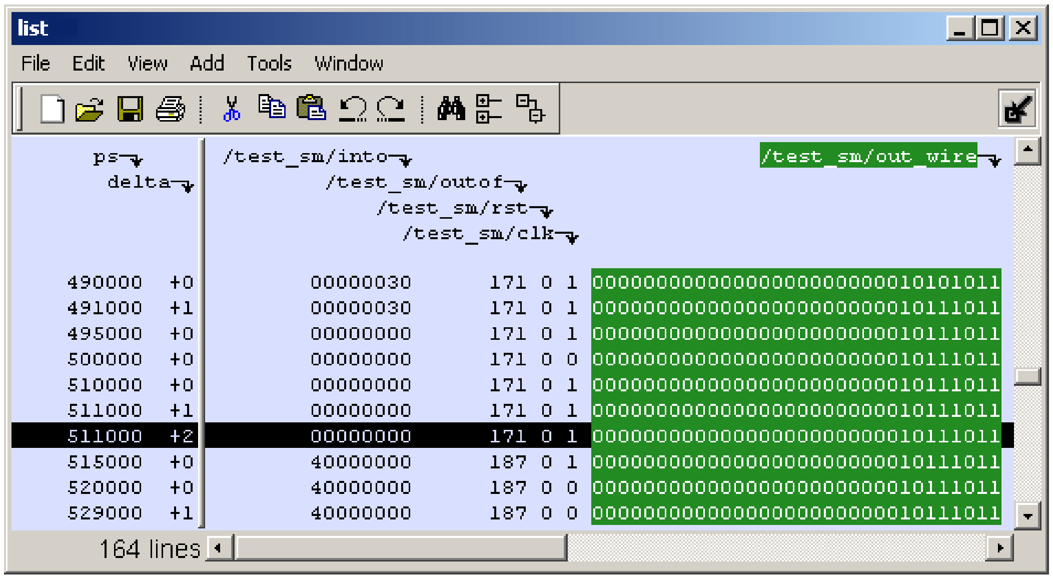
Figure 2-10. List Window Docked in Main Window MDI Frame



```
      ns      /top/clk      /top/paddr       /top/pdata      /top/sa
     delta      /top/prw                         /top/srw
                /top/pstrb                       /top/sstrb
                   /top/prdy                        /top/srdy

   0   +0          0 X X X XXXXXXXX XXXXXXXXXXXXXXXX X X X XXX
   5   +0          0 0 1 1 00000000 ZZZZZZZZZZZZZZZZ 0 1 1 000
  20   +0          1 0 1 1 00000000 ZZZZZZZZZZZZZZZZ 0 1 1 000
  25   +0          1 0 0 1 00000000 ZZZZZZZZZZZZZZZZ 0 1 1 000
  40   +0          0 0 0 1 00000000 ZZZZZZZZZZZZZZZZ 0 1 1 000
  60   +0          1 0 0 1 00000000 ZZZZZZZZZZZZZZZZ 0 1 1 000
  65   +0          1 0 1 1 00000000 0000000000000000 0 0 1 000
  80   +0          0 0 1 1 00000000 0000000000000000 0 0 1 000
 100   +0          1 0 1 1 00000000 0000000000000000 0 0 1 000
 105   +0          1 0 1 1 00000000 0000000000000000 0 1 1 000
 120   +0          0 0 1 1 00000000 0000000000000000 0 1 1 000
```

The window can be undocked from the Main window by clicking the Undock button in the window header or by using the **view -undock list** command.

Figure 2-11. List Window Undocked



```
      ps            /test_sm/into
         delta              /test_sm/outof
                                   /test_sm/rst
                                      /test_sm/clk                         /test_sm/out_wire

  490000  +0       00000030     171  0  1  00000000000000000000000010101011
  491000  +1       00000030     171  0  1  00000000000000000000000010111011
  495000  +0       00000000     171  0  1  00000000000000000000000010111011
  500000  +0       00000000     171  0  0  00000000000000000000000010111011
  510000  +0       00000000     171  0  1  00000000000000000000000010111011
  511000  +1       00000000     171  0  1  00000000000000000000000010111011
  511000  +2       00000000     171  0  1  00000000000000000000000010111011
  515000  +0       40000000     187  0  1  00000000000000000000000010111011
  520000  +0       40000000     187  0  0  00000000000000000000000010111011
  529000  +1       40000000     187  0  0  00000000000000000000000010111011
    164 lines
```

The following type of objects can be viewed in the List pane:

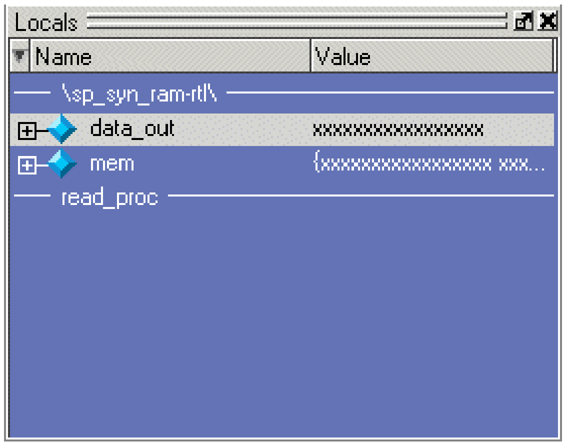
- VHDL — signals, aliases, process variables, and shared variables
- Verilog — nets, registers, and variables
- Virtuals — Virtual signals and functions

Locals Pane

The Locals pane displays data objects that are immediately visible from the statement that will be executed next (that statement is denoted by a blue arrow in the Source editor window). The contents of the window change from one statement to the next.

The Locals pane includes two columns. The first column lists the names of the immediately visible data objects. The second column lists the current value(s) associated with each name.

Figure 2-12. Locals Pane

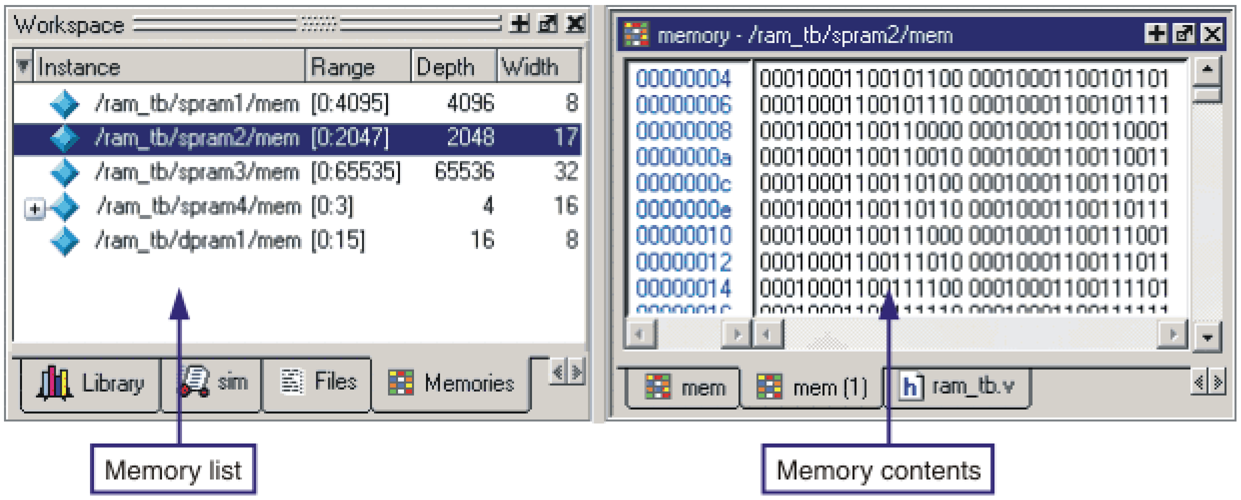


Name	Value
\sp_syn_ram-rtl\	
data_out	xxxxxxxxxxxxxxxx
mem	{xxxxxxxxxxxxxxxx xxx...
read_proc	

Memory Panes

The Main window lists all memories in your design in the Memories tab of the Main window Workspace and displays the contents of a selected memory in the Main window MDI frame.

Figure 2-13. Memory Panes



Instance	Range	Depth	Width
/ram_tb/spram1/mem	[0:4095]	4096	8
/ram_tb/spram2/mem	[0:2047]	2048	17
/ram_tb/spram3/mem	[0:65535]	65536	32
/ram_tb/spram4/mem	[0:3]	4	16
/ram_tb/dpram1/mem	[0:15]	16	8

```
00000004  00010001100101100 00010001100101101
00000006  00010001100101110 00010001100101111
00000008  00010001100110000 00010001100110001
0000000a  00010001100110010 00010001100110011
0000000c  00010001100110100 00010001100110101
0000000e  00010001100110110 00010001100110111
00000010  00010001100111000 00010001100111001
00000012  00010001100111010 00010001100111011
00000014  00010001100111100 00010001100111101
```

The memory list is from the top-level of the design. In other words, it is not sensitive to the context selected in the Structure tab.

ModelSim identifies certain kinds of arrays in various scopes as memories. Memory identification depends on the array element kind as well as the overall array kind (i.e. associative array, unpacked array, etc.).

Table 2-9. Memories

	VHDL	Verilog/SystemVerilog
Element kind	enum[1], std_logic_vector, std_bit_vector, or integer.	any integral type. (i.e. integer_type): shortint, int, longint, byte, bit (2 state), logic, reg, integer, time (4 state), packed_struct / packed_union (2 state), packed_struct / packed_union (4 state), packed_array (single-Dim, multi-D, 2 state and 4 state), enum or string.
Scope: recognizable in	architecture, process, or record	module, interface, package, compilation unit, struct, or static variables within a task / function / named block / class
Array kind	single-dimensional or multi-dimensional	any combination of unpacked, dynamic and associative arrays[2]

1. These enumerated type value sets must have values that are longer than one character. The listed width is the number of entries in the enumerated type definition and the depth is the size of the array itself.
2. Any combination of unpacked, dynamic, and associative arrays is considered a memory, provided the leaf level of the data structure is a string or an integral type.

Associative Arrays in Verilog/SystemVerilog

For an associative array to be recognized as a memory, the index must be of an integral type (see above) or wildcard type.

For associative arrays, the element kind can be any type allowed for fixed-size arrays.

Viewing Single and Multidimensional Memories

Single dimensional arrays of integers are interpreted as 2D memory arrays. In these cases, the word width listed in the Memory List pane is equal to the integer size, and the depth is the size of the array itself.

Memories with three or more dimensions display with a plus sign ’+’ next to their names in the Memory List. Click the ’+’ to show the array indices under that level. When you finally expand down to the 2D level, you can double-click on the index, and the data for the selected 2D slice of the memory will appear in a memory contents pane in the MDI frame.

Viewing Packed Arrays

By default packed dimensions are treated as single vectors in the memory contents pane. To expand packed dimensions of packed arrays, select **View > Memory Contents > Expand Packed Memories**.

To change the permanent default, edit the PrefMemory(ExpandPackedMem) variable. This variable affects only packed arrays. If the variable is set to 1, the packed arrays are treated as unpacked arrays and are expanded along the packed dimensions such that they appear as a linearized bit vector. See Simulator GUI Preferences for details on setting preference variables.

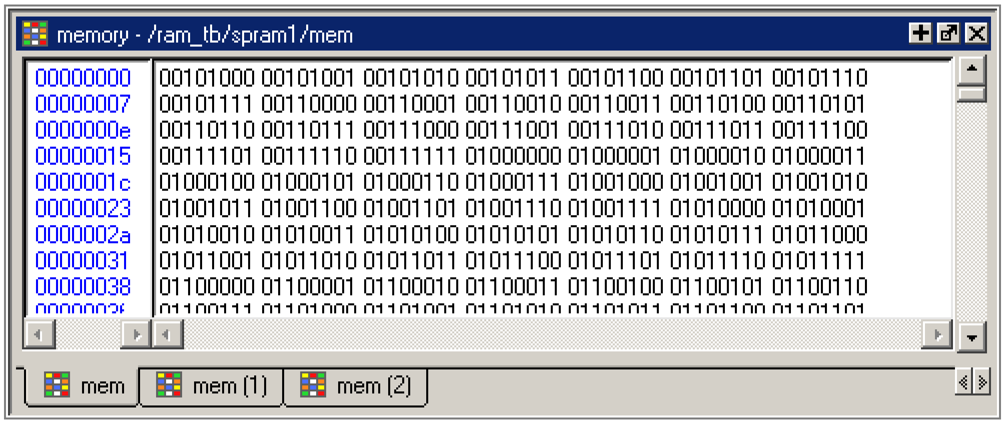
Viewing Memory Contents

When you double-click an instance on the Memory tab, ModelSim automatically displays a memory contents pane in the MDI frame (see Multiple Document Interface (MDI) Frame). You can also enter the command **add mem <instance>** at the **vsim** command prompt.

Viewing Multiple Memory Instances

You can view multiple memory instances simultaneously. A memory tab appears in the MDI frame for each instance you double-click in the Memory list.

Figure 2-14. Viewing Multiple Memories



```
00000000 00101000 00101001 00101010 00101011 00101100 00101101 00101110
00000007 00101111 00110000 00110001 00110010 00110011 00110100 00110101
0000000e 00110110 00110111 00111000 00111001 00111010 00111011 00111100
00000015 00111101 00111110 00111111 01000000 01000001 01000010 01000011
0000001c 01000100 01000101 01000110 01000111 01001000 01001001 01001010
00000023 01001011 01001100 01001101 01001110 01001111 01010000 01010001
0000002a 01010010 01010011 01010100 01010101 01010110 01010111 01011000
00000031 01011001 01011010 01011011 01011100 01011101 01011110 01011111
00000038 01100000 01100001 01100010 01100011 01100100 01100101 01100110
0000003f 01100111 01101000 01101001 01101010 01101011 01101100 01101101
```

See Organizing Windows with Tab Groups for more information on tabs.

Saving Memory Formats in a DO File

You can save all open memory instances and their formats (e.g., address radix, data radix, etc.) by creating a DO file. With the memory tab active, select **File > Save As**. The Save memory format dialog box opens, where you can specify the name for the saved file. By default it is named *mem.do*. The file will contain all open memory instances and their formats. To load it at a later time, select **File > Load**.

Direct Address Navigation

You can navigate to any address location directly by editing the address in the address column. Double-click on any address, type in the desired address, and hit **Enter**. The address display scrolls to the specified location.

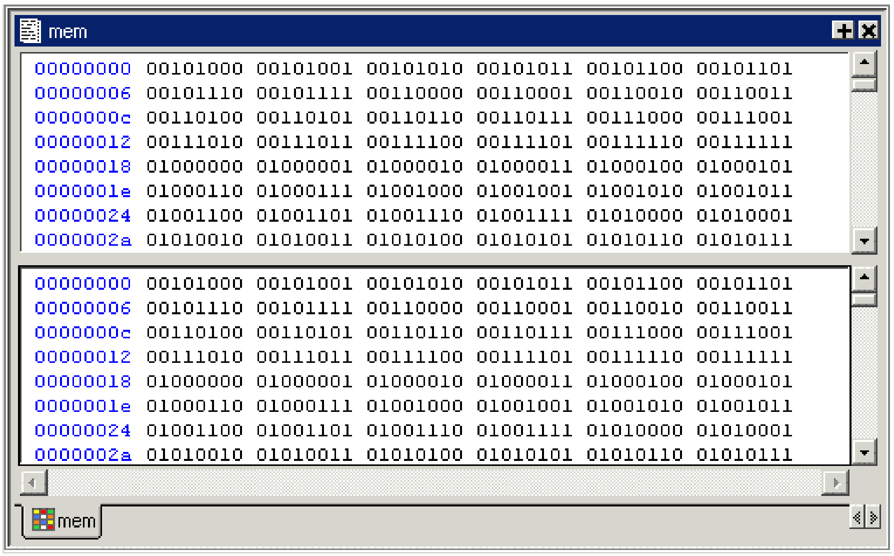
Splitting the Memory Contents Pane

To split a memory contents window into two screens displaying the contents of a single memory instance, so any one of the following:

- select **Memories > Split Screen** if the Memory Contents Pane is docked in the Main window,
- select **View > Split Screen** if the Memory Contents Pane is undocked,
- right-click in the pane and select **Split Screen** from the pop-up menu.

This allows you to view different address locations within the same memory instance simultaneously.

Figure 2-15. Split Screen View of Memory Contents



```
00000000 00101000 00101001 00101010 00101011 00101100 00101101
00000006 00101110 00101111 00110000 00110001 00110010 00110011
0000000c 00110100 00110101 00110110 00110111 00111000 00111001
00000012 00111010 00111011 00111100 00111101 00111110 00111111
00000018 01000000 01000001 01000010 01000011 01000100 01000101
0000001e 01000110 01000111 01001000 01001001 01001010 01001011
00000024 01001100 01001101 01001110 01001111 01010000 01010001
0000002a 01010010 01010011 01010100 01010101 01010110 01010111
```

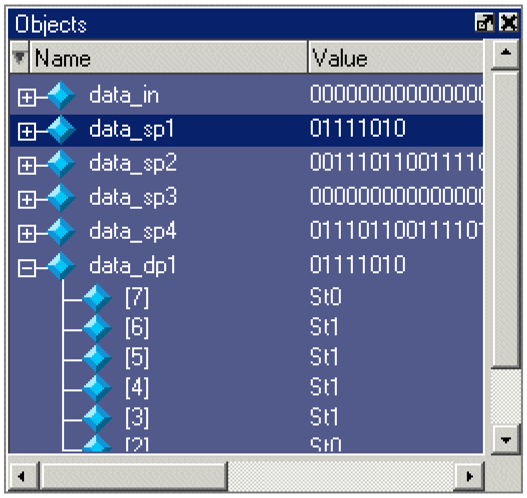
```
00000000 00101000 00101001 00101010 00101011 00101100 00101101
00000006 00101110 00101111 00110000 00110001 00110010 00110011
0000000c 00110100 00110101 00110110 00110111 00111000 00111001
00000012 00111010 00111011 00111100 00111101 00111110 00111111
00000018 01000000 01000001 01000010 01000011 01000100 01000101
0000001e 01000110 01000111 01001000 01001001 01001010 01001011
00000024 01001100 01001101 01001110 01001111 01010000 01010001
0000002a 01010010 01010011 01010100 01010101 01010110 01010111
```

Objects Pane

The Objects pane shows the names and current values of declared data objects in the current region (selected in the structure tabs of the Workspace). Data objects include signals, nets, registers, constants and variables not declared in a process, generics, parameters.

Clicking an entry in the window highlights that object in the Dataflow and Wave windows. Double-clicking an entry highlights that object in a Source editor window (opening a Source editor window if one is not open already). You can also right click an object name and add it to the List or Wave window, or the current log file.

Figure 2-16. Objects Pane



Name	Value
data_in	00000000000000
data_sp1	01111010
data_sp2	00111011001111
data_sp3	00000000000000
data_sp4	01110110011110
data_dp1	01111010
[7]	St0
[6]	St1
[5]	St1
[4]	St1
[3]	St1
[2]	St0

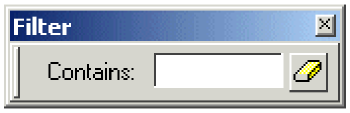
Filtering the Objects List

You can filter the objects list by name or by object type.

Filtering by Name

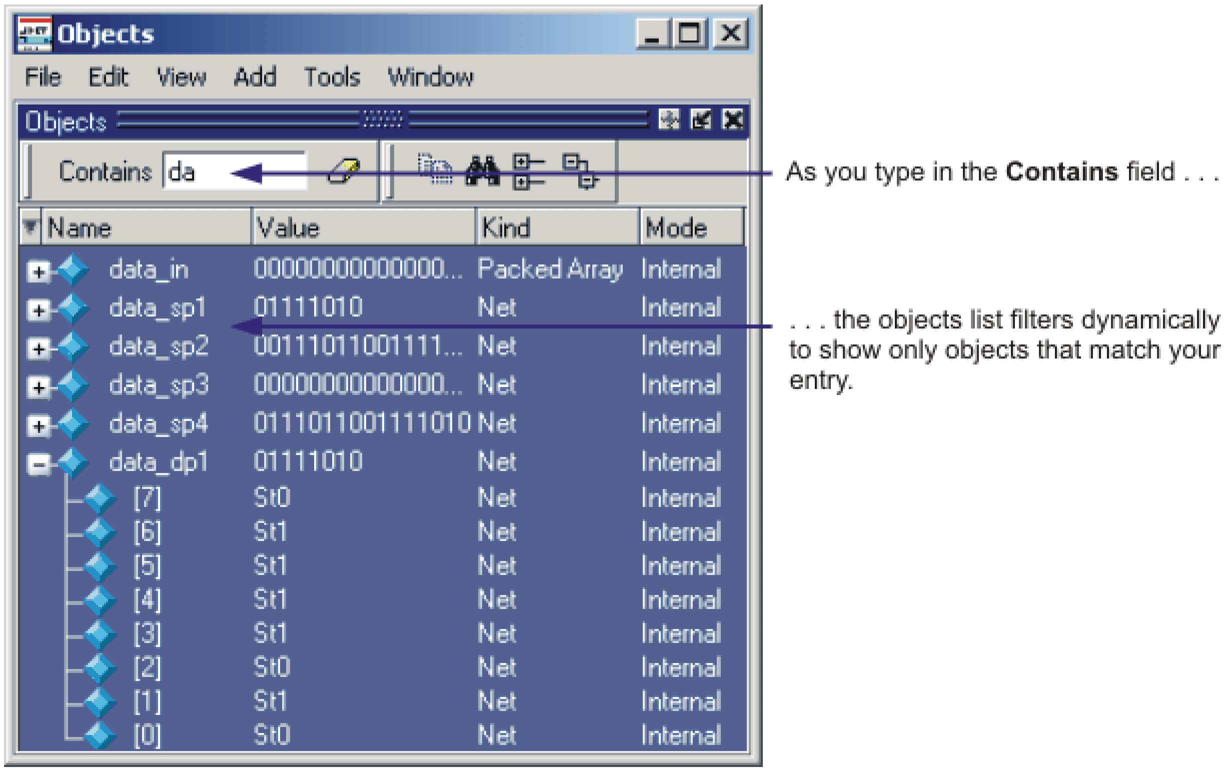
To filter by name, undock the Objects pane from the Main window and start typing letters in the **Contains** field in the toolbar.

Figure 2-17. Objects Filter



As you type, the objects list filters to show only those signals that contain those letters.

Figure 2-18. Filtering the Objects List by Name



As you type in the **Contains** field . . .

. . . the objects list filters dynamically to show only objects that match your entry.

Name	Value	Kind	Mode
data_in	00000000000000...	Packed Array	Internal
data_sp1	01111010	Net	Internal
data_sp2	00111011001111...	Net	Internal
data_sp3	00000000000000...	Net	Internal
data_sp4	0111011001111010	Net	Internal
data_dp1	01111010	Net	Internal
[7]	St0	Net	Internal
[6]	St1	Net	Internal
[5]	St1	Net	Internal
[4]	St1	Net	Internal
[3]	St1	Net	Internal
[2]	St0	Net	Internal
[1]	St1	Net	Internal
[0]	St0	Net	Internal

To display all objects again, click the Eraser icon to clear the entry.

Filters are stored relative to the region selected in the Structure window. If you re-select a region that had a filter applied, that filter is restored. This allows you to apply different filters to different regions.

Filtering by Signal Type

The **View** > **Filter** menu selection allows you to specify which signal types to display in the Objects window. Multiple options can be selected.

Source Window

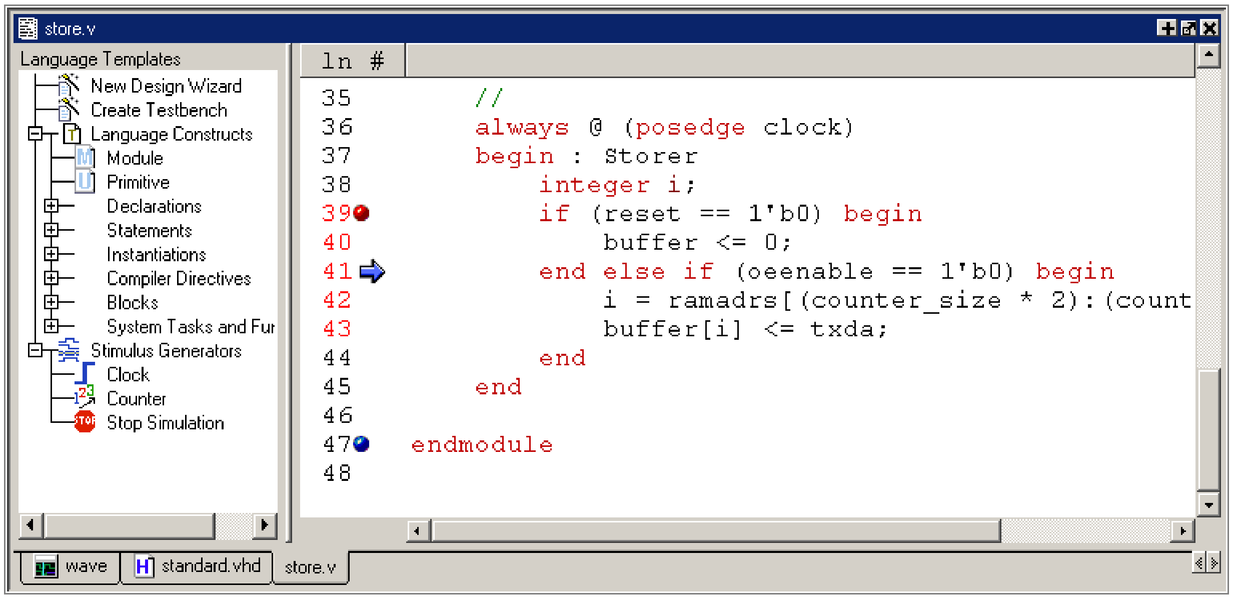
Source files display by default in the MDI frame of the Main window. The window can be undocked from the Main window by pressing the Undock button in the window header or by using the **view -undock source** command.

You can edit source files as well as set breakpoints, step through design files, and view code coverage statistics.

By default, the Source window displays your source code with line numbers. You may also see the following graphic elements:

- Red line numbers — denote lines on which you can set a breakpoint
- Blue arrow — denotes the currently active line or a process that you have selected in the Active Processes Pane
- Red circles — denote file-line breakpoints; gray circles denote breakpoints that are currently disabled
- Blue circles — denote line bookmarks
- Language Templates pane — displays Language Templates (Figure 2-19)

Figure 2-19. Source Window Showing Language Templates



```
      //
      always @ (posedge clock)
      begin : Storer
          integer i;
          if (reset == 1'b0) begin
              buffer <= 0;
          end else if (oeenable == 1'b0) begin
              i = ramadrs[(counter_size * 2):(count
              buffer[i] <= txda;
          end
      end

  endmodule

```

Opening Source Files

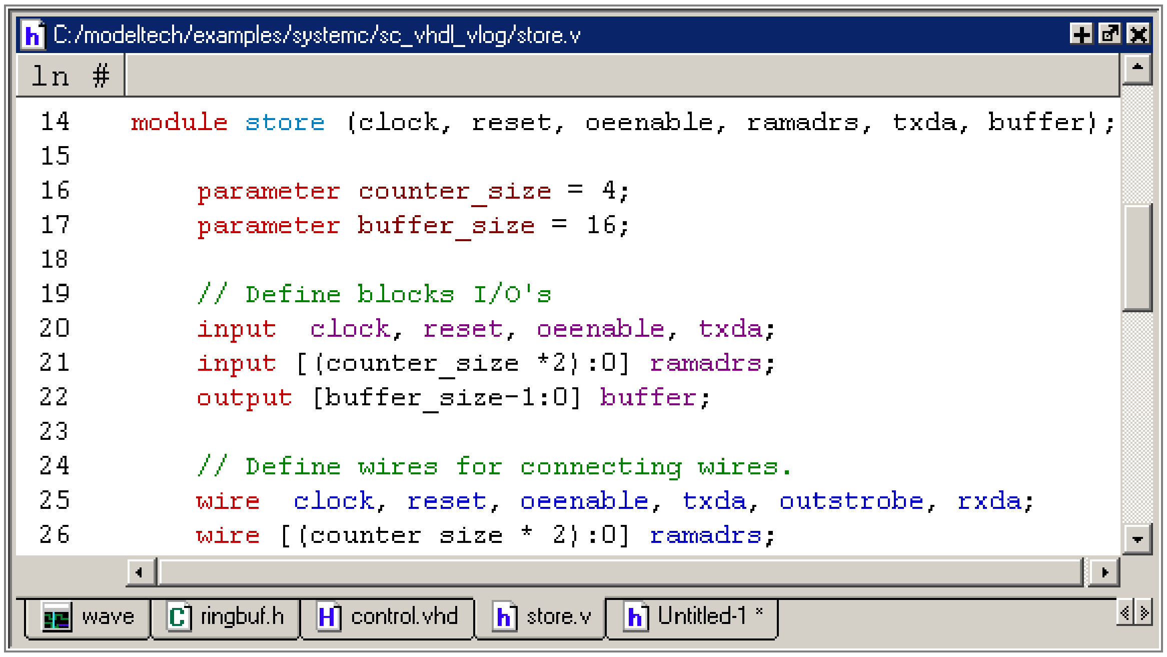
You can open source files using the **File > Open** command. Alternatively, you can open source files by double-clicking objects in other windows. For example, if you double-click an item in the Objects window or in the structure tab of the Workspace, the underlying source file for the object will open, and the cursor will scroll to the line where the object is defined.

By default files you open from within the design (e.g., by double-clicking an object in the Objects pane) open in Read Only mode. To make the file editable, right-click in the Source window and select Read Only. To change this default behavior, set the PrefSource(ReadOnly) variable to 0. See Simulator GUI Preferences for details on setting preference variables.

Displaying Multiple Source Files

By default each file you open or create is marked by a window tab, as shown in the graphic below.

Figure 2-20. Displaying Multiple Source Files



```
module store (clock, reset, oeenable, ramadrs, txda, buffer);

    parameter counter_size = 4;
    parameter buffer_size = 16;

    // Define blocks I/O's
    input  clock, reset, oeenable, txda;
    input [(counter_size *2):0] ramadrs;
    output [buffer_size-1:0] buffer;

    // Define wires for connecting wires.
    wire  clock, reset, oeenable, txda, outstrobe, rxda;
    wire [(counter_size * 2):0] ramadrs;
```

See Organizing Windows with Tab Groups for more information on these tabs.

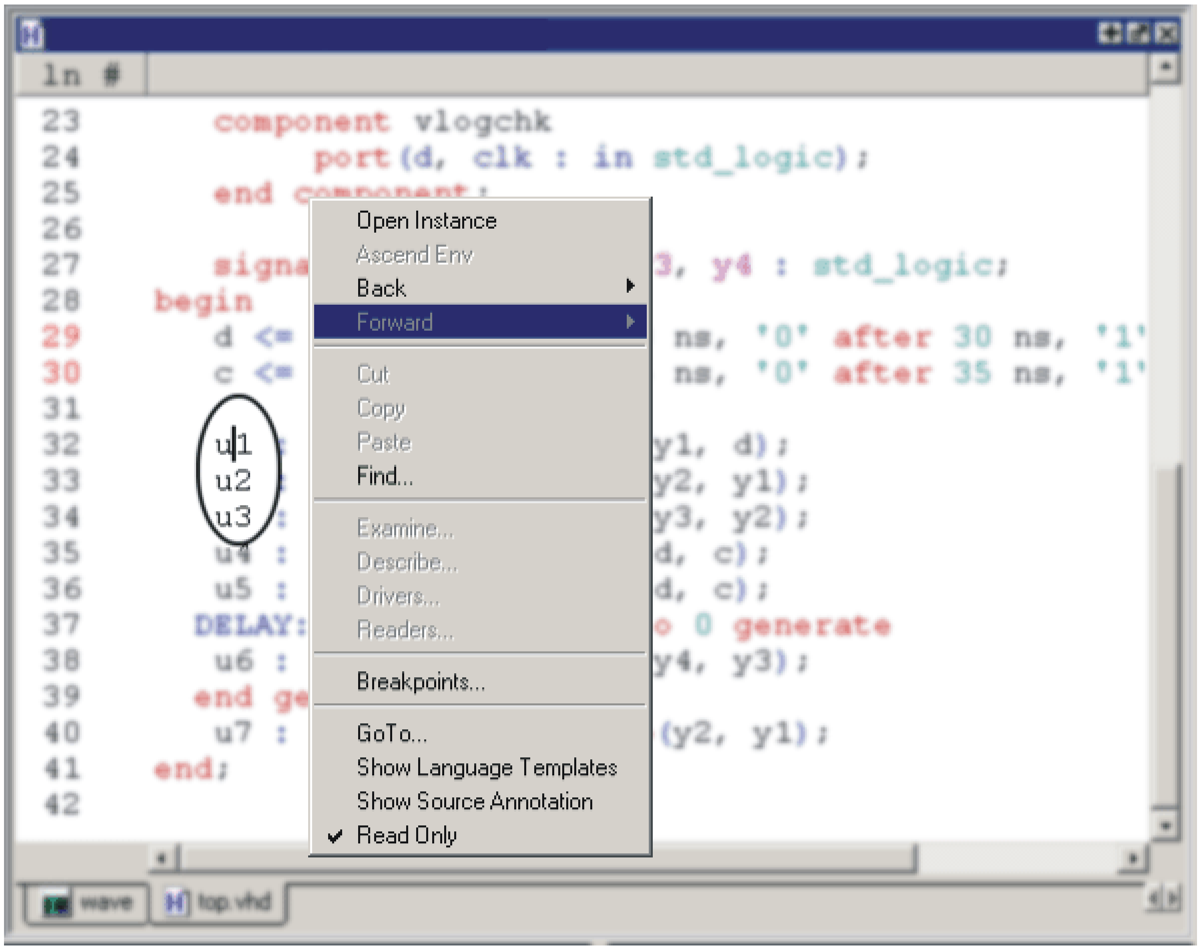
Dragging and Dropping Objects into the Wave and List Windows

ModelSim allows you to drag and drop objects from the Source window to the Wave and List windows. Double-click an object to highlight it, then drag the object to the Wave or List window. To place a group of objects into the Wave and List windows, drag and drop any section of highlighted code.

Setting your Context by Navigating Source Files

When debugging your design from within the GUI, you can change your context while analyzing your source files. Figure 2-21 shows the pop-up menu the tool displays after you select then right-click an instance name in a source file.

Figure 2-21. Setting Context from Source Files



```
        component vlogchk
            port (d, clk : in std_logic);
        end component;

        signa                     3, y4 : std_logic;
    begin
        d <=                      ns, '0' after 30 ns, '1'
        c <=                      ns, '0' after 35 ns, '1'

        u1 :                      y1, d);
        u2 :                      y2, y1);
        u3 :                      y3, y2);
        u4 :                      d, c);
        u5 :                      d, c);
       DELAY:                     o 0 generate
        u6 :                      y4, y3);
       end ge
        u7 :                      (y2, y1);
    end;
```

This functionality allows you to easily navigate your design for debugging purposes by remembering where you have been, similar to the functionality in most web browsers. The navigation options in the pop-up menu function as follows:

- **Open Instance** — changes your context to the instance you have selected within the source file. This is not available if you have not placed your cursor in, or highlighted the name of, an instance within your source file.

  If any ambiguities exists, most likely due to generate statements, this option opens a dialog box allowing you to choose from all available instances.

- **Ascend Env** — changes your context to the next level up within the design. This is not available if you are at the top-level of your design.

- **Forward/Back** — allows you to change to previously selected contexts. This is not available if you have not changed your context.

The Open Instance option is essentially executing an environment command to change your context, therefore any time you use this command manually at the command prompt, that information is also saved for use with the Forward/Back options.

Language Templates

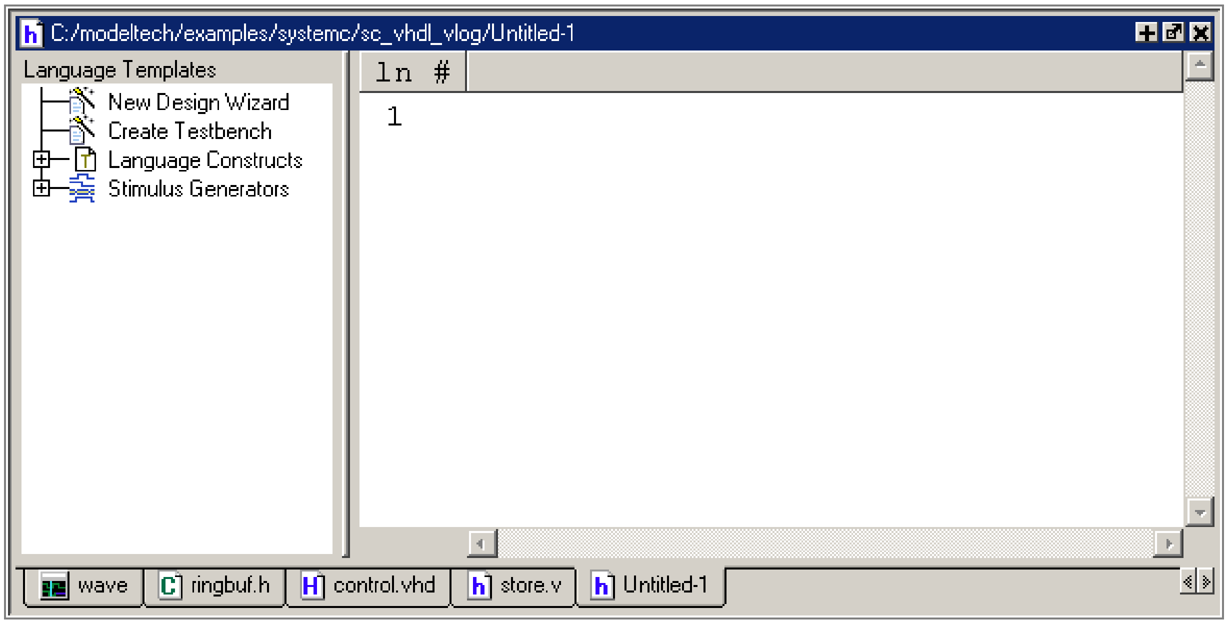
ModelSim language templates help you write code. They are a collection of wizards, menus, and dialogs that produce code for new designs, testbenches, language constructs, logic blocks, etc.

Note

The language templates are not intended to replace thorough knowledge of coding. They are intended as an interactive "reference" for creating small sections of code. If you are unfamiliar with a particular language, you should attend a training class or consult one of the many available books.

To use the templates, either open an existing file, or select **File** > **New** > **Source** to create a new file. Once the file is open, select **Source** > **Show Language Templates** if the Source window is docked in the Main window; select **View** > **Show Language Templates** of the Source window is undocked. This displays a pane that shows the available templates.

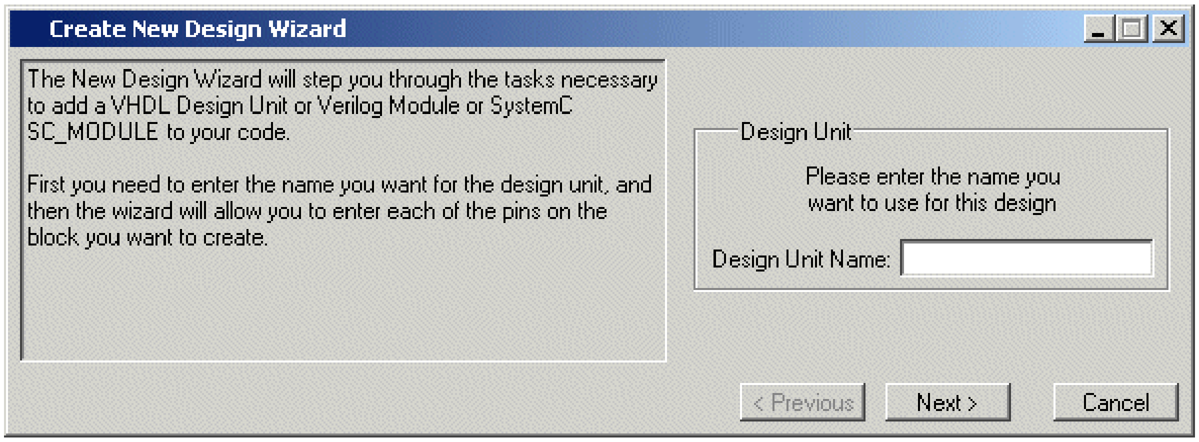
Figure 2-22. Language Templates



The templates that appear depend on the type of file you create. For example Module and Primitive templates are available for Verilog files, and Entity and Architecture templates are available for VHDL files.

Double-click an object in the list to open a wizard or to begin creating code. Some of the objects bring up wizards while others insert code into your source file. The dialog below is part of the wizard for creating a new design. Simply follow the directions in the wizards.

Figure 2-23. Create New Design Wizard



Create New Design Wizard

The New Design Wizard will step you through the tasks necessary to add a VHDL Design Unit or Verilog Module or SystemC SC_MODULE to your code.

First you need to enter the name you want for the design unit, and then the wizard will allow you to enter each of the pins on the block you want to create.

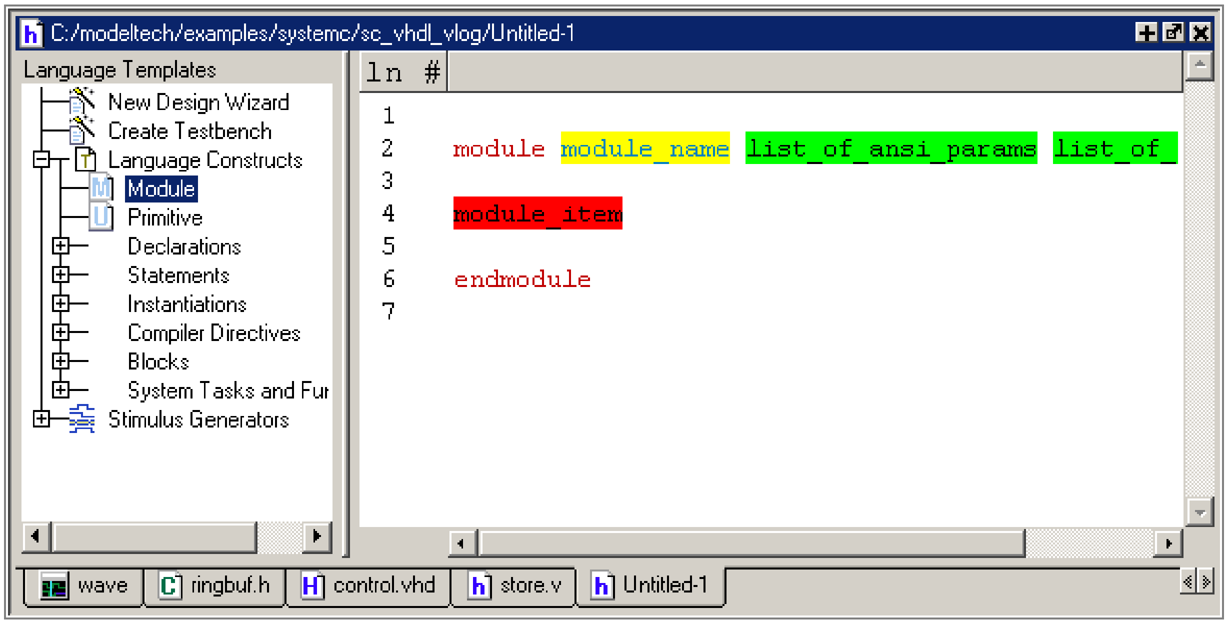
Design Unit

Please enter the name you want to use for this design

Design Unit Name:

Code inserted into your source contains a variety of highlighted fields. The example below shows a module statement inserted from the Verilog template.

Figure 2-24. Inserting Module Statement from Verilog Language Template

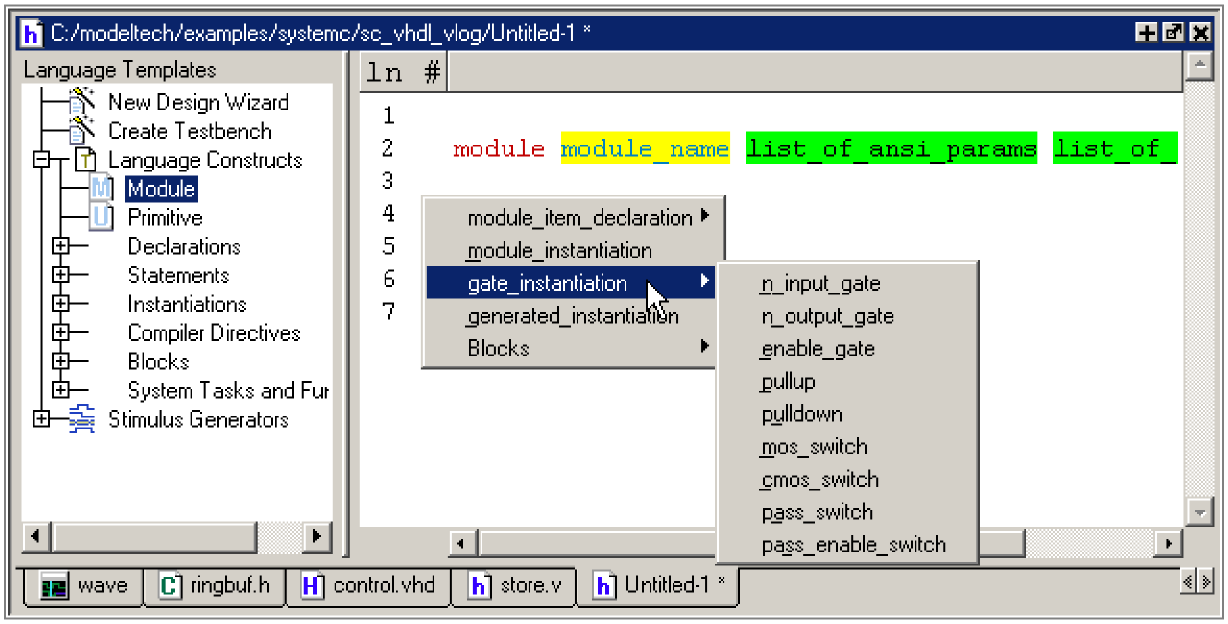


C:/modeltech/examples/systemc/sc_vhdl_vlog/Untitled-1

```

module module_name list_of_ansi_params list_of_

module_item

endmodule

```

Some of the fields, such as *module_name* in the example above, are to be replaced with names you type. Other fields can be expanded by double-clicking and still others offer a context menu of options when double-clicked. The example below shows the menu that appears when you double-click *module_item* then select *gate_instantiation*.

Figure 2-25. Language Template Context Menus



Setting File-Line Breakpoints

You can easily set File-line breakpoints in a Source window using your mouse. Click on a red line number at the left side of the Source window, and a red circle denoting a breakpoint will appear. The breakpoints are toggles – click once to create the breakpoint; click again to disable or enable the breakpoint.

To delete the breakpoint completely, right click the red circle, and select **Remove Breakpoint**. Other options on the context menu include:

- **Disable/Enable Breakpoint** — Deactivate or activate the selected breakpoint.
- **Edit Breakpoint** — Open the File Breakpoint dialog to change breakpoint arguments.
- **Edit All Breakpoints** — Open the Modify Breakpoints dialog

Checking Object Values and Descriptions

There are two quick methods to determine the value and description of an object displayed in the Source window:

- select an object, then right-click and select **Examine** or **Describe** from the context menu
- pause over an object with your mouse pointer to see an examine pop-up

Select **Tools** > **Options** > **Examine Now** or **Tools** > **Options** > **Examine Current Cursor** to choose at what simulation time the object is examined or described.

You can also invoke the examine and/or describe commands on the command line or in a macro.

Marking Lines with Bookmarks

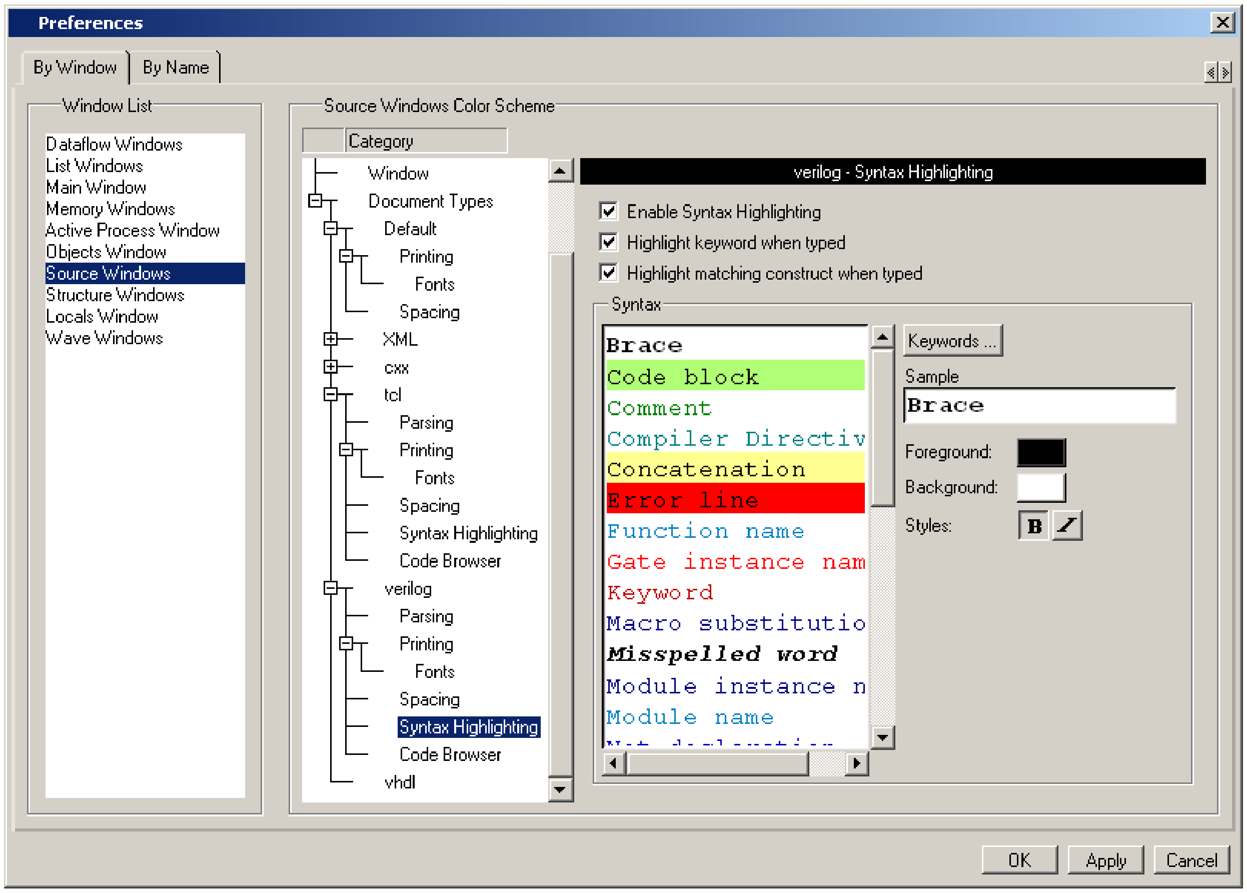
Source window bookmarks are blue circles that mark lines in a source file. These graphical icons may ease navigation through a large source file by "highlighting" certain lines.

As noted above in the discussion about finding text in the Source window, you can insert bookmarks on any line containing the text for which you are searching. The other method for inserting bookmarks is to right-click a line number and select **Add/Remove Bookmark**. To remove a bookmark, right-click the line number and select Add/Remove Bookmark again.

Customizing the Source Window

You can customize a variety of settings for Source windows. For example, you can change fonts, spacing, colors, syntax highlighting, and so forth. To customize Source window settings, select **Tools** > **Edit Preferences**. This opens the Preferences dialog. Select **Source Windows** from the Window List.

Figure 2-26. Preferences Dialog for Customizing Source Window



☑ Enable Syntax Highlighting
☑ Highlight keyword when typed
☑ Highlight matching construct when typed

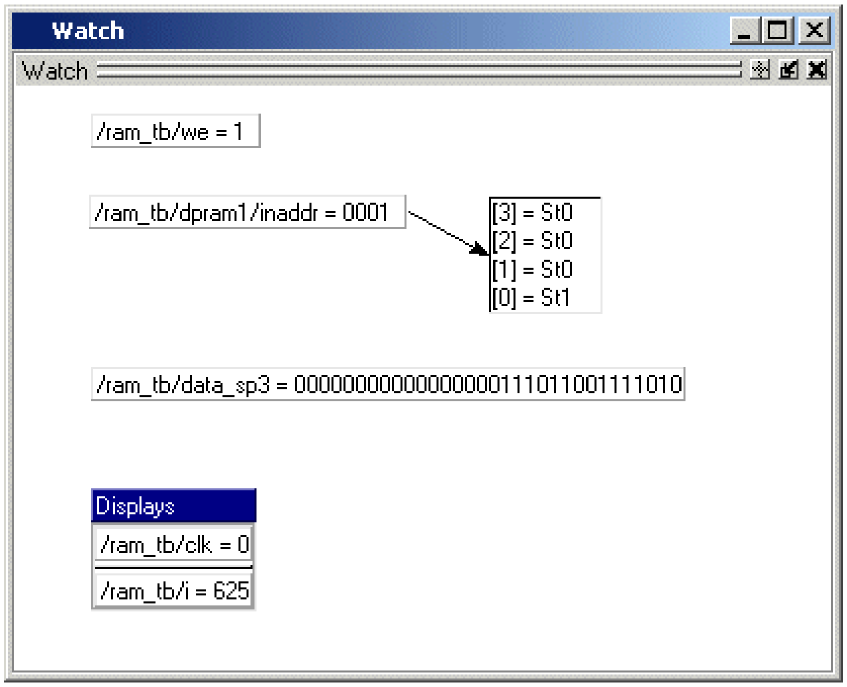
Select an item from the Category list and then edit the available properties on the right. Click OK or Apply to accept the changes.

The changes will be active for the next Source window you open. The changes are saved automatically when you quit ModelSim.

Watch Pane

The Watch pane shows values for signals and variables at the current simulation time. Unlike the Objects or Locals pane, the Watch pane allows you to view any signal or variable in the design regardless of the current context.

Figure 2-27. .Watch Pane



```
/ram_tb/we = 1
/ram_tb/dpram1/inaddr = 0001
[3] = St0
[2] = St0
[1] = St0
[0] = St1
/ram_tb/data_sp3 = 00000000000000000111011001111010
/ram_tb/clk = 0
/ram_tb/i = 625
```

You can view the following objects in the watch pane.

- VHDL objects — signals, aliases, generics, constants, and variables
- Verilog objects — nets, registers, variables, named events, and module parameters
- Virtual objects — virtual signals and virtual functions

Adding Objects to the Pane

To add objects to the Watch pane, drag-and-drop objects from the Structure tab, Objects pane, or Locals pane. Alternatively, use the add watch command.

Expanding Objects to Show Individual Bits

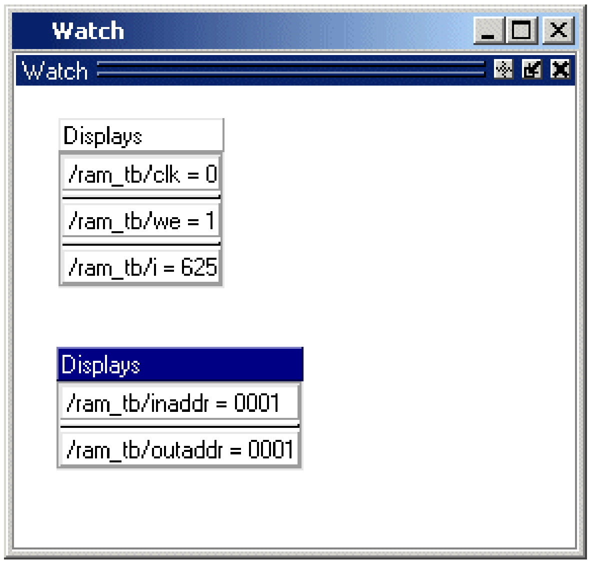
If you add an array or record to the Watch pane, you can view individual bit values by double-clicking the array or record. As shown in the graphic above, */ram_tb/dpram1/inaddr* has been expanded to show all the individual bit values. Notice the arrow that "ties" the array to the individual bit display.

Grouping and Ungrouping Objects

You can group objects in the Watch pane so they display and move together. Select the objects, then right click one of the objects and choose Group.

In the graphic below, two different sets of objects have been grouped together.

Figure 2-28. Grouping Objects in the Watch Pane



/ram_tb/clk = 0
/ram_tb/we = 1
/ram_tb/i = 625

/ram_tb/inaddr = 0001
/ram_tb/outaddr = 0001

To ungroup them, right-click the group and select Ungroup.

Saving and Reloading Format Files

You can save a format file (a DO file, actually) that will redraw the contents of the Watch window. Right-click anywhere in the window and select **Save Format**.

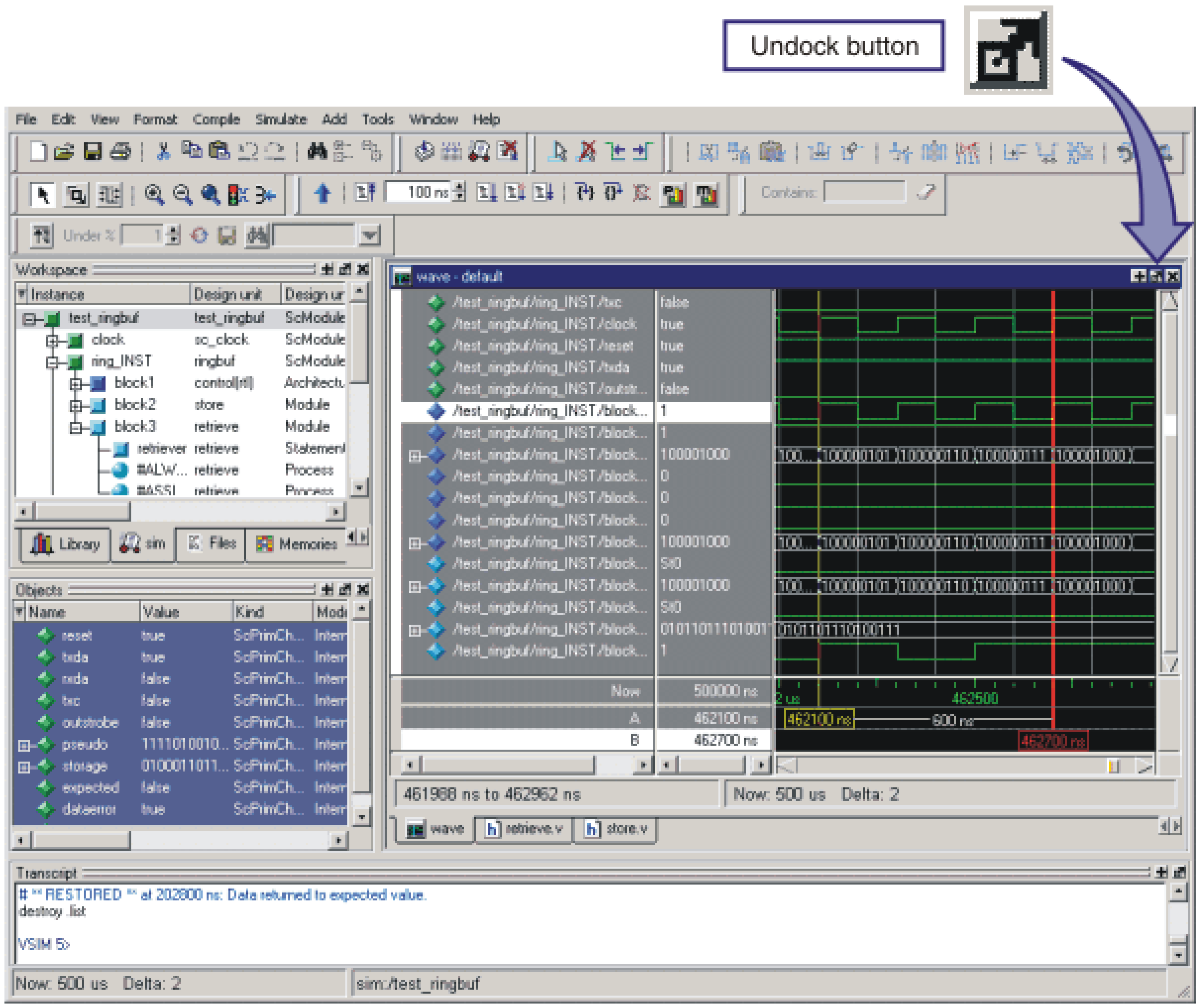
Once you have saved the file, you can reload it by right-clicking and selecting **Load Format**.

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 waveforms and their values.

The Wave window opens by default in the MDI frame of the Main window as shown below. The window can be undocked from the main window by clicking the Undock button in the window header or by using the **view -undock wave** command. The preference variable **PrefMain(ViewUnDocked) wave** can be used to control this default behavior. Setting this variable will open the Wave Window undocked each time you start ModelSim.

Figure 2-29. Wave Window Undock Button



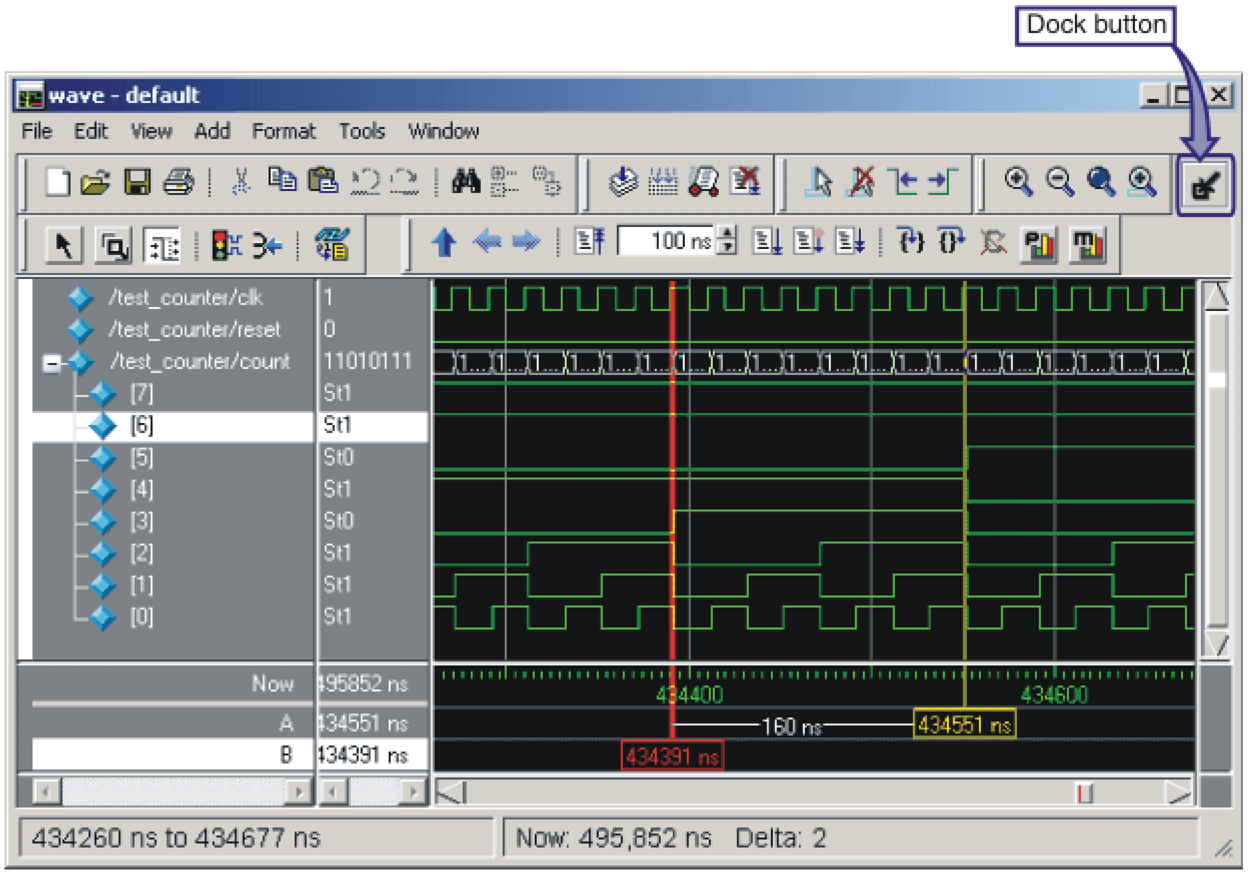
```
# ** RESTORED ** at 202800 ns: Data returned to expected value.
destroy .list

VSIM 5>
```

461968 ns to 462962 ns   Now: 500 us Delta: 2

Now: 500 us Delta: 2   sim:/test_ringbuf

Here is an example of a Wave window that is undocked from the MDI frame. All menus and icons associated with Wave window functions now appear in the menu and toolbar areas of the Wave window.

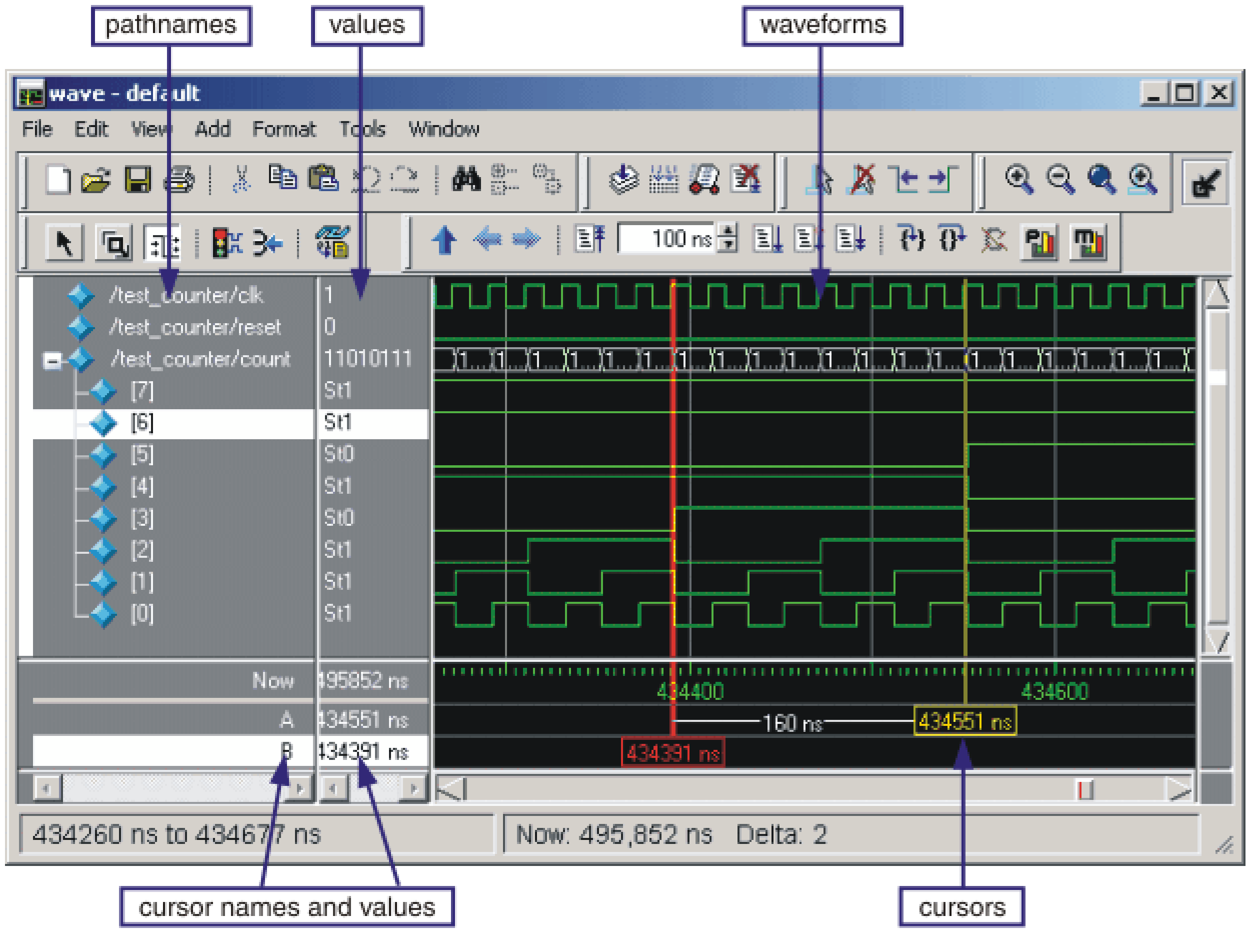
Figure 2-30. Wave Window Dock Button



If the Wave window is docked into the Main window MDI frame, all menus and icons that were in the standalone version of the Wave window move into the Main window menu bar and toolbar.

The Wave window is divided into a number of window panes. All window panes in the Wave window can be resized by clicking and dragging the bar between any two panes.

Example 2-1. Wave Window Panes



The following types of objects can be viewed in the Wave window

- VHDL objects (indicated by a dark blue diamond) — signals, aliases, process variables, and shared variables
- Verilog objects (indicated by a light blue diamond) — nets, registers, variables, and named events
- Virtual objects (indicated by an orange diamond) — virtual signals, buses, and functions, see; Virtual Objects for more information

The data in the object values pane is very similar to the Objects window, except that the values change dynamically whenever a cursor in the waveform pane is moved.

At the bottom of the waveform pane you can see a time line, tick marks, and the time value 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. You can reuse any formatting changes you make by saving a Wave window format file (see Saving the Window Format).

Wave Window Panes

The sections below describe the various Wave window panes.

Pathname Pane

The pathname pane displays signal pathnames. Signals can be displayed with full pathnames, as shown here, or with only the leaf element displayed. You can increase the size of the pane by clicking and dragging on the right border. The selected signal is highlighted.

The white bar along the left margin indicates the selected dataset (see Splitting Wave Window Panes).

Value Pane

The value pane displays the values of the displayed signals.

The radix for each signal can be symbolic, binary, octal, decimal, unsigned, hexadecimal, ASCII, or default. The default radix can be set by selecting **Simulate > Runtime Options**.

Note

When the symbolic radix is chosen for SystemVerilog reg and integer types, the values are treated as binary. When the symbolic radix is chosen for SystemVerilog bit and int types, the values are considered to be decimal.

The data in this pane is similar to that shown in the Objects Pane, except that the values change dynamically whenever a cursor in the waveform pane is moved.

Waveform Pane

The waveform pane displays the waveforms that correspond to the displayed signal pathnames. It also displays up to 20 cursors. Signal values can be displayed in analog step, analog interpolated, analog backstep, literal, logic, and event formats. The radix of each signal can be set individually by selecting the signal and then choosing . The default radix is logic.

If you rest your mouse pointer on a signal in the waveform pane, a popup displays with information about the signal. You can toggle this popup on and off in the **Wave Window Properties** dialog.

Cursor Panes

There are three cursor panes–the left pane shows the cursor names; the middle pane shows the current simulation time and the value for each cursor; and the right pane shows the absolute time value for each cursor and relative time between cursors. Up to 20 cursors can be displayed. See Measuring Time with Cursors in the Wave Window for more information.

Wave Window Toolbar

The Wave window toolbar (in the undocked Wave window) gives you quick access to these ModelSim commands and functions.

Table 2-10. Wave Window Toolbar Buttons and Menu Selections

Button	Menu equivalent	Other options
Open Dataset open a previously saved dataset	File > Open	File > Open from Main window when Transcript window sim tab is active
Save Format save the current Wave window display and signal preferences to a DO (macro) file	File > Save	none
Print print a user-selected range of the current Wave window display to a printer or a file	File > Print File > Print Postscript	none
Export Waveform export a created waveform	File > Export > Waveform	none
Cut cut the selected signal from the Wave window	Edit > Cut	right mouse in pathname pane > Cut
Copy copy the signal selected in the pathname pane	Edit > Copy	right mouse in pathname pane > Copy
Paste paste the copied signal above another selected signal	Edit > Paste	right mouse in pathname pane > Paste
Find find a name or value in the Wave window	Edit > Find	<control-f> Windows <control-s> UNIX

Table 2-10. Wave Window Toolbar Buttons and Menu Selections

Button	Menu equivalent	Other options
Insert Cursor add a cursor to the waveform pane	Add > Wave > Cursor (Main window) Add > Cursor (undocked Wave window)	right click in cursor pane and select New Cursor
Delete Cursor delete the selected cursor from the window	Edit > Delete Cursor	right mouse in cursor pane > Delete Cursor n
Find Previous Transition locate the previous signal value change for the selected signal	Edit > Search (Search Reverse)	keyboard: Shift + Tab
Find Next Transition locate the next signal value change for the selected signal	Edit > Search (Search Forward)	keyboard: Tab
Select Mode set mouse to Select Mode – click left mouse button to select, drag middle mouse button to zoom	View > Zoom > Mouse Mode > Select Mode	none
Zoom Mode set mouse to Zoom Mode – drag left mouse button to zoom, click middle mouse button to select	View > Zoom > Mouse Mode > Zoom Mode	none
Zoom In 2x zoom in by a factor of two from the current view	View > Zoom > Zoom In	keyboard: i I or + right mouse in wave pane > Zoom In
Zoom Out 2x zoom out by a factor of two from current view	View > Zoom > Zoom Out	keyboard: o O or - right mouse in wave pane > Zoom Out
Zoom in on Active Cursor center active cursor in the display and zoom in	View > Zoom > Zoom Cursor	keyboard: c or C
Zoom Full zoom out to view the full range of the simulation from time 0 to the current time	View > Zoom > Zoom Full	keyboard: f or F right mouse in wave pane > Zoom Full

Table 2-10. Wave Window Toolbar Buttons and Menu Selections

Button	Menu equivalent	Other options
Stop Wave Drawing halts any waves currently being drawn in the Wave window	none	
Show Drivers display driver(s) of the selected signal, net, or register in the Dataflow window	[Dataflow window] Navigate > Expand net to drivers	**[Dataflow window] Expand net to all drivers** right mouse in wave pane > Show Drivers
Restart reloads the design elements and resets the simulation time to zero, with the option of keeping the current formatting, breakpoints, and WLF file	Main menu: Simulate > Run > Restart	restart <arguments>
Run run the current simulation for the default time length	Main menu: Simulate > Run > Run <default_length>	use the run command at the VSIM prompt
Continue Run continue the current simulation run	Main menu: Simulate > Run > Continue	use the run **-continue** command at the VSIM prompt
Run -All run the current simulation forever, or until it hits a breakpoint or specified break event	Main menu: Simulate > Run > Run -All	use the run **-all** command at the VSIM prompt
Break stop the current simulation run	none	none
Find First Difference find the first difference in a waveform comparison	none	none
Find Previous Annotated Difference find the previous annotated difference in a waveform comparison	none	none
Find Previous Difference find the previous difference in a waveform comparison	none	none

Table 2-10. Wave Window Toolbar Buttons and Menu Selections

Button	Menu equivalent	Other options
Find Next Difference find the next difference in a waveform comparison	none	none
Find Next Annotated Difference find the next annotated difference in a waveform comparison	none	none
Find Last Difference find the last difference in a waveform comparison	none	none

Chapter 3 Projects

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

What are Projects?

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

- Source files or references to source files
- other files such as READMEs or other project documentation
- local libraries
- references to global libraries
- Simulation Configurations (see Creating a Simulation Configuration)
- Folders (see Organizing Projects with Folders)

Note

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

What are the Benefits of Projects?

Projects offer benefits to both new and advanced users. Projects

- simplify interaction with ModelSim; you don't need to understand the intricacies of compiler switches and library mappings
- eliminate the need to remember a conceptual model of the design; the compile order is maintained for you in the project. Compile order is maintained for HDL-only designs.
- remove the necessity to re-establish compiler switches and settings at each session; these are stored in the project metadata as are mappings to source files

- allow users to share libraries without copying files to a local directory; you can establish references to source files that are stored remotely or locally
- allow you to change individual parameters across multiple files; in previous versions you could only set parameters one file at a time
- enable "what-if" analysis; you can copy a project, manipulate the settings, and rerun it to observe the new results
- reload the initial settings from the project *.mpf* file every time the project is opened

Project Conversion Between Versions

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

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

Getting Started with Projects

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

- Step 1 — Creating a New Project

  This creates a .mpf file and a working library.
- Step 2 — Adding Items to the Project

  Projects can reference or include source files, folders for organization, simulations, and any other files you want to associate with the project. You can copy files into the project directory or simply create mappings to files in other locations.
- Step 3 — Compiling the Files

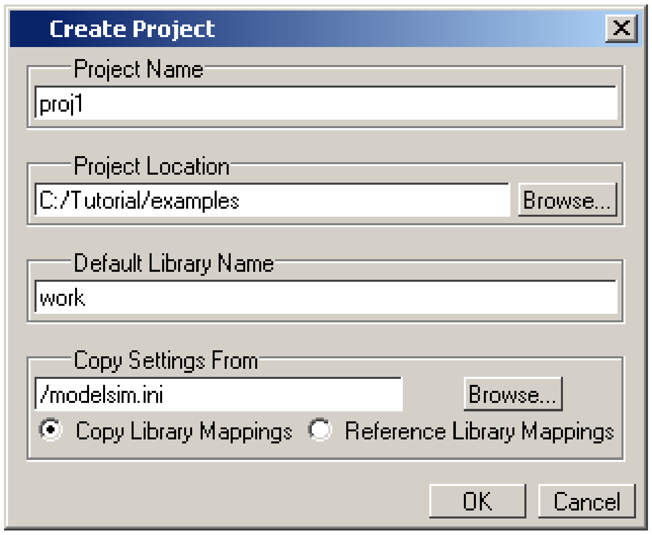
  This checks syntax and semantics and creates the pseudo machine code ModelSim uses for simulation.
- Step 4 — Simulating a Design

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

Step 1 — Creating a New Project

Select **File** > **New** > **Project** to create a new project. This opens the **Create Project** dialog where you can specify a project name, location, and default library name. You can generally leave the **Default Library Name** set to "work." The name you specify will be used to create a working library subdirectory within the Project Location. This dialog also allows you to reference library settings from a selected .ini file or copy them directly into the project.

Figure 3-1. Create Project Dialog



Create Project

Project Name: proj1

Project Location: C:/Tutorial/examples

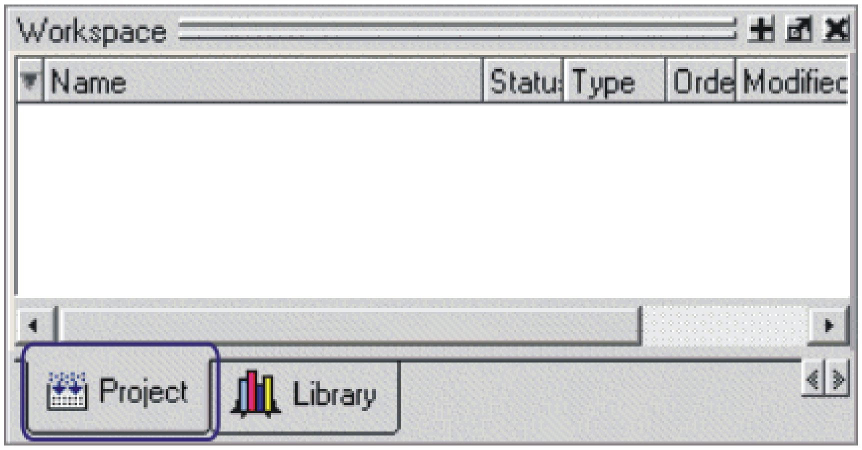
Default Library Name: work

Copy Settings From: /modelsim.ini

[✓ Copy Library Mappings] Reference Library Mappings

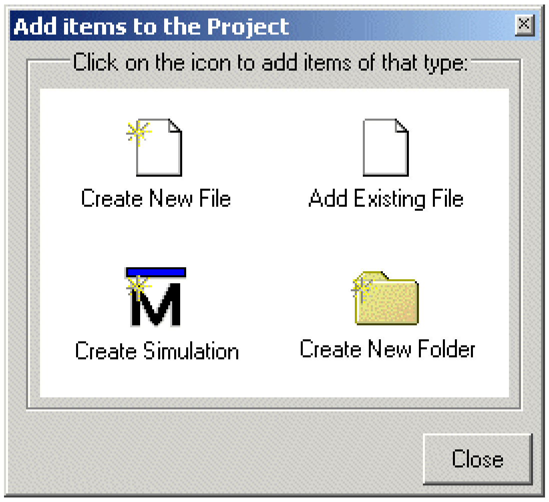
After selecting OK, you will see a blank Project tab in the Workspace pane of the Main window (Figure 3-2)

Figure 3-2. Project Tab in Workspace Pane



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

Figure 3-3. Add items to the Project Dialog



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

Step 2 — Adding Items to the Project

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

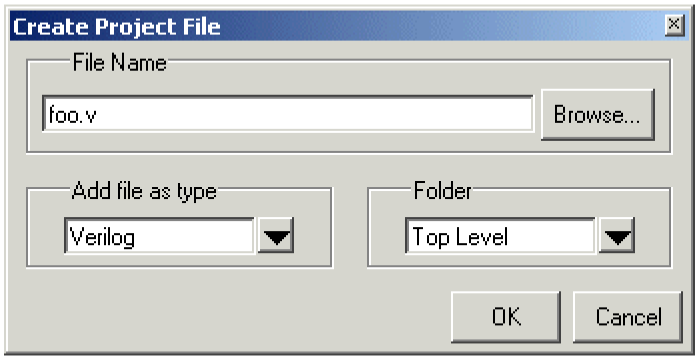
- **Create New File** — Create a new VHDL, Verilog, Tcl, or text file using the Source editor. See below for details.
- **Add Existing File** — Add an existing file. See below for details.
- **Create Simulation** — Create a Simulation Configuration that specifies source files and simulator options. See Creating a Simulation Configuration for details.
- **Create New Folder** — Create an organization folder. See Organizing Projects with Folders for details.

Create New File

The **File** > **New** > **Source** menu selections allow you to create a new VHDL, Verilog, Tcl, or text file using the Source editor.

You can also create a new project file by selecting **Project** > **Add to Project** > **New File** (the Project tab in the Workspace must be active) or right-clicking in the Project tab and selecting **Add to Project** > **New File**. This will open the Create Project File dialog (Figure 3-4).

Figure 3-4. Create Project File Dialog



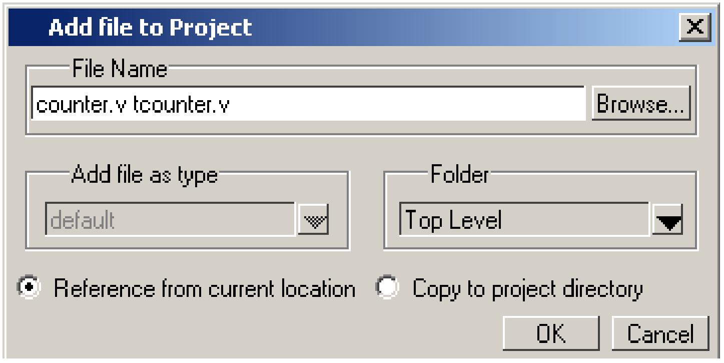
Specify a name, file type, and folder location for the new file.

When you select OK, the file is listed in the Project tab. Double-click the name of the new file and a Source editor window will open, allowing you to create source code.

Add Existing File

You can add an existing file to the project by selecting **Project** > **Add to Project** > **Existing File** or by right-clicking in the Project tab and selecting **Add to Project** > **Existing File**.

Figure 3-5. Add file to Project Dialog

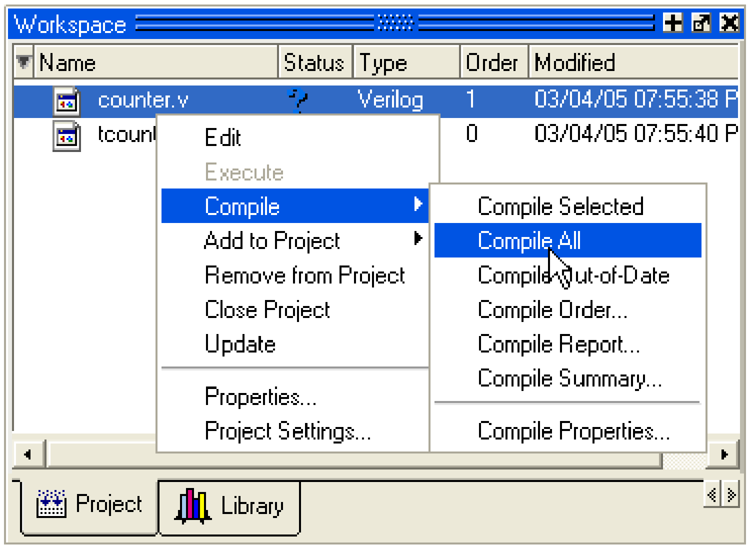


When you select OK, the file(s) is added to the Project tab.

Step 3 — Compiling the Files

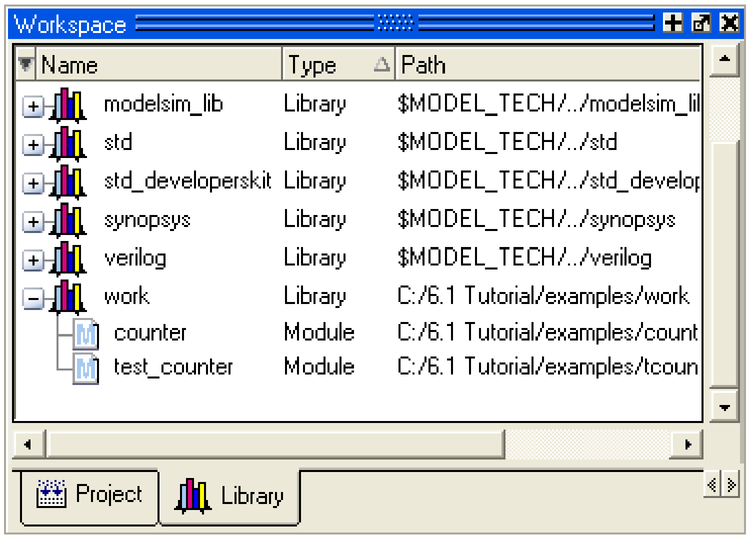
The question marks in the Status column in the Project tab denote either the files haven't been compiled into the project or the source has changed since the last compile. To compile the files, select **Compile** > **Compile All** or right click in the Project tab and select **Compile** > **Compile All**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

Figure 3-6. Right-click Compile Menu in Project Tab of Workspace



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

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



Name	Type	Path
modelsim_lib	Library	$MODEL_TECH/../modelsim_lil
std	Library	$MODEL_TECH/../std
std_developerskit	Library	$MODEL_TECH/../std_develop
synopsys	Library	$MODEL_TECH/../synopsys
verilog	Library	$MODEL_TECH/../verilog
work	Library	C:/6.1 Tutorial/examples/work
counter	Module	C:/6.1 Tutorial/examples/count
test_counter	Module	C:/6.1 Tutorial/examples/tcoun

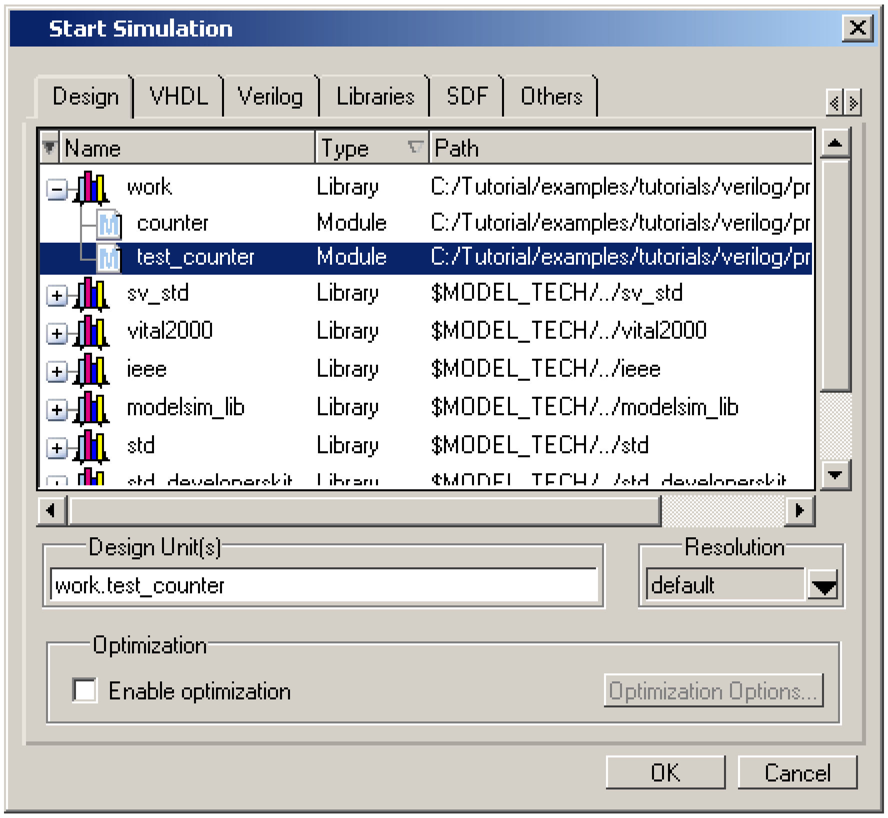
Step 4 — Simulating a Design

To simulate a design, do one of the following:

- double-click the Name of an appropriate design object (such as a testbench module or entity) in the Library tab of the Workspace
- right-click the Name of an appropriate design object and select **Simulate** from the popup menu

- select **Simulate** > **Start Simulation** from the menus to open the Start Simulation dialog (Figure 3-8). Select a design unit in the Design tab. Set other options in the VHDL, Verilog, Libraries, SDF, and Others tabs. Then click OK to start the simulation.

Figure 3-8. Start Simulation Dialog

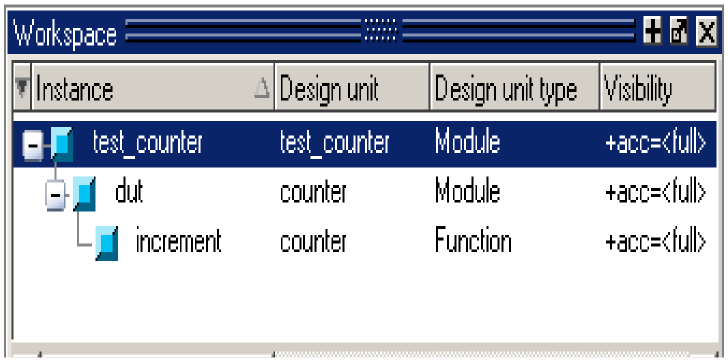


Name	Type	Path
work	Library	C:/Tutorial/examples/tutorials/verilog/pr
counter	Module	C:/Tutorial/examples/tutorials/verilog/pr
test_counter	Module	C:/Tutorial/examples/tutorials/verilog/pr
sv_std	Library	$MODEL_TECH/../sv_std
vital2000	Library	$MODEL_TECH/../vital2000
ieee	Library	$MODEL_TECH/../ieee
modelsim_lib	Library	$MODEL_TECH/../modelsim_lib
std	Library	$MODEL_TECH/../std
std_developerskit	Library	$MODEL_TECH/../std_developerskit

Design Unit(s): work.test_counter
Resolution: default
Optimization
☐ Enable optimization

A new tab named *sim* appears that shows the structure of the active simulation (Figure 3-9).

Figure 3-9. Structure Tab of the Workspace



Instance	Design unit	Design unit type	Visibility
test_counter	test_counter	Module	+acc=<full>
dut	counter	Module	+acc=<full>
increment	counter	Function	+acc=<full>

At this point you are ready to run the simulation and analyze your results. You often do this by adding signals to the Wave window and running the simulation for a given period of time. See the *ModelSim Tutorial* for examples.

Other Basic Project Operations

Open an Existing Project

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

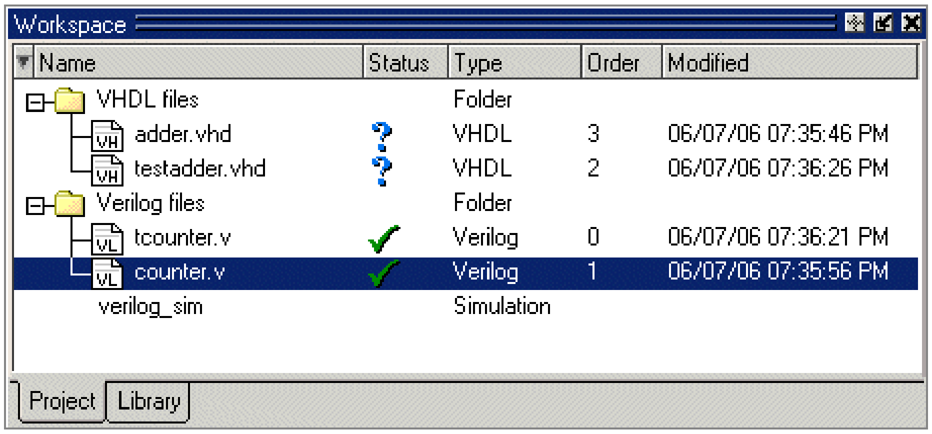
Close a Project

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

The Project Tab

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

Figure 3-10. Project Displayed in Workspace



Name	Status	Type	Order	Modified
VHDL files		Folder		
adder.vhd	?	VHDL	3	06/07/06 07:35:46 PM
testadder.vhd	?	VHDL	2	06/07/06 07:36:26 PM
Verilog files		Folder		
tcounter.v	✓	Verilog	0	06/07/06 07:36:21 PM
counter.v	✓	Verilog	1	06/07/06 07:35:56 PM
verilog_sim		Simulation		

- **Name** – The name of a file or object.
- **Status** – Identifies whether a source file has been successfully compiled. Applies only to VHDL or Verilog files. A question mark means the file hasn't been compiled or the source file has changed since the last successful compile; an X means the compile failed; a check mark means the compile succeeded; a checkmark with a yellow triangle behind it means the file compiled but there were warnings generated.
- **Type** – The file type as determined by registered file types on Windows or the type you specify when you add the file to the project.
- **Order** – The order in which the file will be compiled when you execute a Compile All command.

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

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

Sorting the List

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

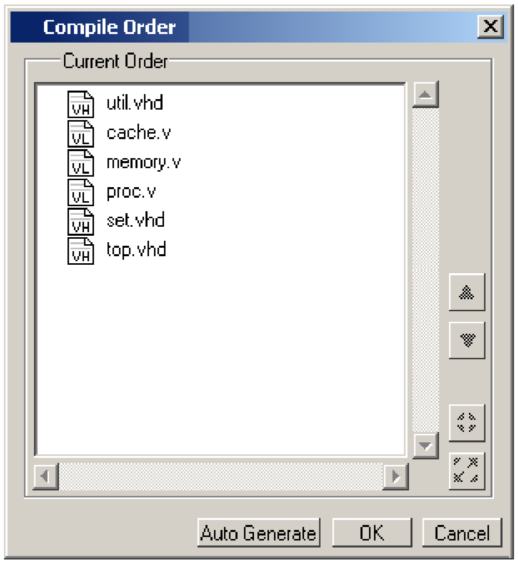
Changing Compile Order

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

To specify a custom compile order, follow these steps:

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

Figure 3-11. Setting Compile Order



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

Auto-Generating Compile Order

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

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

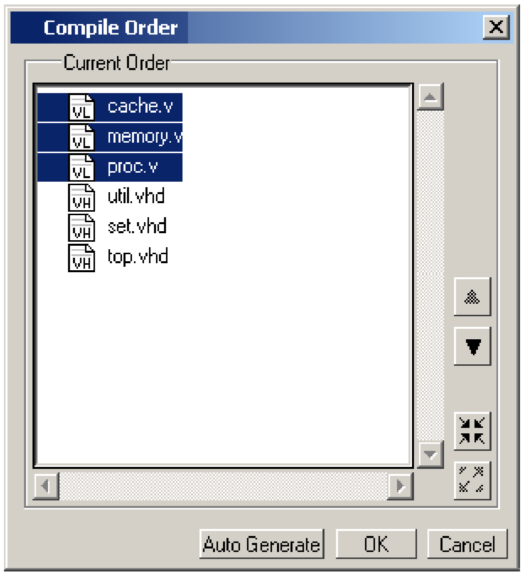
Grouping Files

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

To group files, follow these steps:

1. Select the files you want to group.

Figure 3-12. Grouping Files



2. Click the Group button.

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

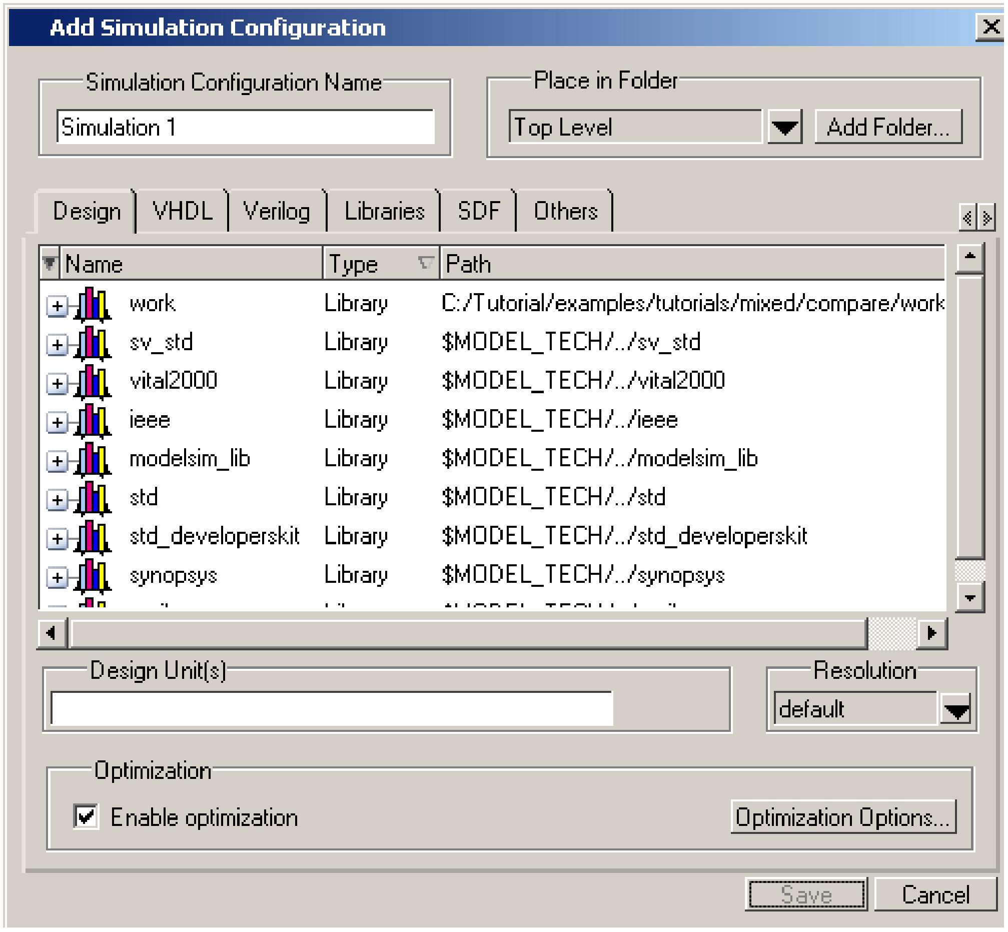
Creating a Simulation Configuration

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

To create a Simulation Configuration, follow these steps:

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

Figure 3-13. Simulation Configuration Dialog



Simulation Configuration Name: Simulation 1
Place in Folder: Top Level

Name	Type	Path
work	Library	C:/Tutorial/examples/tutorials/mixed/compare/work
sv_std	Library	$MODEL_TECH/../sv_std
vital2000	Library	$MODEL_TECH/../vital2000
ieee	Library	$MODEL_TECH/../ieee
modelsim_lib	Library	$MODEL_TECH/../modelsim_lib
std	Library	$MODEL_TECH/../std
std_developerskit	Library	$MODEL_TECH/../std_developerskit
synopsys	Library	$MODEL_TECH/../synopsys

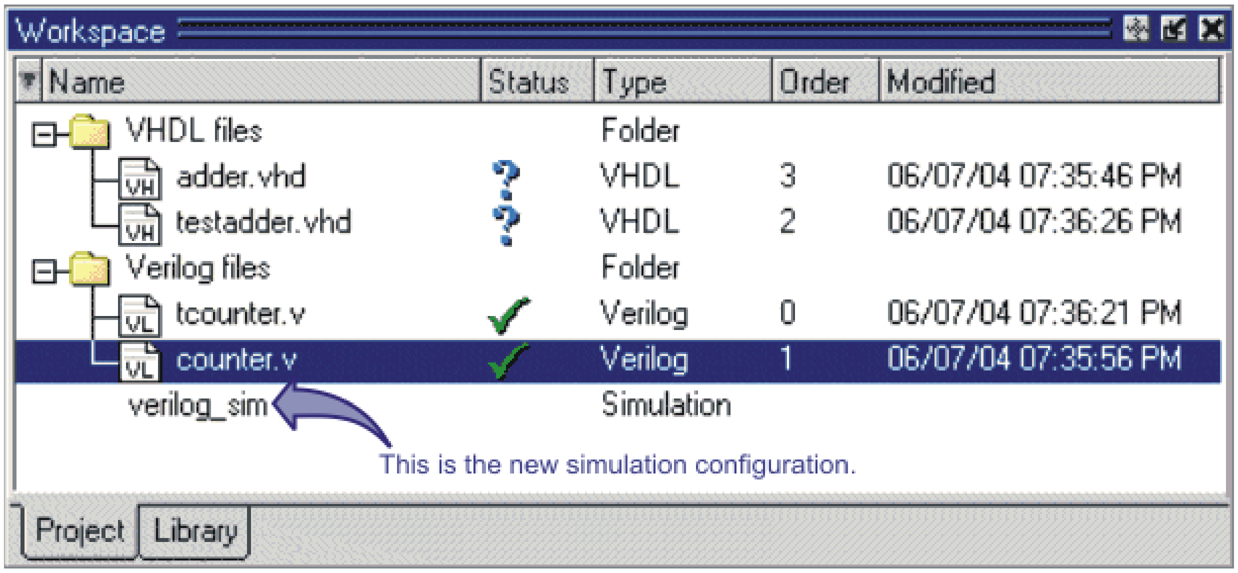
Design Unit(s):
Resolution: default
Optimization: ☑ Enable optimization

2. Specify a name in the **Simulation Configuration Name** field.

3. Specify the folder in which you want to place the configuration (see Organizing Projects with Folders).
4. Select one or more design unit(s). Use the Control and/or Shift keys to select more than one design unit. The design unit names appear in the **Simulate** field when you select them.
5. Use the other tabs in the dialog to specify any required simulation options.

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

Figure 3-14. Simulation Configuration in the Project Tab



Name	Status	Type	Order	Modified
VHDL files		Folder		
adder.vhd	?	VHDL	3	06/07/04 07:35:46 PM
testadder.vhd	?	VHDL	2	06/07/04 07:36:26 PM
Verilog files		Folder		
tcounter.v	✓	Verilog	0	06/07/04 07:36:21 PM
counter.v	✓	Verilog	1	06/07/04 07:35:56 PM
verilog_sim		Simulation		

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

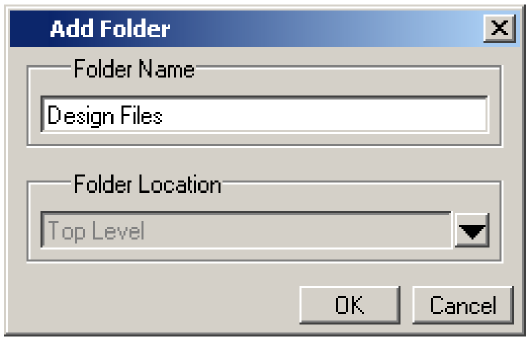
Organizing Projects with Folders

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

Adding a Folder

To add a folder to your project, select **Project** > **Add to Project** > **Folder** or right-click in the Project tab and select **Add to Project** > **Folder** (Figure 3-15).

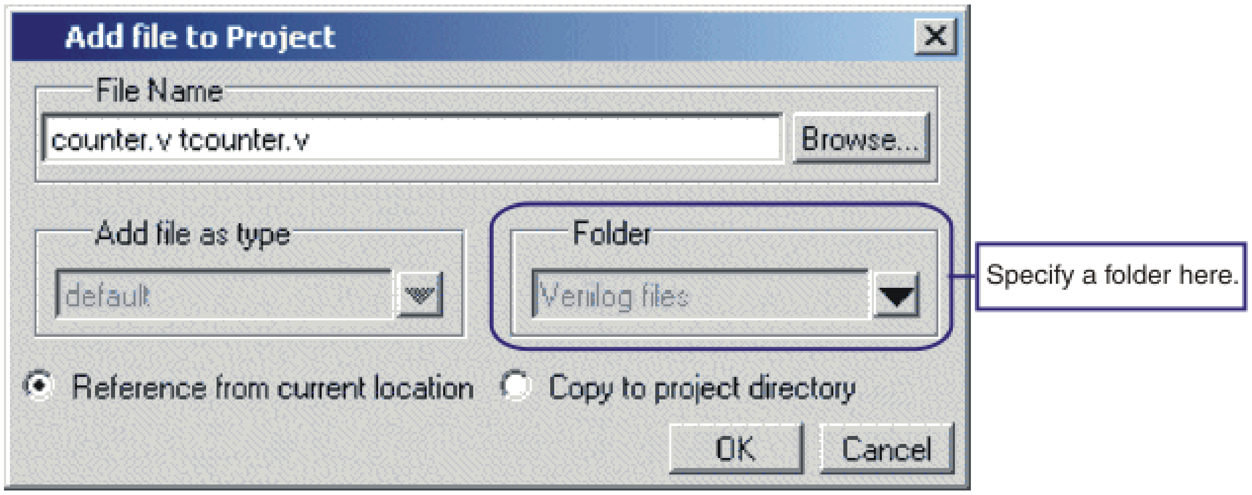
Figure 3-15. Add Folder Dialog



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

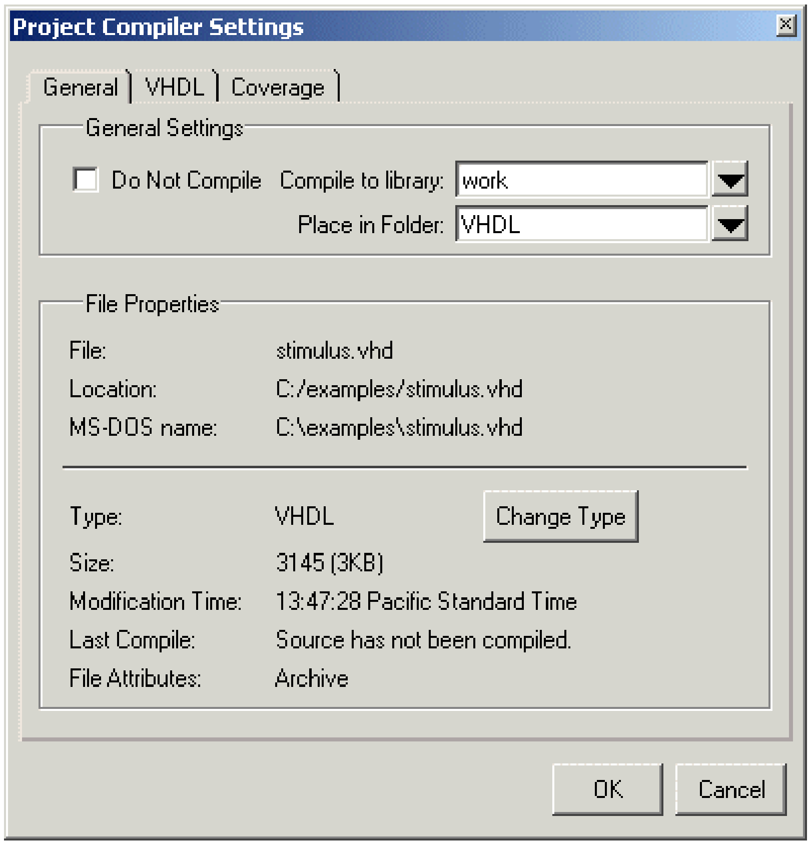
You use the folders when you add new objects to the project. For example, when you add a file, you can select which folder to place it in.

Figure 3-16. Specifying a Project Folder



If you want to move a file into a folder later on, you can do so using the Properties dialog for the file. Simply right-click on the filename in the Project tab and select Properties from the context menu that appears. This will open the Project Compiler Settings Dialog (Figure 3-17). Use the Place in Folder field to specify a folder.

Figure 3-17. Project Compiler Settings Dialog


Project Compiler Settings

General | VHDL | Coverage

General Settings

☐ Do Not Compile Compile to library: work

Place in Folder: VHDL

File Properties

File:	stimulus.vhd
Location:	C:/examples/stimulus.vhd
MS-DOS name:	C:\examples\stimulus.vhd
Type:	VHDL
Size:	3145 (3KB)
Modification Time:	13:47:28 Pacific Standard Time
Last Compile:	Source has not been compiled.
File Attributes:	Archive

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

Specifying File Properties and Project Settings

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

File Compilation Properties

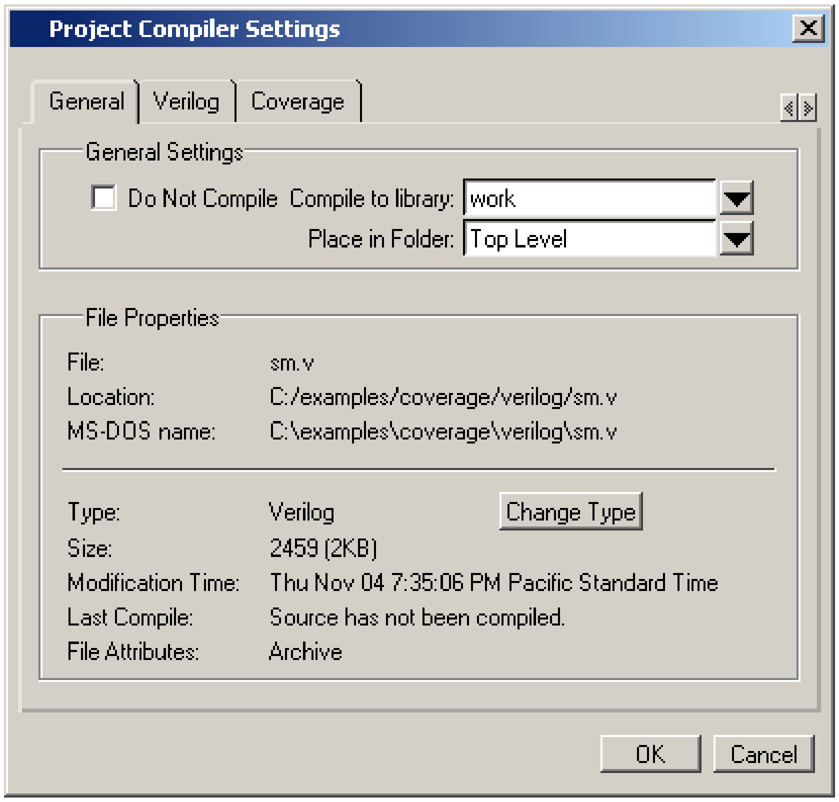
The VHDL and Verilog compilers (**vcom** and **vlog**, respectively) have numerous options that affect how a design is compiled and subsequently simulated. You can customize the settings on individual files or a group of files.

Note

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

To customize specific files, select the file(s) in the Project tab, right click on the file names, and select **Properties**. The resulting Project Compiler Settings dialog (Figure 3-18) varies depending on the number and type of files you have selected. If you select a single VHDL or Verilog file, you will see the General tab, Coverage tab, and the VHDL or Verilog tab, respectively. On the General tab, you will see file properties such as Type, Location, and Size. If you select multiple files, the file properties on the General tab are not listed. Finally, if you select both a VHDL file and a Verilog file, you will see all tabs but no file information on the General tab.

Figure 3-18. Specifying File Properties



Compile to library: work
Place in Folder: Top Level

File:	sm.v
Location:	C:/examples/coverage/verilog/sm.v
MS-DOS name:	C:\examples\coverage\verilog\sm.v
Type:	Verilog
Size:	2459 (2KB)
Modification Time:	Thu Nov 04 7:35:06 PM Pacific Standard Time
Last Compile:	Source has not been compiled.
File Attributes:	Archive

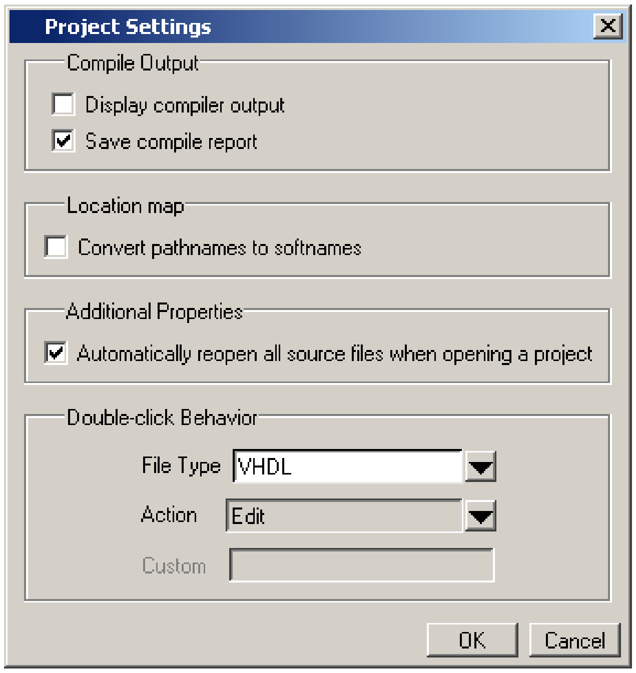
When setting options on a group of files, keep in mind the following:

- If two or more files have different settings for the same option, the checkbox in the dialog will be "grayed out." If you change the option, you cannot change it back to a "multi- state setting" without cancelling out of the dialog. Once you click OK, ModelSim will set the option the same for all selected files.
- If you select a combination of VHDL and Verilog files, the options you set on the VHDL and Verilog tabs apply only to those file types.

Project Settings

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

Figure 3-19. Project Settings Dialog


Project Settings

Compile Output
- [ ] Display compiler output
- [✓] Save compile report

Location map
- [ ] Convert pathnames to softnames

Additional Properties
- [✓] Automatically reopen all source files when opening a project

Double-click Behavior
File Type: VHDL
Action: Edit
Custom:

Converting Pathnames to Softnames for Location Mapping

If you are using location mapping, you can convert a relative pathname, full pathname, or pathname with an environment variable to a softname. A softname is a term for a pathname that uses the location mapping (MGC_LOCATION_MAP). It looks like a pathname containing an environment variable, however it is resolved using the location map rather than the environment.

To convert the pathname to a softname for projects using location mapping, follow these steps:

1. Right-click anywhere within the Project tab and select **Project Settings**
2. Enable the **Convert pathnames to softnames** within the Location map area of the dialog (Figure 3-19).

Once enabled, all pathnames currently in the project and any that are added later are then converted to softnames.

During conversion, if there is no softname in the mgc location map matching the entry, the pathname is converted in to a full (hardened) pathname. A pathname is hardened by removing the environment variable or the relative portion of the path. If this happens, any existing pathnames that are either relative or use environment variables are also changed: either to softnames if possible, or to hardened pathnames if not.

For more information on location mapping and pathnames, see Location Mapping.

Accessing Projects from the Command Line

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

You can also use the project command from the command line to perform common operations on projects.

Chapter 4
Design Libraries

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

Design Library Overview

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

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

Design Unit Information

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

- retargetable, executable code
- debugging information
- dependency information

Working Library Versus Resource Libraries

Design libraries can be used in two ways:

1. as a local working library that contains the compiled version of your design;
2. as a resource library.

The contents of your working library will change as you update your design and recompile. A resource library is typically static and serves as a parts source for your design. You can create your own resource libraries or they may be supplied by another design team or a third party (e.g., a silicon vendor).

Only one library can be the working library.

Any number of libraries can be resource libraries during a compilation. You specify which resource libraries will be used when the design is compiled, and there are rules to specify in which order they are searched (refer to Specifying the Resource Libraries).

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

The Library Named "work"

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

Archives

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

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

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

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

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

Working with Design Libraries

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

Creating a Library

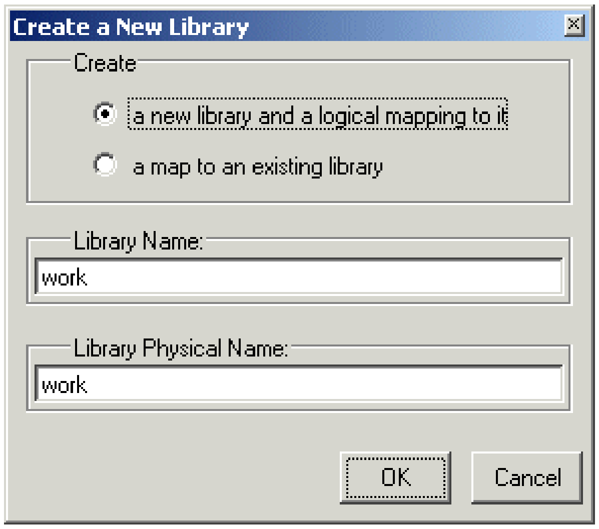
When you create a project (refer to Getting Started with Projects), ModelSim automatically creates a working design library. If you don't create a project, you need to create a working design library before you run the compiler. This can be done from either the command line or from the ModelSim graphic interface.

From the ModelSim prompt or a UNIX/DOS prompt, use this vlib command:

```
vlib <directory_pathname>
```

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

Figure 4-1. Creating a New Library



Create a New Library

Create
[✓ a new library and a logical mapping to it]
a map to an existing library

Library Name: work

Library Physical Name: work

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

The new map entry is written to the *modelsim.ini* file in the [Library] section. Refer to Library Path Variables for more information.

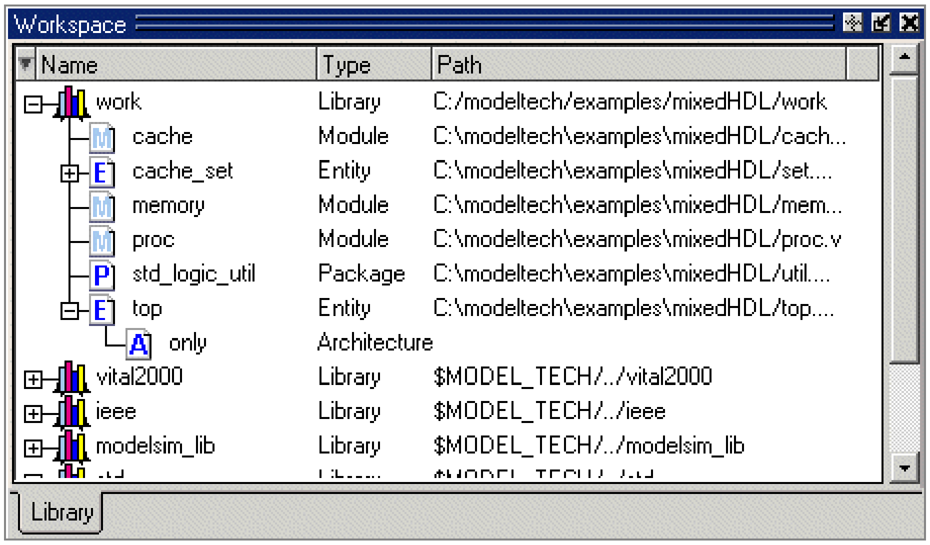
Note
Remember that a design library is a special kind of directory. The **only** way to create a library is to use the ModelSim GUI or the vlib command. Do not try to create libraries using UNIX, DOS, or Windows commands.

Managing Library Contents

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

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

Figure 4-2. Design Unit Information in the Workspace



Name	Type	Path
work	Library	C:/modeltech/examples/mixedHDL/work
cache	Module	C:\modeltech\examples\mixedHDL/cach...
cache_set	Entity	C:\modeltech\examples\mixedHDL/set....
memory	Module	C:\modeltech\examples\mixedHDL/mem...
proc	Module	C:\modeltech\examples\mixedHDL/proc.v
std_logic_util	Package	C:\modeltech\examples\mixedHDL/util....
top	Entity	C:\modeltech\examples\mixedHDL/top....
only	Architecture	
vital2000	Library	$MODEL_TECH/../vital2000
ieee	Library	$MODEL_TECH/../ieee
modelsim_lib	Library	$MODEL_TECH/../modelsim_lib

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

The context menu includes the following commands:

- **Simulate** — Loads the selected design unit and opens structure and Files tabs in the workspace. Related command line command is vsim.
- **Edit** — Opens the selected design unit in the Source window; or, if a library is selected, opens the Edit Library Mapping dialog (refer to Library Mappings with the GUI).
- **Refresh** — Rebuilds the library image of the selected library without using source code. Related command line command is vcom or vlog with the **-refresh** argument.
- **Recompile** — Recompiles the selected design unit. Related command line command is vcom or vlog.
- **Update** — Updates the display of available libraries and design units.

Assigning a Logical Name to a Design Library

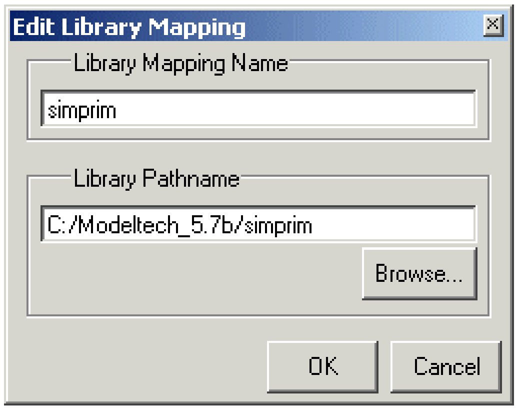
VHDL uses logical library names that can be mapped to ModelSim library directories. By default, ModelSim can find libraries in your current directory (assuming they have the right name), but for it to find libraries located elsewhere, you need to map a logical library name to the pathname of the library.

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

Library Mappings with the GUI

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

Figure 4-3. Edit Library Mapping Dialog



simprim
C:/Modeltech_5.7b/simprim

The dialog box includes these options:

- **Library Mapping Name** — The logical name of the library.
- **Library Pathname** — The pathname to the library.

Library Mapping from the Command Line

You can set the mapping between a logical library name and a directory with the vmap command using the following syntax:

vmap <logical_name> <directory_pathname>

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

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

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

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

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

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

Unix Symbolic Links

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

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

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

```
vmap <logical_name>
```

Library Search Rules

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

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

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

Moving a Library

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

Setting Up Libraries for Group Use

By adding an "others" clause to your *modelsim.ini* file, you can have a hierarchy of library mappings. If the tool does not find a mapping in the *modelsim.ini* file, then it will search the [library] section of the initialization file specified by the "others" clause. For example:

```
[library]
asic_lib = /cae/asic_lib
work = my_work
others = /usr/modeltech/modelsim.ini
```

You can specify only one "others" clause in the library section of a given *modelsim.ini* file.

The others clause only instructs the tool to look in the specified *modelsim.ini* file for a library, it does not load any other part of the specified file.

Specifying the Resource Libraries

Verilog Resource Libraries

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

Note

Resource libraries are specified differently for Verilog and VHDL. For Verilog you use either the **-L** or **-Lf** argument to vlog. Refer to Library Usage for more information.

VHDL Resource Libraries

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

Note that the **library** clause is not used to specify the working library into which the design unit is placed after compilation. The vcom command adds compiled design units to the current working library. By default, this is the library named **work**. To change the current working library, you can use vcom **-work** and specify the name of the desired target library.

Predefined Libraries

Certain resource libraries are predefined in standard VHDL. The library named **std** contains the packages **standard** and **textio**, which should not be modified. The contents of these packages and other aspects of the predefined language environment are documented in the *IEEE Standard VHDL Language Reference Manual, Std 1076*. Refer also to, Using the TextIO Package.

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

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

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

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

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

Alternate IEEE Libraries Supplied

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

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

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

Regenerating Your Design Libraries

Depending on your current ModelSim version, you may need to regenerate your design libraries before running a simulation. Check the installation README file to see if your libraries require an update. You can regenerate your design libraries using the **Refresh** command from the Library tab context menu (refer to Managing Library Contents), or by using the **-refresh** argument to vcom and **vlog**.

From the command line, you would use **vcom** with the **-refresh** argument to update VHDL design units in a library, and **vlog** with the **-refresh** argument to update Verilog design units. By default, the work library is updated. Use either **vcom** or **vlog** with the **-work <library>** argument to update a different library. For example, if you have a library named *mylib* that contains both VHDL and Verilog design units:

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

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

Note

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

Maintaining 32- and 64-bit Versions in the Same Library

ModelSim allows you to maintain 32-bit and 64-bit versions of a design in the same library.

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

Using the 32-bit version of ModelSim:

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

Next, using the 64-bit version of ModelSim:

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

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

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

Importing FPGA Libraries

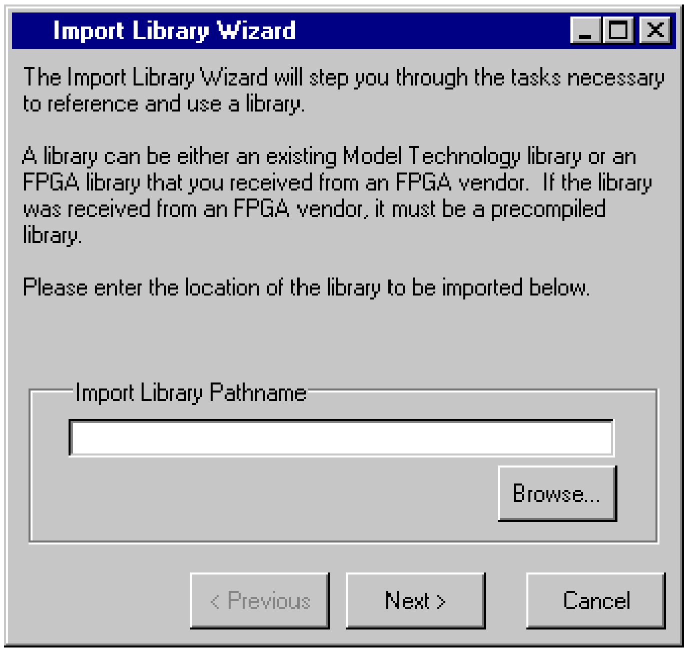
ModelSim includes an import wizard for referencing and using vendor FPGA libraries. The wizard scans for and enforces dependencies in the libraries and determines the correct mappings and target directories.

Note

The FPGA libraries you import must be pre-compiled. Most FPGA vendors supply pre-compiled libraries configured for use with ModelSim.

To import an FPGA library, select **File** > **Import** > **Library**.

Figure 4-4. Import Library Wizard



Import Library Wizard

The Import Library Wizard will step you through the tasks necessary to reference and use a library.

A library can be either an existing Model Technology library or an FPGA library that you received from an FPGA vendor. If the library was received from an FPGA vendor, it must be a precompiled library.

Please enter the location of the library to be imported below.

Follow the instructions in the wizard to complete the import.

Chapter 5
VHDL Simulation

This chapter describes how to compile, optimize, and simulate VHDL designs in ModelSim. It also discusses using the TextIO package with ModelSim*;* ModelSim's implementation of the VITAL (VHDL Initiative Towards ASIC Libraries) specification for ASIC modeling; and ModelSim's special built-in utilities package.

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

Basic VHDL Flow

Simulating VHDL designs with ModelSim includes four general steps:

1. Compile your VHDL code into one or more libraries using the vcom command. See Compiling VHDL Files for details.
2. Load your design with the vsim command. See Simulating VHDL Designs for details.
3. Run and debug your design.

Compiling VHDL Files

Creating a Design Library for VHDL

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

```
vlib work
```

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

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

See Design Libraries for additional information on working with libraries.

Invoking the VHDL Compiler

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

You can simulate a design containing units written with 1076 -1987, 1076 -1993, and 1076-2002 versions of VHDL. To do so you will need to compile units from each VHDL version separately. The vcom command compiles using 1076 -2002 rules by default; use the **-87** or **-93** argument to vcom to compile units written with version 1076-1987 or 1076 -1993, respectively. You can also change the default by modifying the VHDL93 variable in the *modelsim.ini* file (see Simulator Control Variables for more information).

Dependency Checking

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

Range and Index Checking

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

Range and index checks are performed by default when you compile your design. You can disable range checks (potentially offering a performance advantage) and index checks using arguments to the vcom command. Or, you can use the **NoRangeCheck** and **NoIndexCheck** variables in the *modelsim.ini* file to specify whether or not they are performed. See Simulator Control Variables.

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

Subprogram Inlining

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

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

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

mti_inhibit_inline Attribute

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

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

  ```
  attribute mti_inhibit_inline : boolean;
  ```

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

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

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

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

  Do similarly for entities and architectures.

Differences Between Language Versions

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

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

- Select the appropriate version from the compiler options menu in the GUI
- Invoke vcom using the argument -87, -93, or -2002
- Set the VHDL93 variable in the [vcom] section of the *modelsim.ini* file. Appropriate values for VHDL93 are:

  - 0, 87, or 1987 for VHDL-1987

  - 1, 93, or 1993 for VHDL-1993

- 2, 02, or 2002 for VHDL-2002

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

  ```
  -- Compiling package foopack

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

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

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

  by

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

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

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

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

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

Simulating VHDL Designs

A VHDL design is ready for simulation after it has been compiled with **vcom** . The simulator may then be invoked with the name of the configuration or entity/architecture pair.

Note

This section discusses simulation from the UNIX or Windows/DOS command line. You can also use a project to simulate (see Getting Started with Projects) or the **Simulate** dialog box.

This example invokes vsim on the entity **my_asic** and the architecture **structure**:

```
vsim my_asic structure
```

vsim is capable of annotating a design using VITAL compliant models with timing data from an SDF file. You can specify the min:typ:max delay by invoking vsim with the **-sdfmin**, **-sdftyp**, or **-sdfmax** option. Using the SDF file *f1.sdf* in the current work directory, the following invocation of vsim annotates maximum timing values for the design unit *my_asic*:

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

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

```
vsim +notimingchecks topmod
```

Simulator Resolution Limit (VHDL)

The simulator internally represents time as a 64-bit integer in units equivalent to the smallest unit of simulation time, also known as the simulator resolution limit. The default resolution limit is set to the value specified by the Resolution variable in the *modelsim.ini* file. You can view the current resolution by invoking the report command with the **simulator state** option.

Overriding the Resolution

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

For example this command chooses 10 ps resolution:

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

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

Choosing the Resolution for VHDL

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

Default Binding

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

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

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

Default Binding Rules

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

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

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

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

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

Disabling Default Binding

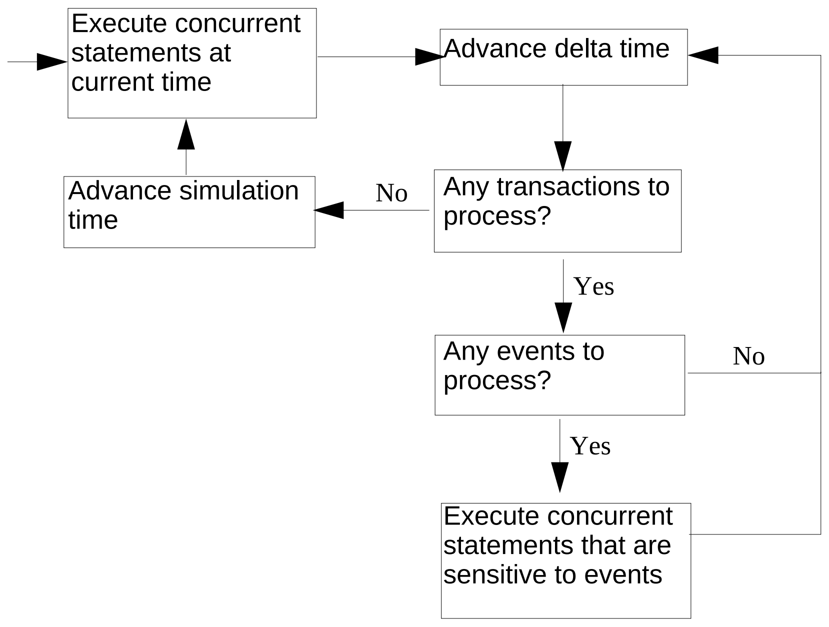
If you want default binding to occur only via configurations, you can disable ModelSim's normal default binding methods by setting the RequireConfigForAllDefaultBinding variable in the *modelsim.ini* to 1 (true).

Delta Delays

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

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

Figure 5-1. VHDL Delta Delay Process


Execute concurrent statements at current time
Advance delta time
Advance simulation time
No
Any transactions to process?
Yes
Any events to process?
No
Yes
Execute concurrent statements that are sensitive to events

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

```
    clk2 <= clk;

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

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

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

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

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

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

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

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

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

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

Detecting Infinite Zero-Delay Loops

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

The iteration limit default value is 1000. If you receive an iteration limit warning, first increase the iteration limit and try to continue simulation. You can set the iteration limit from the **Simulate** > **Runtime Options** menu or by modifying the IterationLimit variable in the *modelsim.ini*. See Simulator Control Variables for more information on modifying the *modelsim.ini* file.

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

Using the TextIO Package

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

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

A simple example using the package TextIO is:

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

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

Syntax for File Declaration

The VHDL'87 syntax for a file declaration is:

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

where "file_logical_name" must be a string expression.

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

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

where "file_open_information" is:

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

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

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

Using STD_INPUT and STD_OUTPUT Within the Tool

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

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

Updated versions of the TextIO package contain these file declarations:

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

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

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

TextIO Implementation Issues

Writing Strings and Aggregates

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

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

will cause the following error:

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

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

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

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

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

The following procedure call also generates an error:

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

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

There are two possible solutions to this problem:

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

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

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

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

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

Reading and Writing Hexadecimal Numbers

The reading and writing of hexadecimal numbers is not specified in standard VHDL. The Issues Screening and Analysis Committee of the VHDL Analysis and Standardization Group (ISAC-VASG) has specified that the TextIO package reads and writes only decimal numbers.

To expand this functionality, ModelSim supplies hexadecimal routines in the package io_utils, which is located in the file *<install_dir>/modeltech/examples/misc/io_utils.vhd*. To use these routines, compile the io_utils package and then include the following use clauses in your VHDL source code:

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

Dangling Pointers

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

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

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

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

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

The ENDLINE Function

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

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

```
(L = NULL) OR (L’LENGTH = 0)
```

The ENDFILE Function

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

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

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

Using Alternative Input/Output Files

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

The VHDL’87 declaration is:

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

The VHDL’93 declaration is:

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

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

Flushing the TEXTIO Buffer

Flushing of the TEXTIO buffer is controlled by the UnbufferedOutput variable in the *modelsim.ini* file.

Providing Stimulus

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

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

VITAL Specification and Source Code

VITAL ASIC Modeling Specification

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

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

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

VITAL source code

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

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

VITAL Packages

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

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

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

Or, alternatively, add use clauses to your code:

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

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

VITAL Compliance

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

VITAL Compliance Checking

If you are using VITAL 2.2b, you must turn off the compliance checking either by not setting the attributes, or by invoking vcom with the option **-novitalcheck**.

Compiling and Simulating with Accelerated VITAL Packages

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

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

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

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

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

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

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

Util Package

The util package serves as a container for various VHDL utilities. The package is part of the modelsim_lib library which is located in the modeltech tree and is mapped in the default *modelsim.ini* file.

To access the utilities in the package, you would add lines like the following to your VHDL code:

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

get_resolution

get_resolution returns the current simulator resolution as a real number. For example, 1 femtosecond corresponds to 1e-15.

Syntax

resval := get_resolution;

Returns

Name	Type	Description
resval	real	The simulator resolution represented as a real

Arguments

None

Related functions

- to_real()
- to_time()

Example

If the simulator resolution is set to 10ps, and you invoke the command:

resval := get_resolution;

the value returned to resval would be 1e-11.

init_signal_driver()

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

See init_signal_driver for complete details.

init_signal_spy()

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

See init_signal_spy for complete details.

signal_force()

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

See signal_force for complete details.

signal_release()

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

See signal_release for complete details.

to_real()

to_real() converts the physical type time value into a real value with respect to the current simulator resolution. The precision of the converted value is determined by the simulator resolution. For example, if you were converting 1900 fs to a real and the simulator resolution was ps, then the real value would be 2.0 (i.e., 2 ps).

Syntax

realval := to_real(timeval);

Returns

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

Arguments

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

Related functions

- get_resolution
- to_time()

Example

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

```
realval := to_real(12.99 ns);
```

then the value returned to realval would be 12990.0. If you wanted the returned value to be in units of nanoseconds (ns) instead, you would use the get_resolution function to recalculate the value:

```
realval := 1e+9 * (to_real(12.99 ns)) * get_resolution();
```

If you wanted the returned value to be in units of femtoseconds (fs), you would enter the function this way:

```
realval := 1e+15 * (to_real(12.99 ns)) * get_resolution();
```

to_time()

to_time() converts a real value into a time value with respect to the current simulator resolution. The precision of the converted value is determined by the simulator resolution. For example, if you were converting 5.9 to a time and the simulator resolution was ps, then the time value would be 6 ps.

Syntax

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

Returns

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

Arguments

Name	Type	Description
realval	real	The value of the type real

Related functions

- get_resolution
- to_real()

Example

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

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

then the value returned to timeval would be 72 ps.

Modeling Memory

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

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

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

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

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

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

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

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

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

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

Example functions are provided below in package "conversions".

For completeness sake we also show an example using VHDL 2002 protected types, though in this example, protected types offer no advantage over shared variables.

VHDL87 and VHDL93 Example

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

entity memory is
    generic(add_bits : integer := 12;
            data_bits : integer := 32);
    port(add_in : in std_ulogic_vector(add_bits-1 downto 0);
        data_in : in std_ulogic_vector(data_bits-1 downto 0);
        data_out : out std_ulogic_vector(data_bits-1 downto 0);
        cs, mwrite : in std_ulogic;
        do_init : in std_ulogic);
    subtype word is std_ulogic_vector(data_bits-1 downto 0);
    constant nwords : integer := 2 ** add_bits;
    type ram_type is array(0 to nwords-1) of word;
end;

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

architecture style_87 of memory is
begin
memory:
process (cs)
    -----------------------
    variable ram : ram_type;
    -----------------------
    variable address : natural;
```

```
        begin
            if rising_edge(cs) then
                address := sulv_to_natural(add_in);
                if (mwrite = '1') then
                     ram(address) := data_in;
                end if;
                data_out <= ram(address);
            end if;
        end process;
    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 := sulv_to_natural(add_in);
                if (mwrite = '1') then
                    ram(address) <= data_in;
                    data_out <= data_in;
                else
                    data_out <= ram(address);
                end if;
            end if;
        end process;
    end bad_style_87;

    ------------------------------------------------------------
    ------------------------------------------------------------
    library ieee;
    use ieee.std_logic_1164.all;

    package conversions is
        function sulv_to_natural(x : std_ulogic_vector) return
                    natural;
        function natural_to_sulv(n, bits : natural) return
                    std_ulogic_vector;
    end conversions;

    package body conversions is

        function sulv_to_natural(x : std_ulogic_vector) return
                    natural is
            variable n : natural := 0;
            variable failure : boolean := false;
        begin
            assert (x'high - x'low + 1) <= 31
                report "Range of sulv_to_natural argument exceeds
                    natural range"
                severity error;
            for i in x'range loop
                n := n * 2;
                case x(i) is
                    when '1' | 'H' => n := n + 1;
                    when '0' | 'L' => null;
                    when others    => failure := true;
                end case;
            end loop;
            assert not failure
                report "sulv_to_natural cannot convert indefinite
                    std_ulogic_vector"
                severity error;

            if failure then
                return 0;
            else
                return n;
            end if;
        end sulv_to_natural;

        function natural_to_sulv(n, bits : natural) return
                    std_ulogic_vector is
            variable x : std_ulogic_vector(bits-1 downto 0) :=
                    (others => '0');
            variable tempn : natural := n;
        begin
            for i in x'reverse_range loop
                if (tempn mod 2) = 1 then
                    x(i) := '1';
                end if;
                tempn := tempn / 2;
            end loop;
            return x;
        end natural_to_sulv;

    end conversions;
```

VHDL02 example

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

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

END sp_syn_ram_protected;


ARCHITECTURE intarch OF sp_syn_ram_protected IS

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

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

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

     IMPURE FUNCTION read  ( addr : IN unsigned(addr_width-1 DOWNTO 0))
RETURN
       std_logic_vector IS
      BEGIN
         return mem(to_integer(addr));
      END;

    END PROTECTED BODY mem_type;
    SHARED VARIABLE memory : mem_type;

BEGIN

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

    control_proc : PROCESS (inclk, outclk)

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

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

END intarch;

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

ENTITY ram_tb IS
END ram_tb;

ARCHITECTURE testbench OF ram_tb IS

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

    -------------------------------------------
    -- Intermediate signals and constants
    -------------------------------------------
    SIGNAL   addr     : unsigned(19 DOWNTO 0);
    SIGNAL   inaddr   : unsigned(3 DOWNTO 0);
    SIGNAL   outaddr  : unsigned(3 DOWNTO 0);
    SIGNAL   data_in  : unsigned(31 DOWNTO 0);
    SIGNAL   data_in1 : std_logic_vector(7 DOWNTO 0);
    SIGNAL   data_sp1 : std_logic_vector(7 DOWNTO 0);
    SIGNAL   we       : std_logic;
    SIGNAL   clk      : std_logic;
    CONSTANT clk_pd   : time := 100 ns;


BEGIN

    ---------------------------------------------------
    -- instantiations of single-port RAM architectures.
    -- All architectures behave equivalently, but they
    -- have different implementations.  The signal-based
    -- architecture (rtl) is not a recommended style.
    ---------------------------------------------------
    spram1 : entity work.sp_syn_ram_protected
        GENERIC MAP (
            data_width => 8,
            addr_width => 12)
        PORT MAP (
            inclk    => clk,
            outclk   => clk,
            we       => we,
            addr     => addr(11 downto 0),
            data_in  => data_in1,
            data_out => data_sp1);

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

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

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

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

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

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

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

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

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

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

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

END testbench;
```

Affecting Performance by Cancelling Scheduled Events

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

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

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

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

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

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

For projected waveforms the following would behave the same way:

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

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

Converting an Integer Into a bit_vector

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

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

entity test is
end test;

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

Chapter 6
Verilog and SystemVerilog Simulation

This chapter describes how to compile and simulate Verilog and SystemVerilog designs with ModelSim. ModelSim implements the Verilog language as defined by the IEEE Standards 1364-1995 and 1364-2005. We recommend that you obtain these specifications for reference.

The following functionality is partially implemented in ModelSim:

- Verilog Procedural Interface (VPI) (see */<install_dir>/modeltech/docs/technotes/Verilog_VPI.note* for details)
- IEEE Std P1800-2005 SystemVerilog (see */<install_dir>/modeltech/docs/technotes/sysvlog.note* for implementation details)

Terminology

This chapter uses the term "Verilog" to represent both Verilog and SystemVerilog, unless otherwise noted.

Basic Verilog Flow

Simulating Verilog designs with ModelSim includes four general steps:

1. Compile your Verilog code into one or more libraries using the vlog command. See Compiling Verilog Files for details.
2. Load your design with the vsim command. See Simulating Verilog Designs for details.
3. Run and debug your design.

Compiling Verilog Files

The first time you compile a design there is a two-step process:

1. Create a working library with **vlib** or select **File > New > Library**.
2. Compile the design using **vlog** or select **Compile > Compile**.

Creating a Working Library

Before you can compile your design, you must create a library in which to store the compilation results. Use the vlib command or select **File > New > Library** to create a new library. For example:

```
vlib work
```

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

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

See Design Libraries for additional information on working with libraries.

Invoking the Verilog Compiler

The Verilog compiler, **vlog**, compiles Verilog source code into retargetable, executable code. The library format is compatible across all supported platforms, and you can simulate your design on any platform without having to recompile your design.

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

Example 6-1. Invocation of the Verilog Compiler

Here is a sample invocation of **vlog**:

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

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

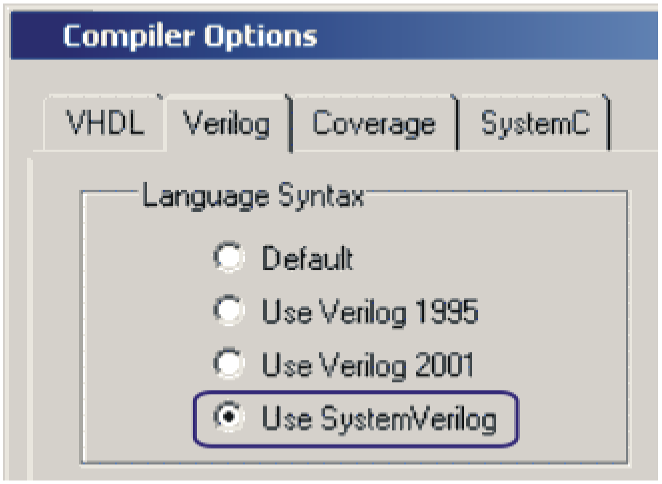
Parsing SystemVerilog Keywords

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

- any file within the design contains the *.sv* file extension,
- the **-sv** argument is used with the vlog command,

- the Use System Verilog option is selected in the Verilog tab of the Compiler Options dialog. Access this dialog by selecting **Compile** > **Compile Options** from the Main window menu bar.

Figure 6-1. Selecting 'Use System Verilog' Compile Option



( ) Default
( ) Use Verilog 1995
( ) Use Verilog 2001
[✓ Use SystemVerilog]

Here are two examples of the **vlog** command that will enable SystemVerilog features and keywords in ModelSim:

```
vlog testbench.sv top.v memory.v cache.v

vlog -sv testbench.v proc.v
```

In the first example, the *.sv* extension for *testbench* automatically instructs ModelSim to parse SystemVerilog keywords. The -*sv* option used in the second example enables SystemVerilog features and keywords.

Though a primary goal of the SystemVerilog standardization efforts has been to ensure full backward compatibility with the Verilog standard, there is an issue with keywords. SystemVerilog adds several new keywords to the Verilog language (see Table B-1 in Appendix B of the P1800 SystemVerilog standard). If your design uses one of these keywords as a regular identifier for a variable, module, task, function, etc., your design will not compile in ModelSim.

Incremental Compilation

ModelSim Verilog supports incremental compilation of designs. Unlike other Verilog simulators, there is no requirement that you compile the entire design in one invocation of the compiler.

You are not required to compile your design in any particular order (unless you are using SystemVerilog packages; see note below) because all module and UDP instantiations and external hierarchical references are resolved when the design is loaded by the simulator.

Note

Compilation order may matter when using SystemVerilog packages. As stated in the IEEE std p1800-2005 LRM, section entitled *Referencing data in packages*, which states: "Packages must exist in order for the items they define to be recognized by the scopes in which they are imported."

Incremental compilation is made possible by deferring these bindings, and as a result some errors cannot be detected during compilation. Commonly, these errors include: modules that were referenced but not compiled, incorrect port connections, and incorrect hierarchical references.

Example 6-2. Incremental Compilation Example

Contents of testbench.sv

```
module testbench;
   timeunit 1ns;
   timeprecision 10ps;
   bit d=1, clk = 0;
   wire q;
   initial
      for (int cycles=0; cycles < 100; cycles++)
         #100 clk = !clk;

   design dut(q, d, clk);
endmodule
```

Contents of design.v:

```
module design(output bit q, input bit d, clk);
   timeunit 1ns;
   timeprecision 10ps;
   always @(posedge clk)
      q = d;
endmodule
```

Compile the design incrementally as follows:

```
ModelSim> vlog testbench.sv
.
# Top level modules:
#  testbench
ModelSim> vlog -sv test1.v
.
# Top level modules:
#  dut
```

Note that the compiler lists each module as a top-level module, although, ultimately, only *testbench* is a top-level module. If a module is not referenced by another module compiled in the same invocation of the compiler, then it is listed as a top-level module. This is just an informative message and can be ignored during incremental compilation.

The message is more useful when you compile an entire design in one invocation of the compiler and need to know the top-level module names for the simulator. For example,

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

Automatic Incremental Compilation with -incr

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

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

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

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

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

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

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

Note

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

Library Usage

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

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

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

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

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

Library Search Rules for vlog

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

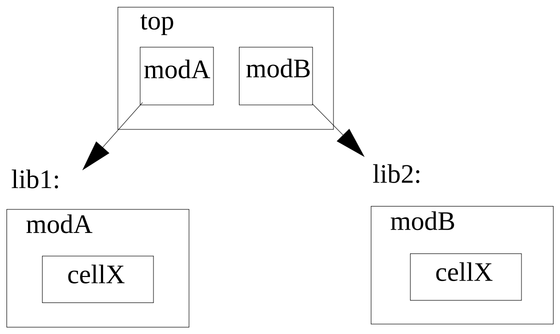
- Search libraries specified with **-Lf** arguments in the order they appear on the command line.
- Search the library specified in the Verilog-XL uselib Compiler Directive section.
- Search libraries specified with **-L** arguments in the order they appear on the command line.
- Search the **work** library.
- Search the library explicitly named in the special escaped identifier instance name.

Handling Sub-Modules with Common Names

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

For example, say you have the following design configuration:

Example 6-3. Sub-Modules with Common Names



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

vsim -L lib1 -L lib2 top

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

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

In the example above you would invoke vsim as follows:

vsim -L work -L lib1 -L lib2 top

SystemVerilog Multi-File Compilation Issues

Declarations in Compilation Unit Scope

SystemVerilog allows the declaration of types, variables, functions, tasks, and other constructs in compilation unit scope ($unit). The visibility of declarations in **$unit** scope does not extend outside the current compilation unit. Thus, it is important to understand how compilation units are defined by the tool during compilation.

By default, vlog operates in Single File Compilation Unit mode (SFCU). This means the visibility of declarations in **$unit** scope terminates at the end of each source file. Visibility does not carry forward from one file to another, except when a module, interface, or package declaration begins in one file and ends in another file. In that case, the compilation unit spans from the file containing the beginning of the declaration to the file containing the end of the declaration.

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

- For a specific compilation -- with the **-mfcu** argument to vlog.
- For all compilations -- by setting the variable **MultiFileCompilationUnit = 1** in the *modelsim.ini* file.

By using either of these methods, you allow declarations in **$unit** scope to remain in effect throughout the compilation of all files.

In case you have made MFCU the default behavior by setting **MultiFileCompilationUnit = 1** in your modelsim.ini file, it is possible to override the default behavior on specific compilations by using the **-sfcu** argument to vlog.

Macro Definitions and Compiler Directives in Compilation Unit Scope

According to the SystemVerilog IEEE Std p1800-2005 LRM, the visibility of macro definitions and compiler directives span the lifetime of a single compilation unit. By default, this means the definitions of macros and settings of compiler directives terminate at the end of each source file. They do not carry forward from one file to another, except when a module, interface, or package declaration begins in one file and ends in another file. In that case, the compilation unit spans from the file containing the beginning of the definition to the file containing the end of the definition.

See Declarations in Compilation Unit Scope for instructions on how to control vlog's handling of compilation units.

Note

Compiler directives revert to their default values at the end of a compilation unit.

If a compiler directive is specified as an option to the compiler, this setting is used for all compilation units present in the current compilation.

Verilog-XL Compatible Compiler Arguments

The compiler arguments listed below are equivalent to Verilog-XL arguments and may ease the porting of a design to ModelSim. See the vlog command for a description of each argument.

```
+define+<macro_name>[=<macro_text>]
+delay_mode_distributed
+delay_mode_path
+delay_mode_unit
+delay_mode_zero
-f <filename>
+incdir+<directory>
+mindelays
+maxdelays
+nowarn<mnemonic>
+typdelays
-u
```

Arguments Supporting Source Libraries

The compiler arguments listed below support source libraries in the same manner as Verilog-XL. See the vlog command for a description of each argument.

Note that these source libraries are very different from the libraries that the ModelSim compiler uses to store compilation results. You may find it convenient to use these arguments if you are porting a design to ModelSim or if you are familiar with these arguments and prefer to use them.

Source libraries are searched after the source files on the command line are compiled. If there are any unresolved references to modules or UDPs, then the compiler searches the source libraries to satisfy them. The modules compiled from source libraries may in turn have additional unresolved references that cause the source libraries to be searched again. This process is repeated until all references are resolved or until no new unresolved references are found. Source libraries are searched in the order they appear on the command line.

```
-v <filename>
-y <directory>
+libext+<suffix>
+librescan
+nolibcell
-R [<simargs>]
```

Verilog-XL uselib Compiler Directive

The **`uselib** compiler directive is an alternative source library management scheme to the **-v**, **-y**, and **+libext** compiler arguments. It has the advantage that a design may reference different modules having the same name. You compile designs that contain **`uselib** directive statements using the **-compile_uselibs** argument (described below) to vlog.

The syntax for the **`uselib** directive is:

`uselib <library_reference>...

where <library_reference> can be one or more of the following:

- **dir=<library_directory>**, which is equivalent to the command line argument:

  ```
  -y <library_directory>
  ```

- **file=<library_file>**, which is equivalent to the command line argument:

  ```
  -v <library_file>
  ```

- **libext=<file_extension>**, which is equivalent to the command line argument:

  ```
  +libext+<file_extension>
  ```

- **lib=<library_name>**, which references a library for instantiated objects. This behaves similarly to a LIBRARY/USE clause in VHDL. You must ensure the correct mappings are set up if the library does not exist in the current working directory. The **-compile_uselibs** argument does not affect this usage of **`uselib**.

For example, the following directive

```
`uselib dir=/h/vendorA libext=.v
```

is equivalent to the following command line arguments:

```
-y /h/vendorA +libext+.v
```

Since the **`uselib** directives are embedded in the Verilog source code, there is more flexibility in defining the source libraries for the instantiations in the design. The appearance of a **`uselib** directive in the source code explicitly defines how instantiations that follow it are resolved, completely overriding any previous **`uselib** directives.

-compile_uselibs Argument

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

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

- The directory name specified by the **-compile_uselibs** argument. For example,

  ```
  -compile_uselibs=./mydir
  ```

- The directory specified by the MTI_USELIB_DIR environment variable (see Environment Variables)

- A directory named *mti_uselibs* that is created in the current working directory

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

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

vlog -compile_uselibs top

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

uselib is Persistent

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

vlog -compile_uselibs dut.v srtr.v

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

Verilog Configurations

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

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

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

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

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

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

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

Configurations and the Library Named work

The library named "work" is treated specially by ModelSim (see The Library Named "work" for details) for Verilog configurations. Consider the following code example:

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

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

Verilog Generate Statements

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

Name Visibility in Generate Statements

Consider the following code example:

```
module m;
   parameter p = 1;

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

   initial $display(x);
endmodule
```

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

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

```
module m;
   parameter p = 1;

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

   initial $display(s.x);
endmodule
```

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

Note too that the keywords **generate - endgenerate** are optional under the 2005 rules and are excluded in the second example.

Simulating Verilog Designs

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

vsim testbench globals

After the simulator loads the top-level modules, it iteratively loads the instantiated modules and UDPs in the design hierarchy, linking the design together by connecting the ports and resolving hierarchical references. By default all modules and UDPs are loaded from the library named **work**. Modules and UDPs from other libraries can be specified using the **-L** or **-Lf** arguments to **vsim** (see Library Usage for details).

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

Simulator Resolution Limit (Verilog)

The simulator internally represents time as a 64-bit integer in units equivalent to the smallest unit of simulation time, also known as the simulator resolution limit. The resolution limit defaults to the smallest time precision found among all of the **`timescale** compiler directives in the design. Here is an example of a **`timescale** directive:

`timescale 1 ns / 100 ps

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

Time units and precision can also be specified with SystemVerilog keywords as follows:

```
timeunit 1 ns
timeprecision 100 ps
```

Modules Without Timescale Directives

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

Table 6-1. Sample Modules With and Without Timescale Directive

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

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

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

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

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

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

These warnings should ALWAYS be investigated.

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

-timescale Option

The **-timescale** option can be used with the **vlog** and **vopt** to specifies the default timescale for modules not having an explicit **`timescale** directive in effect during compilation. The format of the **-timescale** argument is the same as that of the **`timescale** directive

-timescale <time_units>/<time_precision>

The format for <*time_units*> and <*time_precision*> is <*n*><*units*>. The value of <*n*> must be 1, 10, or 100. The value of <*units*> must be fs, ps, ns, us, ms, or s. In addition, the <*time_units*> must be greater than or equal to the <*time_precision*>. For example:

-timescale "1ns / 1ps"

The argument above needs quotes because it contains white space.

Multiple Timescale Directives

As alluded to above, your design can have multiple timescale directives. The timescale directive takes effect where it appears in a source file and applies to all source files which follow in the same **vlog** command. Separately compiled modules can also have different timescales. The simulator determines the smallest timescale of all the modules in a design and uses that as the simulator resolution.

timescale, -t, and Rounding

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

```
`timescale 1 ns / 100 ps

module foo;

  initial
    #12.536 $display
```

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

- **-t** not set

  The delay will be rounded to 12.5 as directed by the module's ‘timescale directive.

- **-t** is set to 1 fs

  The delay will be rounded to 12.5. Again, the module's precision is determined by the ‘timescale directive. ModelSim does not override the module's precision.

- **-t** is set to 1 ns

  The delay will be rounded to 12. The module's precision is determined by the **-t** setting. ModelSim has no choice but to round the module's time values because the entire simulation is operating at 1 ns.

Choosing the Resolution for Verilog

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

Event Ordering in Verilog Designs

Event-based simulators such as ModelSim may process multiple events at a given simulation time. The Verilog language is defined such that you cannot explicitly control the order in which simultaneous events are processed. Unfortunately, some designs rely on a particular event order, and these designs may behave differently than you expect.

Event Queues

Section 5 of the IEEE Std 1364-1995 LRM defines several event queues that determine the order in which events are evaluated. At the current simulation time, the simulator has the following pending events:

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

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

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

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

Say you have these four statements:

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

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

The tables below show two of the many valid evaluations of these statements. Evaluation events are denoted by a number where the number is the statement to be evaluated. Update events are denoted *<name>(old->new)* where *<name>* indicates the reg being updated and *new* is the updated value.\

Table 6-2. Evaluation 1 of always Statements

Event being processed	Active event queue
	q(0 -> 1)
q(0 -> 1)	1, 2
1	p(0 -> 1), 2
p(0 -> 1)	3, 2
3	clk(0 -> 1), 2
clk(0 -> 1)	4, 2
4	2
2	p2(1 -> 0)

Table 6-2. Evaluation 1 of always Statements (cont.)

Event being processed	Active event queue
p2(1 -> 0)	3
3	clk(1 -> 0)
clk(1 -> 0)	<empty>

Table 6-3. Evaluation 2 of always Statement

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

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

Controlling Event Queues with Blocking or Non-Blocking Assignments

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

Blocking Assignments

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

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

Non-Blocking Assignments

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

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

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

```
gen1: always @(master)
  clk1 = master;

gen2: always @(clk1)
  clk2 = clk1;

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

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

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

Debugging Event Order Issues

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

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

See the **vlog** command for descriptions of **-compat** and **-hazards**.

Hazard Detection

The **-hazard** argument to vsim detects event order hazards involving simultaneous reading and writing of the same register in concurrently executing processes. **vsim** detects the following kinds of hazards:

- WRITE/WRITE — Two processes writing to the same variable at the same time.
- READ/WRITE — One process reading a variable at the same time it is being written to by another process. ModelSim calls this a READ/WRITE hazard if it executed the read first.
- WRITE/READ — Same as a READ/WRITE hazard except that ModelSim executed the write first.

vsim issues an error message when it detects a hazard. The message pinpoints the variable and the two processes involved. You can have the simulator break on the statement where the hazard is detected by setting the **break on assertion** level to **Error**.

To enable hazard detection you must invoke vlog with the **-hazards** argument when you compile your source code and you must also invoke **vsim** with the **-hazards** argument when you simulate.

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

Hazard Detection and Optimization Levels

In certain cases hazard detection results are affected by the optimization level used in the simulation. Some optimizations change the read/write operations performed on a variable if the transformation is determined to yield equivalent results. Since the hazard detection algorithm doesn't know whether or not the read/write operations can affect the simulation results, the optimizations can result in different hazard detection results. Generally, the optimizations reduce the number of false hazards by eliminating unnecessary reads and writes, but there are also optimizations that can produce additional false hazards.

Limitations of Hazard Detection

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

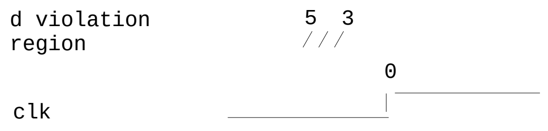
- Glitches on nets caused by non-guaranteed event ordering are not detected.
- A non-blocking assignment is not treated as a WRITE for hazard detection purposes. This is because non-blocking assignments are not normally involved in hazards. (In fact, they should be used to avoid hazards.)
- Hazards caused by simultaneous forces are not detected.

Negative Timing Check Limits

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

Example 6-4. Negative Timing Check

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



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

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

Negative Timing Constraint Algorithm

The algorithm ModelSim uses to calculate delays for delayed nets isn't described in IEEE Std 1364. Rather, ModelSim matches Verilog-XL behavior. The algorithm attempts to find a set of delays so the data net is valid when the clock net transitions and the timing checks are satisfied. The algorithm is iterative because a set of delays can be selected that satisfies all timing checks for a pair of inputs but then causes mis-ordering of another pair (where both pairs of inputs share a common input). When a set of delays that satisfies all timing checks is found, the delays are said to converge.

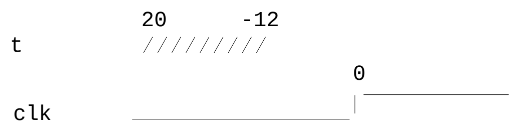
Using Delayed Inputs for Timing Checks

By default ModelSim performs timing checks on inputs specified in the timing check. If you want timing checks performed on the delayed inputs, use the **+delayed_timing_checks** argument with the vsim command.

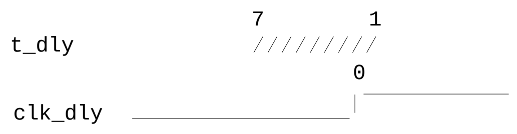
Consider an example. This timing check:

$setuphold(posedge clk, posedge t, 20, -12, NOTIFIER,,, clk_dly, t_dly);

reports a timing violation when posedge *t* occurs in the violation region:



With the **+delayed_timing_checks** argument, the violation region between the delayed inputs is:



Although the check is performed on the delayed inputs, the timing check violation message is adjusted to reference the undelayed inputs. Only the report time of the violation message is noticeably different between the delayed and undelayed timing checks.

By far the greatest difference between these modes is evident when there are conditions on a delayed check event because the condition is not implicitly delayed. Also, timing checks specified without explicit delayed signals are delayed, if necessary, when they reference an input that is delayed for a negative timing check limit.

Other simulators perform timing checks on the delayed inputs. To be compatible, ModelSim supports both methods.

Verilog-XL Compatible Simulator Arguments

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

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

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

Using Escaped Identifiers

ModelSim **always** converts Verilog escaped identifiers to VHDL syntax.

In Verilog, escaped identifiers start with the backslash character and end with a white space. Neither the backslash at the beginning or the white space at the end are considered to be a part of the identifier. When a ModelSim displays Verilog escaped identifiers, however, a backslash is added at the end in order to match the VHDL syntax for escaped identifiers. This is because all Verilog escaped identifiers can be easily converted to VHDL but the converse is not true.

So, for example, a Verilog escaped identifier like the following:

```
\/top/dut/03
```

will be displayed as follows:

```
\/top/dut/03\
```

When entering Verilog identifiers with the ModelSim command line interface, you should use the VHDL syntax, with a backslash at the beginning and end of the identifier.

In Tcl, the backslash is one of a number of characters that have a special meaning. For example,

```
\n
```

creates a new line.

When a Tcl command is used in the command line interface, the TCL backslash should be escaped by adding another backslash. For example:

```
force -freeze /top/ix/iy/\\yw\[1\]\\ 10 0, 01 {50 ns} -r 100
```

The Verilog identifier, in this example, is \yw[1]. Here, double backslashes are used because it is necessary to escape the square brackets ([]), which have a special meaning in Tcl.

For a more detailed description of special characters in Tcl and how backslashes should be used with those characters, click **Help > Tcl Syntax** in the menu bar of the graphic interface, or simply open the *docs/tcl_help_html/TclCmd* directory in your ModelSim installation.

Cell Libraries

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

The cell models generally contain Verilog "specify blocks" that describe the path delays and timing constraints for the cells. See section 13 in the IEEE Std 1364-1995 for details on specify blocks, and section 14.5 for details on timing constraints. ModelSim Verilog fully implements specify blocks and timing constraints as defined in IEEE Std 1364 along with some Verilog-XL compatible extensions.

SDF Timing Annotation

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

Delay Modes

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

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

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

Distributed Delay Mode

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

Path Delay Mode

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

Unit Delay Mode

In unit delay mode the non-zero distributed delays are set to one unit of simulation resolution (determined by the minimum time_precision argument in all ‘timescale directives in your design or the value specified with the -t argument to vsim), and the specify path delays and timing constraints are ignored. Select this delay mode with the **+delay_mode_unit** compiler argument or the **`delay_mode_unit** compiler directive.

Zero Delay Mode

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

System Tasks and Functions

ModelSim supports system tasks and functions as follows:

- All system tasks and functions defined in IEEE Std 1364
- Some system tasks and functions defined in SystemVerilog IEEE std p1800-2005 LRM
- Several system tasks and functions that are specific to ModelSim
- Several non-standard, Verilog-XL system tasks

The system tasks and functions listed in this section are built into the simulator, although some designs depend on user-defined system tasks implemented with the Programming Language Interface (PLI), Verilog Procedural Interface (VPI), or the SystemVerilog DPI (Direct Programming Interface). If the simulator issues warnings regarding undefined system tasks or functions, then it is likely that these tasks or functions are defined by a PLI/VPI application that must be loaded by the simulator.

IEEE Std 1364 System Tasks and Functions

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

Table 6-4. IEEE Std 1364 System Tasks and Functions - 1

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

Table 6-5. IEEE Std 1364 System Tasks and Functions - 2

Probabilistic distribution functions	Conversion functions	Stochastic analysis tasks	Timing check tasks
$dist_chi_square	$bitstoreal	$q_add	$hold
$dist_erlang	$itor	$q_exam	$nochange
$dist_exponential	$realtobits	$q_full	$period
$dist_normal	$rtoi	$q_initialize	$recovery
$dist_poisson	$signed	$q_remove	$setup
$dist_t	$unsigned		$setuphold
$dist_uniform			$skew
$random			$width[1]
			$removal
			$recrem

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

Table 6-6. IEEE Std 1364 System Tasks

Display tasks	PLA modeling tasks	Value change dump (VCD) file tasks
$display	$async$and$array	$dumpall

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

Display tasks	PLA modeling tasks	Value change dump (VCD) file tasks
$displayb	$async$nand$array	$dumpfile
$displayh	$async$or$array	$dumpflush
$displayo	$async$nor$array	$dumplimit
$monitor	$async$and$plane	$dumpoff
$monitorb	$async$nand$plane	$dumpon
$monitorh	$async$or$plane	$dumpvars
$monitoro	$async$nor$plane	
$monitoroff	$sync$and$array	
$monitoron	$sync$nand$array	
$strobe	$sync$or$array	
$strobeb	$sync$nor$array	
$strobeh	$sync$and$plane	
$strobeo	$sync$nand$plane	
$write	$sync$or$plane	
$writeb	$sync$nor$plane	
$writeh		
$writeo		

Table 6-7. IEEE Std 1364 File I/O Tasks

File I/O tasks		
$fclose	$fmonitoro	$fwriteh
$fdisplay	$fopen	$fwriteo
$fdisplayb	$fread	$readmemb
$fdisplayh	$fscanf	$readmemh
$fdisplayo	$fseek	$rewind
$feof	$fstrobe	$sdf_annotate
$ferror	$fstrobeb	$sformat
$fflush	$fstrobeh	$sscanf
$fgetc	$fstrobeo	$swrite

Table 6-7. IEEE Std 1364 File I/O Tasks (cont.)

File I/O tasks		
$fgets	$ftell	$swriteb
$fmonitor	$fwrite	$swriteh
$fmonitorb	$fwriteb	$swriteo
$fmonitorh		$ungetc

SystemVerilog System Tasks and Functions

The following ModelSim-supported system tasks and functions are described in detail in the SystemVerilog IEEE Std p1800-2005 LRM.

Table 6-8. SystemVerilog System Tasks and Functions - 1

Expression size function	Range function
$bits	$isunbounded

Table 6-9. SystemVerilog System Tasks and Functions - 2

Shortreal conversions	Array querying functions
$shortrealbits	$dimensions
$bitstoshortreal	$left
	$right
	$low
	$high
	$increment
	$size

Table 6-10. SystemVerilog System Tasks and Functions - 4

Reading packed data functions	Writing packed data functions	Other functions
$readmemb	$writememb	$root
$readmemh	$writememh	$unit

System Tasks and Functions Specific to the Tool

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

$init_signal_driver

The $init_signal_driver() system task drives the value of a VHDL signal or Verilog net onto an existing VHDL signal or Verilog net. This allows you to drive signals or nets at any level of the design hierarchy from within a Verilog module (e.g., a testbench). See $init_signal_driver for complete details.

$init_signal_spy

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

$psprintf()

The $psprintf() system function behaves like the $sformat() file I/O task except that the string result is passed back to the user as the function return value for $psprintf(), not placed in the first argument as for $sformat(). Thus $psprintf() can be used where a string is valid. Note that at this time, unlike other system tasks and functions, $psprintf() cannot be overridden by a user-defined system function in the PLI.

$signal_force

The $signal_force() system task forces the value specified onto an existing VHDL signal or Verilog register or net. This allows you to force signals, registers, or nets at any level of the design hierarchy from within a Verilog module (e.g., a testbench). A $signal_force works the same as the force command with the exception that you cannot issue a repeating force. See $signal_force for complete details.

$signal_release

The $signal_release() system task releases a value that had previously been forced onto an existing VHDL signal or Verilog register or net. A $signal_release works the same as the noforce command. See $signal_release for complete details.

$sdf_done

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

Verilog-XL Compatible System Tasks and Functions

ModelSim supports a number of Verilog-XL specific system tasks and functions.

Supported Tasks and Functions Mentioned in IEEE Std 1364

The following supported system tasks and functions, though not part of the IEEE standard, are described in an annex of the IEEE Std 1364.

$countdrivers
$getpattern
$sreadmemb
$sreadmemh

Supported Tasks not Described in the IEEE Std 1364

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

$deposit(variable, value);

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

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

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

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

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

$system("command");

This system function takes a literal string argument, executes the specified operating system command, and displays the status of the underlying OS process. Double quotes are required for the OS command. For example, to list the contents of the working directory on Unix:

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

Return value of the **$system** function is a 32-bit integer that is set to the exit status code of the underlying OS process.

Note

There is a known issue in the return value of this system function on the win32 platform. If the OS command is built with a cygwin compiler, the exit status code may not be reported correctly when an exception is thrown, and thus the return code may be wrong. The workaround is to avoid building the application using cygwin or to use the switch **-mno-cygwin** in cygwin the gcc command line.

$systemf(list_of_args)

This system function can take any number of arguments. The list_of_args is treated exactly the same as with the $display() function. The OS command that will be run is the final output from $display() given the same list_of_args. Return value of the $systemf function is a 32-bit integer that is set to the exit status code of the underlying OS process.

Note

There is a known issue in the return value of this system function on the win32 platform. If the OS command is built with a cygwin compiler, the exit status code may not be reported correctly when an exception is thrown, and thus the return code may be wrong. The workaround is to avoid building the application using cygwin or to use the switch **-mno-cygwin** in cygwin the gcc command line.

Supported Tasks that Have Been Extended

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

$recovery(reference event, data_event, removal_limit, recovery_limit, [notifier], [tstamp_cond], [tcheck_cond], [delayed_reference], [delayed_data])

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

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

The tstamp_cond argument conditions the data_event for the setup check and the clk_event for the hold check. This alternate method of conditioning precludes specifying conditions in the clk_event and data_event arguments.

The tcheck_cond argument conditions the data_event for the hold check and the clk_event for the setup check. This alternate method of conditioning precludes specifying conditions in the clk_event and data_event arguments.

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

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

The delayed_clk and delayed_data arguments are provided to ease the modeling of devices that may have negative timing constraints. The model's logic should reference the delayed_clk and delayed_data nets in place of the normal clk and data nets. This ensures that the correct data is latched in the presence of negative constraints. The simulator automatically calculates the delays for delayed_clk and delayed_data such that the correct data is latched as long as a timing constraint has not been violated. See Negative Timing Check Limits for more details.

Unsupported Verilog-XL System Tasks

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

$input("filename")

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

$list[(hierarchical_name)]

This system task lists the source code for the specified scope. The equivalent functionality is provided by selecting a module in the structure pane of the Workspace. The corresponding source code is displayed in a Source window.

$reset

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

$restart("filename")

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

$save("filename")

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

$scope(hierarchical_name)

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

$showscopes

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

$showvars

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

Compiler Directives

ModelSim Verilog supports all of the compiler directives defined in the IEEE Std 1364, some Verilog-XL compiler directives, and some that are proprietary. The SystemVerilog IEEE Std P1800-2005 version of the *‘define* and *‘include* compiler directives are not currently supported.

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

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

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

ModelSim Verilog implicitly defines the following macro:

`define MODEL_TECH

IEEE Std 1364 Compiler Directives

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

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

Verilog-XL Compatible Compiler Directives

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

‘default_decay_time <time>

This directive specifies the default decay time to be used in trireg net declarations that do not explicitly declare a decay time. The decay time can be expressed as a real or integer number, or as "infinite" to specify that the charge never decays.

`delay_mode_distributed

This directive disables path delays in favor of distributed delays. See Delay Modes for details.

`delay_mode_path

This directive sets distributed delays to zero in favor of path delays. See Delay Modes for details.

`delay_mode_unit

This directive sets path delays to zero and non-zero distributed delays to one time unit. See Delay Modes for details.

`delay_mode_zero

This directive sets path delays and distributed delays to zero. See Delay Modes for details.

`uselib

This directive is an alternative to the **-v**, **-y**, and **+libext** source library compiler arguments. See Verilog-XL uselib Compiler Directive for details.

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

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

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

`default_trireg_strength
`signed
`unsigned

Verilog PLI/VPI and SystemVerilog DPI

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

Chapter 7 WLF Files (Datasets) and Virtuals

This chapter describes the Wave Log Format (WLF) file and how you should and can use it in your simulation flow.

A ModelSim simulation can be saved to a wave log format (WLF) file for future viewing or comparison to a current simulation. We use the term "dataset" to refer to a WLF file that has been reopened for viewing.

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

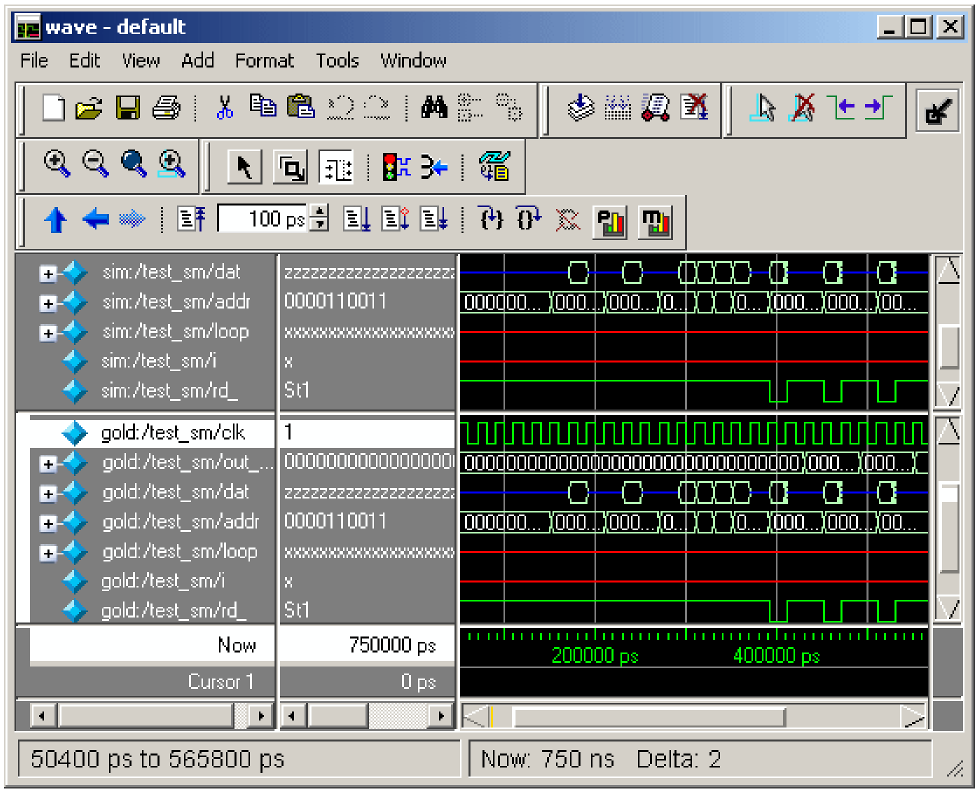
WLF files are recordings of simulation runs. The WLF file is written as an archive file in binary format and is used to drive the debug windows at a later time. The files contain data from logged objects (e.g., signals and variables) and the design hierarchy in which the logged objects are found. You can record the entire design or choose specific objects.

The WLF file provides you with precise in-simulation and post-simulation debugging capability. Any number of WLF files can be reloaded for viewing or comparing to the active simulation.

A dataset is a previously recorded simulation that has been loaded into ModelSim. Each dataset has a logical name to let you indicate the dataset to which any command applies. This logical name is displayed as a prefix. The current, active simulation is prefixed by "sim:", while any other datasets are prefixed by the name of the WLF file by default.

Two datasets are displayed in the Wave window in Figure 7-1. The current simulation is shown in the top pane and is indicated by the "sim" prefix. A dataset from a previous simulation is shown in the bottom pane and is indicated by the "gold" prefix.

Figure 7-1. Displaying Two Datasets in the Wave Window



sim:/test_sm/dat	zzzzzzzzzzzzzzzzzzz
sim:/test_sm/addr	0000110011
sim:/test_sm/loop	xxxxxxxxxxxxxxxxxxx
sim:/test_sm/i	x
sim:/test_sm/rd_	St1
gold:/test_sm/clk	1
gold:/test_sm/out_...	0000000000000000
gold:/test_sm/dat	zzzzzzzzzzzzzzzzzzz
gold:/test_sm/addr	0000110011
gold:/test_sm/loop	xxxxxxxxxxxxxxxxxxx
gold:/test_sm/i	x
gold:/test_sm/rd_	St1
Now	750000 ps
Cursor 1	0 ps

50400 ps to 565800 ps

Now: 750 ns Delta: 2

The simulator resolution (see Simulator Resolution Limit (Verilog) or Simulator Resolution Limit (VHDL)) must be the same for all datasets you are comparing, including the current simulation. If you have a WLF file that is in a different resolution, you can use the wlfman command to change it.

Saving a Simulation to a WLF File

If you add objects to the Dataflow, List, or Wave windows, or log objects with the **log** command, the results of each simulation run are automatically saved to a WLF file called *vsim.wlf* in the current directory. If you then run a new simulation in the same directory, the *vsim.wlf* file is overwritten with the new results.

If you want to save the WLF file and not have it be overwritten, select the dataset tab in the Workspace and then select **File** > **Save**. Or, you can use the **-wlf <filename>** argument to the vsim command or the dataset save command.

Note

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

WLF File Parameter Overview

There are a number of WLF file parameters that you can control via the *modelsim.ini* file or a simulator argument. This section summarizes the various parameters.

Table 7-1. WLF File Parameters

Feature	**vsim argument**	**modelsim.ini**	**Default**
WLF Filename	-wlf <filename>	WLFFilename=<filename>	*vsim.wlf*
WLF Size Limit	-wlfslim <n>	WLFSizeLimit = <n>	no limit
WLF Time Limit	-wlftlim <t>	WLFTimeLimit = <t>	no limit
WLF Compression	-wlfcompress -wlfnocompress	WLFCompress = 0|1	1 (-wlfcompress)
WLF Optimization[1]	-wlfopt -wlfnoopt	WLFOptimize = 0|1	1 (-wlfopt)
WLF Delete on Quit[a]	-wlfdeleteonquit -wlfnodeleteonquit	WLFDeleteOnQuit = 0|1	0
WLF Cache Size[a]	-wlfcachesize <n>	WLFCacheSize = <n>	256
WLF Collapse Mode	-wlfnocollapse -wlfcollapsedelta -wlfcollapsetime	WLFCollapseModel = 0|1|2	1

1. These parameters can also be set using the dataset config command.

- WLF Filename — Specify the name of the WLF file.
- WLF Size Limit — Limit the size of a WLF file to <n> megabytes by truncating from the front of the file as necessary.
- WLF Time Limit — Limit the size of a WLF file to <t> time by truncating from the front of the file as necessary.
- WLF Compression — Compress the data in the WLF file.
- WLF Optimization — Write additional data to the WLF file to improve draw performance at large zoom ranges. Optimization results in approximately 15% larger

WLF files. Disabling WLF optimization also prevents ModelSim from reading a previously generated WLF file that contains optimized data.

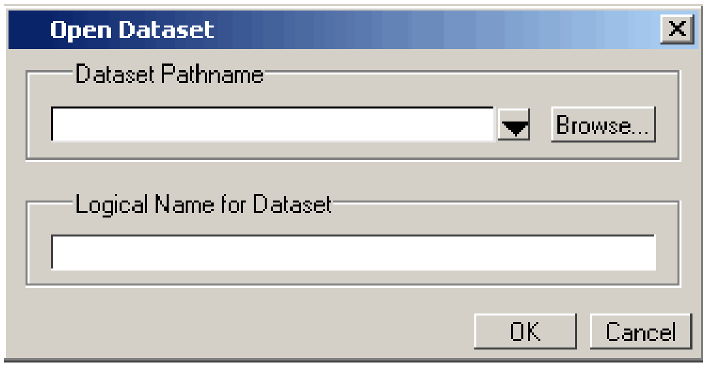
- WLF Delete on Quit — Delete the WLF file automatically when the simulation exits. Valid for current simulation dataset (*vsim.wlf*) only.
- WLF Cache Size — Specify the size in megabytes of the WLF reader cache. WLF reader cache is enabled by default. The default value is 256. This feature caches blocks of the WLF file to reduce redundant file I/O. If the cache is made smaller or disabled, least recently used data will be freed to reduce the cache to the specified size.
- WLF Collapse Mode —WLF event collapsing has three settings: disabled, delta, time:
    - o When disabled, all events and event order are preserved.
    - o Delta mode records an object's value at the end of a simulation delta (iteration) only. Default.
    - o Time mode records an object's value at the end of a simulation time step only.

Opening Datasets

To open a dataset, do one of the following:

- Select **File** > **Open** and choose Log Files or use the dataset open command.

Figure 7-2. Open Dataset Dialog Box



The Open Dataset dialog includes the following options:

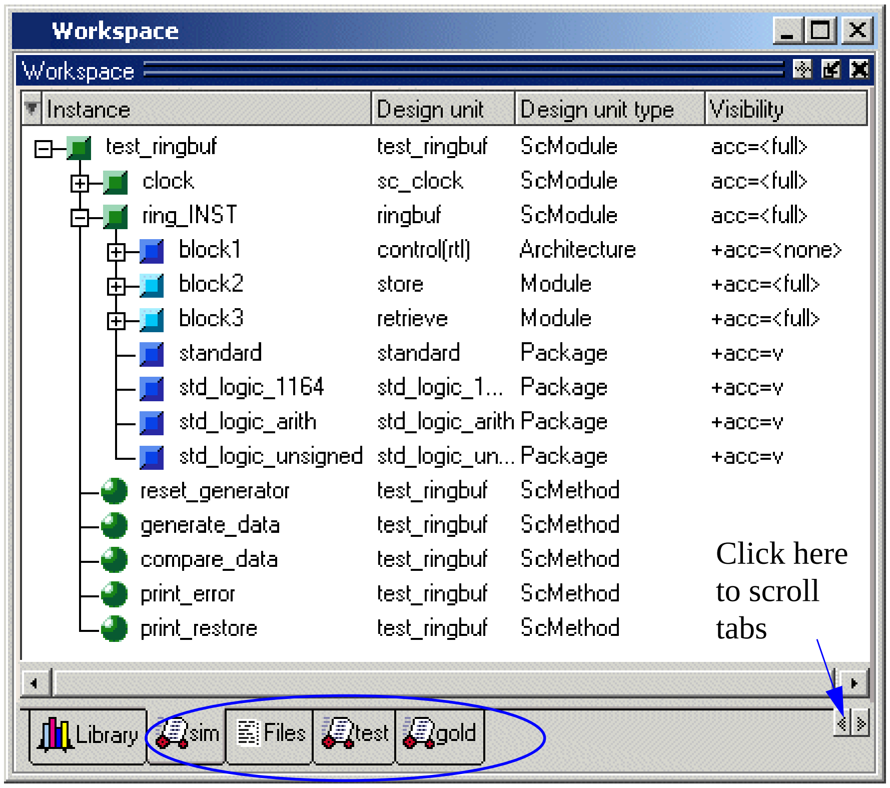
- **Dataset Pathname** — Identifies the path and filename of the WLF file you want to open.
- **Logical Name for Dataset** — This is the name by which the dataset will be referred. By default this is the name of the WLF file.

Viewing Dataset Structure

Each dataset you open creates a structure tab in the Main window workspace. The tab is labeled with the name of the dataset and displays a hierarchy of the design units in that dataset.

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

Figure 7-3. Structure Tabs in Workspace Pane



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

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

Structure Tab Columns

Each structure tab displays three columns by default:

Table 7-2. Structure Tab Columns

Column name	Description
Instance	the name of the instance
Design unit	the name of the design unit

Table 7-2. Structure Tab Columns (cont.)

Column name	Description
Design unit type	the type (e.g., Module, Entity, etc.) of the design unit

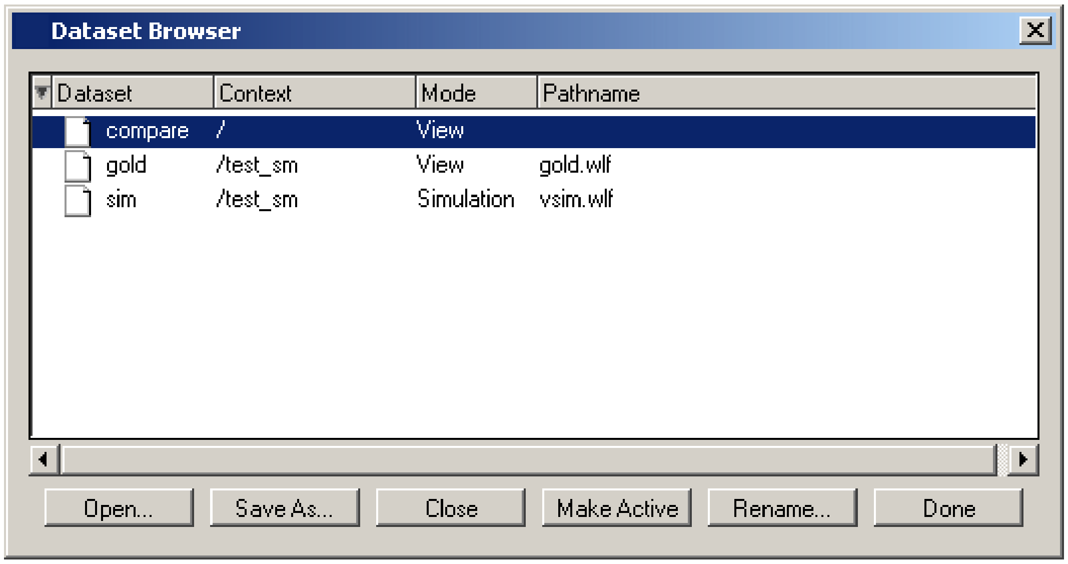
You can hide or show columns by right-clicking a column name and selecting the name on the list.

Managing Multiple Datasets

GUI

When you have one or more datasets open, you can manage them using the **Dataset Browser**. To open the browser, select **File > Datasets**.

Figure 7-4. The Dataset Browser



Dataset	Context	Mode	Pathname
compare	/	View	
gold	/test_sm	View	gold.wlf
sim	/test_sm	Simulation	vsim.wlf

Command Line

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

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

For example:

```
vsim -view foo=vsim.wlf
```

ModelSim designates one of the datasets to be the "active" dataset, and refers all names without dataset prefixes to that dataset. The active dataset is displayed in the context path at the bottom of the Main window. When you select a design unit in a dataset's structure tab, that dataset becomes active automatically. Alternatively, you can use the Dataset Browser or the environment command to change the active dataset.

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

```
sim:/top/alu/out

view:/top/alu/out

golden:.top.alu.out
```

Dataset prefixes are not required unless more than one dataset is open, and you want to refer to something outside the active dataset. When more than one dataset is open, ModelSim will automatically prefix names in the Wave and List windows with the dataset name. You can change this default by selecting **Tools > Window Preferences** (Wave and List windows).

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

```
env foo:
```

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

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

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

Restricting the Dataset Prefix Display

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

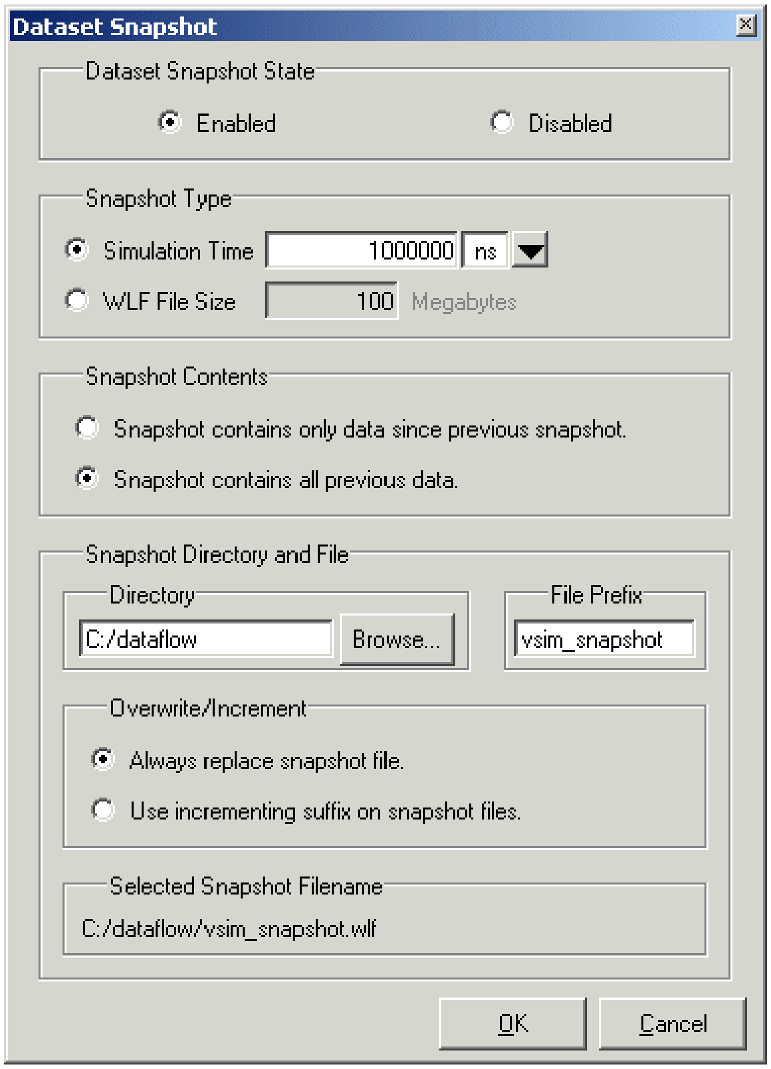
Additionally, you can restrict display of the dataset prefix if you use the **environment -nodataset** command to view a dataset. To display the prefix use the environment command with the **-dataset** option (you won't need to specify this option if the variable noted above is set to 1). The **environment**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

Saving at Intervals with Dataset Snapshot

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

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

Figure 7-5. Dataset Snapshot Dialog



Dataset Snapshot

Dataset Snapshot State
[✓ Enabled] Disabled

Snapshot Type
[✓ Simulation Time] 1000000 ns
WLF File Size 100 Megabytes

Snapshot Contents
Snapshot contains only data since previous snapshot.
[✓ Snapshot contains all previous data.]

Snapshot Directory and File
Directory: C:/dataflow
File Prefix: vsim_snapshot

Overwrite/Increment
[✓ Always replace snapshot file.]
Use incrementing suffix on snapshot files.

Selected Snapshot Filename
C:/dataflow/vsim_snapshot.wlf

Collapsing Time and Delta Steps

By default ModelSim collapses delta steps. This means each logged signal that has events during a simulation delta has its final value recorded to the WLF file when the delta has expired. The event order in the WLF file matches the order of the first events of each signal.

You can configure how ModelSim collapses time and delta steps using arguments to the vsim command or by setting the WLFCollapseMode variable in the *modelsim.ini* file. The table below summarizes the arguments and how they affect event recording.

Table 7-3. vsim Arguments for Collapsing Time and Delta Steps

vsim argument	effect	modelsim.ini setting
-wlfnocollapse	All events for each logged signal are recorded to the WLF file in the exact order they occur in the simulation.	WLFCollapseMode = 0
-wlfdeltacollapse	Each logged signal which has events during a simulation delta has its final value recorded to the WLF file when the delta has expired. Default.	WLFCollapseMode = 1
-wlftimecollapse	Same as delta collapsing but at the timestep granularity.	WLFCollapseMode = 2

When a run completes that includes single stepping or hitting a breakpoint, all events are flushed to the WLF file regardless of the time collapse mode. It's possible that single stepping through part of a simulation may yield a slightly different WLF file than just running over that piece of code. If particular detail is required in debugging, you should disable time collapsing.

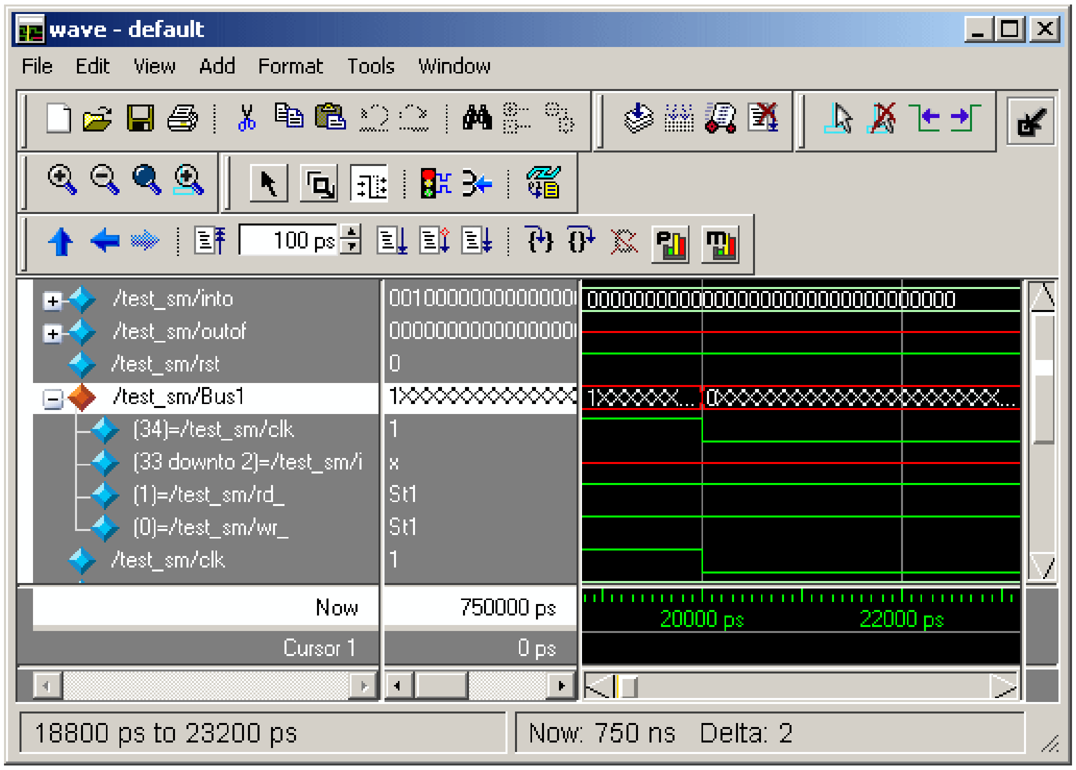
Virtual Objects

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

- Virtual Signals
- Virtual Functions
- Virtual Regions
- Virtual Types

Virtual objects are indicated by an orange diamond as illustrated by *bus* in Figure 7-6:

Figure 7-6. Virtual Objects Indicated by Orange Diamond



Virtual Signals

Virtual signals are aliases for combinations or subelements of signals written to the WLF file by the simulation kernel. They can be displayed in the Objects, List, and Wave windows, accessed by the **examine** command, and set using the **force** command. You can create virtual signals using the **Tools > Combine Signals** (Wave and List windows) menu selections or by using the virtual signal command. Once created, virtual signals can be dragged and dropped from the Objects pane to the Wave and List windows.

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

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

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

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

The definitions of virtuals can be saved to a macro file using the virtual save command. By default, when quitting, ModelSim will append any newly-created virtuals (that have not been saved) to the *virtuals.do* file in the local directory.

If you have virtual signals displayed in the Wave or List window when you save the Wave or List format, you will need to execute the *virtuals.do* file (or some other equivalent) to restore the virtual signal definitions before you re-load the Wave or List format during a later run. There is one exception: "implicit virtuals" are automatically saved with the Wave or List format.

Implicit and Explicit Virtuals

An implicit virtual is a virtual signal that was automatically created by ModelSim without your knowledge and without you providing a name for it. An example would be if you expand a bus in the Wave window, then drag one bit out of the bus to display it separately. That action creates a one-bit virtual signal whose definition is stored in a special location, and is not visible in the Objects pane or to the normal virtual commands.

All other virtual signals are considered "explicit virtuals".

Virtual Functions

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

Examples of virtual functions include the following:

- a function defined as the inverse of a given signal
- a function defined as the exclusive-OR of two signals
- a function defined as a repetitive clock
- a function defined as "the rising edge of CLK delayed by 1.34 ns"

Virtual functions can also be used to convert signal types and map signal values.

The result type of a virtual function can be any of the types supported in the GUI expression syntax: integer, real, boolean, std_logic, std_logic_vector, and arrays and records of these types. Verilog types are converted to VHDL 9-state std_logic equivalents and Verilog net strengths are ignored.

Virtual functions can be created using the virtual function command.

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

Virtual Regions

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

Virtual regions are created and attached using the virtual region command.

Virtual Types

User-defined enumerated types can be defined in order to display signal bit sequences as meaningful alphanumeric names. The virtual type is then used in a type conversion expression to convert a signal to values of the new type. When the converted signal is displayed in any of the windows, the value will be displayed as the enumeration string corresponding to the value of the original signal.

Virtual types are created using the virtual type command.

Chapter 8
Waveform Analysis

When your simulation finishes, you will often want to analyze waveforms to assess and debug your design. Designers typically use the Wave window for waveform analysis. However, you can also look at waveform data in a textual format in the List window.

To analyze waveforms in ModelSim, follow these steps:

1. Compile your files.
2. Load your design.
3. Add objects to the Wave or List window.

```
add wave <object_name>
add list <object_name>
```

4. Run the design.

Objects You Can View

The list below identifies the types of objects can be viewed in the Wave or List window.

- **VHDL objects** — (indicated by dark blue diamond in the Wave window)

  signals, aliases, process variables, and shared variables
- **Verilog objects** — (indicated by light blue diamond in the Wave window)

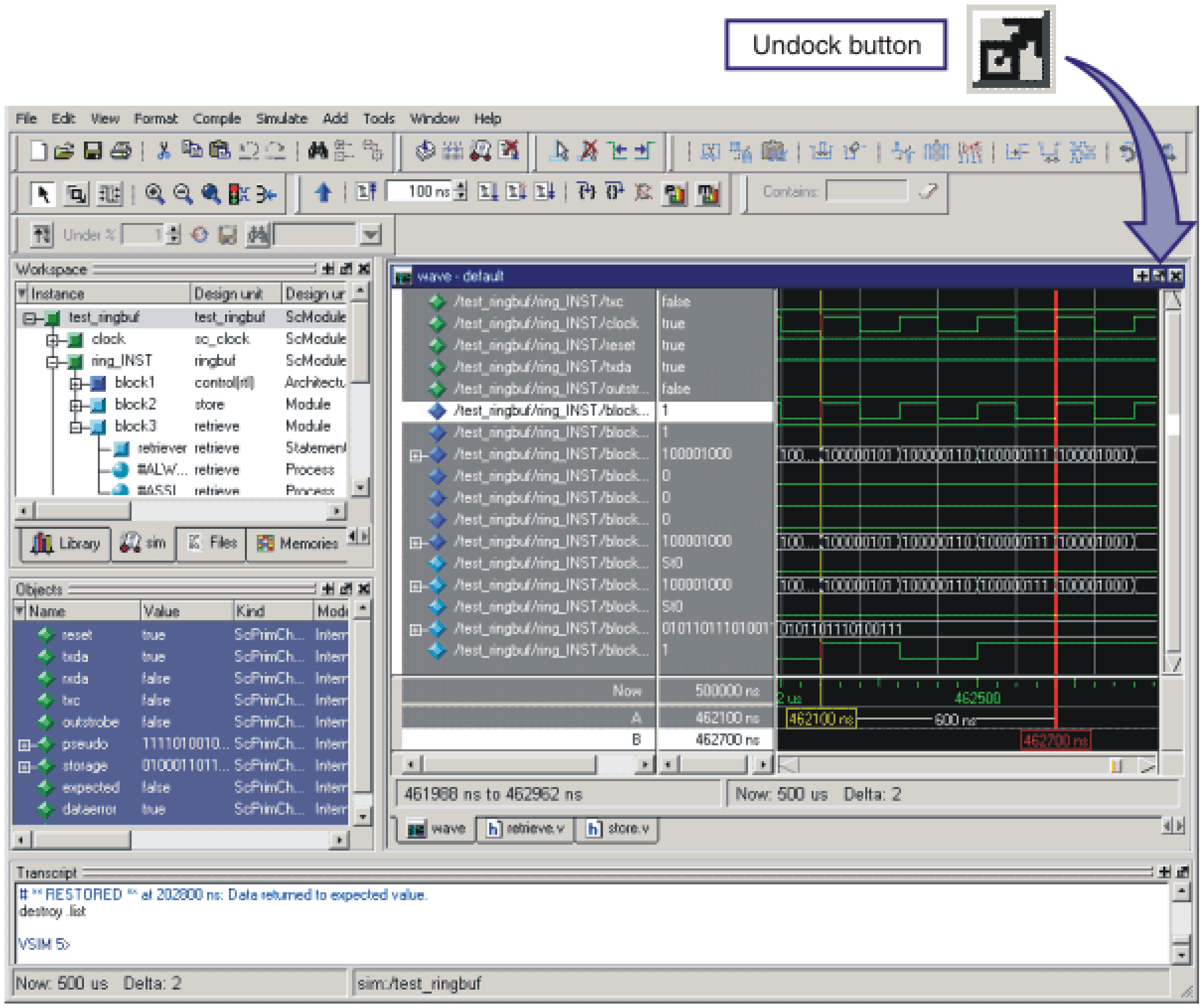
  nets, registers, variables, and named events
- **Virtual objects** — (indicated by an orange diamond in the Wave window)

  virtual signals, buses, and functions, see; Virtual Objects for more information

Wave Window Overview

The Wave window opens by default in the MDI frame of the Main window as shown below. The window can be undocked from the main window by pressing the Undock button in the window header or by using the **view -undock wave** command. The preference variable **PrefMain(ViewUnDocked) wave** can be used to control this default behavior. Setting this variable will open the Wave Window undocked each time you start ModelSim.

Figure 8-1. Undocking the Wave Window

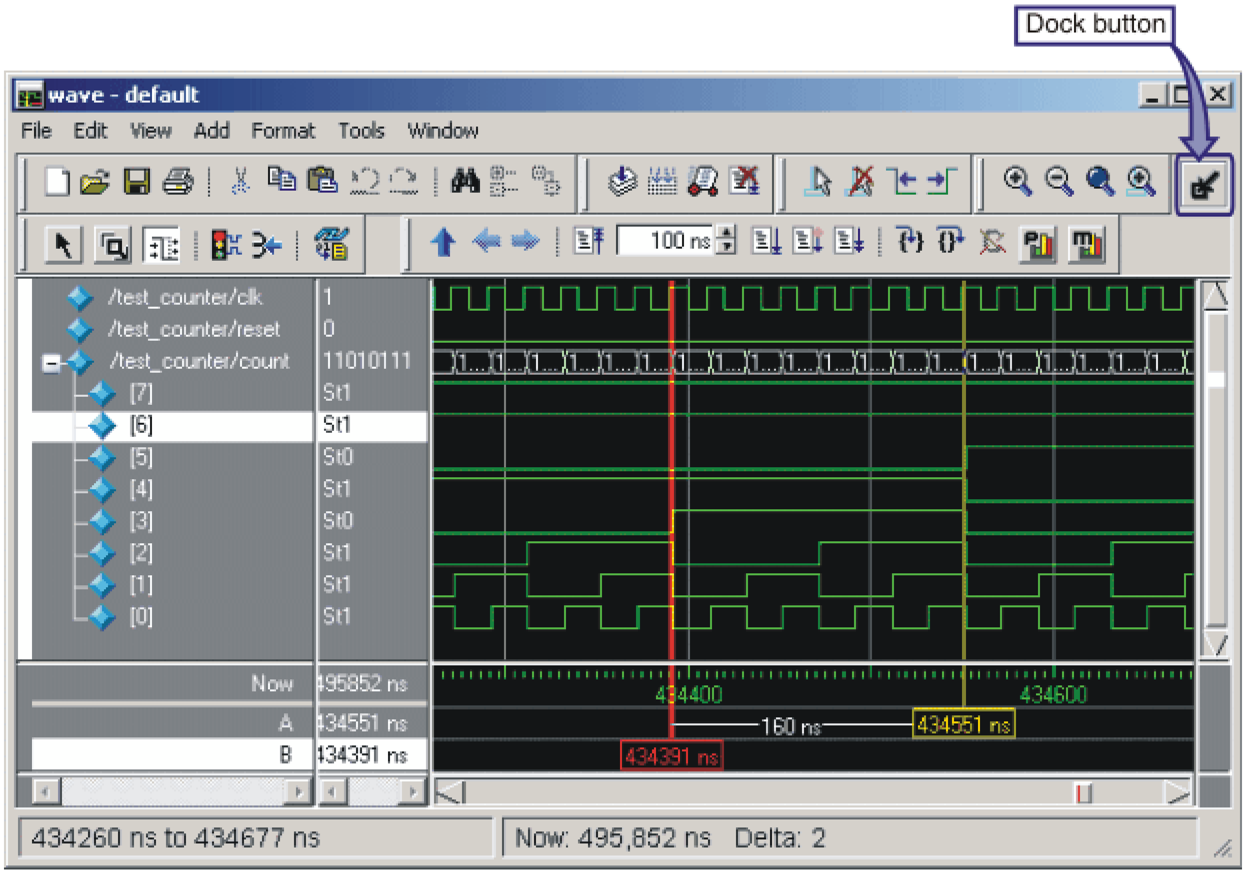


```
# ** RESTORED ** at 202800 ns: Data returned to expected value.
destroy .list

VSIM 5>
```

Now: 500 us Delta: 2 sim:/test_ringbuf

Here is an example of a Wave window that is undocked from the MDI frame. All menus and icons associated with Wave window functions now appear in the menu and toolbar areas of the Wave window.

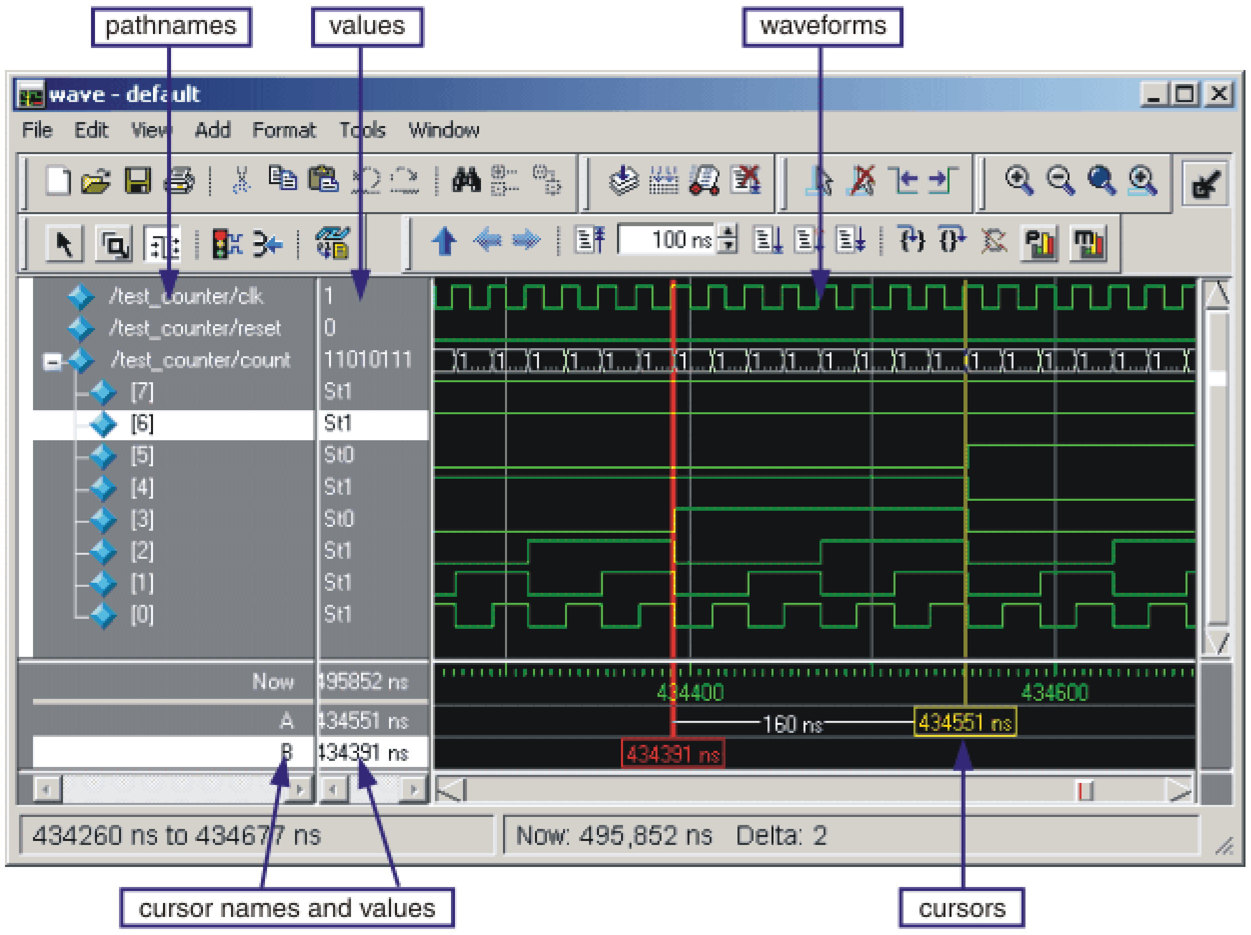
Figure 8-2. Docking the Wave Window



If the Wave window is docked into the Main window MDI frame, all menus and icons that were in the standalone version of the Wave window move into the Main window menu bar and toolbar.

The Wave window is divided into a number of window panes. All window panes in the Wave window can be resized by clicking and dragging the bar between any two panes.

Figure 8-3. Panes in the Wave Window



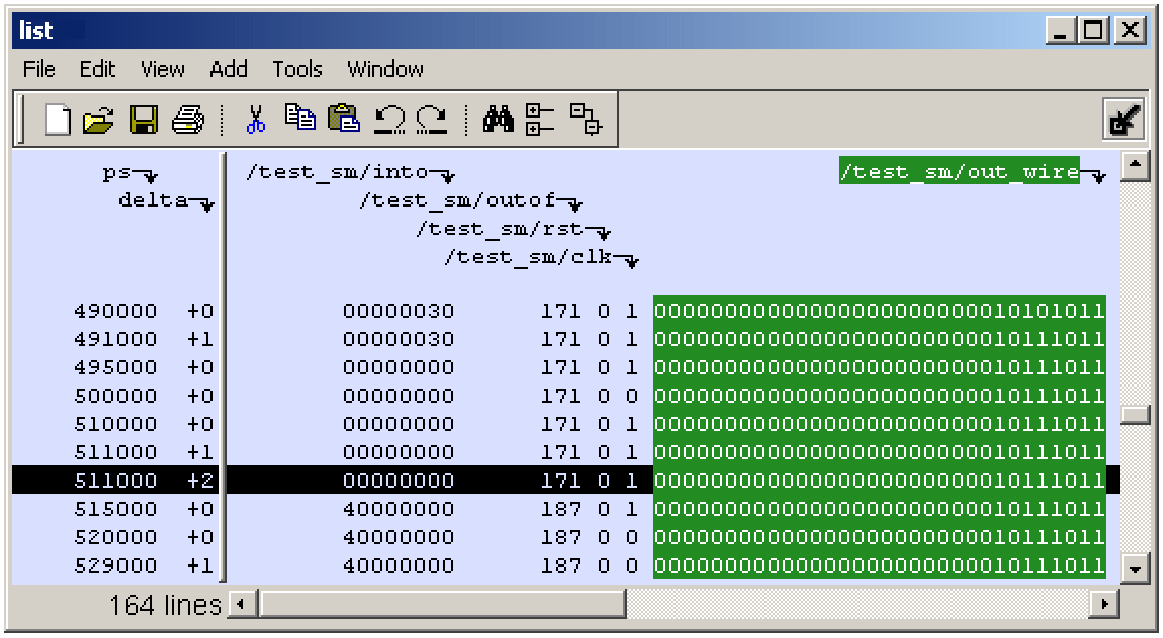
List Window Overview

The List window displays simulation results in tabular format. Common tasks that people use the window for include:

- Using gating expressions and trigger settings to focus in on particular signals or events. See Configuring New Line Triggering in the List Window.
- Debugging delta delay issues. See Delta Delays for more information.

The window is divided into two adjustable panes, which allows you to scroll horizontally through the listing on the right, while keeping time and delta visible on the left.

Figure 8-4. Tabular Format of the List Window



```
         ps        /test_sm/into                               /test_sm/out_wire
           delta              /test_sm/outof
                                     /test_sm/rst
                                        /test_sm/clk

     490000  +0          00000030        171 0 1 00000000000000000000000010101011
     491000  +1          00000030        171 0 1 00000000000000000000000010111011
     495000  +0          00000000        171 0 1 00000000000000000000000010111011
     500000  +0          00000000        171 0 0 00000000000000000000000010111011
     510000  +0          00000000        171 0 1 00000000000000000000000010111011
     511000  +1          00000000        171 0 1 00000000000000000000000010111011
     511000  +2          00000000        171 0 1 00000000000000000000000010111011
     515000  +0          40000000        187 0 1 00000000000000000000000010111011
     520000  +0          40000000        187 0 0 00000000000000000000000010111011
     529000  +1          40000000        187 0 0 00000000000000000000000010111011
     164 lines
```

Adding Objects to the Wave or List Window

You can add objects to the Wave or List window in several ways.

Adding Objects with Drag and Drop

You can drag and drop objects into the Wave or List window from the Workspace, Active Processes, Memory, Objects, Source, or Locals panes. You can also drag objects from the Wave window to the List window and vice versa.

Select the objects in the first window, then drop them into the Wave window. Depending on what you select, all objects or any portion of the design can be added.

Adding Objects with a Menu Command

The **Add** menu in the Main windows let you add objects to the Wave window, List window, or Log file.

Adding Objects with a Command

Use the add list or add wave commands to add objects from the command line. For example:

VSIM> add wave /proc/a

Adds signal */proc/a* to the Wave window.

VSIM> add list *

Adds all the objects in the current region to the List window.

VSIM> add wave -r /*

Adds all objects in the design to the Wave window.

Adding Objects with a Window Format File

Select **File** > **Open** > **Format** and specify a previously saved format file. See Saving the Window Format for details on how to create a format file.

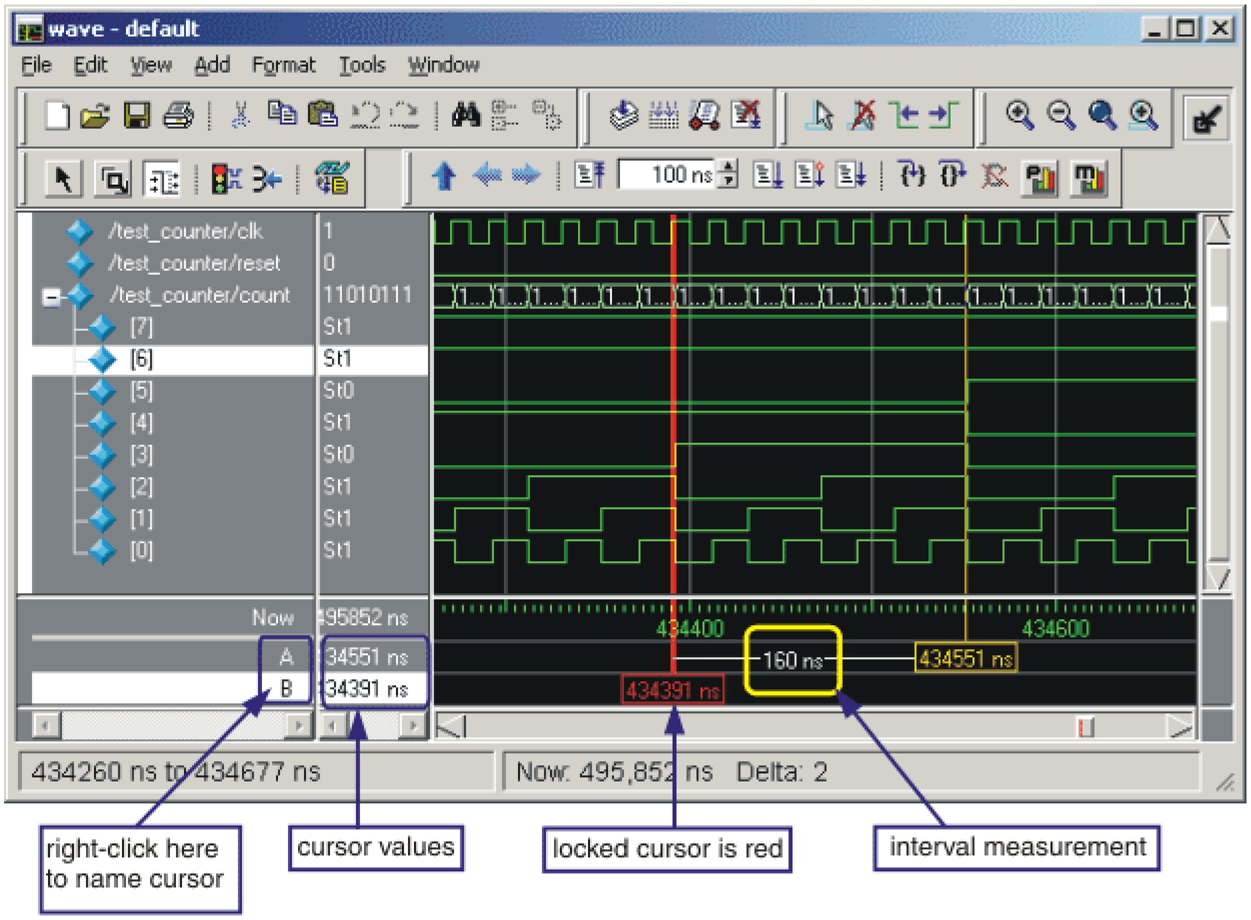
Measuring Time with Cursors in the Wave Window

ModelSim uses cursors to measure time in the Wave window. Cursors extend a vertical line over the waveform display and identify a specific simulation time. Multiple cursors can be used to measure time intervals, as shown in the graphic below.

When the Wave window is first drawn, there is one cursor located at time zero. Clicking anywhere in the waveform display brings that cursor to the mouse location. The selected cursor is drawn as a bold solid line; all other cursors are drawn with thin lines.

As shown in the graphic below, three window panes relate to cursors: the cursor name pane on the bottom left, the cursor value pane in the bottom middle, and the cursor pane with horizontal "tracks" on the bottom right.

Figure 8-5. Cursor Names, Values and Time Measurements



Working with Cursors

The table below summarizes common cursor actions.

Table 8-1. Actions for Cursors

Action	Menu command (Wave window docked)	Menu command (Wave window undocked)	Toolbar button
Add cursor	**Add > Wave > Cursor**	**Add > Cursor**	
Delete cursor	**Wave > Delete Cursor**	**Edit > Delete Cursor**	
Zoom In on Active Cursor	**Wave > Zoom > Zoom Cursor**	**View > Zoom > Zoom Cursor**	

Table 8-1. Actions for Cursors (cont.)

Action	**Menu command** (Wave window docked)	**Menu command** (Wave window undocked)	**Toolbar button**
Lock cursor	**Wave > Edit Cursor**	**Edit > Edit Cursor**	NA
Name cursor	**Wave > Edit Cursor**	**Edit > Edit Cursor**	NA
Select cursor	**Wave > Cursors**	**View > Cursors**	NA

Shortcuts for Working with Cursors

There are a number of useful keyboard and mouse shortcuts related to the actions listed above:

- Select a cursor by clicking the cursor name.
- Jump to a "hidden" cursor (one that is out of view) by double-clicking the cursor name.
- Name a cursor by right-clicking the cursor name and entering a new value. Press <Enter> on your keyboard after you have typed the new name.
- Move a locked cursor by holding down the <shift> key and then clicking-and-dragging the cursor.
- Move a cursor to a particular time by right-clicking the cursor value and typing the value to which you want to scroll. Press <Enter> on your keyboard after you have typed the new value.

Understanding Cursor Behavior

The following list describes how cursors "behave" when you click in various panes of the Wave window:

- If you click in the waveform pane, the cursor closest to the mouse position is selected and then moved to the mouse position.
- Clicking in a horizontal "track" in the cursor pane selects that cursor and moves it to the mouse position.
- Cursors "snap" to a waveform edge if you click or drag a cursor along the selected waveform to within ten pixels of a waveform edge. You can set the snap distance in the Window Preferences dialog. Select **Tools > Options > Wave Preferences** when the Wave window is docked in the Main window MDI frame. Select **Tools > Window Preferences** when the Wave window is a stand-alone, undocked window.
- You can position a cursor without snapping by dragging in the cursor pane below the waveforms.

Jumping to a Signal Transition

You can move the active cursor to the next or previous transition on the selected signal using these two buttons on the toolbar:

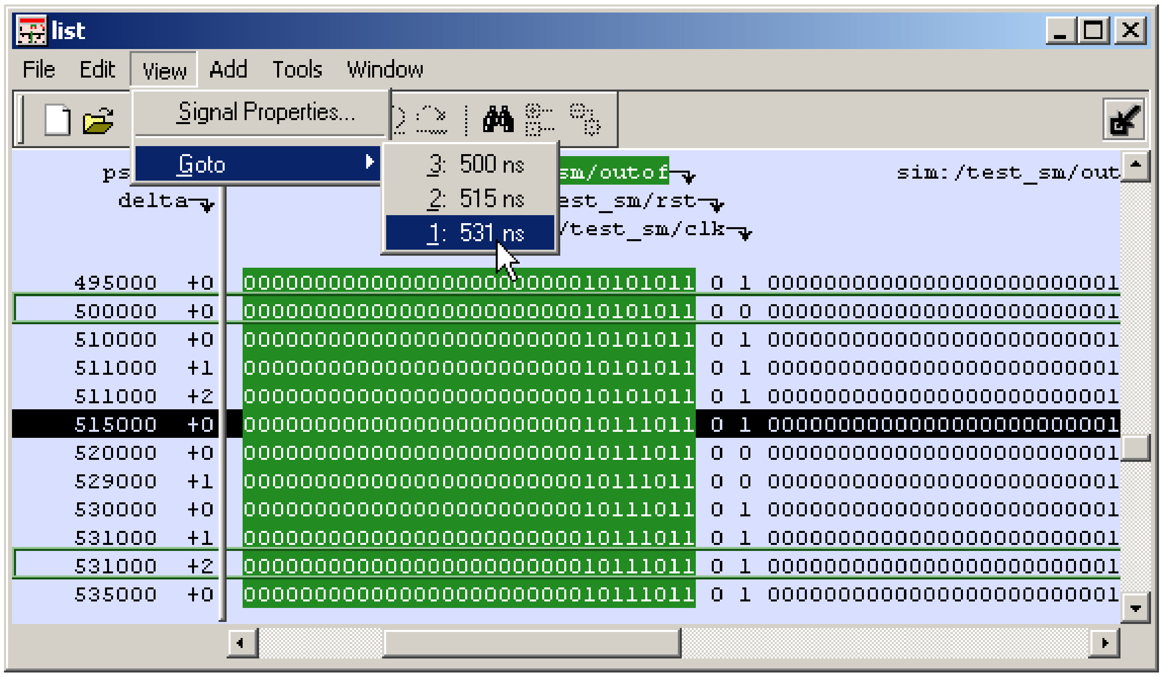
Find Previous Transition locate the previous signal value change for the selected signal

Find Next Transition locate the next signal value change for the selected signal

Setting Time Markers in the List Window

Time markers in the List window are similar to cursors in the Wave window. Time markers tag lines in the data table so you can quickly jump back to that time. Markers are indicated by a thin box surrounding the marked line.

Figure 8-6. Time Markers in the List Window



```
     ps              sm/outof                     sim:/test_sm/out
  delta              est_sm/rst
                     /test_sm/clk

 495000  +0  00000000000000000000000010101011  0 1  0000000000000000000000001
 500000  +0  00000000000000000000000010101011  0 0  0000000000000000000000001
 510000  +0  00000000000000000000000010101011  0 1  0000000000000000000000001
 511000  +1  00000000000000000000000010101011  0 1  0000000000000000000000001
 511000  +2  00000000000000000000000010101011  0 1  0000000000000000000000001
 515000  +0  00000000000000000000000010111011  0 1  0000000000000000000000001
 520000  +0  00000000000000000000000010111011  0 0  0000000000000000000000001
 529000  +1  00000000000000000000000010111011  0 0  0000000000000000000000001
 530000  +0  00000000000000000000000010111011  0 1  0000000000000000000000001
 531000  +1  00000000000000000000000010111011  0 1  0000000000000000000000001
 531000  +2  00000000000000000000000010111011  0 1  0000000000000000000000001
 535000  +0  00000000000000000000000010111011  0 1  0000000000000000000000001
```

Working with Markers

The table below summarizes actions you can take with markers.

Table 8-2. Actions for Time Markers

Action	Method
Add marker	Select a line and then select **Edit** > **Add Marker**

Table 8-2. Actions for Time Markers (cont.)

Action	Method
Delete marker	Select a tagged line and then select **Edit** > **Delete Marker**
Goto marker	Select **View** > **Goto** > **<time>**

Zooming the Wave Window Display

Zooming lets you change the simulation range in the waveform pane. You can zoom using the context menu, toolbar buttons, mouse, keyboard, or commands.

Zooming with the Menu, Toolbar and Mouse

You can access Zoom commands from the **View** menu in the Wave window when it is undocked, from the **Wave** > **Zoom** menu selections in the Main window when the Wave window is docked, or by clicking the right mouse button in the waveform pane of the Wave window.

These zoom buttons are available on the toolbar:

Zoom In 2x
zoom in by a factor of two from the current view

Zoom Out 2x
zoom out by a factor of two from current view

Zoom In on Active Cursor
centers the active cursor in the waveform display and zooms in

Zoom Full
zoom out to view the full range of the simulation from time 0 to the current time

Zoom Mode
change mouse pointer to zoom mode; see below

To zoom with the mouse, first enter zoom mode by selecting **View** > **Zoom** > **Mouse Mode** > **Zoom Mode**. The left mouse button then offers 3 zoom options by clicking and dragging in different directions:

- Down-Right *or* Down-Left: Zoom Area (In)
- Up-Right: Zoom Out
- Up-Left: Zoom Fit

Also note the following about zooming with the mouse:

- The zoom amount is displayed at the mouse cursor. A zoom operation must be more than 10 pixels to activate.
- You can enter zoom mode temporarily by holding the <Ctrl> key down while in select mode.
- With the mouse in the Select Mode, the middle mouse button will perform the above zoom operations.

Saving Zoom Range and Scroll Position with Bookmarks

Bookmarks save a particular zoom range and scroll position. This lets you return easily to a specific view later. You save the bookmark with a name and then access the named bookmark from the Bookmark menu. Bookmarks are saved in the Wave format file (see Adding Objects with a Window Format File) and are restored when the format file is read.

Managing Bookmarks

The table below summarizes actions you can take with bookmarks.

Table 8-3. Actions for Bookmarks

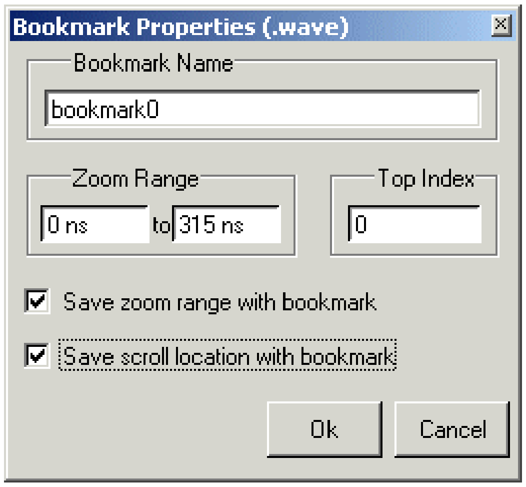
Action	Menu commands (Wave window docked)	Menu commands (Wave window undocked)	Command
Add bookmark	**Add** > **Wave** > **Bookmark**	**Add** > **Bookmark**	bookmark add wave
View bookmark	**Wave** > **Bookmarks** > <bookmark_name>	**View** > **Bookmarks** > <bookmark_name>	bookmark goto wave
Delete bookmark	**Wave** > **Bookmarks** > **Bookmarks** > <select bookmark then **Delete**>	**View** > **Bookmarks** > **Bookmarks** > <select bookmark then **Delete**>	bookmark delete wave

Adding Bookmarks

To add a bookmark, follow these steps:

1. Zoom the wave window as you see fit using one of the techniques discussed in Zooming the Wave Window Display.
2. If the Wave window is docked, select **Add > Wave > Bookmark**. If the Wave window is undocked, select **Add > Bookmark**.

Figure 8-7. Bookmark Properties Dialog


Bookmark Properties (.wave)
Bookmark Name
bookmark0
Zoom Range
0 ns
to
315 ns
Top Index
0
Save zoom range with bookmark
Save scroll location with bookmark
Ok
Cancel

3. Give the bookmark a name and click OK.

Editing Bookmarks

Once a bookmark exists, you can change its properties by selecting **Wave** > **Bookmarks** > **Bookmarks** if the Wave window is docked; or by selecting **Tools** > **Bookmarks** if the Wave window is undocked.

Searching in the Wave and List Windows

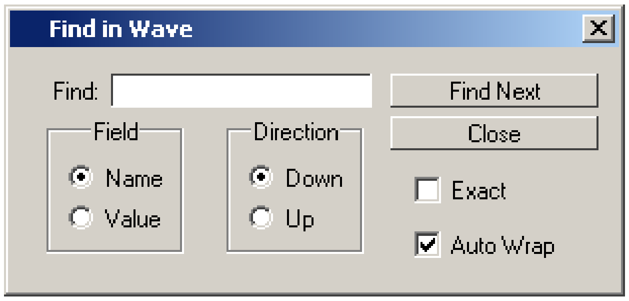
The Wave and List windows provide two methods for locating objects:

- Finding signal names – Select **Edit** > **Find** or use the find command to search for the name of a signal.
- Search for values or transitions – Select **Edit** > **Signal Search** to locate transitions or signal values. The search feature is not available in all versions of ModelSim.

Finding Signal Names

The Find command is used to locate a signal name or value in the Wave or List window. When you select **Edit** > **Find**, the Find dialog appears.

Figure 8-8. Find Signals by Name or Value



Field: [✓ Name] Value
Direction: [✓ Down] Up
☐ Exact
☑ Auto Wrap

One option of note is the "Exact" checkbox. Check **Exact** if you only want to find objects that match your search exactly. For example, searching for "clk" without **Exact** will find */top/clk* and *clk1*.

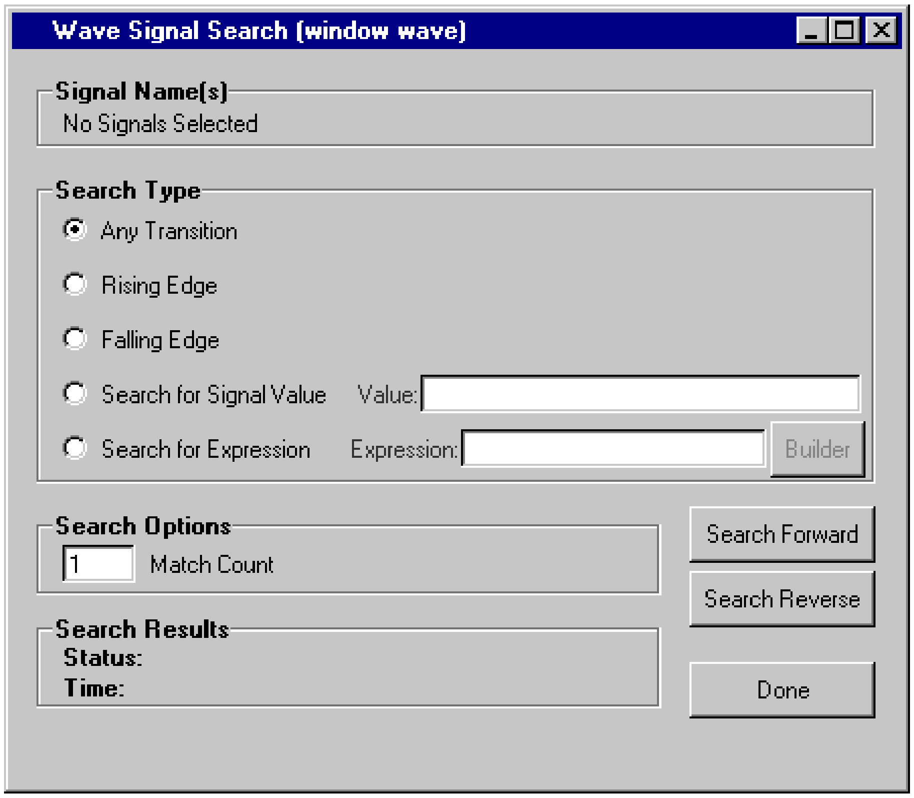
There are two differences between the Wave and List windows as it relates to the Find feature:

- In the Wave window you can specify a value to search for in the values pane.
- The find operation works only within the active pane in the Wave window.

Searching for Values or Transitions

Available in some versions of ModelSim, the Search command lets you search for transitions or values on selected signals. When you select **Edit** > **Search Signals**, the Signal Search dialog appears.

Figure 8-9. Wave Signal Search Dialog



Signal Name(s)
No Signals Selected

Search Type
(● Any Transition) Rising Edge / Falling Edge / Search for Signal Value Value: / Search for Expression Expression:

Search Options
1 Match Count

Search Results
Status:
Time:

One option of note is **Search for Expression**. The expression can involve more than one signal but is limited to signals currently in the window. Expressions can include constants, variables, and DO files. See Expression Syntax for more information.

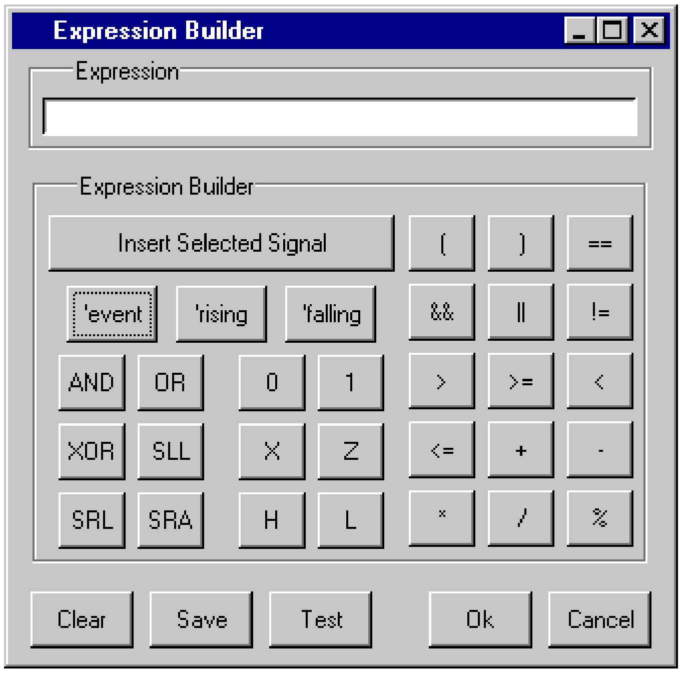
Using the Expression Builder for Expression Searches

The Expression Builder is a feature of the Wave and List Signal Search dialog boxes, and the List trigger properties dialog box. It aids in building a search expression that follows the GUI_expression_format.

To locate the Builder:

- select **Edit > Search Signals** (List or Wave window)
- select the **Search for Expression** option in the resulting dialog box
- select the **Builder** button

Figure 8-10. Expression Builder Dialog



The Expression Builder dialog box provides an array of buttons that help you build a GUI expression. For instance, rather than typing in a signal name, you can select the signal in the associated Wave or List window and press Insert Selected Signal. All Expression Builder buttons correspond to the Expression Syntax.

Saving an Expression to a Tcl Variable

Clicking the **Save** button will save the expression to a Tcl variable. Once saved this variable can be used in place of the expression. For example, say you save an expression to the variable "foo". Here are some operations you could do with the saved variable:

- Read the value of *foo* with the set command:

```
set foo
```

- Put $foo in the Expression: entry box for the Search for Expression selection.
- Issue a searchlog command using foo:

```
searchlog -expr $foo 0
```

Searching for when a Signal Reaches a Particular Value

Select the signal in the Wave window and click **Insert Selected Signal** and ==. Then, click the value buttons or type a value.

Evaluating Only on Clock Edges

Click the **&&** button to AND this condition with the rest of the expression. Then select the clock in the Wave window and click **Insert Selected Signal** and **'rising**. You can also select the falling edge or both edges.

Operators

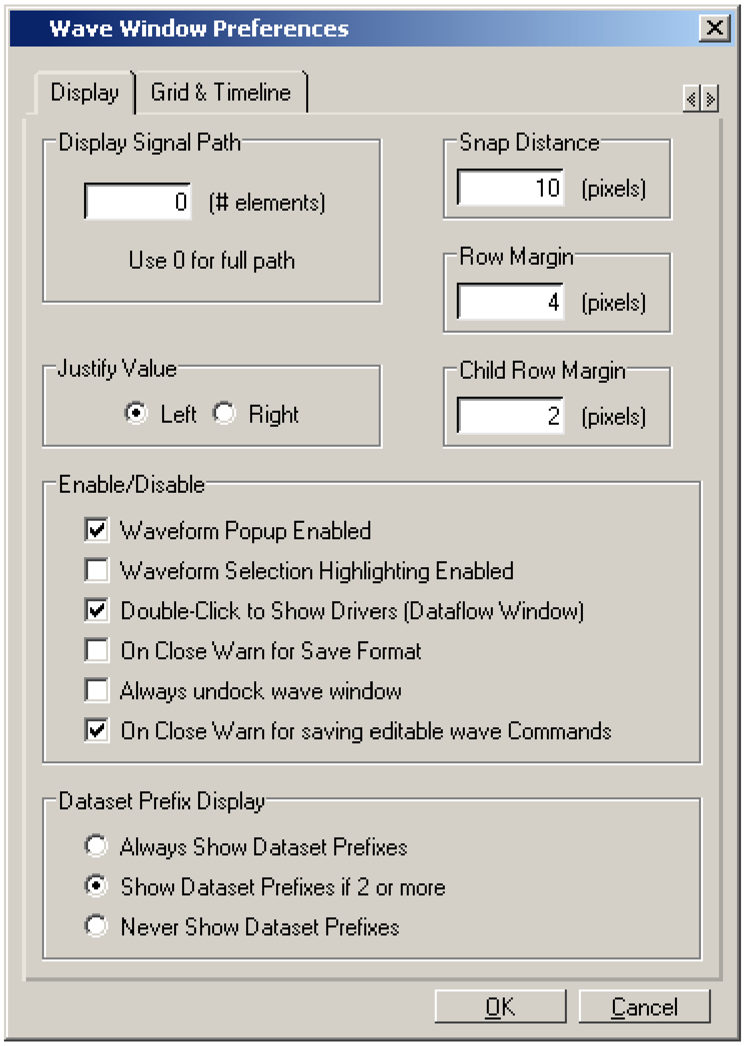
Other buttons will add operators of various kinds (see Expression Syntax), or you can type them in.

Formatting the Wave Window

Setting Wave Window Display Preferences

You can set Wave Window display preferences by selecting **Tools** > **Options** > **Wave Preferences** (when the window is docked in the MDI frame) or **Tools** > **Window Preferences** (when the window is undocked). These commands open the Wave Window Preferences dialog (Figure 8-11).

Figure 8-11. Display Tab of the Wave Window Preferences Dialog



Display Signal Path: 0 (# elements)
Use 0 for full path

Snap Distance: 10 (pixels)

Row Margin: 4 (pixels)

Justify Value: [✓ Left] Right

Child Row Margin: 2 (pixels)

Enable/Disable
- [✓] Waveform Popup Enabled
- [ ] Waveform Selection Highlighting Enabled
- [✓] Double-Click to Show Drivers (Dataflow Window)
- [ ] On Close Warn for Save Format
- [ ] Always undock wave window
- [✓] On Close Warn for saving editable wave Commands

Dataset Prefix Display
- Always Show Dataset Prefixes
- [✓ Show Dataset Prefixes if 2 or more]
- Never Show Dataset Prefixes

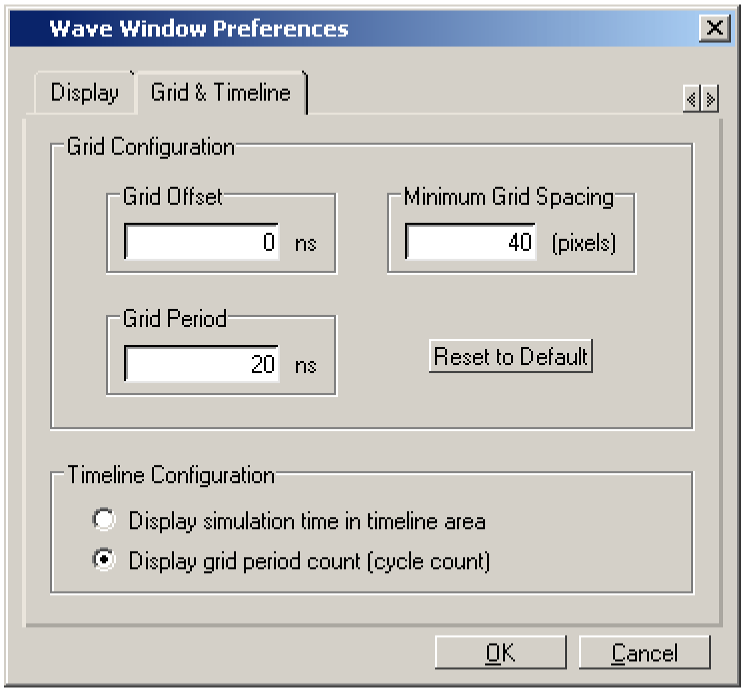
Hiding/Showing Path Hierarchy

You can set how many elements of the object path display by changing the Display Signal Path value in the Wave Window Preferences dialog (Figure 8-11). Zero indicates the full path while a non-zero number indicates the number of path elements to be displayed.

Setting the Timeline to Count Clock Cycles

You can set the timeline of the Wave window to count clock cycles rather than elapsed time. If the Wave window is docked in the MDI frame, open the Wave Window Preferences dialog by selecting **Tools** > **Options** > **Wave Preferences** from the Main window menus. If the Wave window is undocked, select **Tools** > **Window Preferences** from the Wave window menus. This opens the Wave Window Preferences dialog. In the dialog, select the Grid & Timeline tab (Figure 8-12).

Figure 8-12. Grid & Timeline Tab of Wave Window Preferences Dialog



Wave Window Preferences

Display | Grid & Timeline

Grid Configuration

Grid Offset: 0 ns

Minimum Grid Spacing: 40 (pixels)

Grid Period: 20 ns

Reset to Default

Timeline Configuration

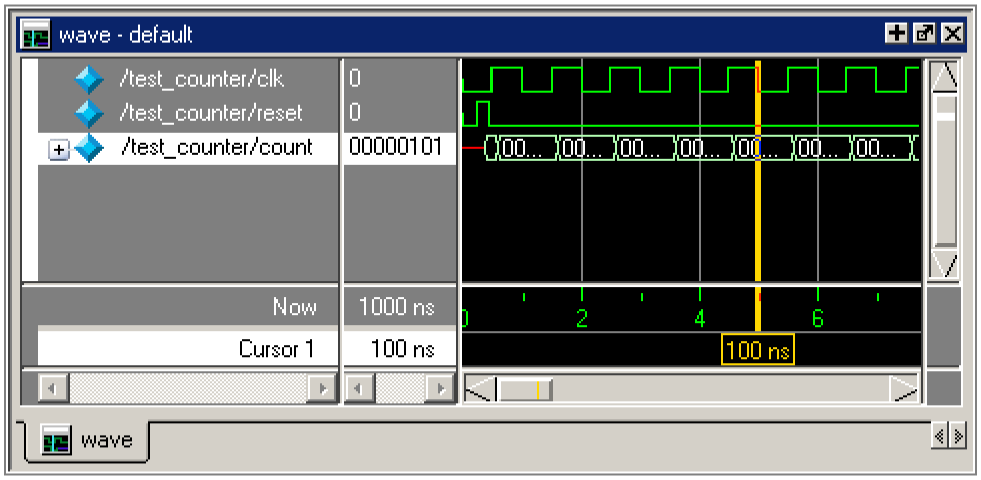
( ) Display simulation time in timeline area

[✓ Display grid period count (cycle count)]

OK | Cancel

Enter the period of your clock in the Grid Period field and select "Display grid period count (cycle count)." The timeline will now show the number of clock cycles, as shown in Figure 8-13.

Figure 8-13. Clock Cycles in Timeline of Wave Window



wave - default

/test_counter/clk	0
/test_counter/reset	0
/test_counter/count	00000101
Now	1000 ns
Cursor 1	100 ns

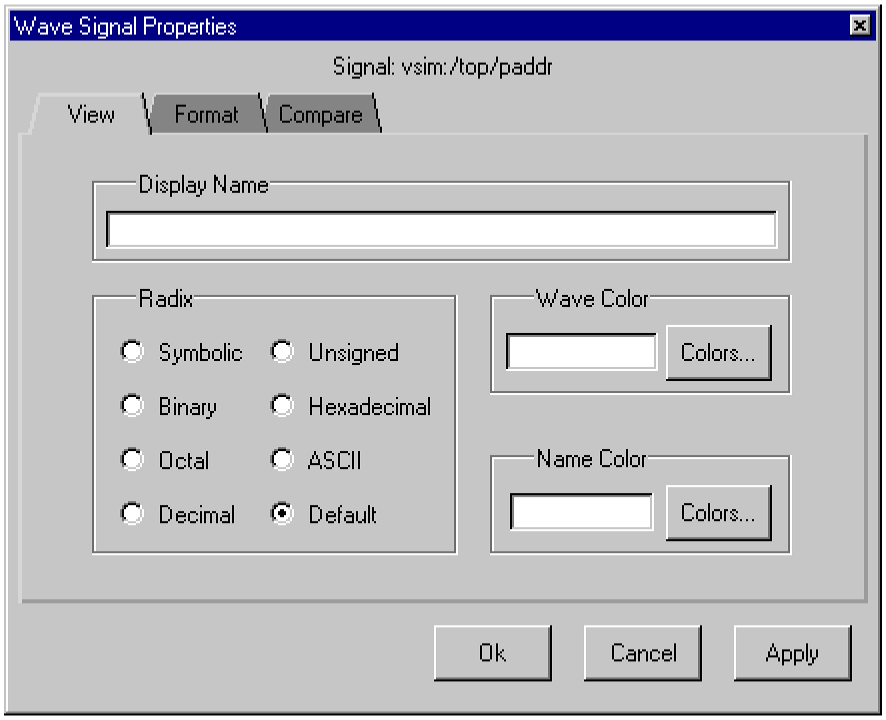
Formatting Objects in the Wave Window

You can adjust various object properties to create the view you find most useful. Select one or more objects and then select **View** > **Properties** or use the selections in the **Format** menu.

Changing Radix (base) for the Wave Window

One common adjustment is changing the radix (base) of an object. When you select **View** > **Properties**, the Wave Signal Properties dialog appears.

Figure 8-14. Changing Signal Radix


Wave Signal Properties

Signal: vsim:/top/paddr

Radix: Symbolic ( ) Unsigned ( ) Binary ( ) Hexadecimal ( ) Octal ( ) ASCII ( ) Decimal ( ) [✓ Default]

The default radix is symbolic, which means that for an enumerated type, the value pane lists the actual values of the enumerated type of that object. For the other radixes - binary, octal, decimal, unsigned, hexadecimal, or ASCII - the object value is converted to an appropriate representation in that radix.

Note
When the symbolic radix is chosen for SystemVerilog reg and integer types, the values are treated as binary. When the symbolic radix is chosen for SystemVerilog bit and int types, the values are considered to be decimal.

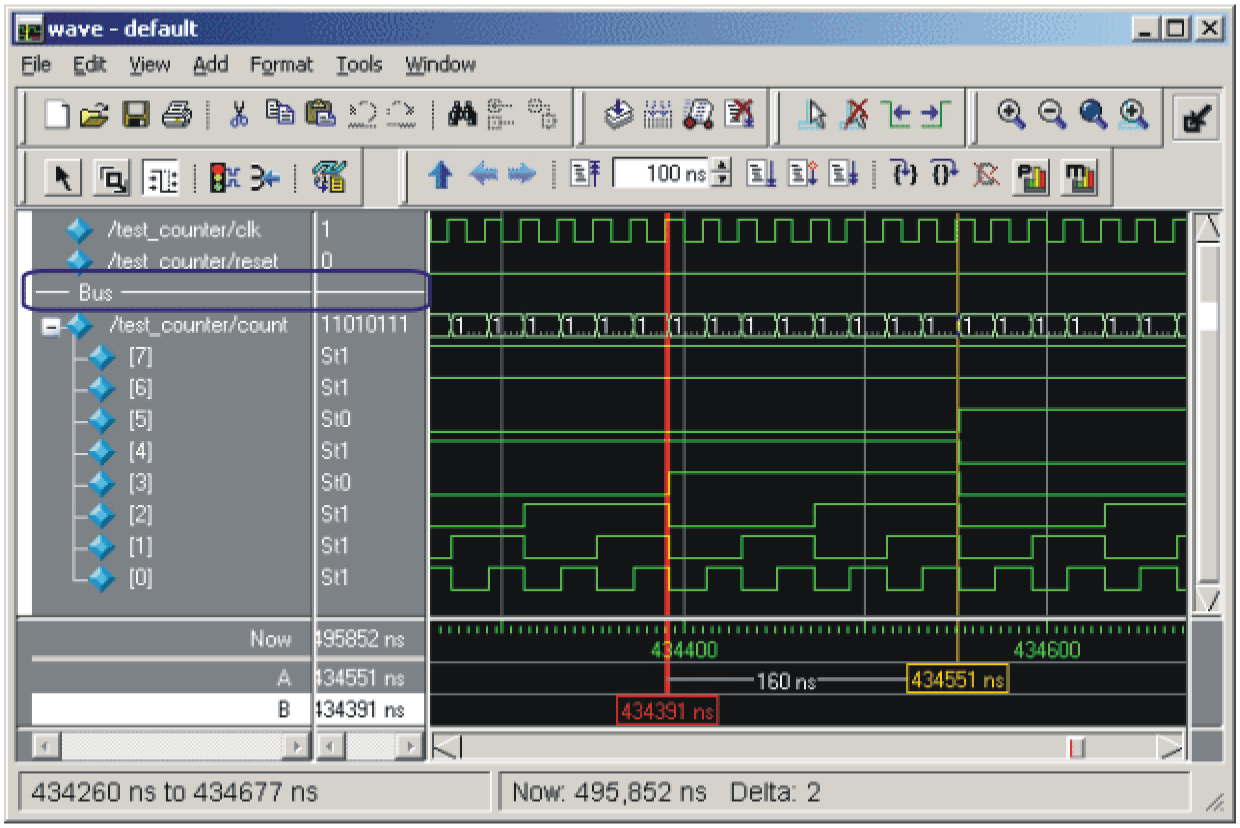
Aside from the Wave Signal Properties dialog, there are three other ways to change the radix:

- Change the default radix for the current simulation using **Simulate** > **Runtime Options** (Main window)
- Change the default radix for the current simulation using the radix command.
- Change the default radix permanently by editing the DefaultRadix variable in the *modelsim.ini* file.

Dividing the Wave Window

Dividers serve as a visual aid for debugging, allowing you to separate signals and waveforms for easier viewing. In the graphic below, a bus is separated from the two signals above it with a divider called "Bus."

Figure 8-15. Separate Signals with Wave Window Dividers



To insert a divider, follow these steps:

1. Select the signal above which you want to place the divider.
2. If the Wave pane is docked in MDI frame of the Main window, select **Add** > **Wave** > **Divider** from the Main window menu bar. If the Wave window stands alone, undocked from the Main window, select **Add** > **Divider** from the Wave window menu bar.
3. Specify the divider name in the Wave Divider Properties dialog. The default name is New Divider. Unnamed dividers are permitted. Simply delete "New Divider" in the Divider Name field to create an unnamed divider.
4. Specify the divider height (default height is 17 pixels) and then click OK.

You can also insert dividers with the **-divider** argument to the add wave command.

Working with Dividers

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

Table 8-4. Actions for Dividers

Action	Method
Move a divider	Click-and-drag the divider to the desired location
Change a divider's name or size	Right-click the divider and select Divider Properties
Delete a divider	Right-click the divider and select Delete

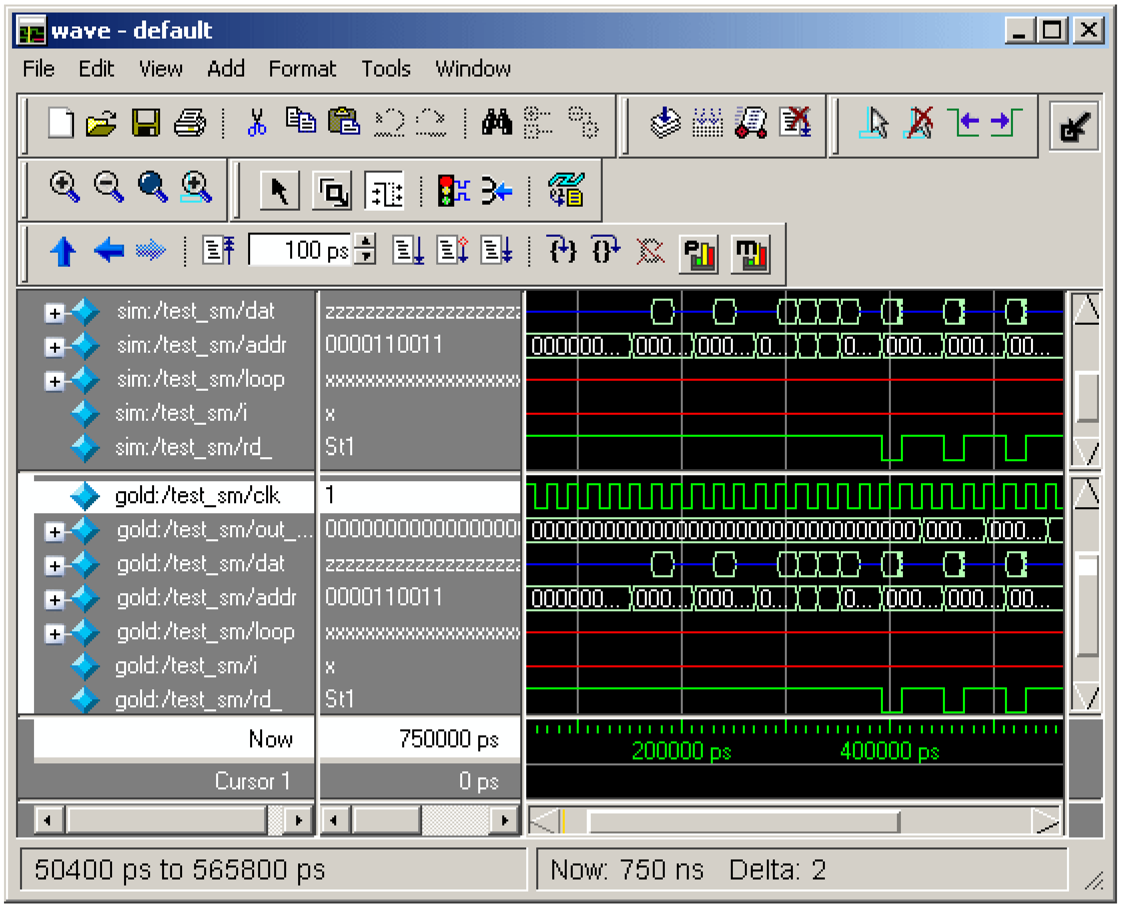
Splitting Wave Window Panes

The pathnames, values, and waveforms panes of the Wave window display can be split to accommodate signals from one or more datasets. For more information on viewing multiple simulations, see WLF Files (Datasets) and Virtuals.

To split the window, select **Add** > **Window Pane**.

In the illustration below, the top split shows the current active simulation with the prefix "sim," and the bottom split shows a second dataset with the prefix "gold".

Figure 8-16. Splitting Wave Window Panes



sim:/test_sm/dat	zzzzzzzzzzzzzzzzzzzz	
sim:/test_sm/addr	0000110011	
sim:/test_sm/loop	xxxxxxxxxxxxxxxxxxxx	
sim:/test_sm/i	x	
sim:/test_sm/rd_	St1	
gold:/test_sm/clk	1	
gold:/test_sm/out_...	0000000000000000	
gold:/test_sm/dat	zzzzzzzzzzzzzzzzzzzz	
gold:/test_sm/addr	0000110011	
gold:/test_sm/loop	xxxxxxxxxxxxxxxxxxxx	
gold:/test_sm/i	x	
gold:/test_sm/rd_	St1	
Now	750000 ps	200000 ps 400000 ps
Cursor 1	0 ps	

50400 ps to 565800 ps   Now: 750 ns Delta: 2

The Active Split

The active split is denoted with a solid white bar to the left of the signal names. The active split becomes the target for objects added to the Wave window.

Wave Groups

Wave groups are a wave window specific container object for creating arbitrary groups of items. A wave group may contain 0, 1 or many items. The command line as well as drag and drop may be used to add or remove items from a group. Groups themselves may be dragged around the wave window or to another wave window.

Currently, groups may not be nested.

Creating a Wave Group

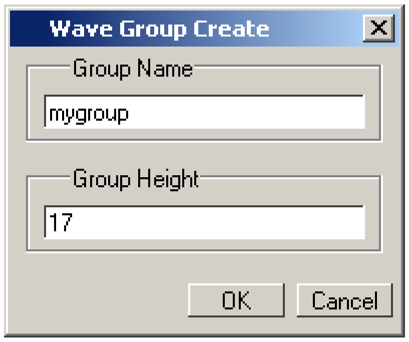
There are two ways to create a wave group.

1. Use the **Tools** > **Group** menu selection.

a. Select a set of signals in the wave window.

b. Select the **Tools** > **Group** menu item. The Wave Group Create dialog will appear.

Figure 8-17. Fill in the name of the group in the Group Name field.



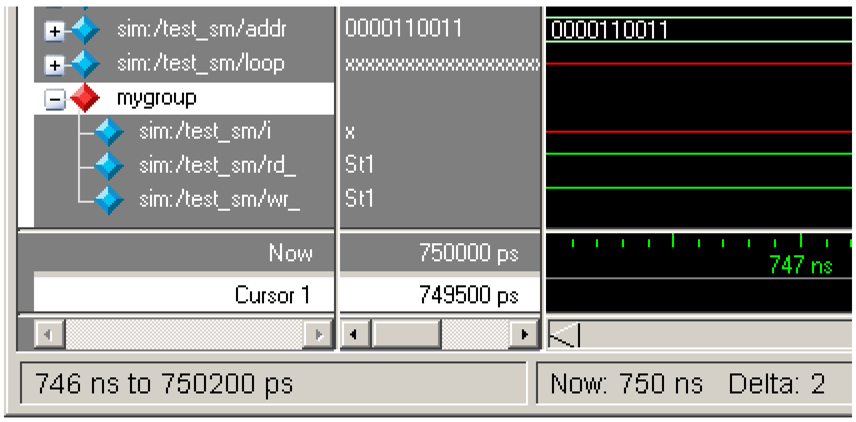
Wave Group Create

Group Name: mygroup

Group Height: 17

c. Click Ok. The new wave group will be denoted by a red diamond in the Wave window pathnames.

Figure 8-18. Wave groups denoted by red diamond



```
sim:/test_sm/addr     0000110011
sim:/test_sm/loop     xxxxxxxxxxxxxxxxxxxx
mygroup
  sim:/test_sm/i      x
  sim:/test_sm/rd_    St1
  sim:/test_sm/wr_    St1
Now                   750000 ps
Cursor 1              749500 ps
746 ns to 750200 ps        Now: 750 ns   Delta: 2
```

2. Use the -group argument to the add wave command.

Example 1 — The following command will create a group named *mygroup* containing three items:

```
add wave -group mygroup sig1 sig2 sig3
```

Example 2 — The following command will create an empty group named *mygroup*:

```
add wave -group mygroup
```

Deleting or Ungrouping a Wave Group

If a wave group is selected and cut or deleted the entire group and all its contents will be removed from the wave window. Likewise, the delete wave command will remove the entire group if the group name is specified.

If a wave group is selected and the **Tools** > **Ungroup** menu item is selected the group will be removed and all of its contents will remain in the Wave window in existing order.

Adding Items to an Existing Wave Group

There are three ways to add items to an existing wave group.

1. Using the drag and drop capability to move items outside of the group or from other windows within ModelSim into the group. The insertion indicator will show the position the item will be dropped into the group. If the cursor is moved over the lower portion of the group item name a box will be drawn around the group name indicating the item will be dropped into the last position in the group.
2. The cut/copy/paste functions may be used to paste items into a group.
3. Use the **add wave -group** command.

   The following example adds two more signals to an existing group called *mygroup*.

   ```
   add wave -group mygroup sig4 sig5
   ```

Removing Items from an Existing Wave Group

You can use any of the following methods to remove an item from a wave group.

1. Use the drag and drop capability to move an item outside of the group.
2. Use menu or icon selections to cut or delete an item or items from the group.
3. Use the delete wave command to specify a signal to be removed from the group.

Note

The delete wave command removes all occurrences of a specified name from the wave window, not just an occurrence within a group.

Miscellaneous Wave Group Features

Dragging a wave group from the Wave window to the List window will result in all of the items within the group being added to the List window.

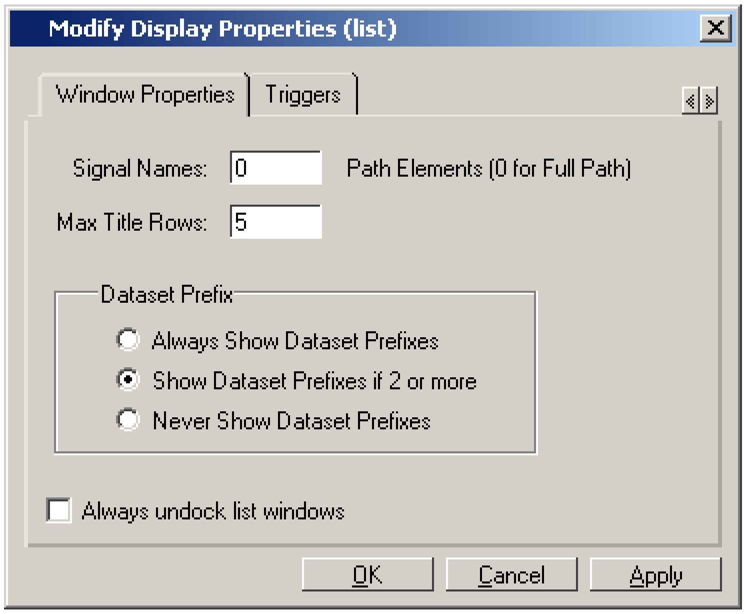
Dragging a group from the Wave window to the Transcript window will result in a list of all of the items within the group being added to the existing command line, if any.

Formatting the List Window

Setting List Window Display Properties

Before you add objects to the List window, you can set the window's display properties. To change when and how a signal is displayed in the List window, select **Tools > List Preferences** from the List window menu bar (when the window is undocked).

Figure 8-19. Modifying List Window Display Properties



Signal Names: 0 Path Elements (0 for Full Path)

Max Title Rows: 5

Dataset Prefix

- Always Show Dataset Prefixes
- [✓ Show Dataset Prefixes if 2 or more]
- Never Show Dataset Prefixes

☐ Always undock list windows

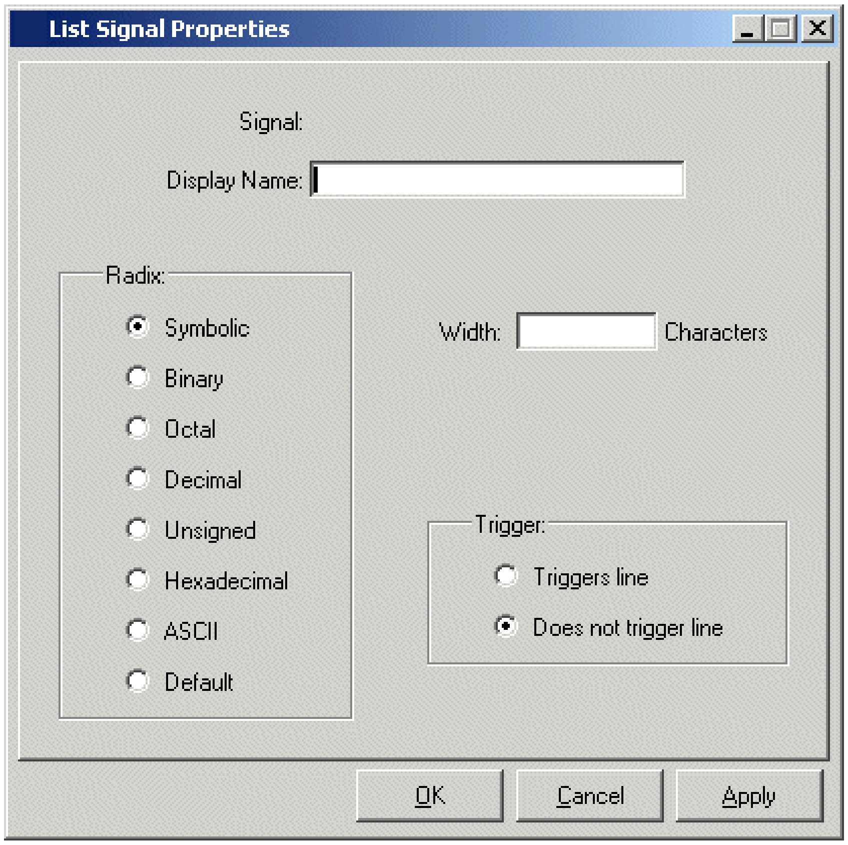
Formatting Objects in the List Window

You can adjust various properties of objects to create the view you find most useful. Select one or more objects and then select **View** > **Signal Properties** from the List window menu bar (when the window is undocked).

Changing Radix (base) for the List Window

One common adjustment is changing the radix (base) of an object. When you select **View** > **Signal Properties**, the List Signal Properties dialog appears (Figure 8-20).

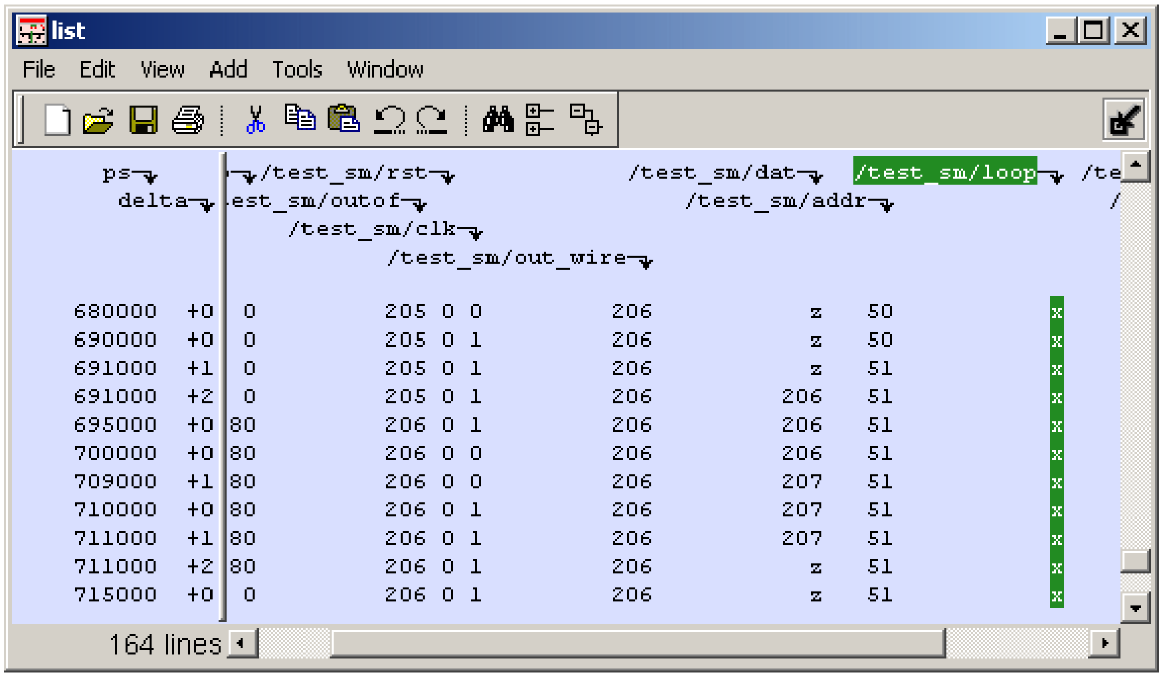
Figure 8-20. List Signal Properties Dialog



The default radix is symbolic, which means that for an enumerated type, the window lists the actual values of the enumerated type of that object. For the other radixes - binary, octal, decimal, unsigned, hexadecimal, or ASCII - the object value is converted to an appropriate representation in that radix.

Changing the radix can make it easier to view information in the List window. Compare the image below (with decimal values) with the image in the section List Window Overview (with symbolic values).

Figure 8-21. Changing the Radix in the List Window



```
       ps     /test_sm/rst                 /test_sm/dat    /test_sm/loop  /te
    delta    est_sm/outof                    /test_sm/addr                 /
                 /test_sm/clk
                          /test_sm/out_wire

 680000  +0  0         205 0 0         206            z    50               x
 690000  +0  0         205 0 1         206            z    50               x
 691000  +1  0         205 0 1         206            z    51               x
 691000  +2  0         205 0 1         206          206    51               x
 695000  +0 80         206 0 1         206          206    51               x
 700000  +0 80         206 0 0         206          206    51               x
 709000  +1 80         206 0 0         206          207    51               x
 710000  +0 80         206 0 1         206          207    51               x
 711000  +1 80         206 0 1         206          207    51               x
 711000  +2 80         206 0 1         206            z    51               x
 715000  +0  0         206 0 1         206            z    51               x
```

164 lines

Aside from the List Signal Properties dialog, there are three other ways to change the radix:

- Change the default radix for the current simulation using **Simulate > Runtime Options** (Main window)
- Change the default radix for the current simulation using the radix command.
- Change the default radix permanently by editing the DefaultRadix variable in the *modelsim.ini* file.

Saving the Window Format

By default all Wave and List window information is forgotten once you close the windows. If you want to restore the windows to a previously configured layout, you must save a window format file. Follow these steps:

1. Add the objects you want to the Wave or List window.
2. Edit and format the objects to create the view you want.
3. Save the format to a file by selecting **File > Save > Format**.

To use the format file, start with a blank Wave or List window and run the DO file in one of two ways:

- Invoke the do command from the command line:

```
VSIM> do <my_format_file>
```

- Select **File** > **Load**.

Note
Window format files are design-specific. Use them only with the design you were simulating when they were created.

Printing and Saving Waveforms in the Wave window

You can print the waveform display or save it as an encapsulated postscript (EPS) file.

Saving a .eps Waveform File and Printing in UNIX

Select **File** > **Print Postscript** (Wave window) to print all or part of the waveform in the current Wave window in UNIX, or save the waveform as a .eps file on any platform (see also the write wave command).

Printing from the Wave Window on Windows Platforms

Select **File** > **Print** (Wave window) to print all or part of the waveform in the current Wave window, or save the waveform as a printer file (a Postscript file for Postscript printers).

Printer Page Setup

Select **File** > **Page setup** or click the Setup button in the Write Postscript or Print dialog box to define how the printed page will appear.

Saving List Window Data to a File

Select **File** > **Write List** in the List window to save the data in one of these formats:

- **Tabular** — writes a text file that looks like the window listing

```
ns    delta      /a        /b      /cin     /sum    /cout
0       +0        X         X        U        X        U
0       +1        0         1        0        X        U
2       +0        0         1        0        X        U
```

- **Events** — writes a text file containing transitions during simulation

```
@0 +0
/a X
/b X
/cin U
/sum X
/cout U
@0 +1
/a 0
/b 1
/cin 0
```

- **TSSI** — writes a file in standard TSSI format; see also, the write tssi command.

```
0 00000000000000010?????????
2 00000000000000010???????1?
3 00000000000000010??????010
4 00000000000000010000000010
100 00000001000000010000000010
```

You can also save List window output using the write list command.

Combining Objects into Buses

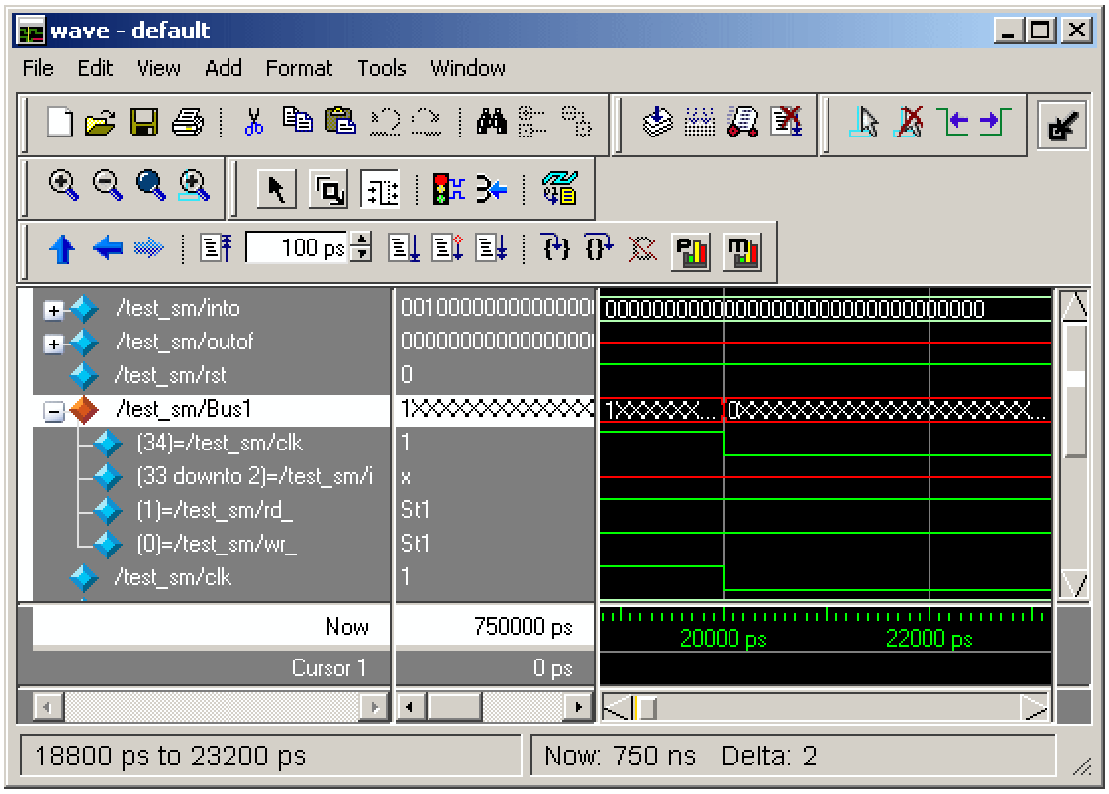
You can combine signals in the Wave or List window into buses. A bus is a collection of signals concatenated in a specific order to create a new virtual signal with a specific value. A virtual compare signal (the result of a comparison simulation) is not supported for combination with any other signal.

To combine signals into a bus, use one of the following methods:

- Select two or more signals in the Wave or List window and then choose **Tools > Combine Signals** from the menu bar. A virtual signal that is the result of a comparison simulation is not supported for combining with any other signal.
- Use the virtual signal command at the Main window command prompt.

In the illustration below, three signals have been combined to form a new bus called "Bus1". Note that the component signals are listed in the order in which they were selected in the Wave window. Also note that the value of the bus is made up of the values of its component signals, arranged in a specific order.

Figure 8-22. Signals Combined to Create Virtual Bus



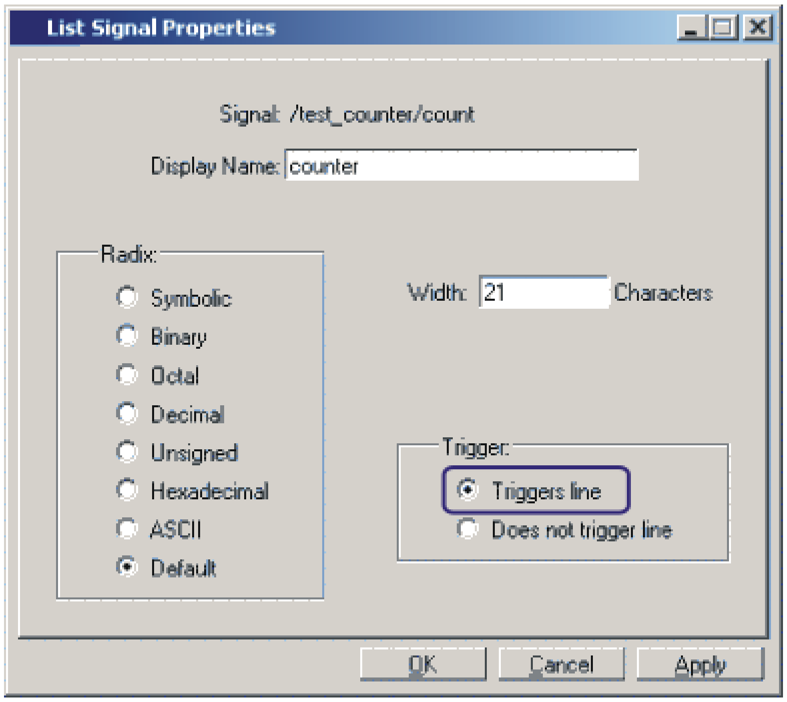
```
/test_sm/into                   0010000000000000  000000000000000000000000000000000
/test_sm/outof                  0000000000000000
/test_sm/rst                    0
/test_sm/Bus1                   1XXXXXXXXXXXXX    1XXXXXX...  0XXXXXXXXXXXXXXXXXXXXXX...
  (34)=/test_sm/clk             1
  (33 downto 2)=/test_sm/i      x
  (1)=/test_sm/rd_              St1
  (0)=/test_sm/wr_              St1
/test_sm/clk                    1
                          Now   750000 ps         20000 ps        22000 ps
                     Cursor 1   0 ps
18800 ps to 23200 ps            Now: 750 ns   Delta: 2
```

Configuring New Line Triggering in the List Window

New line triggering refers to what events cause a new line of data to be added to the List window. By default ModelSim adds a new line for any signal change including deltas within a single unit of time resolution.

You can set new line triggering on a signal-by-signal basis or for the whole simulation. To set for a single signal, select **View > Signal Properties** from the List window menu bar (when the window is undocked) and select the **Triggers line** setting. Individual signal settings override global settings.

Figure 8-23. Line Triggering in the List Window


List Signal Properties

Signal: /test_counter/count

Display Name: counter

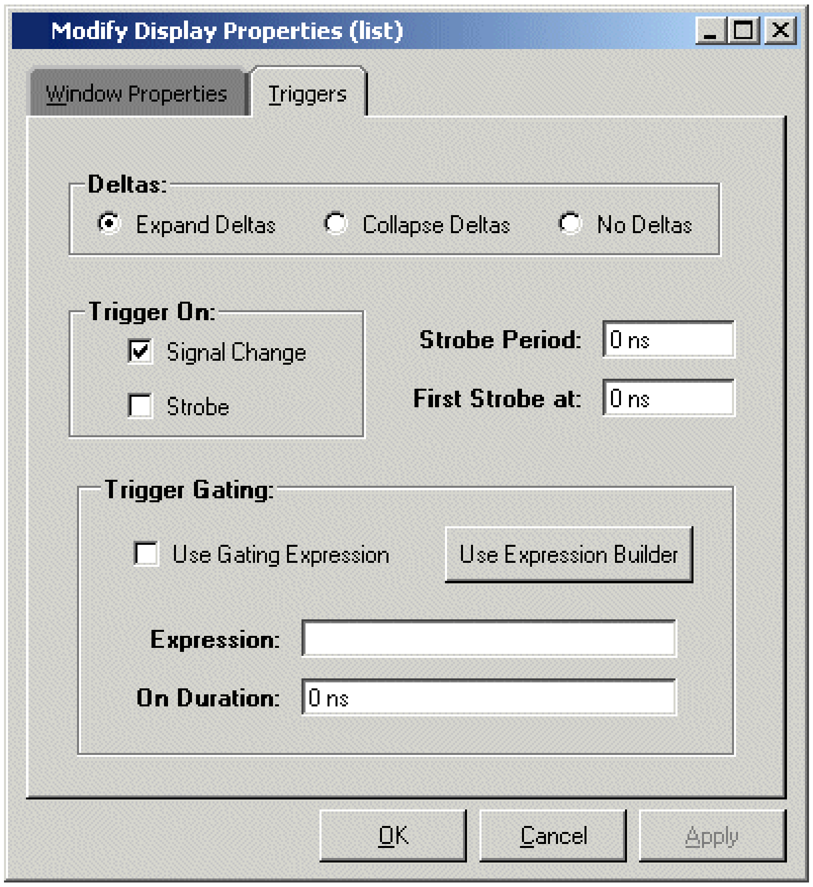
Radix: Symbolic / Binary / Octal / Decimal / Unsigned / Hexadecimal / ASCII / [✓ Default]

Width: 21 Characters

Trigger: [✓ Triggers line] / Does not trigger line

To modify new line triggering for the whole simulation, select **Tools > List Preferences** from the List window menu bar (when the window is undocked), or use the configure command. When you select **Tools > List Preferences**, the Modify Display Properties dialog appears:

Figure 8-24. Setting Trigger Properties



Deltas: [✓ Expand Deltas] Collapse Deltas No Deltas

Trigger On: [✓ Signal Change] Strobe

Strobe Period: 0 ns

First Strobe at: 0 ns

Trigger Gating: Use Gating Expression

Expression:

On Duration: 0 ns

The following table summaries the triggering options:

Table 8-5. Triggering Options

Option	Description
Deltas	Choose between displaying all deltas (Expand Deltas), displaying the value at the final delta (Collapse Delta). You can also hide the delta column all together (No Delta), however this will display the value at the final delta.
Strobe trigger	Specify an interval at which you want to trigger data display
Trigger gating	Use a gating expression to control triggering; see Using Gating Expressions to Control Triggering for more details

Using Gating Expressions to Control Triggering

Trigger gating controls the display of data based on an expression. Triggering is enabled once the gating expression evaluates to true. This setup behaves much like a hardware signal analyzer that starts recording data on a specified setup of address bits and clock edges.

Here are some points about gating expressions:

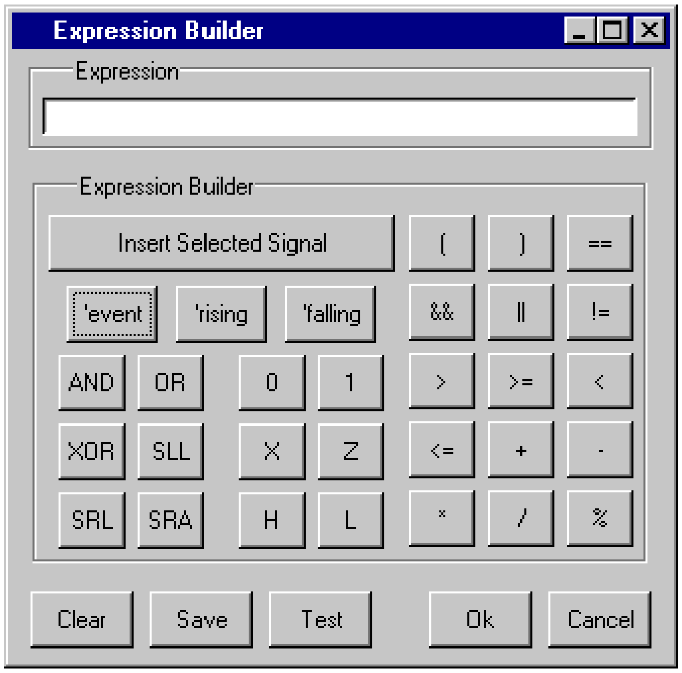
- Gating expressions affect the display of data but not acquisition of the data.
- The expression is evaluated when the List window would normally have displayed a row of data (given the other trigger settings).
- The duration determines for how long triggering stays enabled after the gating expression returns to false (0). The default of 0 duration will enable triggering only while the expression is true (1). The duration is expressed in x number of default timescale units.
- Gating is level-sensitive rather than edge-triggered.

Trigger Gating Example Using the Expression Builder

This example shows how to create a gating expression with the ModelSim Expression Builder. Here is the procedure:

1. Select **Tools > Window Preferences** from the List window menu bar (when the window is undocked) and select the Triggers tab.
2. Click the **Use Expression Builder** button.

Figure 8-25. Trigger Gating Using Expression Builder



3. Select the signal in the List window that you want to be the enable signal by clicking on its name in the header area of the List window.
4. Click **Insert Selected Signal** and then **'rising** in the Expression Builder.
5. Click OK to close the Expression Builder.

   You should see the name of the signal plus "'rising" added to the Expression entry box of the Modify Display Properties dialog box.
6. Click **OK** to close the dialog.

If you already have simulation data in the List window, the display should immediately switch to showing only those cycles for which the gating signal is rising. If that isn't quite what you want, you can go back to the expression builder and play with it until you get it the way you want it.

If you want the enable signal to work like a "One-Shot" that would display all values for the next, say 10 ns, after the rising edge of enable, then set the **On Duration** value to **10 ns**.

Trigger Gating Example Using Commands

The following commands show the gating portion of a trigger configuration statement:

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

See the configure command for more details.

Sampling Signals at a Clock Change

You easily can sample signals at a clock change using the add list command with the **-notrigger** argument. The **-notrigger** argument disables triggering the display on the specified signals. For example:

```
add list clk -notrigger a b c
```

When you run the simulation, List window entries for *clk*, *a*, *b*, and *c* appear only when *clk* changes.

If you want to display on rising edges only, you have two options:

1. Turn off the List window triggering on the clock signal, and then define a repeating strobe for the List window.
2. Define a "gating expression" for the List window that requires the clock to be in a specified state. See above.

Miscellaneous Tasks

Examining Waveform Values

You can use your mouse to display a dialog that shows the value of a waveform at a particular time. You can do this two ways:

- Rest your mouse pointer on a waveform. After a short delay, a dialog will pop-up that displays the value for the time at which your mouse pointer is positioned. If you'd prefer that this popup not display, it can be toggled off in the display properties. See Setting Wave Window Display Preferences.
- Right-click a waveform and select **Examine**. A dialog displays the value for the time at which you clicked your mouse. This method works in the List window as well.

Displaying Drivers of the Selected Waveform

You can automatically display in the Dataflow window the drivers of a signal selected in the Wave window. You can do this three ways:

- Select a waveform and click the Show Drivers button on the toolbar.
- Select a waveform and select Show Drivers from the shortcut menu
- Double-click a waveform edge (you can enable/disable this option in the display properties dialog; see Setting Wave Window Display Preferences)

This operation opens the Dataflow window and displays the drivers of the signal selected in the Wave window. The Wave pane in the Dataflow window also opens to show the selected signal with a cursor at the selected time. The Dataflow window shows the signal(s) values at the current cursor position.

Sorting a Group of Objects in the Wave Window

Select **View** > **Sort** to sort the objects in the pathname and values panes.

Creating and managing breakpoints

ModelSim supports both signal (i.e., when conditions) and file-line breakpoints. Breakpoints can be set from multiple locations in the GUI or from the command line.

Signal breakpoints

Signal breakpoints (when conditions) instruct ModelSim to perform actions when the specified conditions are met. For example, you can break on a signal value or at a specific simulator time (see the when command for additional details). When a breakpoint is hit, a message in the Main window transcript identifies the signal that caused the breakpoint.

Setting signal breakpoints from the command line

You use the when command to set a signal breakpoint from the VSIM> prompt.

Setting signal breakpoints from the GUI

Signal breakpoints are most easily set in the Objects Pane and the Wave Window Overview. Right-click a signal and select **Insert Breakpoint** from the context menu. A breakpoint is set on that signal and will be listed in the **Breakpoints** dialog.

File-line breakpoints

File-line breakpoints are set on executable lines in your source files. When the line is hit, the simulator stops and the Source window opens to show the line with the breakpoint. You can change this behavior by editing the PrefSource(OpenOnBreak) variable. See Simulator GUI Preferences for details on setting preference variables.

Setting file-line breakpoints from the command line

You use the bp command to set a file-line breakpoint from the VSIM> prompt.

Setting file-line breakpoints from the GUI

File-line breakpoints are most easily set using your mouse in the Source Window. Click on a blue line number at the left side of the Source window, and a red diamond denoting a breakpoint will appear. The breakpoints are toggles – click once to create the colored diamond; click again to disable or enable the breakpoint. To delete the breakpoint completely, click the red diamond with your right mouse button, and select **Remove Breakpoint**.

Chapter 9
Tracing Signals with the Dataflow Window

This chapter discusses how to use the Dataflow window for tracing signal values and browsing the physical connectivity of your design.

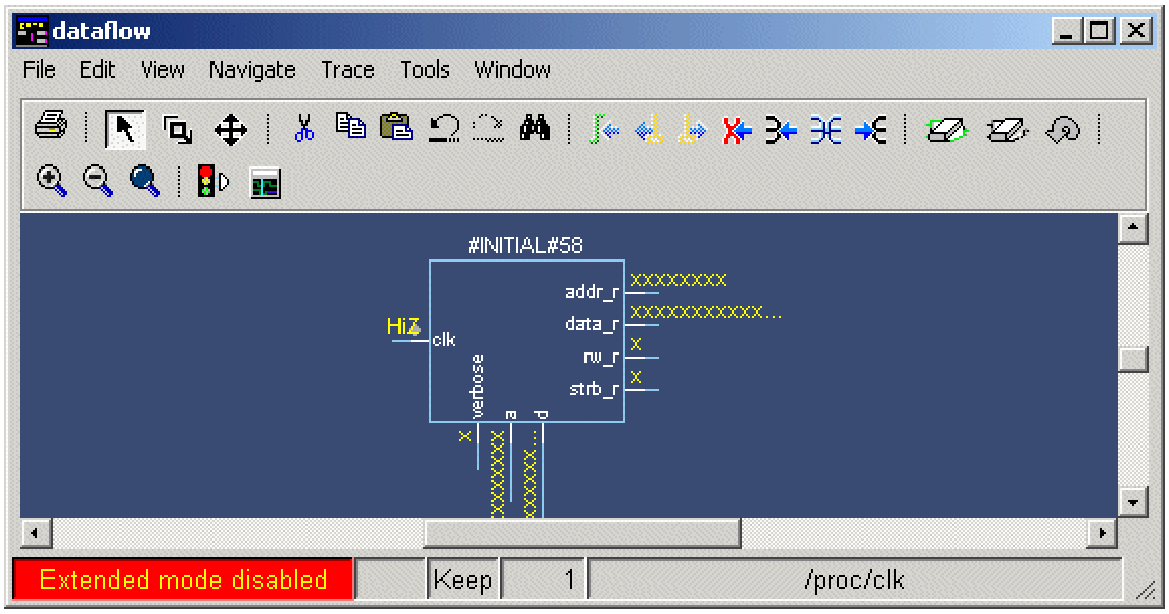
Dataflow Window Overview

The Dataflow window allows you to explore the "physical" connectivity of your design.

Note

OEM versions of ModelSim have limited Dataflow functionality. Many of the features described below will operate differently. The window will show only one process and its attached signals or one signal and its attached processes, as displayed in Figure 9-1.

Figure 9-1. The Dataflow Window (undocked)



Objects You Can View in the Dataflow Window

The Dataflow window displays:

- processes

- signals, nets, and registers

The window has built-in mappings for all Verilog primitive gates (i.e., AND, OR, etc.). For components other than Verilog primitives, you can define a mapping between processes and built-in symbols. See Symbol Mapping for details.

Adding Objects to the Window

You can use any of the following methods to add objects to the Dataflow window:

- drag and drop objects from other windows
- use the Navigate menu options in the Dataflow window
- use the add dataflow command
- double-click any waveform in the Wave window display

The **Navigate** menu offers four commands that will add objects to the window. The commands include:

- **View region** — clear the window and display all signals from the current region
- **Add region** — display all signals from the current region without first clearing window
- **View all nets** — clear the window and display all signals from the entire design
- **Add ports** — add port symbols to the port signals in the current region

When you view regions or entire nets, the window initially displays only the drivers of the added objects in order to reduce clutter. You can easily view readers by selecting an object and invoking **Navigate** > **Expand net to readers**.

A small circle above an input signal on a block denotes a trigger signal that is on the process' sensitivity list.

Links to Other Windows

The Dataflow window has links to other windows as described below:

Table 9-1. Dataflow Window Links to Other Windows and Panes

Window	Link
Main Window	select a signal or process in the Dataflow window, and the structure tab updates if that object is in a different design unit
Active Processes Pane	select a process in either window, and that process is highlighted in the other

Table 9-1. Dataflow Window Links to Other Windows and Panes (cont.)

Window	Link
Objects Pane	select a design object in either window, and that object is highlighted in the other
Wave Window	• trace through the design in the Dataflow window, and the associated signals are added to the Wave window • move a cursor in the Wave window, and the values update in the Dataflow window
Source Window	select an object in the Dataflow window, and the Source window updates if that object is in a different source file

Exploring the Connectivity of the Design

A primary use of the Dataflow window is exploring the "physical" connectivity of your design. One way of doing this is by expanding the view from process to process. This allows you to see the drivers/receivers of a particular signal, net, or register.

You can expand the view of your design using menu commands or your mouse. To expand with the mouse, simply double click a signal, register, or process. Depending on the specific object you click, the view will expand to show the driving process and interconnect, the reading process and interconnect, or both.

Alternatively, you can select a signal, register, or net, and use one of the toolbar buttons or menu commands described in Table 9-2:

Table 9-2. Icon and Menu Selections for Exploring Design Connectivity

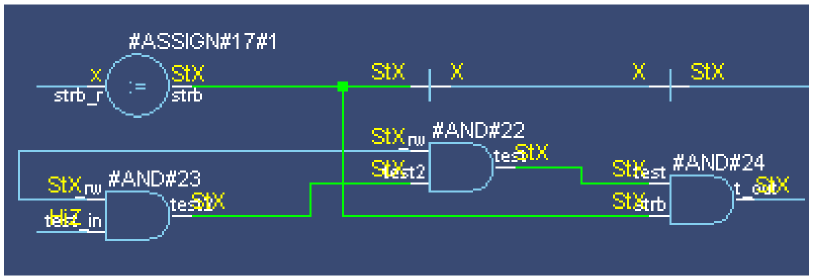
	Expand net to all drivers display driver(s) of the selected signal, net, or register	Navigate > Expand net to drivers
	Expand net to all drivers and readers display driver(s) and reader(s) of the selected signal, net, or register	Navigate > Expand net
	Expand net to all readers display reader(s) of the selected signal, net, or register	Navigate > Expand net to readers

As you expand the view, note that the "layout" of the design may adjust to best show the connectivity. For example, the location of an input signal may shift from the bottom to the top of a process.

Tracking Your Path Through the Design

You can quickly traverse through many components in your design. To help mark your path, the objects that you have expanded are highlighted in green.

Figure 9-2. Green Highlighting Shows Your Path Through the Design



You can clear this highlighting using the **Edit > Erase highlight** command or by clicking the Erase highlight icon in the toolbar.

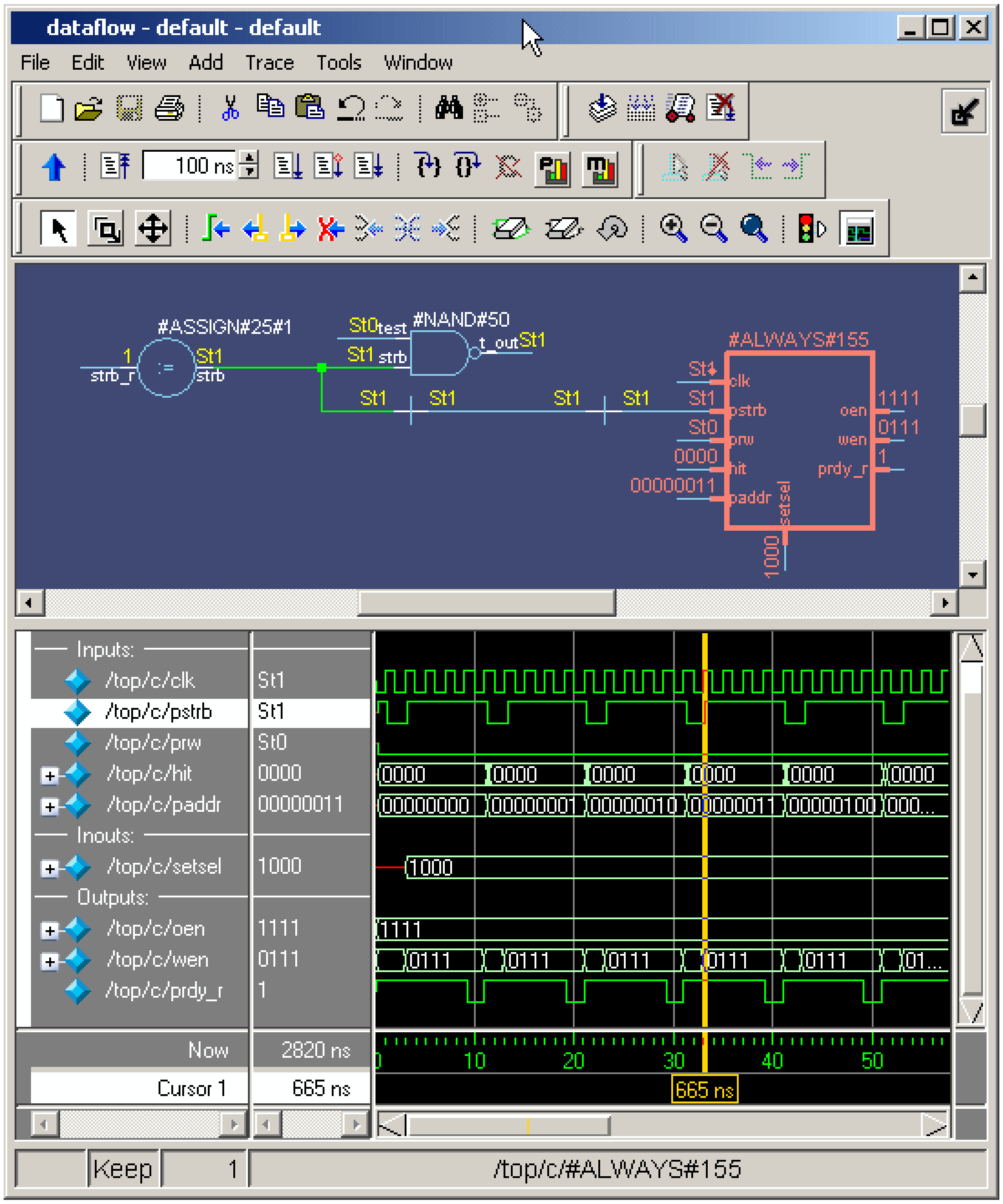
The Embedded Wave Viewer

Another way of exploring your design is to use the Dataflow window's embedded wave viewer. This viewer closely resembles, in appearance and operation, the stand-alone Wave window (see Waveform Analysis for more information).

The wave viewer is opened using the **View > Show Wave** command or by clicking the Show Wave icon.

One common scenario is to place signals in the wave viewer and the Dataflow panes, run the design for some amount of time, and then use time cursors to investigate value changes. In other words, as you place and move cursors in the wave viewer pane (see Measuring Time with Cursors in the Wave Window for details), the signal values update in the Dataflow pane.

Figure 9-3. Wave Viewer Displays Inputs and Outputs of Selected Process



Another scenario is to select a process in the Dataflow pane, which automatically adds to the wave viewer pane all signals attached to the process.

See Tracing Events (Causality) for another example of using the embedded wave viewer.

Zooming and Panning

The Dataflow window offers several tools for zooming and panning the display.

These zoom buttons are available on the toolbar:

Zoom In
zoom in by a factor of two from the current view

Zoom Out
zoom out by a factor of two from current view

Zoom Full
zoom out to view the entire schematic

To zoom with the mouse, you can either use the middle mouse button or enter Zoom Mode by selecting **View** > **Zoom** and then use the left mouse button.

Four zoom options are possible by clicking and dragging in different directions:

- Down-Right: Zoom Area (In)
- Up-Right: Zoom Out (zoom amount is displayed at the mouse cursor)
- Down-Left: Zoom Selected
- Up-Left: Zoom Full

The zoom amount is displayed at the mouse cursor. A zoom operation must be more than 10 pixels to activate.

Panning with the Mouse

You can pan with the mouse in two ways: 1) enter Pan Mode by selecting **View** > **Pan** and then drag with the left mouse button to move the design; 2) hold down the <Ctrl> key and drag with the middle mouse button to move the design.

Tracing Events (Causality)

One of the most useful features of the Dataflow window is tracing an event to see the cause of an unexpected output. This feature uses the Dataflow window's embedded wave viewer (see The Embedded Wave Viewer for more details).

In short you identify an output of interest in the Dataflow pane and then use time cursors in the wave viewer pane to identify events that contribute to the output.

The process for tracing events is as follows:

1. Log all signals before starting the simulation (add log -r /*).
2. After running a simulation for some period of time, open the Dataflow window and the wave viewer pane.
3. Add a process or signal of interest into the Dataflow window (if adding a signal, find its driving process). Select the process and all signals attached to the selected process will appear in the wave viewer pane.

4. Place a time cursor on an edge of interest; the edge should be on a signal that is an output of the process.
5. Select **Trace > Trace input net to event**.

   A second cursor is added at the most recent input event.
6. Keep selecting **Trace** > **Trace next event** until you've reached an input event of interest. Note that the signals with the events are selected in the wave pane.
7. Now select **Trace** > **Trace Set**.

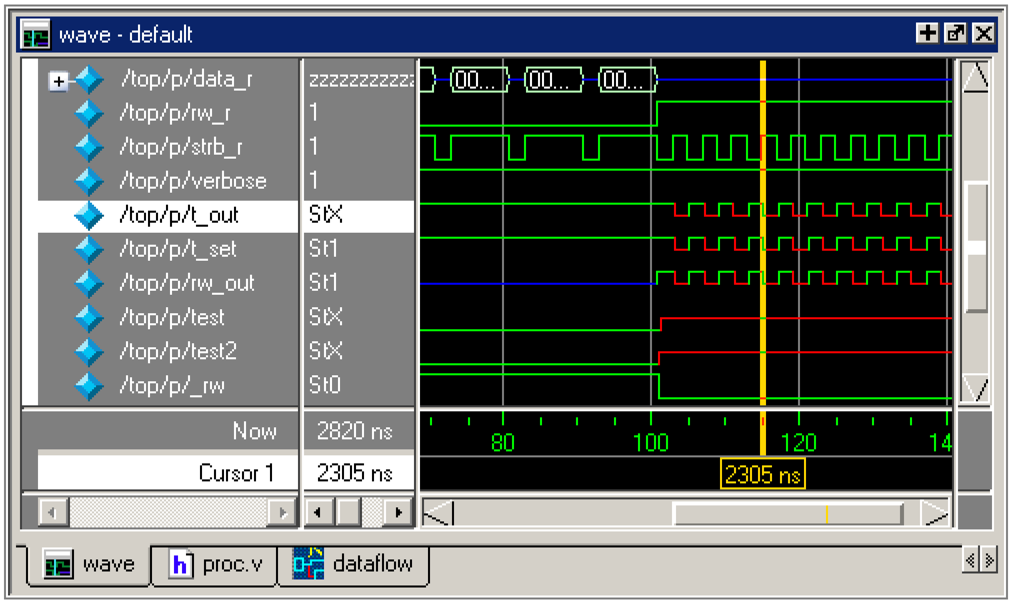
   The Dataflow display "jumps" to the source of the selected input event(s). The operation follows all signals selected in the wave viewer pane. You can change which signals are followed by changing the selection.
8. To continue tracing, go back to step 5 and repeat.

If you want to start over at the originally selected output, select **Trace** > **Trace event reset**.

Tracing the Source of an Unknown State (StX)

Another useful Dataflow window debugging tool is the ability to trace an unknown state (StX) back to its source. Unknown values are indicated by red lines in the Wave window (Figure 9-4) and in the wave viewer of the Dataflow window.

Figure 9-4. Unknown States Shown as Red Lines in Wave Window



The procedure for tracing to the source of an unknown state in the Dataflow window is as follows:

1. Load your design.

2. Log all signals in the design or any signals that may possibly contribute to the unknown value (**log -r /*** will log all signals in the design).
3. Add signals to the Wave window or wave viewer pane, and run your design the desired length of time.
4. Put a Wave window cursor on the time at which the signal value is unknown (StX). In Figure 9-4, Cursor 1 at time 2305 shows an unknown state on signal *t_out*.
5. Add the signal of interest to the Dataflow window by doing one of the following:
   - double-clicking on the signal's waveform in the Wave window,
   - right-clicking the signal in the Objects window and selecting **Add to Dataflow > Selected Signals** from the popup menu,
   - selecting the signal in the Objects window and selecting **Add > Dataflow > Selected Signals** from the menu bar.
6. In the Dataflow window, make sure the signal of interest is selected.
7. Trace to the source of the unknown by doing one of the following:
   - If the Dataflow window is docked, select **Tools > Trace > TraceX**, **Tools > Trace > TraceX Delay, Tools > Trace > ChaseX**, or **Tools > Trace > ChaseX Delay**.
   - If the Dataflow window is undocked, select **Trace > TraceX**, **Trace > TraceX Delay, Trace > ChaseX**, or **Trace > ChaseX Delay**.

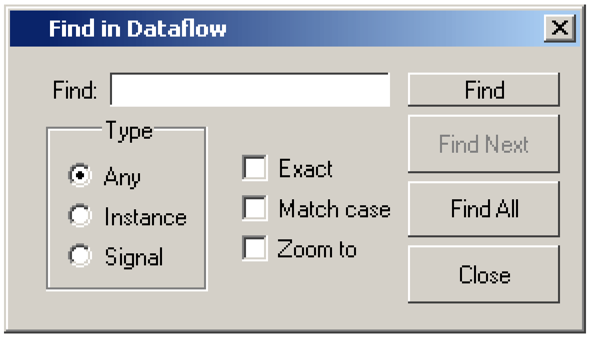
     These commands behave as follows:

     - **TraceX / TraceX Delay**— Steps back to the last driver of an X value. **TraceX Delay** works similarly but it steps back in time to the last driver of an X value. **TraceX** should be used for RTL designs; **TraceX Delay** should be used for gate-level netlists with back annotated delays.
     - **ChaseX / ChaseX Delay** — "Jumps" through a design from output to input, following X values. **ChaseX Delay** acts the same as **ChaseX** but also moves backwards in time to the point where the output value transitions to X. **ChaseX** should be used for RTL designs; **ChaseX Delay** should be used for gate-level netlists with back annotated delays.

Finding Objects by Name in the Dataflow Window

Select **Edit > Find** from the menu bar, or click the Find icon in the toolbar, to search for signal, net, or register names or an instance of a component. This opens the Find in Dataflow dialog (Figure 9-5).

Figure 9-5. Find in Dataflow Dialog



With the Find in Dataflow dialog you can limit the search by type to instances or signals. You select Exact to find an item that exactly matches the entry you've typed in the Find field. The Match case selection will enforce case-sensitive matching of your entry. And the Zoom to selection will zoom in to the item in Find field.

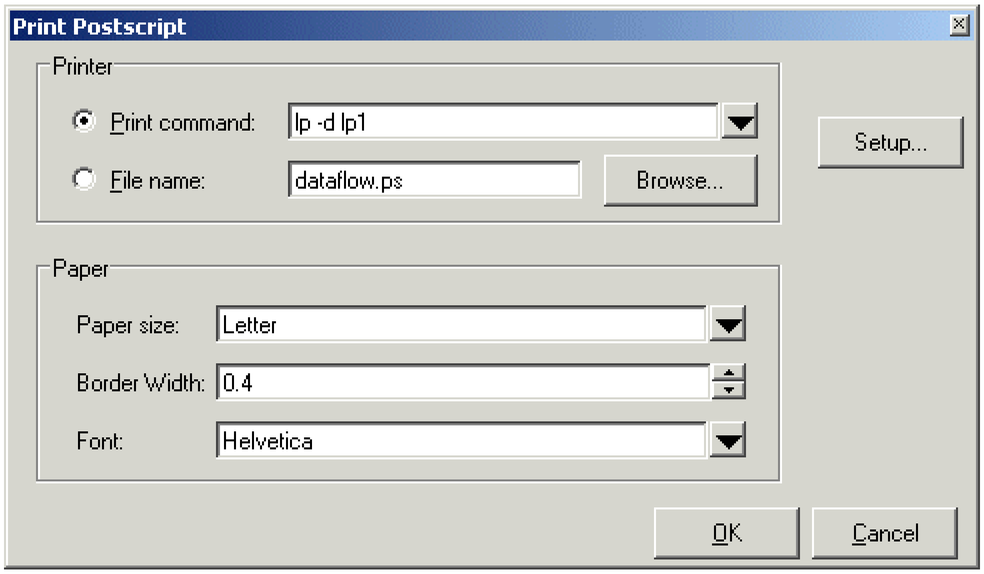
The Find All button allows you to find and highlight all occurrences of the item in the Find field. If Zoom to is checked, the view will change such that all selected items are viewable. If Zoom to is not selected, then no change is made to zoom or scroll state.

Printing and Saving the Display

Saving a .eps File and Printing the Dataflow Display from UNIX

Select **File** > **Print Postscript** to setup and print the Dataflow display in UNIX, or save the waveform as a .eps file on any platform.

Figure 9-6. The Print Postscript Dialog



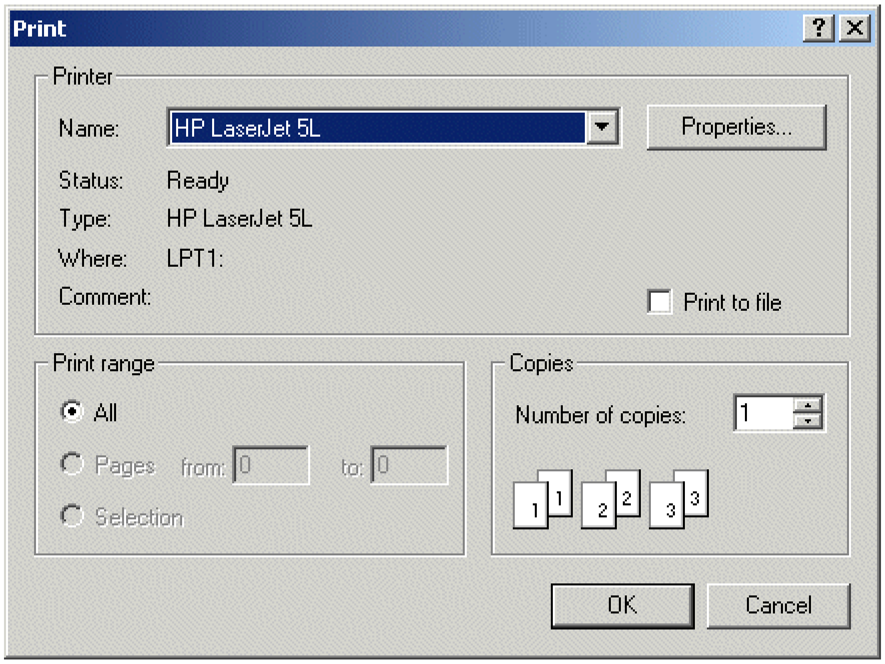
- [✓ Print command: lp -d lp1]
- File name: dataflow.ps

Paper size: Letter
Border Width: 0.4
Font: Helvetica

Printing from the Dataflow Display on Windows Platforms

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

Figure 9-7. The Print Dialog



Name: HP LaserJet 5L
Status: Ready
Type: HP LaserJet 5L
Where: LPT1:
Comment:
Print to file [ ]

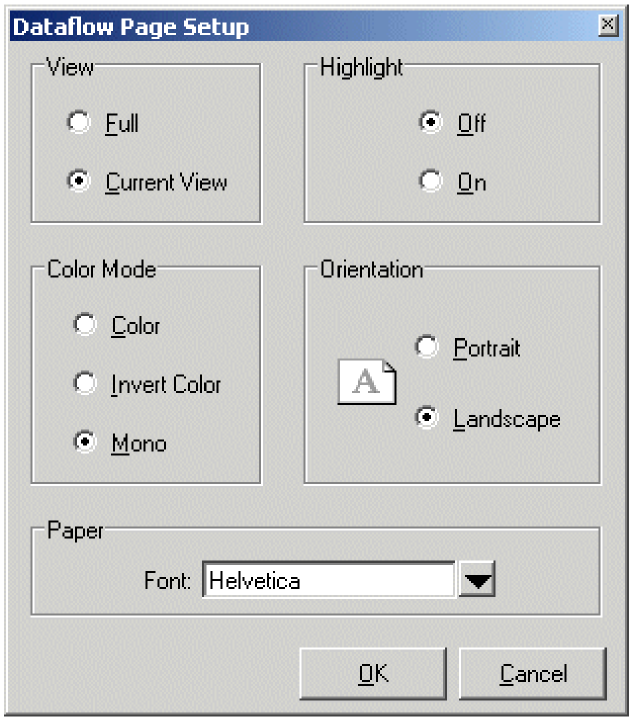
Print range: [✓ All] Pages from: 0 to: 0 Selection

Number of copies: 1

Configuring Page Setup

Clicking the Setup button in the Print Postscript or Print dialog box allows you to configure page view, highlight, color mode, orientation, and paper options (this is the same dialog that opens via **File > Page setup**).

Figure 9-8. The Dataflow Page Setup Dialog



Dataflow Page Setup

View: Full [✓ Current View]

Highlight: [✓ Off] On

Color Mode: Color Invert Color [✓ Mono]

Orientation: Portrait [✓ Landscape]

Paper — Font: Helvetica

Symbol Mapping

The Dataflow window has built-in mappings for all Verilog primitive gates (i.e., AND, OR, etc.). For components other than Verilog primitives, you can define a mapping between processes and built-in symbols. This is done through a file containing name pairs, one per line, where the first name is the concatenation of the design unit and process names, (DUname.Processname), and the second name is the name of a built-in symbol. For example:

```
xorg(only).p1 XOR
org(only).p1 OR
andg(only).p1 AND
```

Entities and modules are mapped the same way:

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

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

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

User-defined symbols

You can also define your own symbols using an ASCII symbol library file format for defining symbol shapes. This capability is delivered via Concept Engineering's Nlview^{TM} widget Symlib format.

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

```
symbol adder(structural) * DEF \
   port a in -loc -12 -15 0 -15 \
   pinattrdsp @name -cl 2 -15 8 \
   port b in -loc -12 15 0 15 \
   pinattrdsp @name -cl 2 15 8 \
   port cin in -loc 20 -40 20 -28 \
   pinattrdsp @name -uc 19 -26 8 \
   port cout out -loc 20 40 20 28 \
pinattrdsp @name -lc 19 26 8 \
   port sum out -loc 63 0 51 0 \
   pinattrdsp @name -cr 49 0 8 \
   path 10 0 0 7 \
   path 0 7 0 35 \
   path 0 35 51 17 \
   path 51 17 51 -17 \
   path 51 -17 0 -35 \
   path 0 -35 0 -7 \
   path 0 -7 10 0
```

Port mapping is done by name for these symbols, so the port names in the symbol definition must match the port names of the Entity|Module|Process (in the case of the process, it's the signal names that the process reads/writes).

Note

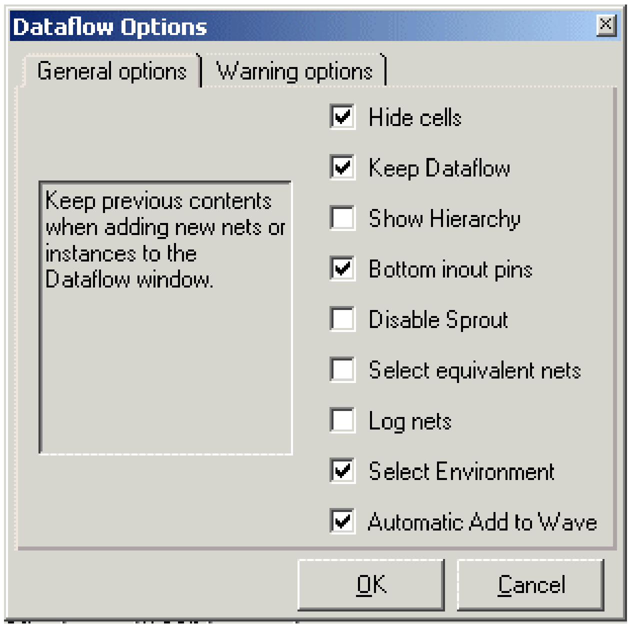
When you create or modify a symlib file, you must generate a file index. This index is how the Nlview widget finds and extracts symbols from the file. To generate the index, select **Tools** > **Create symlib index** (Dataflow window) and specify the symlib file. The file will be rewritten with a correct, up-to-date index.

Configuring Window Options

You can configure several options that determine how the Dataflow window behaves. The settings affect only the current session.

Select **Tools** > **Options** to open the Dataflow Options dialog box.

Figure 9-9. Configuring Dataflow Options



Keep previous contents when adding new nets or instances to the Dataflow window.

- [✓] Hide cells
- [✓] Keep Dataflow
- [ ] Show Hierarchy
- [✓] Bottom inout pins
- [ ] Disable Sprout
- [ ] Select equivalent nets
- [ ] Log nets
- [✓] Select Environment
- [✓] Automatic Add to Wave

Chapter 10
Signal Spy

The Verilog language allows access to any signal from any other hierarchical block without having to route it via the interface. This means you can use hierarchical notation to either assign or determine the value of a signal in the design hierarchy from a testbench. This capability fails when a Verilog testbench attempts to reference a signal in a VHDL block or reference a signal in a Verilog block through a VHDL level of hierarchy.

This limitation exists because VHDL does not allow hierarchical notation. In order to reference internal hierarchical signals, you have to resort to defining signals in a global package and then utilize those signals in the hierarchical blocks in question. But, this requires that you keep making changes depending on the signals that you want to reference.

The Signal Spy procedures and system tasks overcome the aforementioned limitations. They allow you to monitor (spy), drive, force, or release hierarchical objects in a VHDL or mixed design.

The VHDL procedures are provided via the Util Package within the *modelsim_lib* library. To access the procedures you would add lines like the following to your VHDL code:

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

The Verilog tasks are available as built-in System Tasks and Functions. The table below shows the VHDL procedures and their corresponding Verilog system tasks.

Table 10-1. Signal Spy: Mapping VHDL Procedures to Verilog System Tasks

VHDL procedures	Verilog system tasks
disable_signal_spy	$disable_signal_spy
enable_signal_spy	$enable_signal_spy
init_signal_driver	$init_signal_driver
init_signal_spy	$init_signal_spy
signal_force	$signal_force
signal_release	$signal_release

Designed for Testbenches

Signal Spy limits the portability of your code. HDL code with Signal Spy procedures or tasks works only in ModelSim, not other simulators. We therefore recommend using Signal Spy only in testbenches, where portability is less of a concern, and the need for such a tool is more applicable.

disable_signal_spy

The disable_signal_spy() procedure disables the associated init_signal_spy. The association between the disable_signal_spy call and the init_signal_spy call is based on specifying the same src_object and dest_object arguments to both functions. The disable_signal_spy call can only affect init_signal_spy calls that had their control_state argument set to "0" or "1".

Syntax

disable_signal_spy(<src_object>, <dest_object>, <verbose>)

Returns

Nothing

Arguments

- src_object

  Required string. A full hierarchical path (or relative downward path with reference to the calling block) to a VHDL signal or Verilog register/net. This path should match the path that was specified in the init_signal_spy call that you wish to disable.

- dest_object

  Required string. A full hierarchical path (or relative downward path with reference to the calling block) to a VHDL signal or Verilog register/net. This path should match the path that was specified in the init_signal_spy call that you wish to disable.

- verbose

  Optional integer. Possible values are 0 or 1. Specifies whether you want a message reported in the transcript stating that a disable occurred and the simulation time that it occurred. Default is 0, no message

Related procedures

init_signal_spy, enable_signal_spy

Example

See init_signal_spy Example

enable_signal_spy

The enable_signal_spy() procedure enables the associated init_signal_spy. The association between the enable_signal_spy call and the init_signal_spy call is based on specifying the same src_object and dest_object arguments to both functions. The enable_signal_spy call can only affect init_signal_spy calls that had their control_state argument set to "0" or "1".

Syntax

enable_signal_spy(<src_object>, <dest_object>, <verbose>)

Returns

Nothing

Arguments

- src_object

  Required string. A full hierarchical path (or relative downward path with reference to the calling block) to a VHDL signal or Verilog register/net. This path should match the path that was specified in the init_signal_spy call that you wish to enable.

- dest_object

  Required string. A full hierarchical path (or relative downward path with reference to the calling block) to a VHDL signal or Verilog register/net. This path should match the path that was specified in the init_signal_spy call that you wish to enable.

- verbose

  Optional integer. Possible values are 0 or 1. Specifies whether you want a message reported in the transcript stating that an enable occurred and the simulation time that it occurred. Default is 0, no message

Related procedures

init_signal_spy, disable_signal_spy

Example

See init_signal_spy Example

init_signal_driver

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

The init_signal_driver procedure drives the value onto the destination signal just as if the signals were directly connected in the HDL code. Any existing or subsequent drive or force of the destination signal, by some other means, will be considered with the init_signal_driver value in the resolution of the signal.

Call only once

The init_signal_driver procedure creates a persistent relationship between the source and destination signals. Hence, you need to call init_signal_driver only once for a particular pair of signals. Once init_signal_driver is called, any change on the source signal will be driven on the destination signal until the end of the simulation.

Thus, we recommend that you place all init_signal_driver calls in a VHDL process. You need to code the VHDL process correctly so that it is executed only once. The VHDL process should not be sensitive to any signals and should contain only init_signal_driver calls and a simple wait statement. The process will execute once and then wait forever. See the example below.

Syntax

init_signal_driver(<src_object>, <dest_object>, <delay>, <delay_type>, <verbose>)

Returns

Nothing

Arguments

- src_object

  Required string. A full hierarchical path (or relative downward path with reference to the calling block) to a VHDL signal or Verilog net. Use the path separator to which your simulation is set (i.e., "/" or "."). A full hierarchical path must begin with a "/" or ".". The path must be contained within double quotes.

- dest_object

  Required string. A full hierarchical path (or relative downward path with reference to the calling block) to an existing VHDL signal or Verilog net. Use the path separator to which your simulation is set (i.e., "/" or "."). A full hierarchical path must begin with a "/" or ".". The path must be contained within double quotes.

- delay

  Optional time value. Specifies a delay relative to the time at which the src_object changes. The delay can be an inertial or transport delay. If no delay is specified, then a delay of zero is assumed.

- delay_type

  Optional del_mode. Specifies the type of delay that will be applied. The value must be either mti_inertial or mti_transport. The default is mti_inertial.

- verbose

  Optional integer. Possible values are 0 or 1. Specifies whether you want a message reported in the Transcript stating that the src_object is driving the dest_object. Default is 0, no message.

Related procedures

init_signal_spy, signal_force, signal_release

Limitations

- When driving a Verilog net, the only *delay_type* allowed is inertial. If you set the delay type to *mti_transport*, the setting will be ignored and the delay type will be *mti_inertial*.
- Any delays that are set to a value less than the simulator resolution will be rounded to the nearest resolution unit; no special warning will be issued.

init_signal_driver Example

This example creates a local clock (*clk0*) and connects it to two clocks within the design hierarchy. The *.../blk1/clk* will match local *clk0* and a message will be displayed. The *open* entries allow the default delay and delay_type while setting the verbose parameter to a 1. The *.../blk2/clk* will match the local *clk0* but be delayed by 100 ps.

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

entity testbench is
end;

architecture only of testbench is
  signal clk0 : std_logic;
begin
  gen_clk0 : process
  begin
    clk0 <= '1' after 0 ps, '0' after 20 ps;
    wait for 40 ps;
  end process gen_clk0;

  drive_sig_process : process
  begin
    init_signal_driver("clk0", "/testbench/uut/blk1/clk", open, open, 1);
    init_signal_driver("clk0", "/testbench/uut/blk2/clk", 100 ps,
                        mti_transport);
    wait;
  end process drive_sig_process;
  ...
end;
```

init_signal_spy

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

The init_signal_spy procedure only sets the value onto the destination signal and does not drive or force the value. Any existing or subsequent drive or force of the destination signal, by some other means, will override the value that was set by init_signal_spy.

Call only once

The init_signal_spy procedure creates a persistent relationship between the source and destination signals. Hence, you need to call init_signal_spy once for a particular pair of signals. Once init_signal_spy is called, any change on the source signal will mirror on the destination signal until the end of the simulation unless the control_state is set.

The control_state determines whether the mirroring of values can be enabled/disabled and what the initial state is. Subsequent control of whether the mirroring of values is enabled/disabled is handled by the enable_signal_spy and disable_signal_spy calls.

We recommend that you place all init_signal_spy calls in a VHDL process. You need to code the VHDL process correctly so that it is executed only once. The VHDL process should not be sensitive to any signals and should contain only init_signal_spy calls and a simple wait statement. The process will execute once and then wait forever, which is the desired behavior. See the example below.

Syntax

init_signal_spy(<src_object>, <dest_object>, <verbose>, <control_state>)

Returns

Nothing

Arguments

- src_object

  Required string. A full hierarchical path (or relative downward path with reference to the calling block) to a VHDL signal or Verilog register/net. Use the path separator to which your simulation is set (i.e., "/" or "."). A full hierarchical path must begin with a "/" or ".". The path must be contained within double quotes.

- dest_object

  Required string. A full hierarchical path (or relative downward path with reference to the calling block) to an existing VHDL signal or Verilog register. Use the path separator to which your simulation is set (i.e., "/" or "."). A full hierarchical path must begin with a "/" or ".". The path must be contained within double quotes.

- verbose

  Optional integer. Possible values are 0 or 1. Specifies whether you want a message reported in the Transcript stating that the src_object's value is mirrored onto the dest_object. Default is 0, no message.

- control_state

  Optional integer. Possible values are -1, 0, or 1. Specifies whether or not you want the ability to enable/disable mirroring of values and, if so, specifies the initial state. The default is -1, no ability to enable/disable and mirroring is enabled. "0" turns on the ability to enable/disable and initially disables mirroring. "1" turns on the ability to enable/disable and initially enables mirroring.

Related procedures

init_signal_driver, signal_force, signal_release, enable_signal_spy, disable_signal_spy

Limitations

- When mirroring the value of a Verilog register/net onto a VHDL signal, the VHDL signal must be of type bit, bit_vector, std_logic, or std_logic_vector.
- Verilog memories (arrays of registers) are not supported.

init_signal_spy Example

In this example, the value of */top/uut/inst1/sig1* is mirrored onto */top/top_sig1*. A message is issued to the transcript. The ability to control the mirroring of values is turned on and the init_signal_spy is initially enabled.

The mirroring of values will be disabled when enable_sig transitions to a '0' and enable when enable_sig transitions to a '1'.

```
library ieee;
library modelsim_lib;
use ieee.std_logic_1164.all;
use modelsim_lib.util.all;
entity top is
end;
architecture only of top is
  signal top_sig1 : std_logic;
begin
  ...
  spy_process : process
  begin
    init_signal_spy("/top/uut/inst1/sig1","/top/top_sig1",1,1);
    wait;
  end process spy_process;
  ...
  spy_enable_disable : process(enable_sig)
  begin
    if (enable_sig = '1') then
      enable_signal_spy("/top/uut/inst1/sig1","/top/top_sig1",0);
    elseif (enable_sig = '0')
```

```
        disable_signal_spy("/top/uut/inst1/sig1","/top/top_sig1",0);
      end if;
    end process spy_enable_disable;
    ...
  end;
```

signal_force

The signal_force() procedure forces the value specified onto an existing VHDL signal or Verilog register or net (called the dest_object). This allows you to force signals, registers, or nets at any level of the design hierarchy from within a VHDL architecture (e.g., a testbench).

A signal_force works the same as the force command with the exception that you cannot issue a repeating force. The force will remain on the signal until a signal_release, a force or release command, or a subsequent signal_force is issued. Signal_force can be called concurrently or sequentially in a process.

This command acquires displays any signals using your radix setting (either the default, or as you specify) unless you specify the radix in the *value* you set.

Syntax

signal_force(<dest_object>, <value>, <rel_time>, <force_type>, <cancel_period>, <verbose>)

Returns

Nothing

Arguments

- dest_object

  Required string. A full hierarchical path (or relative downward path with reference to the calling block) to an existing VHDL signal or Verilog register/net. Use the path separator to which your simulation is set (i.e., "/" or "."). A full hierarchical path must begin with a "/" or ".". The path must be contained within double quotes.

- value

  Required string. Specifies the value to which the dest_object is to be forced. The specified value must be appropriate for the type.

- rel_time

  Optional time. Specifies a time relative to the current simulation time for the force to occur. The default is 0.

- force_type

  Optional forcetype. Specifies the type of force that will be applied. The value must be one of the following; default, deposit, drive, or freeze. The default is "default" (which is "freeze" for unresolved objects or "drive" for resolved objects). See the force command for further details on force type.

- cancel_period

  Optional time. Cancels the signal_force command after the specified period of time units. Cancellation occurs at the last simulation delta cycle of a time unit. A value of zero cancels the force at the end of the current time period. Default is -1 ms. A negative value means that the force will not be cancelled.

- verbose

  Optional integer. Possible values are 0 or 1. Specifies whether you want a message reported in the Transcript stating that the value is being forced on the dest_object at the specified time. Default is 0, no message.

Related procedures

init_signal_driver, init_signal_spy, signal_release

Limitations

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

signal_force Example

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

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

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

entity testbench is
end;

architecture only of testbench is
begin

  force_process : process
  begin
    signal_force("/testbench/uut/blk1/reset", "1", 0 ns, freeze, open, 1);
    signal_force("/testbench/uut/blk1/reset", "0", 40 ns, freeze, 2 ms,
1);
    wait;
  end process force_process;

  ...

end;
```

signal_release

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

A signal_release works the same as the noforce command. Signal_release can be called concurrently or sequentially in a process.

Syntax

signal_release(<dest_object>, <verbose>)

Returns

Nothing

Arguments

- dest_object

  Required string. A full hierarchical path (or relative downward path with reference to the calling block) to an existing VHDL signal or Verilog register/net. Use the path separator to which your simulation is set (i.e., "/" or "."). A full hierarchical path must begin with a "/" or ".". The path must be contained within double quotes.

- verbose

  Optional integer. Possible values are 0 or 1. Specifies whether you want a message reported in the Transcript stating that the signal is being released and the time of the release. Default is 0, no message.

Related procedures

init_signal_driver, init_signal_spy, signal_force

Limitations

- You cannot release a bit or slice of a register; you can release only the entire register.

signal_release Example

This example releases any forces on the signals data and *clk* when the signal *release_flag* is a "1". Both calls will send a message to the transcript stating which signal was released and when.

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

entity testbench is
end;

architecture only of testbench is

  signal release_flag : std_logic;
```

```
begin

  stim_design : process
  begin
    ...
    wait until release_flag = '1';
    signal_release("/testbench/dut/blk1/data", 1);
    signal_release("/testbench/dut/blk1/clk", 1);
    ...
  end process stim_design;

  ...

end;
```

$disable_signal_spy

The $disable_signal_spy() system task disables the associated $init_signal_spy task. The association between the $disable_signal_spy task and the $init_signal_spy task is based on specifying the same src_object and dest_object arguments to both tasks. The $disable_signal_spy task can only affect $init_signal_spy tasks that had their control_state argument set to "0" or "1".

Syntax

$disable_signal_spy(<src_object>, <dest_object>, <verbose>)

Returns

Nothing

Arguments

- src_object

  Required string. A full hierarchical path (or relative downward path with reference to the calling block) to a VHDL signal or Verilog register/net. This path should match the path that was specified in the init_signal_spy call that you wish to disable.

- dest_object

  Required string. A full hierarchical path (or relative downward path with reference to the calling block) to a VHDL signal or Verilog register/net. This path should match the path that was specified in the init_signal_spy call that you wish to disable.

- verbose

  Optional integer. Possible values are 0 or 1. Specifies whether you want a message reported in the transcript stating that a disable occurred and the simulation time that it occurred. Default is 0, no message

Related tasks

$init_signal_spy, $enable_signal_spy

Example

See $init_signal_spy Example

$enable_signal_spy

The $enable_signal_spy() system task enables the associated $init_signal_spy task. The association between the $enable_signal_spy task and the $init_signal_spy task is based on specifying the same src_object and dest_object arguments to both tasks. The $enable_signal_spy task can only affect $init_signal_spys tasks that had their control_state argument set to "0" or "1".

Syntax

$enable_signal_spy(<src_object>, <dest_object>, <verbose>)

Returns

Nothing

Arguments

- src_object

  Required string. A full hierarchical path (or relative downward path with reference to the calling block) to a VHDL signal or Verilog register/net. This path should match the path that was specified in the init_signal_spy call that you wish to enable.

- dest_object

  Required string. A full hierarchical path (or relative downward path with reference to the calling block) to a VHDL signal or Verilog register/net. This path should match the path that was specified in the init_signal_spy call that you wish to enable.

- verbose

  Optional integer. Possible values are 0 or 1. Specifies whether you want a message reported in the transcript stating that an enable occurred and the simulation time that it occurred. Default is 0, no message

Related tasks

$init_signal_spy, $disable_signal_spy

Example

See $init_signal_spy Example

$init_signal_driver

The $init_signal_driver() system task drives the value of a VHDL signal or Verilog net (called the src_object) onto an existing VHDL signal or Verilog register/net (called the dest_object). This allows you to drive signals or nets at any level of the design hierarchy from within a Verilog module (e.g., a testbench).

The $init_signal_driver system task drives the value onto the destination signal just as if the signals were directly connected in the HDL code. Any existing or subsequent drive or force of the destination signal, by some other means, will be considered with the $init_signal_driver value in the resolution of the signal.

Call only once

The $init_signal_driver system task creates a persistent relationship between the source and destination signals. Hence, you need to call $init_signal_driver only once for a particular pair of signals. Once $init_signal_driver is called, any change on the source signal will be driven on the destination signal until the end of the simulation.

Thus, we recommend that you place all $init_signal_driver calls in a Verilog initial block. See the example below.

Syntax

$init_signal_driver(<src_object>, <dest_object>, <delay>, <delay_type>, <verbose>)

Returns

Nothing

Arguments

- src_object

  Required string. A full hierarchical path (or relative downward path with reference to the calling block) to a VHDL signal or Verilog net. Use the path separator to which your simulation is set (i.e., "/" or "."). A full hierarchical path must begin with a "/" or ".". The path must be contained within double quotes.

- dest_object

  Required string. A full hierarchical path (or relative downward path with reference to the calling block) to an existing VHDL signal or Verilog net. Use the path separator to which your simulation is set (i.e., "/" or "."). A full hierarchical path must begin with a "/" or ".". The path must be contained within double quotes.

- delay

  Optional integer, real, or time. Specifies a delay relative to the time at which the src_object changes. The delay can be an inertial or transport delay. If no delay is specified, then a delay of zero is assumed.

- delay_type

  Optional integer. Specifies the type of delay that will be applied. The value must be either 0 (inertial) or 1 (transport). The default is 0.

- verbose

  Optional integer. Possible values are 0 or 1. Specifies whether you want a message reported in the Transcript stating that the src_object is driving the dest_object. Default is 0, no message.

Related tasks

$init_signal_spy, $signal_force, $signal_release

Limitations

- When driving a Verilog net, the only *delay_type* allowed is inertial. If you set the delay type to 1 (transport), the setting will be ignored, and the delay type will be inertial.
- Any delays that are set to a value less than the simulator resolution will be rounded to the nearest resolution unit; no special warning will be issued.
- Verilog memories (arrays of registers) are not supported.

$init_signal_driver Example

This example creates a local clock (*clk0*) and connects it to two clocks within the design hierarchy. The *.../blk1/clk* will match local *clk0* and a message will be displayed. The *.../blk2/clk* will match the local *clk0* but be delayed by 100 ps. For the second call to work, the *.../blk2/clk* must be a VHDL based signal, because if it were a Verilog net a 100 ps inertial delay would consume the 40 ps clock period. Verilog nets are limited to only inertial delays and thus the setting of 1 (transport delay) would be ignored.

```
`timescale 1 ps / 1 ps

module testbench;

reg clk0;

initial begin
  clk0 = 1;
  forever begin
   #20 clk0 = ~clk0;
  end
end

initial begin
   $init_signal_driver("clk0", "/testbench/uut/blk1/clk", , , 1);
   $init_signal_driver("clk0", "/testbench/uut/blk2/clk", 100, 1);
end

  ...

endmodule
```

$init_signal_spy

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

The $init_signal_spy system task only sets the value onto the destination signal and does not drive or force the value. Any existing or subsequent drive or force of the destination signal, by some other means, will override the value set by $init_signal_spy.

Call only once

The $init_signal_spy system task creates a persistent relationship between the source and the destination signal. Hence, you need to call $init_signal_spy only once for a particular pair of signals. Once $init_signal_spy is called, any change on the source signal will mirror on the destination signal until the end of the simulation unless the control_state is set.

The control_state determines whether the mirroring of values can be enabled/disabled and what the initial state is. Subsequent control of whether the mirroring of values is enabled/disabled is handled by the $enable_signal_spy and $disable_signal_spy tasks.

We recommend that you place all $init_signal_spy tasks in a Verilog initial block. See the example below.

Syntax

$init_signal_spy(<src_object>, <dest_object>, <verbose>, <control_state>)

Returns

Nothing

Arguments

- src_object

  Required string. A full hierarchical path (or relative downward path with reference to the calling block) to a VHDL signal or Verilog register/net. Use the path separator to which your simulation is set (i.e., "/" or "."). A full hierarchical path must begin with a "/" or ".". The path must be contained within double quotes.

- dest_object

  Required string. A full hierarchical path (or relative downward path with reference to the calling block) to a Verilog register or VHDL signal. Use the path separator to which your simulation is set (i.e., "/" or "."). A full hierarchical path must begin with a "/" or ".". The path must be contained within double quotes.

- verbose

  Optional integer. Possible values are 0 or 1. Specifies whether you want a message reported in the Transcript stating that the src_object's value is mirrored onto the dest_object. Default is 0, no message.

- control_state

  Optional integer. Possible values are -1, 0, or 1. Specifies whether or not you want the ability to enable/disable mirroring of values and, if so, specifies the initial state. The default is -1, no ability to enable/disable and mirroring is enabled. "0" turns on the ability to enable/disable and initially disables mirroring. "1" turns on the ability to enable/disable and initially enables mirroring.

Related tasks

$init_signal_driver, $signal_force, $signal_release, $disable_signal_spy

Limitations

- When mirroring the value of a VHDL signal onto a Verilog register, the VHDL signal must be of type bit, bit_vector, std_logic, or std_logic_vector.
- Verilog memories (arrays of registers) are not supported.

$init_signal_spy Example

In this example, the value of *.top.uut.inst1.sig1* is mirrored onto *.top.top_sig1*. A message is issued to the transcript. The ability to control the mirroring of values is turned on and the init_signal_spy is initially enabled.

The mirroring of values will be disabled when enable_reg transitions to a ’0’ and enabled when enable_reg transitions to a ’1’.

```
module top;
...
reg top_sig1;
reg enable_reg;
...
initial
  begin
  $init_signal_spy(".top.uut.inst1.sig1",".top.top_sig1",1,1);
  end
  always @ (posedge enable_reg)
  begin
  $enable_signal_spy(".top.uut.inst1.sig1",".top.top_sig1",0);
  end
  always @ (negedge enable_reg)
  begin
  $disable_signal_spy(".top.uut.inst1.sig1",".top.top_sig1",0);
  end
...
endmodule
```

$signal_force

The $signal_force() system task forces the value specified onto an existing VHDL signal or Verilog register/net (called the dest_object). This allows you to force signals, registers, or nets at any level of the design hierarchy from within a Verilog module (e.g., a testbench).

A $signal_force works the same as the force command with the exception that you cannot issue a repeating force. The force will remain on the signal until a $signal_release, a force or release command, or a subsequent $signal_force is issued. $signal_force can be called concurrently or sequentially in a process.

Syntax

$signal_force(<dest_object>, <value>, <rel_time>, <force_type>, <cancel_period>,
    <verbose>)

Returns

Nothing

Arguments

- dest_object

  Required string. A full hierarchical path (or relative downward path with reference to the calling block) to an existing VHDL signal or Verilog register/net. Use the path separator to which your simulation is set (i.e., "/" or "."). A full hierarchical path must begin with a "/" or ".". The path must be contained within double quotes.

- value

  Required string. Specifies the value to which the dest_object is to be forced. The specified value must be appropriate for the type.

- rel_time

  Optional integer, real, or time. Specifies a time relative to the current simulation time for the force to occur. The default is 0.

- force_type

  Optional integer. Specifies the type of force that will be applied. The value must be one of the following; 0 (default), 1 (deposit), 2 (drive), or 3 (freeze). The default is "default" (which is "freeze" for unresolved objects or "drive" for resolved objects). See the force command for further details on force type.

- cancel_period

  Optional integer, real, time. Cancels the $signal_force command after the specified period of time units. Cancellation occurs at the last simulation delta cycle of a time unit. A value of zero cancels the force at the end of the current time period. Default is -1. A negative value means that the force will not be cancelled.

- verbose

  Optional integer. Possible values are 0 or 1. Specifies whether you want a message reported in the Transcript stating that the value is being forced on the dest_object at the specified time. Default is 0, no message.

Related tasks

$init_signal_driver, $init_signal_spy, $signal_release

Limitations

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

$signal_force Example

This example forces *reset* to a "1" from time 0 ns to 40 ns. At 40 ns, *reset* is forced to a "0", 200000 ns after the second $signal_force call was executed.

```
`timescale 1 ns / 1 ns

module testbench;

initial
  begin
   $signal_force("/testbench/uut/blk1/reset", "1", 0, 3, , 1);
   $signal_force("/testbench/uut/blk1/reset", "0", 40, 3, 200000, 1);
  end

...

endmodule
```

$signal_release

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

A $signal_release works the same as the noforce command. $signal_release can be called concurrently or sequentially in a process.

Syntax

$signal_release(<dest_object>, <verbose>)

Returns

Nothing

Arguments

- dest_object

  Required string. A full hierarchical path (or relative downward path with reference to the calling block) to an existing VHDL signal or Verilog register/net. Use the path separator to which your simulation is set (i.e., "/" or "."). A full hierarchical path must begin with a "/" or ".". The path must be contained within double quotes.

- verbose

  Optional integer. Possible values are 0 or 1. Specifies whether you want a message reported in the Transcript stating that the signal is being released and the time of the release. Default is 0, no message.

Related tasks

$init_signal_driver, $init_signal_spy, $signal_force

Limitations

- You cannot release a bit or slice of a register; you can release only the entire register.

$signal_release Example

This example releases any forces on the signals *data* and *clk* when the register *release_flag* transitions to a "1". Both calls will send a message to the transcript stating which signal was released and when.

```
module testbench;

reg release_flag;

always @(posedge release_flag) begin
  $signal_release("/testbench/dut/blk1/data", 1);
  $signal_release("/testbench/dut/blk1/clk", 1);
end

...

endmodule
```

Chapter 11
Standard Delay Format (SDF) Timing Annotation

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

Verilog and VHDL VITAL timing data can be annotated from SDF files by using the simulator's built-in SDF annotator.

Note

SDF timing annotations can be applied only to your FPGA vendor's libraries; all other libraries will simulate without annotation.

Specifying SDF Files for Simulation

ModelSim supports SDF versions 1.0 through 4.0 (except the NETDELAY statement). The simulator's built-in SDF annotator automatically adjusts to the version of the file. Use the following vsim command-line options to specify the SDF files, the desired timing values, and their associated design instances:

```
-sdfmin [<instance>=]<filename>
-sdftyp [<instance>=]<filename>
-sdfmax [<instance>=]<filename>
```

Any number of SDF files can be applied to any instance in the design by specifying one of the above options for each file. Use **-sdfmin** to select minimum, **-sdftyp** to select typical, and **-sdfmax** to select maximum timing values from the SDF file.

Instance Specification

The instance paths in the SDF file are relative to the instance to which the SDF is applied. Usually, this instance is an ASIC or FPGA model instantiated under a testbench. For example, to annotate maximum timing values from the SDF file *myasic.sdf* to an instance *u1* under a top-level named *testbench*, invoke the simulator as follows:

```
vsim -sdfmax /testbench/u1=myasic.sdf testbench
```

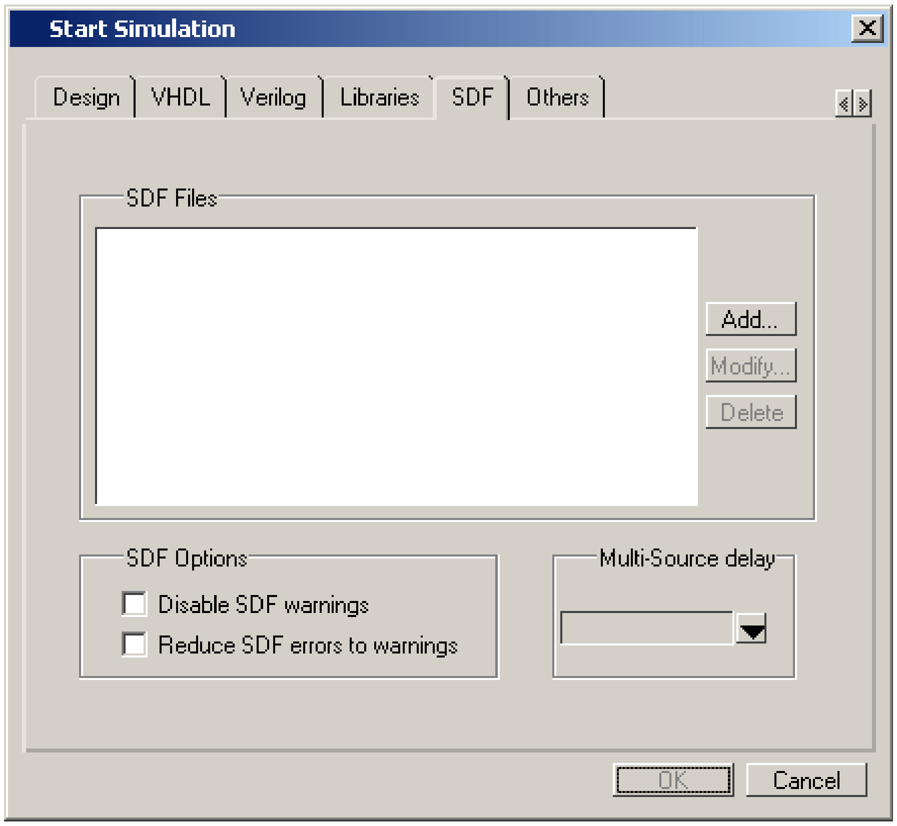
If the instance name is omitted then the SDF file is applied to the top-level. *This is usually incorrect* because in most cases the model is instantiated under a testbench or within a larger system level simulation. In fact, the design can have several models, each having its own SDF file. In this case, specify an SDF file for each instance. For example,

vsim -sdfmax /system/u1=asic1.sdf -sdfmax /system/u2=asic2.sdf system

SDF Specification with the GUI

As an alternative to the command-line options, you can specify SDF files in the **Start Simulation** dialog box under the SDF tab.

Figure 11-1. SDF Tab in Start Simulation Dialog



You can access this dialog by invoking the simulator without any arguments or by selecting **Simulate > Start Simulation**. See the GUI chapter for a description of this dialog.

For Verilog designs, you can also specify SDF files by using the **$sdf_annotate** system task. See $sdf_annotate for more details.

Errors and Warnings

Errors issued by the SDF annotator while loading the design prevent the simulation from continuing, whereas warnings do not. Use the **-sdfnoerror** option with vsim to change SDF errors to warnings so that the simulation can continue. Warning messages can be suppressed by using vsim with either the **-sdfnowarn** or **+nosdfwarn** options.

Another option is to use the **SDF** tab from the **Start Simulation** dialog box (shown above). Select **Disable SDF warnings** (-sdfnowarn +nosdfwarn) to disable warnings, or select **Reduce SDF errors to warnings** (-sdfnoerror) to change errors to warnings.

See Troubleshooting for more information on errors and warnings and how to avoid them.

VHDL VITAL SDF

VHDL SDF annotation works on VITAL cells only. The IEEE 1076.4 VITAL ASIC Modeling Specification describes how cells must be written to support SDF annotation. Once again, the designer does not need to know the details of this specification because the library provider has already written the VITAL cells and tools that create compatible SDF files. However, the following summary may help you understand simulator error messages. For additional VITAL specification information, see VITAL Specification and Source Code.

SDF to VHDL Generic Matching

An SDF file contains delay and timing constraint data for cell instances in the design. The annotator must locate the cell instances and the placeholders (VHDL generics) for the timing data. Each type of SDF timing construct is mapped to the name of a generic as specified by the VITAL modeling specification. The annotator locates the generic and updates it with the timing value from the SDF file. It is an error if the annotator fails to find the cell instance or the named generic. The following are examples of SDF constructs and their associated generic names:

Table 11-1. Matching SDF to VHDL Generics

SDF construct	Matching VHDL generic name
(IOPATH a y (3))	tpd_a_y
(IOPATH (posedge clk) q (1) (2))	tpd_clk_q_posedge
(INTERCONNECT u1/y u2/a (5))	tipd_a
(SETUP d (posedge clk) (5))	tsetup_d_clk_noedge_posedge
(HOLD (negedge d) (posedge clk) (5))	thold_d_clk_negedge_posedge
(SETUPHOLD d clk (5) (5))	tsetup_d_clk & thold_d_clk
(WIDTH (COND (reset==1’b0) clk) (5))	tpw_clk_reset_eq_0

The SDF statement CONDELSE, when targeted for Vital cells, is annotated to a **tpd** generic of the form **tpd_<inputPort>_<outputPort>**.

Resolving Errors

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

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

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

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

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

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

For more information on resolving errors see Troubleshooting.

Verilog SDF

Verilog designs can be annotated using either the simulator command-line options or the **$sdf_annotate** system task (also commonly used in other Verilog simulators). The command-line options annotate the design immediately after it is loaded, but before any simulation events take place. The **$sdf_annotate** task annotates the design at the time it is called in the Verilog source code. This provides more flexibility than the command-line options.

$sdf_annotate

Syntax

$sdf_annotate
 (["<sdffile>"], [<instance>], ["<config_file>"], ["<log_file>"], ["<mtm_spec>"],
 ["<scale_factor>"], ["<scale_type>"]);

Arguments

- "<sdffile>"

  String that specifies the SDF file. Required.

- <instance>

  Hierarchical name of the instance to be annotated. Optional. Defaults to the instance where the $sdf_annotate call is made.

- "<config_file>"

  String that specifies the configuration file. Optional. Currently not supported, this argument is ignored.

- "<log_file>"

  String that specifies the logfile. Optional. Currently not supported, this argument is ignored.

- "<mtm_spec>"

  String that specifies the delay selection. Optional. The allowed strings are "minimum", "typical", "maximum", and "tool_control". Case is ignored and the default is "tool_control". The "tool_control" argument means to use the delay specified on the command line by +mindelays, +typdelays, or +maxdelays (defaults to +typdelays).

- "<scale_factor>"

  String that specifies delay scaling factors. Optional. The format is "<min_mult>:<typ_mult>:<max_mult>". Each multiplier is a real number that is used to scale the corresponding delay in the SDF file.

- "<scale_type>"

  String that overrides the **<mtm_spec>** delay selection. Optional. The **<mtm_spec>** delay selection is always used to select the delay scaling factor, but if a **<scale_type>** is specified, then it will determine the min/typ/max selection from the SDF file. The allowed strings are "from_min", "from_minimum", "from_typ", "from_typical", "from_max", "from_maximum", and "from_mtm". Case is ignored, and the default is "from_mtm", which means to use the **<mtm_spec>** value.

Examples

Optional arguments can be omitted by using commas or by leaving them out if they are at the end of the argument list. For example, to specify only the SDF file and the instance to which it applies:

```
$sdf_annotate("myasic.sdf", testbench.u1);
```

To also specify maximum delay values:

```
$sdf_annotate("myasic.sdf", testbench.u1, , , "maximum");
```

SDF to Verilog Construct Matching

The annotator matches SDF constructs to corresponding Verilog constructs in the cells. Usually, the cells contain path delays and timing checks within specify blocks. For each SDF construct, the annotator locates the cell instance and updates each specify path delay or timing check that matches. An SDF construct can have multiple matches, in which case each matching specify statement is updated with the SDF timing value. SDF constructs are matched to Verilog constructs as follows.

- **IOPATH** is matched to specify path delays or primitives:

Table 11-2. Matching SDF IOPATH to Verilog

SDF	Verilog
(IOPATH (posedge clk) q (3) (4))	(posedge clk => q) = 0;
(IOPATH a y (3) (4))	buf u1 (y, a);

The IOPATH construct usually annotates path delays. If ModelSim can't locate a corresponding specify path delay, it returns an error unless you use the **+sdf_iopath_to_prim_ok** argument to vsim. If you specify that argument and the module contains no path delays, then all primitives that drive the specified output port are annotated.

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

Table 11-3. Matching SDF INTERCONNECT and PORT to Verilog

SDF	Verilog
(INTERCONNECT u1.y u2.a (5))	input a;
(PORT u2.a (5))	inout a;

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

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

Table 11-4. Matching SDF PATHPULSE and GLOBALPATHPULSE to Verilog

SDF	Verilog
(PATHPULSE a y (5) (10))	(a => y) = 0;
(GLOBALPATHPULSE a y (30) (60))	(a => y) = 0;

If the input and output ports are omitted in the SDF, then all path delays are matched in the cell.

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

Table 11-5. Matching SDF DEVICE to Verilog

SDF	Verilog
(DEVICE y (5))	and u1(y, a, b);
(DEVICE y (5))	(a => y) = 0; (b => y) = 0;

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

SETUP is matched to $setup and $setuphold:

Table 11-6. Matching SDF SETUP to Verilog

SDF	Verilog
(SETUP d (posedge clk) (5))	$setup(d, posedge clk, 0);
(SETUP d (posedge clk) (5))	$setuphold(posedge clk, d, 0, 0);

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

Table 11-7. Matching SDF HOLD to Verilog

SDF	Verilog
(HOLD d (posedge clk) (5))	$hold(posedge clk, d, 0);
(HOLD d (posedge clk) (5))	$setuphold(posedge clk, d, 0, 0);

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

Table 11-8. Matching SDF SETUPHOLD to Verilog

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:

Table 11-9. Matching SDF RECOVERY to Verilog

SDF	Verilog
(RECOVERY (negedge reset) (posedge clk) (5))	$recovery(negedge reset, posedge clk, 0);

- **REMOVAL** is matched to $removal:

Table 11-10. Matching SDF REMOVAL to Verilog

SDF	Verilog
(REMOVAL (negedge reset) (posedge clk) (5))	$removal(negedge reset, posedge clk, 0);

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

Table 11-11. Matching SDF RECREM to Verilog

SDF	Verilog
(RECREM (negedge reset) (posedge clk) (5) (5))	$recovery(negedge reset, posedge clk, 0);
(RECREM (negedge reset) (posedge clk) (5) (5))	$removal(negedge reset, posedge clk, 0);
(RECREM (negedge reset) (posedge clk) (5) (5))	$recrem(negedge reset, posedge clk, 0);

- **SKEW** is matched to $skew:

Table 11-12. Matching SDF SKEW to Verilog

SDF	Verilog
(SKEW (posedge clk1) (posedge clk2) (5))	$skew(posedge clk1, posedge clk2, 0);

- **WIDTH** is matched to $width:

Table 11-13. Matching SDF WIDTH to Verilog

SDF	Verilog
(WIDTH (posedge clk) (5))	$width(posedge clk, 0);

- **PERIOD** is matched to $period:

Table 11-14. Matching SDF PERIOD to Verilog

SDF	Verilog
(PERIOD (posedge clk) (5))	$period(posedge clk, 0);

- **NOCHANGE** is matched to $nochange:

Table 11-15. Matching SDF NOCHANGE to Verilog

SDF	Verilog
(NOCHANGE (negedge write) addr (5) (5))	$nochange(negedge write, addr, 0, 0);

Optional Edge Specifications

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

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

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

Table 11-16. Matching Verilog Timing Checks to SDF SETUP

SDF	Verilog
(SETUP data (posedge clock) (5))	$setup(posedge data, posedge clk, 0);
(SETUP data (posedge clock) (5))	$setup(negedge data, posedge clk, 0);

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

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

Table 11-17. SDF Data May Be More Accurate Than Model

SDF	Verilog
(SETUP (posedge data) (posedge clock) (4))	$setup(data, posedge clk, 0);
(SETUP (negedge data) (posedge clock) (6))	$setup(data, posedge clk, 0);

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

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

Table 11-18. Matching Explicit Verilog Edge Transitions to Verilog

SDF	Verilog
(SETUP data (posedge clock) (5))	$setup(data, edge[01, 0x] clk, 0);

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

Optional Conditions

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

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

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

Table 11-19. SDF Timing Check Conditions

SDF	Verilog
(SETUP data (COND (reset!=1) (posedge clock)) (5))	$setup(data, posedge clk &&& (reset==0),0);

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

Table 11-20. SDF Path Delay Conditions

SDF	Verilog
(COND (r1 || r2) (IOPATH clk q (5)))	if (r1 || r2) (clk => q) = 5; // matches
(COND (r1 || r2) (IOPATH clk q (5)))	if (r2 || r1) (clk => q) = 5; // does not match

The annotator does not match the second condition above because the order of r1 and r2 are reversed.

Rounded Timing Values

The SDF **TIMESCALE** construct specifies time units of values in the SDF file. The annotator rounds timing values from the SDF file to the time precision of the module that is annotated. For example, if the SDF TIMESCALE is 1ns and a value of .016 is annotated to a path delay in a module having a time precision of 10ps (from the timescale directive), then the path delay receives a value of 20ps. The SDF value of 16ps is rounded to 20ps. Interconnect delays are rounded to the time precision of the module that contains the annotated MIPD.

SDF for Mixed VHDL and Verilog Designs

Annotation of a mixed VHDL and Verilog design is very flexible. VHDL VITAL cells and Verilog cells can be annotated from the same SDF file. This flexibility is available only by using the simulator's SDF command-line options. The Verilog $sdf_annotate system task can annotate Verilog cells only. See the vsim command for more information on SDF command-line options.

Interconnect Delays

An interconnect delay represents the delay from the output of one device to the input of another. ModelSim can model single interconnect delays or multisource interconnect delays for Verilog, VHDL/VITAL, or mixed designs. See the **vsim** command for more information on the relevant command-line arguments.

Timing checks are performed on the interconnect delayed versions of input ports. This may result in misleading timing constraint violations, because the ports may satisfy the constraint while the delayed versions may not. If the simulator seems to report incorrect violations, be sure to account for the effect of interconnect delays.

Disabling Timing Checks

ModelSim offers a number of options for disabling timing checks on a "global" or individual basis. The table below provides a summary of those options. See the command and argument descriptions in the Reference Manual for more details.

Table 11-21. Disabling Timing Checks

Command and argument	Effect
vlog +notimingchecks	disables timing check system tasks for all instances in the specified Verilog design
vlog +nospecify	disables specify path delays and timing checks for all instances in the specified Verilog design
vsim +no_neg_tchk	disables negative timing check limits by setting them to zero for all instances in the specified design
vsim +no_notifier	disables the toggling of the notifier register argument of the timing check system tasks for all instances in the specified design
vsim +no_tchk_msg	disables error messages issued by timing check system tasks when timing check violations occur for all instances in the specified design
vsim +notimingchecks	disables Verilog and VITAL timing checks for all instances in the specified design
vsim +nospecify	disables specify path delays and timing checks for all instances in the specified design

Troubleshooting

Specifying the Wrong Instance

By far, the most common mistake in SDF annotation is to specify the wrong instance to the simulator's SDF options. The most common case is to leave off the instance altogether, which is the same as selecting the top-level design unit. This is generally wrong because the instance paths in the SDF are relative to the ASIC or FPGA model, which is usually instantiated under a top-level testbench. See Instance Specification for an example.

A common example for both VHDL and Verilog testbenches is provided below. For simplicity, the test benches do nothing more than instantiate a model that has no ports.

VHDL Testbench

```
entity testbench is end;
```

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

Verilog Testbench

```
module testbench;
   myasic dut();
endmodule
```

The name of the model is *myasic* and the instance label is *dut*. For either testbench, an appropriate simulator invocation might be:

vsim -sdfmax /testbench/dut=myasic.sdf testbench

Optionally, you can leave off the name of the top-level:

vsim -sdfmax /dut=myasic.sdf testbench

The important thing is to select the instance for which the SDF is intended. If the model is deep within the design hierarchy, an easy way to find the instance name is to first invoke the simulator without SDF options, view the structure pane, navigate to the model instance, select it, and enter the environment command. This command displays the instance name that should be used in the SDF command-line option.

Mistaking a Component or Module Name for an Instance Label

Another common error is to specify the component or module name rather than the instance label. For example, the following invocation is wrong for the above testbenches:

vsim -sdfmax /testbench/myasic=myasic.sdf testbench

This results in the following error message:

```
** Error (vsim-SDF-3250) myasic.sdf(0):
Failed to find INSTANCE '/testbench/myasic'.
```

Forgetting to Specify the Instance

If you leave off the instance altogether, then the simulator issues a message for each instance path in the SDF that is not found in the design. For example,

vsim -sdfmax myasic.sdf testbench

Results in:

```
** Error (vsim-SDF-3250) myasic.sdf(0):
Failed to find INSTANCE ’/testbench/u1’
** Error (vsim-SDF-3250) myasic.sdf(0):
Failed to find INSTANCE ’/testbench/u2’
** Error (vsim-SDF-3250) myasic.sdf(0):
Failed to find INSTANCE ’/testbench/u3’
** Error (vsim-SDF-3250) myasic.sdf(0):
Failed to find INSTANCE ’/testbench/u4’
** Error (vsim-SDF-3250) myasic.sdf(0):
Failed to find INSTANCE ’/testbench/u5’
** Warning (vsim-SDF-3432) myasic.sdf:
This file is probably applied to the wrong instance.
** Warning (vsim-SDF-3432) myasic.sdf:
Ignoring subsequent missing instances from this file.
```

After annotation is done, the simulator issues a summary of how many instances were not found and possibly a suggestion for a qualifying instance:

```
** Warning (vsim-SDF-3440) myasic.sdf:
Failed to find any of the 358 instances from this file.
** Warning (vsim-SDF-3442) myasic.sdf:
Try instance ’/testbench/dut’. It contains all instance paths from this
file.
```

The simulator recommends an instance only if the file was applied to the top-level and a qualifying instance is found one level down.

Also see Resolving Errors for specific VHDL VITAL SDF troubleshooting.

Chapter 12
Value Change Dump (VCD) Files

This chapter describes how to use VCD files in ModelSim. The VCD file format is specified in the IEEE 1364 standard. It is an ASCII file containing header information, variable definitions, and variable value changes.

VCD is in common use for Verilog designs, and is controlled by VCD system task calls in the Verilog source code. ModelSim provides command equivalents for these system tasks and extends VCD support to VHDL designs. The ModelSim commands can be used on VHDL, Verilog, or mixed designs.

If you need vendor-specific ASIC design-flow documentation that incorporates VCD, please contact your ASIC vendor.

Creating a VCD File

There are two flows in ModelSim for creating a VCD file. One flow produces a four-state VCD file with variable changes in 0, 1, x, and z with no strength information; the other produces an extended VCD file with variable changes in all states and strength information and port driver data.

Both flows will also capture port driver changes unless filtered out with optional command-line arguments.

Flow for Four-State VCD File

First, compile and load the design:

```
% cd ~/modeltech/examples/misc
% vlib work
% vlog counter.v tcounter.v
% vsim test_counter
```

Next, with the design loaded, specify the VCD file name with the vcd file command and add objects to the file with the vcd add command:

```
VSIM 1> vcd file myvcdfile.vcd
VSIM 2> vcd add /test_counter/dut/*
VSIM 3> run
VSIM 4> quit -f
```

There will now be a VCD file in the working directory.

Flow for Extended VCD File

First, compile and load the design:

```
% cd ~/modeltech/examples/misc
% vlib work
% vlog counter.v tcounter.v
% vsim test_counter
```

Next, with the design loaded, specify the VCD file name and objects to add with the **vcd dumpports** command:

```
VSIM 1> vcd dumpports -file myvcdfile.vcd /test_counter/dut/*
VSIM 3> run
VSIM 4> quit -f
```

There will now be an extended VCD file called *myvcdfile.vcd* in the working directory.

Note

There is an internal limit to the number of port driver changes that can be created with the **vcd dumpports** command. If that limit is reached, use the **vcd add** command with the **-dumpports** option to create additional port driver changes.

By default ModelSim uses strength ranges for resolving conflicts as specified by IEEE 1364-2005. You can ignore strength ranges using the **-no_strength_range** argument to the vcd dumpports command. See Resolving Values for more details.

Case Sensitivity

VHDL is not case sensitive so ModelSim converts all signal names to lower case when it produces a VCD file. Conversely, Verilog designs are case sensitive so ModelSim maintains case when it produces a VCD file.

Using Extended VCD as Stimulus

You can use an extended VCD file as stimulus to re-simulate your design. There are two ways to do this: 1) simulate the top level of a design unit with the input values from an extended VCD file; and 2) specify one or more instances in a design to be replaced with the output values from the associated VCD file.

Simulating with Input Values from a VCD File

When simulating with inputs from an extended VCD file, you can simulate only one design unit at a time. In other words, you can apply the VCD file inputs only to the top level of the design unit for which you captured port data.

The general procedure includes two steps:

1. Create a VCD file for a single design unit using the vcd dumpports command.
2. Resimulate the single design unit using the **-vcdstim** argument to vsim. Note that **-vcdstim** works only with VCD files that were created by a ModelSim simulation.

Example 12-1. Verilog Counter

First, create the VCD file for the single instance using **vcd dumpports**:

```
% cd ~/modeltech/examples/misc
% vlib work
% vlog counter.v tcounter.v
% vsim test_counter
VSIM 1> vcd dumpports -file counter.vcd /test_counter/dut/*
VSIM 2> run
VSIM 3> quit -f
```

Next, rerun the counter without the testbench, using the **-vcdstim** argument:

```
% vsim -vcdstim counter.vcd counter
VSIM 1> add wave /*
VSIM 2> run 200
```

Example 12-2. VHDL Adder

First, create the VCD file using **vcd dumpports**:

```
% cd ~/modeltech/examples/misc
% vlib work
% vcom gates.vhd adder.vhd stimulus.vhd
% vsim testbench2
VSIM 1> vcd dumpports -file addern.vcd /testbench2/uut/*
VSIM 2> run 1000
VSIM 3> quit -f
```

Next, rerun the adder without the testbench, using the **-vcdstim** argument:

```
% vsim -vcdstim addern.vcd addern -gn=8 -do "add wave /*; run 1000"
```

Example 12-3. Mixed-HDL Design

First, create three VCD files, one for each module:

```
% cd ~/modeltech/examples/tutorials/mixed/projects
% vlib work
% vlog cache.v memory.v proc.v
% vcom util.vhd set.vhd top.vhd
% vsim top
VSIM 1> vcd dumpports -file proc.vcd /top/p/*
VSIM 2> vcd dumpports -file cache.vcd /top/c/*
VSIM 3> vcd dumpports -file memory.vcd /top/m/*
VSIM 4> run 1000
VSIM 5> quit -f
```

Next, rerun each module separately, using the captured VCD stimulus:

```
% vsim -vcdstim proc.vcd proc -do "add wave /*; run 1000"
VSIM 1> quit -f

% vsim -vcdstim cache.vcd cache -do "add wave /*; run 1000"
VSIM 1> quit -f

% vsim -vcdstim memory.vcd memory -do "add wave /*; run 1000"
VSIM 1> quit -f
```

Replacing Instances with Output Values from a VCD File

Replacing instances with output values from a VCD file lets you simulate without the instance's source or even the compiled object. The general procedure includes two steps:

1. Create VCD files for one or more instances in your design using the vcd dumpports command. If necessary, use the **-vcdstim** switch to handle port order problems (see below).
2. Re-simulate your design using the **-vcdstim <instance>=<filename>** argument to vsim. Note that this works only with VCD files that were created by a ModelSim simulation.

Example 12-4. Replacing Instances

In the following example, the three instances */top/p*, */top/c*, and */top/m* are replaced in simulation by the output values found in the corresponding VCD files.

First, create VCD files for all instances you want to replace:

```
vcd dumpports -vcdstim -file proc.vcd /top/p/*
vcd dumpports -vcdstim -file cache.vcd /top/c/*
vcd dumpports -vcdstim -file memory.vcd /top/m/*
run 1000
```

Next, simulate your design and map the instances to the VCD files you created:

```
vsim top -vcdstim /top/p=proc.vcd -vcdstim /top/c=cache.vcd
-vcdstim /top/m=memory.vcd
```

Port Order Issues

The **-vcdstim** argument to the **vcd dumpports** command ensures the order that port names appear in the VCD file matches the order that they are declared in the instance's module or entity declaration. Consider the following module declaration:

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

The order of the ports in the module line (clk, addr, data, ...) does not match the order of those ports in the input, output, and inout lines (clk, rdy, addr, ...). In this case the **-vcdstim** argument to the **vcd dumpports** command needs to be used.

In cases where the order is the same, you do not need to use the **-vcdstim** argument to **vcd dumpports**. Also, module declarations of the form:

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

do not require use of the argument.

VCD Commands and VCD Tasks

ModelSim VCD commands map to IEEE Std 1364 VCD system tasks and appear in the VCD file along with the results of those commands. The table below maps the VCD commands to their associated tasks.

Table 12-1. VCD Commands and SystemTasks

VCD commands	**VCD system tasks**
vcd add	$dumpvars
vcd checkpoint	$dumpall
vcd file	$dumpfile
vcd flush	$dumpflush
vcd limit	$dumplimit
vcd off	$dumpoff
vcd on	$dumpon

ModelSim also supports extended VCD (dumpports system tasks). The table below maps the VCD dumpports commands to their associated tasks.

Table 12-2. VCD Dumpport Commands and System Tasks

VCD dumpports commands	**VCD system tasks**
vcd dumpports	$dumpports
vcd dumpportsall	$dumpportsall
vcd dumpportsflush	$dumpportsflush
vcd dumpportslimit	$dumpportslimit
vcd dumpportsoff	$dumpportsoff
vcd dumpportson	$dumpportson

ModelSim supports multiple VCD files. This functionality is an extension of the IEEE Std 1364 specification. The tasks behave the same as the IEEE equivalent tasks such as $dumpfile, $dumpvar, etc. The difference is that $fdumpfile can be called multiple times to create more than one VCD file, and the remaining tasks require a filename argument to associate their actions with a specific file.

Table 12-3. VCD Commands and System Tasks for Multiple VCD Files

VCD commands	VCD system tasks
vcd add -file <filename>	$fdumpvars
vcd checkpoint <filename>	$fdumpall
vcd files <filename>	$fdumpfile
vcd flush <filename>	$fdumpflush
vcd limit <filename>	$fdumplimit
vcd off <filename>	$fdumpoff
vcd on <filename>	$fdumpon

Compressing Files with VCD Tasks

ModelSim can produce compressed VCD files using the **gzip** compression algorithm. Since we cannot change the syntax of the system tasks, we act on the extension of the output file name. If you specify a *.gz* extension on the filename, ModelSim will compress the output.

VCD File from Source To Output

The following example shows the VHDL source, a set of simulator commands, and the resulting VCD output.

VHDL Source Code

The design is a simple shifter device represented by the following VHDL source code:

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;

entity SHIFTER_MOD is
   port (CLK, RESET, data_in   : IN STD_LOGIC;
      Q : INOUT STD_LOGIC_VECTOR(8 downto 0));
END SHIFTER_MOD ;

architecture RTL of SHIFTER_MOD is
begin
   process (CLK,RESET)
   begin
      if (RESET = '1') then
         Q <= (others => '0') ;
      elsif (CLK'event and CLK = '1') then
         Q <= Q(Q'left - 1 downto 0) & data_in ;
      end if ;
   end process ;
end ;
```

VCD Simulator Commands

At simulator time zero, the designer executes the following commands:

```
vcd file output.vcd
vcd add -r *
force reset 1 0
force data_in 0 0
force clk 0 0
run 100
force clk 1 0, 0 50 -repeat 100
run 100
vcd off
force reset 0 0
force data_in 1 0
run 100
vcd on
run 850
force reset 1 0
run 50
vcd checkpoint
quit -sim
```

VCD Output

The VCD file created as a result of the preceding scenario would be called *output.vcd*. The following pages show how it would look.

```
$date
   Thu Sep 18 11:07:43 2003
$end
$version
   ModelSim Version 6.1
$end
$timescale
   1ns
$end
$scope module shifter_mod $end
$var wire 1 ! clk $end
$var wire 1 " reset $end
$var wire 1 # data_in $end
$var wire 1 $ q [8] $end
$var wire 1 % q [7] $end
$var wire 1 & q [6] $end
$var wire 1 ' q [5] $end
$var wire 1 ( q [4] $end
$var wire 1 ) q [3] $end
$var wire 1 * q [2] $end
$var wire 1 + q [1] $end
$var wire 1 , q [0] $end
$upscope $end
$enddefinitions $end
#0
$dumpvars
0!
1"
0#
0$
0%
0&
0'
0(
0)
0*
0+
0,
$end
#100
1!
#150
0!
#200
1!
$dumpoff
x!
x"
x#
x$
x%
x&
x'
x(
x)
x*
x+
x,
$end
#300
$dumpon
1!
0"
1#
0$
0%
0&
0'
0(
0)
0*
0+
1,
$end
#350
0!
#400
1!
1+
#450
0!
#500
1!
1*
#550
0!
#600
1!
1)
#650
0!
#700
1!
1(
#750
0!
#800
1!
1'
#850
0!
#900
1!
1&
#950
0!
#1000
1!
1%
#1050
0!
#1100
1!
1$
#1150
0!
```

```
1"
0,
0+
0*
0)
0(
0'
0&
0%
0$
#1200
1!
$dumpall
1!
1"
1#
0$
0%
0&
0'
0(
0)
0*
0+
0,
$end
```

Capturing Port Driver Data

Some ASIC vendors' toolkits read a VCD file format that provides details on port drivers. This information can be used, for example, to drive a tester. See the ASIC vendor's documentation for toolkit specific information.

In ModelSim use the vcd dumpports command to create a VCD file that captures port driver data. Each time an external or internal port driver changes values, a new value change is recorded in the VCD file with the following format:

```
p<state> <0 strength> <1 strength> <identifier_code>
```

Driver States

The driver states are recorded as TSSI states if the direction is known, as detailed in this table:

Table 12-4. Driver States

Input (testfixture)	Output (dut)
D low	L low
U high	H high
N unknown	X unknown
Z tri-state	T tri-state

Table 12-4. Driver States (cont.)

Input (testfixture)	Output (dut)
d low (two or more drivers active)	l low (two or more drivers active)
u high (two or more drivers active)	h high (two or more drivers active)

If the direction is unknown, the state will be recorded as one of the following:

Table 12-5. State When Direction is Unknown

Unknown direction
0 low (both input and output are driving low)
1 high (both input and output are driving high)
? unknown (both input and output are driving unknown)
F three-state (input and output unconnected)
A unknown (input driving low and output driving high)
a unknown (input driving low and output driving unknown)
B unknown (input driving high and output driving low)
b unknown (input driving high and output driving unknown)
C unknown (input driving unknown and output driving low)
c unknown (input driving unknown and output driving high)
f unknown (input and output three-stated)

Driver Strength

The recorded 0 and 1 strength values are based on Verilog strengths:

Table 12-6. Driver Strength

Strength	VHDL std_logic mappings
0 highz	’Z’
1 small	

Table 12-6. Driver Strength (cont.)

Strength	VHDL std_logic mappings
2 medium	
3 weak	
4 large	
5 pull	’W’,’H’,’L’
6 strong	’U’,’X’,’0’,’1’,’-’
7 supply	

Identifier Code

The <identifier_code> is an integer preceded by < that starts at zero and is incremented for each port in the order the ports are specified. Also, the variable type recorded in the VCD header is "port".

Resolving Values

The resolved values written to the VCD file depend on which options you specify when creating the file.

Default Behavior

By default ModelSim generates output according to IEEE 1364-2005. The standard states that the values 0 (both input and output are active with value 0) and 1 (both input and output are active with value 1) are conflict states. The standard then defines two strength ranges:

- Strong: strengths 7, 6, and 5
- Weak: strengths 4, 3, 2, 1

The rules for resolving values are as follows:

- If the input and output are driving the same value with the same range of strength, the resolved value is 0 or 1, and the strength is the stronger of the two.
- If the input is driving a strong strength and the output is driving a weak strength, the resolved value is D, d, U or u, and the strength is the strength of the input.
- If the input is driving a weak strength and the output is driving a strong strength, the resolved value is L, l, H or h, and the strength is the strength of the output.

Ignoring Strength Ranges

You may wish to ignore strength ranges and have ModelSim handle each strength separately. Any of the following options will produce this behavior:

- Use the **-no_strength_range** argument to the vcd dumpports command
- Use an optional argument to $dumpports (see Extended $dumpports Syntax below)
- Use the **+dumpports+no_strength_range** argument to vsim command

In this situation, ModelSim reports strengths for both the zero and one components of the value if the strengths are the same. If the strengths are different, ModelSim reports only the "winning" strength. In other words, the two strength values either match (e.g., pA 5 5 !) or the winning strength is shown and the other is zero (e.g., pH 0 5 !).

Extended $dumpports Syntax

ModelSim extends the $dumpports system task in order to support exclusion of strength ranges. The extended syntax is as follows:

```
$dumpports (scope_list, file_pathname, ncsim_file_index, file_format)
```

The **nc_sim_index** argument is required yet ignored by ModelSim. It is required only to be compatible with NCSim's argument list.

The **file_format** argument accepts the following values or an ORed combination thereof (see examples below):

Table 12-7. Values for file_format Argument

File_format value	Meaning
0	Ignore strength range
2	Use strength ranges; produces IEEE 1364-compliant behavior
4	Compress the EVCD output
8	Include port direction information in the EVCD file header; same as using **-direction** argument to **vcd dumpports**

Here are some examples:

```
// ignore strength range
$dumpports(top, "filename", 0, 0)
// compress and ignore strength range
$dumpports(top, "filename", 0, 4)
// print direction and ignore strength range
$dumpports(top, "filename", 0, 8)
```

```
// compress, print direction, and ignore strength range
$dumpports(top, "filename", 0, 12)
```

Example 12-5. VCD Output from vcd dumpports

This example demonstrates how **vcd dumpports** resolves values based on certain combinations of driver values and strengths and whether or not you use strength ranges. Table 12-8 is sample driver data.

Table 12-8. Sample Driver Data

time	in value	out value	in strength value (range)	out strength value (range)
0	0	0	7 (strong)	7 (strong)
100	0	0	6 (strong)	7 (strong)
200	0	0	5 (strong)	7 (strong)
300	0	0	4 (weak)	7 (strong)
900	1	0	6 (strong)	7 (strong)
27400	1	1	5 (strong)	4 (weak)
27500	1	1	4 (weak)	4 (weak)
27600	1	1	3 (weak)	4 (weak)

Given the driver data above and use of 1364 strength ranges, here is what the VCD file output would look like:

```
#0
p0 7 0 <0
#100
p0 7 0 <0
#200
p0 7 0 <0
#300
pL 7 0 <0
#900
pB 7 0 <0
#27400
pU 0 5 <0
#27500
p1 0 4 <0
#27600
p1 0 4 <0
```

Here is what the output would look like if you ignore strength ranges:

```
#0
p0 7 0 <0
#100
pL 7 0 <0
#200
pL 7 0 <0
#300
pL 7 0 <0
#900
pL 7 0 <0
#27400
pU 0 5 <0
#27500
p1 0 4 <0
#27600
pH 0 4 <0
```

Chapter 13
Tcl and Macros (DO Files)

Tcl is a scripting language for controlling and extending ModelSim. Within ModelSim you can develop implementations from Tcl scripts without the use of C code. Because Tcl is interpreted, development is rapid; you can generate and execute Tcl scripts on the fly without stopping to recompile or restart ModelSim. In addition, if ModelSim does not provide the command you need, you can use Tcl to create your own commands.

Tcl Features

Using Tcl with ModelSim gives you these features:

- command history (like that in C shells)
- full expression evaluation and support for all C-language operators
- a full range of math and trig functions
- support of lists and arrays
- regular expression pattern matching
- procedures
- the ability to define your own commands
- command substitution (that is, commands may be nested)
- robust scripting language for macros

Tcl References

Two books about Tcl are *Tcl and the Tk Toolkit* by John K. Ousterhout, published by Addison-Wesley Publishing Company, Inc., and *Practical Programming in Tcl and Tk by* Brent Welch published by Prentice Hall. You can also consult the following online references:

- Select **Help > Tcl Man Pages**.

Tcl Commands

For complete information on Tcl commands, select **Help > Tcl Man Pages**. Also see Simulator GUI Preferences for information on Tcl preference variables.

ModelSim command names that conflict with Tcl commands have been renamed or have been replaced by Tcl commands. See the list below:

Table 13-1.

Previous ModelSim command	**Command changed to (or replaced by)**
continue	run with the **-continue** option
format list | wave	write format with either list or wave specified
if	replaced by the Tcl **if** command, see If Command Syntax for more information
list	add list
nolist | nowave	delete with either list or wave specified
set	replaced by the Tcl **set** command
source	vsource
wave	add wave

Tcl Command Syntax

The following eleven rules define the syntax and semantics of the Tcl language. Additional details on If Command Syntax.

1. A Tcl script is a string containing one or more commands. Semi-colons and newlines are command separators unless quoted as described below. Close brackets ("]") are command terminators during command substitution (see below) unless quoted.
2. A command is evaluated in two steps. First, the Tcl interpreter breaks the command into words and performs substitutions as described below. These substitutions are performed in the same way for all commands. The first word is used to locate a command procedure to carry out the command, then all of the words of the command are passed to the command procedure. The command procedure is free to interpret each of its words in any way it likes, such as an integer, variable name, list, or Tcl script. Different commands interpret their words differently.
3. Words of a command are separated by white space (except for newlines, which are command separators).
4. If the first character of a word is a double-quote (") then the word is terminated by the next double-quote character. If semi-colons, close brackets, or white space characters (including newlines) appear between the quotes then they are treated as ordinary characters and included in the word. Command substitution, variable substitution, and backslash substitution are performed on the characters between the quotes as described below. The double-quotes are not retained as part of the word.

5. If the first character of a word is an open brace ({) then the word is terminated by the matching close brace (}). Braces nest within the word: for each additional open brace there must be an additional close brace (however, if an open brace or close brace within the word is quoted with a backslash then it is not counted in locating the matching close brace). No substitutions are performed on the characters between the braces except for backslash-newline substitutions described below, nor do semi-colons, newlines, close brackets, or white space receive any special interpretation. The word will consist of exactly the characters between the outer braces, not including the braces themselves.
6. If a word contains an open bracket ([) then Tcl performs command substitution. To do this it invokes the Tcl interpreter recursively to process the characters following the open bracket as a Tcl script. The script may contain any number of commands and must be terminated by a close bracket (]). The result of the script (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.

Table 13-2. Tcl Backslash Sequences

Sequence	Value
\a	Audible alert (bell) (0x7)
\b	Backspace (0x8)
\f	Form feed (0xc).
\n	Newline (0xa)
\r	Carriage-return (0xd)
\t	Tab (0x9)
\v	Vertical tab (0xb)
\<newline>whiteSpace	A single space character replaces the backslash, newline, and all spaces and tabs after the newline. This backslash sequence is unique in that it is replaced in a separate pre-pass before the command is actually parsed. This means that it will be replaced even when it occurs between braces, and the resulting space will be treated as a word separator if it isn't in braces or quotes.
\\\\	Backslash ("\\")
\ooo	The digits ooo (one, two, or three of them) give the octal value of the character.
\**x***hh*	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 question mark (?) indicates an optional argument.

Syntax

if *expr1* ?then? *body1* elseif *expr2* ?then? *body2* elseif ... ?else? ?*bodyN*?

Description

The **if** command evaluates *expr1* as an expression. The value of the expression must be a boolean (a numeric value, where 0 is false and anything else is true, or a string value such as **true** or **yes** for true and **false** or **no** for false); if it is true then *body1* is executed by passing it to the Tcl interpreter. Otherwise *expr2* is evaluated as an expression and if it is true then *body2* is executed, and so on. If none of the expressions evaluates to true then *bodyN* is executed. The **then** and **else** arguments are optional "noise words" to make the command easier to read. There may be any number of **elseif** clauses, including zero. *BodyN* may also be omitted as long as **else** is omitted too. The return value from the command is the result of the body script that was executed, or an empty string if none of the expressions was non-zero and there was no *bodyN*.

Command Substitution

Placing a command in square brackets ([ ]) will cause that command to be evaluated first and its results returned in place of the command. 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 braces ({}) is not evaluated immediately. This is important for if-then-else statements, procedures, loops, and so forth.

Tcl Relational Expression Evaluation

When you are comparing values, the following hints may be useful:

- Tcl stores all values as strings, and will convert certain strings to numeric values when appropriate. If you want a literal to be treated as a numeric value, don't quote it.

  ```
  if {[exa var_1] == 345}...
  ```

  The following will also work:

  ```
  if {[exa var_1] == "345"}...
  ```

- However, if a literal cannot be represented as a number, you *must* quote it, or Tcl will give you an error. For instance:

  ```
  if {[exa var_2] == 001Z}...
  ```

  will give an error.

  ```
  if {[exa var_2] == "001Z"}...
  ```

will work okay.

- Don't quote single characters in single quotes:

```
if {[exa var_3] == 'X'}...
```

will give an error

```
if {[exa var_3] == "X"}...
```

will work okay.

- For the equal operator, you must use the C operator (==). For not-equal, you must use the C operator (!=).

Variable Substitution

When a $<var_name> is encountered, the Tcl parser will look for variables that have been defined either by ModelSim or by you, and substitute the value of the variable.

Note
Tcl is case sensitive for variable names.

To access environment variables, use the construct:

```
$env(<var_name>)
echo My user name is $env(USER)
```

Environment variables can also be set using the env array:

```
set env(SHELL) /bin/csh
```

See Simulator State Variables for more information about ModelSim-defined variables.

System Commands

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

```
echo The date is [exec date]
```

List Processing

In Tcl a "list" is a set of strings in curly braces separated by spaces. Several Tcl commands are available for creating lists, indexing into lists, appending to lists, getting the length of lists and shifting lists. These commands are:

Table 13-3. Tcl List Commands

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

Two other commands, **lsearch** and **lsort,** are also available for list manipulation. See the Tcl man pages (**Help** > **Tcl Man Pages**) for more information on these commands.

Simulator Tcl Commands

These additional commands enhance the interface between Tcl and ModelSim. Only brief descriptions are provided here; for more information and command syntax see the Reference Manual.

Table 13-4. Simulator-Specific Tcl Commands

Command	Description
alias	creates a new Tcl procedure that evaluates the specified commands; used to create a user-defined alias
find	locates incrTcl classes and objects
lshift	takes a Tcl list as argument and shifts it in-place one place to the left, eliminating the 0th element
lsublist	returns a sublist of the specified Tcl list that matches the specified Tcl glob pattern

Table 13-4. Simulator-Specific Tcl Commands

Command	Description
printenv	echoes to the Transcript pane the current names and values of all environment variables

Simulator Tcl Time Commands

ModelSim Tcl time commands make simulator-time-based values available for use within other Tcl procedures.

Time values may optionally contain a units specifier where the intervening space is also optional. If the space is present, the value must be quoted (e.g. 10ns, "10 ns"). Time values without units are taken to be in the UserTimeScale. Return values are always in the current Time Scale Units. All time values are converted to a 64-bit integer value in the current Time Scale. This means that values smaller than the current Time Scale will be truncated to 0.

Conversions

Table 13-5. Tcl Time Conversion Commands

Command	Description
intToTime <intHi32> <intLo32>	converts two 32-bit pieces (high and low order) into a 64-bit quantity (Time in ModelSim is a 64-bit integer)
RealToTime <real>	converts a <real> number to a 64-bit integer in the current Time Scale
scaleTime <time> <scaleFactor>	returns the value of <time> multiplied by the <scaleFactor> integer

Relations

Table 13-6. Tcl Time Relation Commands

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

Table 13-7. Tcl Time Arithmetic Commands

Command	Description
addTime <time> <time>	add time
divTime <time> <time>	64-bit integer divide
mulTime <time> <time>	64-bit integer multiply
subTime <time> <time>	subtract time

Tcl Examples

This is an example of using the Tcl **while** loop to copy a list from variable *a* to variable *b*, reversing the order of the elements along the way:

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

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

```
set b [list]
foreach i $a { set b [linsert $b 0 $i] }
```

This example shows a list reversal as above, this time aborting on a particular element using the Tcl **break** command:

```
set b [list]
foreach i $a {
   if {$i = "ZZZ"} break
   set b [linsert $b 0 $i]
}
```

This example is a list reversal that skips a particular element by using the Tcl **continue** command:

```
   set b [list]
   foreach i $a {
      if {$i = "ZZZ"} continue
      set b [linsert $b 0 $i]
   }
```

The next example works in UNIX only. In a Windows environment, the Tcl **exec** command will execute compiled files only, not system commands.) The example shows how you can access system information and transfer it into VHDL variables or signals and Verilog nets or registers. When a particular HDL source breakpoint occurs, a Tcl function is called that gets the date and time and deposits it into a VHDL signal of type STRING. If a particular environment variable (DO_ECHO) is set, the function also echoes the new date and time to the transcript file by examining the VHDL variable.

(in VHDL source):

```
   signal datime : string(1 to 28) := " ";# 28 spaces
```

(on VSIM command line or in macro):

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

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

This next example shows a complete Tcl script that restores multiple Wave windows to their state in a previous simulation, including signals listed, geometry, and screen position. It also adds buttons to the Main window toolbar to ease management of the wave files.

```
   ##  This file contains procedures to manage multiple wave files.
   ##  Source this file from the command line or as a startup script.
   ##  source <path>/wave_mgr.tcl
   ##  add_wave_buttons
   ##      Add wave management buttons to the main toolbar (new, save and
   load)
   ##  new_wave
   ##      Dialog box creates a new wave window with the user provided name
   ##  named_wave <name>
   ##      Creates a new wave window with the specified title
   ##  save_wave <file-root>
   ##      Saves name, window location and contents for all open windows
   ##  wave windows
   ##      Creates <file-root><n>.do file for each window where <n> is 1
   ##      to the number of windows. Default file-root is "wave". Also
   ##       creates windowSet.do file that contains title and geometry info.
```

```
##  load_wave <file-root>
##        Opens and loads wave windows for all files matching <file-
root><n>.do
##      where <n> are the numbers from 1-9. Default <file-root> is "wave".
##       Also runs windowSet.do file if it exists.
## Add wave management buttons to the main toolbar
proc add_wave_buttons {} {
_add_menu main controls right SystemMenu SystemWindowFrame {Load Waves} \
load_wave
_add_menu main controls right SystemMenu SystemWindowFrame {Save Waves} \
save_wave
_add_menu main controls right SystemMenu SystemWindowFrame {New Wave} \
new_wave
}
## Simple Dialog requests name of new wave window. Defaults to Wave<n>

proc new_wave {} {
   global vsimPriv
   set defaultName "Wave[llength $vsimPriv(WaveWindows)]"
   set windowName [GetValue . "Create Named Wave Window:" $defaultName ]
   if {$windowName == ""} {
      # Dialog canceled
      # abort operation
      return
   }
   ## Debug
   puts "Window name: $windowName\n"
   if {$windowName == "{}"} {
      set windowName ""
   }
   if {$windowName != ""} {
      named_wave $windowName
   } else {
      named_wave $defaultName
   }
}

## Creates a new wave window with the provided name (defaults to "Wave")

proc named_wave {{name "Wave"}} {
   set newWave [view -new wave]
   if {[string length $name] > 0} {
      wm title $newWave $name
   }
}

## Writes out format of all wave windows, stores geometry and title info
in
## windowSet.do file. Removes any extra files with the same fileroot.
## Default file name is wave<n> starting from 1.
```

```
proc save_wave {{fileroot "wave"}} {
   global vsimPriv
   set n 1
   if {[catch {open windowSet_$fileroot.do w 755} fileId]} {
      error "Open failure for $fileroot ($fileId)"
   }
   foreach w $vsimPriv(WaveWindows) {
      echo "Saving: [wm title $w]"
      set filename $fileroot$n.do
      if {[file exists $filename]} {
         # Use different file
         set n2 0
         while {[file exists ${fileroot}${n}${n2}.do]} {
            incr n2
         }
         set filename ${fileroot}${n}${n2}.do
      }
      write format wave -window $w $filename
      puts $fileId "wm title $w \"[wm title $w]\""
      puts $fileId "wm geometry $w [wm geometry $w]"
      puts $fileId "mtiGrid_colconfig $w.grid name -width \
         [mtiGrid_colcget $w.grid name -width]"
      puts $fileId "mtiGrid_colconfig $w.grid value -width \
         [mtiGrid_colcget $w.grid value -width]"
      flush $fileId
      incr n
   }

   foreach f [lsort [glob -nocomplain $fileroot\[$n-9\].do]] {
         echo "Removing: $f"
         exec rm $f
      }
   }
}

## Provide file root argument and load_wave restores all saved windows.
## Default file root is "wave".

proc load_wave {{fileroot "wave"}} {
   foreach f [lsort [glob -nocomplain $fileroot\[1-9\].do]] {
      echo "Loading: $f"
      view -new wave
      do $f
   }
   if {[file exists windowSet_$fileroot.do]} {
      do windowSet_$fileroot.do
   }
}

...
```

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

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

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

```
set vhdFiles [list]
set vFiles [list]
set nbrArgs $argc
for {set x 1} {$x <= $nbrArgs} {incr x} {
   if {[string match *.vhd $1]} {
      lappend vhdFiles $1
   } else {
      lappend vFiles $1
   }
   shift
}
if {[llength $vhdFiles] > 0} {
   eval vcom -93 -explicit -noaccel $vhdFiles
}
if {[llength $vFiles] > 0} {
   eval vlog $vFiles
}
```

Macros (DO Files)

ModelSim macros (also called DO files) are simply scripts that contain ModelSim and, optionally, Tcl commands. You invoke these scripts with the **Tools > TCL > Execute Macro** menu selection or the do command.

Creating DO Files

You can create DO files, like any other Tcl script, by typing the required commands in any editor and saving the file. Alternatively, you can save the transcript as a DO file (see Saving the Transcript File).

All "event watching" commands (e.g. onbreak, onerror, etc.) must be placed before run commands within the macros in order to take effect.

The following is a simple DO file that was saved from the transcript. It is used in the dataset exercise in the ModelSim Tutorial. This DO file adds several signals to the Wave window, provides stimulus to those signals, and then advances the simulation.

```
add wave ld
add wave rst
add wave clk
add wave d
add wave q
force -freeze clk 0 0, 1 {50 ns} -r 100
force rst 1
force rst 0 10
force ld 0
force d 1010
onerror {cont}
run 1700
force ld 1
run 100
force ld 0
run 400
force rst 1
run 200
force rst 0 10
run 1500
```

Using Parameters with DO Files

You can increase the flexibility of DO files by using parameters. Parameters specify values that are passed to the corresponding parameters $1 through $9 in the macro file. For example say the macro "*testfile"* contains the line **bp** $1 $2. The command below would place a breakpoint in the source file named *design.vhd* at line 127:

```
do   testfile   design.vhd   127
```

There is no limit 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 to see the other parameters.

Deleting a File from a .do Script

To delete a file from a .do script, use the Tcl **file** command as follows:

```
file delete myfile.log
```

This will delete the file "*myfile.log*."

You can also use the **transcript file** command to perform a deletion:

```
transcript file ()
transcript file my file.log
```

The first line will close the current log file. The second will open a new log file. If it has the same name as an existing file, it will replace the previous one.

Making Macro Parameters Optional

If you want to make macro parameters optional (i.e., be able to specify fewer parameter values with the do command than the number of parameters referenced in the macro), you must use the **argc** simulator state variable. The **argc** simulator state variable returns the number of parameters passed. The examples below show several ways of using **argc**.

Example 1

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

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

Example 2

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

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

Example 3

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

```
variable vhdFiles ""
variable vFiles ""
set nbrArgs $argc
set vhdFilesExist 0
set vFilesExist   0
for {set x 1} {$x <= $nbrArgs} {incr x} {
  if {[string match *.vhd $1]} {
    set vhdFiles [concat $vhdFiles $1]
    set vhdFilesExist 1
  } else {
    set vFiles [concat $vFiles $1]
    set vFilesExist 1
  }
  shift
}
if {$vhdFilesExist == 1} {
  eval vcom -93 -explicit -noaccel $vhdFiles
}
if {$vFilesExist == 1} {
  eval vlog $vFiles
}
```

Useful Commands for Handling Breakpoints and Errors

If you are executing a macro when your simulation hits a breakpoint or causes a run-time error, ModelSim interrupts the macro and returns control to the command line. The following commands may be useful for handling such events. (Any other legal command may be executed as well.)

Table 13-8. Commands for Handling Breakpoints and Errors in Macros

command	**result**
run **-continue**	continue as if the breakpoint had not been executed, completes the run that was interrupted
onbreak	specify a command to run when you hit a breakpoint within a macro
onElabError	specify a command to run when an error is encountered during elaboration
onerror	specify a command to run when an error is encountered within a macro
status	get a traceback of nested macro calls when a macro is interrupted
abort	terminate a macro once the macro has been interrupted or paused
pause	cause the macro to be interrupted; the macro can be resumed by entering a resume command via the command line

You can also set the OnErrorDefaultAction Tcl variable to determine what action ModelSim takes when an error occurs. To set the variable on a permanent basis, you must define the variable in a *modelsim.tcl* file (see The modelsim.tcl File for details).

Error Action in DO Files

If a command in a macro returns an error, ModelSim does the following:

1. If an onerror command has been set in the macro script, ModelSim executes that command. The onerror command must be placed prior to the run command in the DO file to take effect.
2. If no **onerror** command has been specified in the script, ModelSim checks the OnErrorDefaultAction variable. If the variable is defined, its action will be invoked.
3. If neither 1 or 2 is true, the macro aborts.

Using the Tcl Source Command with DO Files

Either the **do** command or Tcl **source** command can execute a DO file, but they behave differently.

With the **source** command, the DO file is executed exactly as if the commands in it were typed in by hand at the prompt. Each time a breakpoint is hit, the Source window is updated to show the breakpoint. This behavior could be inconvenient with a large DO file containing many breakpoints.

When a **do** command is interrupted by an error or breakpoint, it does not update any windows, and keeps the DO file "locked". This keeps the Source window from flashing, scrolling, and moving the arrow when a complex DO file is executed. Typically an **onbreak resume** command is used to keep the macro running as it hits breakpoints. Add an **onbreak abort** command to the DO file if you want to exit the macro and update the Source window.

Appendix A
Simulator Variables

This appendix documents the following types of variables:

- Environment Variables — Variables referenced and set according to operating system conventions. Environment variables prepare the ModelSim environment prior to simulation.
- Simulator Control Variables — Variables used to control compiler, simulator, and various other functions.
- Simulator State Variables — Variables that provide feedback on the state of the current simulation.

Variable Settings Report

The report command 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
```

Environment Variables

Environment Variable Expansion

The shell commands vcom, vlog, vsim, and vmap, no longer expand environment variables in filename arguments and options. Instead, variables should be expanded by the shell beforehand, in the usual manner. The -f option that most of these commands support, now performs environment variable expansion throughout the file.

Environment variable expansion is still performed in the following places:

- Pathname and other values in the *modelsim.ini* file
- Strings used as file pathnames in VHDL and Verilog
- VHDL Foreign attributes
- The PLIOBJS environment variable may contain a path that has an environment variable.
- Verilog `uselib file and dir directives

- Anywhere in the contents of a -f file

The recommended method for using flexible pathnames is to make use of the MGC Location Map system (see Using Location Mapping). When this is used, then pathnames stored in libraries and project files (.mpf) will be converted to logical pathnames.

If a file or path name contains the dollar sign character ($), and must be used in one of the places listed above that accepts environment variables, then the explicit dollar sign must be escaped by using a double dollar sign ($$).

Setting Environment Variables

Before compiling or simulating, several environment variables may be set to provide the functions described below. The variables are set through the System control panel on Windows 2000 and XP machines. For UNIX, the variables are typically found in the *.login* script. The LM_LICENSE_FILE variable is required; all others are optional.

DOPATH

The toolset uses the DOPATH environment variable to search for DO files (macros). DOPATH consists of a colon-separated (semi-colon for Windows) list of paths to directories. You can override this environment variable with the DOPATH Tcl preference variable.

The DOPATH environment variable isn't accessible when you invoke vsim from a UNIX shell or from a Windows command prompt. It is accessible once ModelSim or vsim is invoked. If you need to invoke from a shell or command line and use the DOPATH environment variable, use the following syntax:

```
vsim -do "do <dofile_name>" <design_unit>
```

EDITOR

The EDITOR environment variable specifies the editor to invoke with the edit command

HOME

The toolset uses the HOME environment variable to look for an optional graphical preference file and optional location map file. Refer to Simulator Control Variables for additional information.

HOME_0IN

The HOME_0IN environment variable identifies the location of the 0-In executables directory. Refer to the 0-In documentation for more information.

LD_LIBRARY_PATH

A UNIX shell environment variable setting the search directories for shared libraries. It instructs the OS where to search for the shared libraries for FLI/PLI/VPI/DPI. This variable is used for both 32-bit and 64-bit shared libraries on Solaris/Linux systems.

LD_LIBRARY_PATH_32

A UNIX shell environment variable setting the search directories for shared libraries. It instructs the OS where to search for the shared libraries for FLI/PLI/VPI/DPI. This variable is used only for 32-bit shared libraries on Solaris/Linux systems.

LD_LIBRARY_PATH_64

A UNIX shell environment variable setting the search directories for shared libraries. It instructs the OS where to search for the shared libraries for FLI/PLI/VPI/DPI. This variable is used only for 64-bit shared libraries on Solaris/Linux systems.

LM_LICENSE_FILE

The toolset's file manager uses the LM_LICENSE_FILE environment variable to find the location of the license file. The argument may be a colon-separated (semi-colon for Windows) set of paths, including paths to other vendor license files. The environment variable is required.

MODEL_TECH

The toolset automatically sets the MODEL_TECH environment variable to the directory in which the binary executable resides; DO NOT SET THIS VARIABLE!

MODEL_TECH_TCL

The toolset uses the MODEL_TECH_TCL environment variable to find Tcl libraries for Tcl/Tk 8.3 and vsim, and may also be used to specify a startup DO file. This variable defaults to */modeltech/../tcl*, however you may set it to an alternate path

MGC_LOCATION_MAP

The toolset uses the MGC_LOCATION_MAP environment variable to find source files based on easily reallocated "soft" paths.

MODELSIM

The toolset uses the MODELSIM environment variable to find the *modelsim.ini* file. The argument consists of a path including the file name.

An alternative use of this variable is to set it to the path of a project file (*<Project_Root_Dir>/<Project_Name>.mpf*). This allows you to use project settings with command line tools. However, if you do this, the .mpf file will replace *modelsim.ini* as the initialization file for all tools.

MODELSIM_PREFERENCES

The MODELSIM_PREFERENCES environment variable specifies the location to store user interface preferences. Setting this variable with the path of a file instructs the toolset to use this file instead of the default location (your HOME directory in UNIX or in the registry in Windows). The file does not need to exist beforehand, the toolset will initialize it. Also, if this file is read-only, the toolset will not update or otherwise modify the file. This variable may contain a relative pathname – in which case the file will be relative to the working directory at the time the tool is started.

MODELSIM_TCL

The toolset uses the MODELSIM_TCL environment variable to look for an optional graphical preference file. The argument can be a colon-separated (UNIX) or semi-colon separated (Windows) list of file paths.

MTI_COSIM_TRACE

The MTI_COSIM_TRACE environment variable creates an *mti_trace_cosim* file containing debugging information about FLI/PLI/VPI function calls. You should set this variable to any value before invoking the simulator.

MTI_TF_LIMIT

The MTI_TF_LIMIT environment variable limits the size of the VSOUT temp file (generated by the toolset's kernel). Set the argument of this variable to the size of k-bytes

The environment variable TMPDIR controls the location of this file, while STDOUT controls the name. The default setting is 10, and a value of 0 specifies that there is no limit. This variable does *not* control the size of the transcript file.

MTI_RELEASE_ON_SUSPEND

The MTI_RELEASE_ON_SUSPEND environment variable allows you to turn off or modify the delay for the functionality of releasing all licenses when the tool is suspended. The default setting is 10 (in seconds), which means that if you do not set this variable your licenses will be released 10 seconds after your run is suspended. If you set this environment variable with an argument of 0 (zero) the tool will not release the licenses after being suspended. You can change the default length of time (number of seconds) by setting this environment variable to an integer greater than 0 (zero).

MTI_USELIB_DIR

The MTI_USELIB_DIR environment variable specifies the directory into which object libraries are compiled when using the **-compile_uselibs** argument to the vlog command

NOMMAP

When set to 1, the NOMMAP environment variable disables memory mapping in the toolset. You should only use this variable when running on Linux 7.1 because it will decrease the speed with which the tool reads files.

PLIOBJS

The toolset uses the PLIOBJS environment variable to search for PLI object files for loading. The argument consists of a space-separated list of file or path names

STDOUT

The argument to the STDOUT environment variable specifies a filename to which the simulator saves the VSOUT temp file information. Typically this information is deleted when the simulator exits. The location for this file is set with the TMPDIR variable, which allows you to find and delete the file in the event of a crash, because an unnamed VSOUT file is not deleted after a crash.

TMP

(Windows environments) The TMP environment variable specifies the path to a tempnam() generated file (VSOUT) containing all stdout from the simulation kernel.

TMPDIR

(UNIX environments) The TMPDIR environment variable specifies the path to a tempnam() generated file (VSOUT) containing all stdout from the simulation kernel.

Creating Environment Variables in Windows

In addition to the predefined variables shown above, you can define your own environment variables. This example shows a user-defined library path variable that can be referenced by the **vmap** command to add library mapping to the *modelsim.ini* file.

1. From your desktop, right-click your **My Computer** icon and select **Properties**
2. In the System Properties dialog box, select the Advanced tab
3. Click Environment Variables
4. In the Environment Variables dialog box and User variables for <user> pane, select New:

5. In the New User Variable dialog box, add the new variable with this data

```
Variable ame: MY_PATH
Variable value:\temp\work
```

6. OK (New User Variable, Environment Variable, and System Properties dialog boxes)

Library Mapping with Environment Variables

Once the **MY_PATH** variable is set, you can use it with the vmap command to add library mappings to the current *modelsim.ini* file.

Table A-1. Add Library Mappings to modelsim.ini File

Prompt Type	**Command**	**Result added to *modelsim.ini***
DOS prompt	vmap MY_VITAL %MY_PATH%	MY_VITAL = c:\temp\work
ModelSim or vsim prompt	vmap MY_VITAL \$MY_PATH[1]	MY_VITAL = $MY_PATH

1. The dollar sign ($) character is Tcl syntax that indicates a variable. The backslash (\) character is an escape character that prevents the variable from being evaluated during the execution of **vmap**.

You can easily add additional hierarchy to the path. For example,

vmap MORE_VITAL %MY_PATH%\more_path\and_more_path

vmap MORE_VITAL \$MY_PATH\more_path\and_more_path

Referencing Environment Variables

There are two ways to reference environment variables within ModelSim. Environment variables are allowed in a **FILE** variable being opened in VHDL. For example,

```
use std.textio.all;
entity test is end;
architecture only of test is
begin
   process
      FILE in_file : text is in "$ENV_VAR_NAME";
   begin
      wait;
   end process;
end;
```

Environment variables may also be referenced from the ModelSim command line or in macros using the Tcl **env** array mechanism:

echo "$env(ENV_VAR_NAME)"

Note
Environment variable expansion *does not* occur in files that are referenced via the **-f** argument to **vcom**, **vlog**, or **vsim**.

Removing Temp Files (VSOUT)

The *VSOUT* temp file is the communication mechanism between the simulator kernel and the Graphical User Interface. In normal circumstances the file is deleted when the simulator exits. If the tool 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.

Simulator Control Variables

Initialization (INI) files contain control variables that specify reference library paths and compiler and simulator settings. The default initialization file is *modelsim.ini* and is located in your install directory.

To set these variables, edit the initialization file directly with any text editor. The syntax for variables in the file is:

```
<variable> = <value>
```

Comments within the file are preceded with a semicolon (;).

The following sections contain information about the variables:

- Library Path Variables
- Verilog Compiler Control Variables
- VHDL Compiler Control Variables
- Simulation Control Variables

Library Path Variables

You can find these variables under the heading [Library] in the *modelsim.ini* file.

ieee

This variable sets the path to the library containing IEEE and Synopsys arithmetic packages.

- **Value Range**: any valid path; may include environment variables
- **Default**: $MODEL_TECH/../ieee

modelsim_lib

This variable sets the path to the library containing Model Technology VHDL utilities such as Signal Spy.

- **Value Range**: any valid path; may include environment variables
- **Default**: $MODEL_TECH/../modelsim_lib

std

This variable sets the path to the VHDL STD library.

- **Value Range**: any valid path; may include environment variables
- **Default**: $MODEL_TECH/../std

std_developerskit

This variable sets the path to the libraries for MGC standard developer's kit.

- **Value Range**: any valid path; may include environment variables
- **Default**: $MODEL_TECH/../std_developerskit

synopsys

This variable sets the path to the accelerated arithmetic packages.

- **Value Range**: any valid path; may include environment variables
- **Default**: $MODEL_TECH/../synopsys

sv_std

This variable sets the path to the SystemVerilog STD library.

- **Value Range**: any valid path; may include environment variables
- **Default**: $MODEL_TECH/../sv_std

verilog

This variable sets the path to the library containing VHDL/Verilog type mappings.

- **Value Range**: any valid path; may include environment variables
- **Default**: $MODEL_TECH/../verilog

vital2000

This variable sets the path to the VITAL 2000 library

- **Value Range**: any valid path; may include environment variables
- **Default**: $MODEL_TECH/../vital2000

others

This variable points to another *modelsim.ini* file whose library path variables will also be read; the pathname must include "modelsim.ini"; only one others variable can be specified in any *modelsim.ini* file.

- **Value Range**: any valid path; may include environment variables
- **Default**: none

Verilog Compiler Control Variables

You can find these variables under the heading [vlog] in the *modelsim.ini* file.

DisableOpt

This variable, when on, disables all optimizations enacted by the compiler; same as the **-O0** argument to **vlog**.

- **Value Range**: 0, 1
- **Default**: off (0)

GenerateLoopIterationMax

This variable specifies the maximum number of iterations permitted for a generate loop; restricting this permits the implementation to recognize infinite generate loops.

- **Value Range**: natural integer (>=0)
- **Default**: 100000

GenerateRecursionDepthMax

This variable specifies the maximum depth permitted for a recursive generate instantiation; restricting this permits the implementation to recognize infinite recursions.

- **Value Range**: natural integer (>=0)
- **Default**: 200

Hazard

This variable turns on Verilog hazard checking (order-dependent accessing of global variables).

- **Value Range**: 0, 1
- **Default**: off (0)

Incremental

This variable activates the incremental compilation of modules.

- **Value Range**: 0, 1
- **Default**: off (0)

MultiFileCompilationUnit

Controls how Verilog files are compiled into compilation units. Valid arguments:

- 1 -- (0n) Compiles all files on command line into a single compilation unit. This behavior is called Multi File Compilation Unit (MFCU) mode; same as -mfcu argument to
- 0 -- (Off) Default value. Compiles each file in the compilation command line into separate compilation units. This behavior is called Single File Compilation Unit (SFCU) mode.

Refer to SystemVerilog Multi-File Compilation Issues for details on the implications of these settings.

Note
The default behavior in versions prior to 6.1 was opposite of the current default behavior.

NoDebug

This variable, when on, disables the inclusion of debugging info within design units.

- **Value Range**: 0, 1
- **Default**: off (0)

Quiet

This variable turns off "loading…" messages.

- **Value Range**: 0, 1
- **Default**: off (0)

Show_BadOptionWarning

This variable instructs the tool to generate a warning whenever an unknown plus argument is encountered.

- **Value Range**: 0, 1
- **Default**: off (0)

Show_Lint

This variable instructs the tool to display lint warning messages.

- **Value Range**: 0, 1
- **Default**: off (0)

Show_WarnCantDoCoverage

This variable instructs the tool to display warning messages when the simulator encounters constructs which code coverage cannot handle.

- **Value Range**: 0,1
- **Default**: on (1)

Show_WarnMatchCadence

This variable instructs the tool to display warning messages about non-LRM compliance in order to match Cadence behavior.

- **Value Range**: 0, 1
- **Default**: on (1)

Show_source

This variable instructs the tool to show any source line containing an error.

- **Value Range**: 0, 1
- **Default**: off (0)

vlog95compat

This variable instructs the tool to disable SystemVerilog and Verilog 2001 support, making the compiler compatible with IEEE Std 1364-1995.

- **Value Range**: 0, 1

- **Default**: off (0)

VHDL Compiler Control Variables

You can find these variables under the heading [vcom].

BindAtCompile

This variable instructs the tool to perform VHDL default binding at compile time rather than load time. Refer to Default Binding for more information.

- **Value Range:** 0, 1
- **Default:** off (0)

CheckSynthesis

This variable turns on limited synthesis rule compliance checking, which includes checking only signals used (read) by a process and understanding only combinational logic, not clocked logic.

- **Value Range:** 0, 1
- **Default:** off (0)

DisableOpt

This variable disables all optimizations enacted by the compiler, similar to using the **-O0** argument to **vcom**.

- **Value Range:** 0, 1
- **Default:** off (0)

Explicit

This variable enables the resolving of ambiguous function overloading in favor of the "explicit" function declaration (not the one automatically created by the compiler for each type declaration).

- **Value Range:** 0, 1
- **Default:** on (1)

IgnoreVitalErrors

This variable instructs the tool to ignore VITAL compliance checking errors.

- **Value Range:** 0, 1

- **Default:** off (0)

NoCaseStaticError

This variable changes case statement static errors to warnings.

- **Value Range:** 0, 1
- **Default:** off (0)

NoDebug

This variable disables turns off inclusion of debugging info within design units.

- **Value Range:** 0, 1
- **Default:** off (0)

NoIndexCheck

This variable disables run time index checks.

- **Value Range:** 0, 1
- **Default:** off (0)

NoOthersStaticError

This variable disables errors caused by aggregates that are not locally static.

- **Value Range:** 0, 1
- **Default:** off (0)

NoRangeCheck

This variable disables run time range checking.

- **Value Range:** 0, 1
- **Default:** off (0)

NoVital

This variable disables acceleration of the VITAL packages.

- **Value Range:** 0, 1
- **Default:** off (0)

NoVitalCheck

This variable disables VITAL compliance checking.

- **Value Range:** 0, 1
- **Default:** off (0)

Optimize_1164

This variable disables optimization for the IEEE std_logic_1164 package.

- **Value Range:** 0, 1
- **Default:** on (1)

PedanticErrors

This variable overrides NoCaseStaticError and NoOthersStaticError

- **Value Range:** 0, 1
- **Default:** off(0)

Quiet

This variable disables the “loading…” messages.

- **Value Range:** 0, 1
- **Default:** off (0)

RequireConfigForAllDefaultBinding

This variable instructs the compiler not to generate a default binding during compilation.

- **Value Range:** 0, 1
- **Default:** off (0)

Show_Lint

This variable enables lint-style checking.

- **Value Range:** 0, 1
- **Default:** off (0)

Show_source

This variable shows source line containing error.

- **Value Range:** 0, 1
- **Default:** off (0)

Show_VitalChecksOpt

This variable enables VITAL optimization warnings.

- **Value Range:** 0, 1
- **Default:** on (1)

Show_VitalChecksWarnings

This variable enables VITAL compliance-check warnings.

- **Value Range:** 0, 1
- **Default:** on (1)

Show_WarnCantDoCoverage

This variable enables warnings when the simulator encounters constructs which code coverage cannot handle.

- **Value Range:** 0, 1
- **Default:** on (1)

Show_Warning1

This variable enables unbound-component warnings.

- **Value Range:** 0, 1
- **Default:** on (1)

Show_Warning2

This variable enables process-without-a-wait-statement warnings.

- **Value Range:** 0, 1
- **Default:** on (1)

Show_Warning3

This variable enables null-range warnings.

- **Value Range:** 0, 1

- **Default:** on (1)

Show_Warning4

This variable enables no-space-in-time-literal warnings.

- **Value Range:** 0, 1
- **Default:** on (1)

Show_Warning5

This variable enables multiple-drivers-on-unresolved-signal warnings.

- **Value Range:** 0, 1
- **Default:** on (1)

Show_Warning9

This variable enables warnings about signal value dependency at elaboration.

- **Value Range:** 0, 1
- **Default:** on (1)

Show_Warning10

This variable enables warnings about VHDL-1993 constructs in VHDL-1987 code.

- **Value Range:** 0, 1
- **Default:** on (1)

Show_WarnLocallyStaticError

This variable enables warnings about locally static errors deferred until run time.

- **Value Range:** 0, 1
- **Default:** on (1)

VHDL93

This variable enables support for VHDL-1987, where “1” enables support for VHDL-1993 and “2” enables support for VHDL-2002.

- **Value Range:** 0, 1, 2
- **Default:** 2

Simulation Control Variables

You can find these variables under the heading [vsim] in the *modelsim.ini* file.

AssertFile

This variable specifies an alternative file for storing VHDL assertion messages.

- **Value Range:** any valid filename
- **Default:** transcript

AssertionDebug

This variable specifies that SVA assertion passes are reported.

- **Value Range:** 0, 1
- **Default:** off (0)

AssertionFormat

This variable defines the format of VHDL assertion messages.

- **Value Range:**

Table A-2. AssertionFormat Variable: Accepted Values

Variable	Description
%S	severity level
%R	report message
%T	time of assertion
%D	delta
%I	instance or region pathname (if available)
%i	instance pathname with process
%O	process name
%K	kind of object path points to; returns Instance, Signal, Process, or Unknown
%P	instance or region path without leaf process
%F	file
%L	line number of assertion, or if from subprogram, line from which call is made
%%	print ’%’ character

- Default: "** %S: %R\n Time: %T Iteration: %D%I\n"

AssertionFormatBreak

This variable defines the format of messages for VHDL assertions that trigger a breakpoint.

- **Value Range:** Refer to Table A-2
- **Default:** "** %S: %R\n   Time: %T Iteration: %D %K: %i File: %F\n"

AssertionFormatError

This variable defines the format of messages for VHDL Error assertions.

If undefined, AssertionFormat is used unless assertion causes a breakpoint in which case AssertionFormatBreak is used.

- **Value Range:** Refer to Table A-2
- **Default:** "** %S: %R\n   Time: %T Iteration: %D %K: %i File: %F\n"

AssertionFormatFail

This variable defines the format of messages for VHDL Fail assertions.

If undefined, AssertionFormat is used unless assertion causes a breakpoint in which case AssertionFormatBreak is used

- **Value Range:** Refer to Table A-2
- **Default:** "** %S: %R\n   Time: %T Iteration: %D %K: %i File: %F\n"

AssertionFormatFatal

This variable defines the format of messages for VHDL Fatal assertions

If undefined, AssertionFormat is used unless assertion causes a breakpoint in which case AssertionFormatBreak is used.

- **Value Range:** Refer to Table A-2
- **Default:** "** %S: %R\n   Time: %T Iteration: %D %K: %i File: %F\n"

AssertionFormatNote

This variable defines the format of messages for VHDL Note assertions

If undefined, AssertionFormat is used unless assertion causes a breakpoint in which case AssertionFormatBreak is used

- **Value Range:** Refer to Table A-2
- **Default:** "** %S: %R\n   Time: %T Iteration: %D%I\n"

AssertionFormatWarning

This variable defines the format of messages for VHDL Warning assertions

If undefined, AssertionFormat is used unless assertion causes a breakpoint in which case AssertionFormatBreak is used

- **Value Range:** Refer to Table A-2
- **Default:** "** %S: %R\n   Time: %T Iteration: %D%I\n"

BreakOnAssertion

This variable defines the severity of VHDL assertions that cause a simulation break. It also controls any messages in the source code that use *assertion_failure_**. For example, since most runtime messages use some form of assertion_failure_*, any runtime error will cause the simulation to break if the user sets BreakOnAssertion to 2.

You can set this variable interactively in the GUI; refer to Setting Simulator Control Variables With The GUI.

- **Value Range:** 0 (note), 1 (warning), 2 (error), 3 (failure), 4 (fatal)
- **Default:** 3 (failure)

CheckPlusargs

This variable defines the simulator's behavior when encountering unrecognized plusargs.

- **Value Range:** 0 (ignores), 1 (issues warning, simulates while ignoring), 2 (issues error, exits)
- **Default:** 0 (ignores)

CheckpointCompressMode

This variable specifies that checkpoint files are written in compressed format

- **Value Range:** 0, 1
- **Default:** on (1)

CommandHistory

This variable specifies the name of a file in which to store the Main window command history.

- **Value Range:** any valid filename
- **Default:** commented out (;)

ConcurrentFileLimit

This variable controls the number of VHDL files open concurrently. This number should be less than the current limit setting for max file descriptors.

- **Value Range:** any positive integer or 0 (unlimited)
- **Default:** 40

DatasetSeparator

This variable specifies the dataset separator for fully-rooted contexts, for example:

```
sim:/top
```

The argument to DatasetSeparator must not be the same character as PathSeparator

- **Value Range:** any character except those with special meaning, such as \, {, }, etc.
- **Default:** :

DefaultForceKind

This variable defines the kind of force used when not otherwise specified.

You can set this variable interactively in the GUI; refer to Setting Simulator Control Variables With The GUI.

- **Value Range:** freeze, drive, or deposit
- **Default:** drive, for resolved signals; freeze, for unresolved signals

DefaultRadix

This variable specifies a numeric radix may be specified as a name or number. For example, you can specify binary as “binary” or “2” or octal as “octal” or “8”.

You can set this variable interactively in the GUI; refer to Setting Simulator Control Variables With The GUI.

- **Value Range:** symbolic, binary, octal, decimal, unsigned, hexadecimal, ascii
- **Default:** symbolic

DefaultRestartOptions

This variable sets the default behavior for the restart command

- **Value Range:** one or more of: -force, -noassertions, -nobreakpoint, -nofcovers, -nolist, -nolog, -nowave
- **Default:** commented out (;)

DelayFileOpen

This variable instructs the tool to open VHDL87 files on first read or write, else open files when elaborated.

- **Value Range:** 0, 1
- **Default:** off (0)

DumpportsCollapse

This variable collapses vectors (VCD id entries) in dumpports output.

- Value Range: 0, 1
- Default: off (0)

GenerateFormat

This variable controls the format of a generate statement label. Do not enclose the argument in quotation marks.

- **Value Range:** Any non-quoted string containing at a minimum a %s followed by a %d
- **Default:** %s__%d

GlobalSharedObjectsList

This variable instruct the tool to load the specified PLI/FLI shared objects with global symbol visibility.

- **Value Range:** comma separated list of filenames
- **Default:** commented out (;)

IgnoreError

This variable instructs the tool to ignore VHDL assertion errors.

You can set this variable interactively in the GUI; refer to Setting Simulator Control Variables With The GUI.

- **Value Range:** 0,1
- **Default:** off (0)

IgnoreFailure

This variable instructs the tool to ignore VHDL assertion failures.

You can set this variable interactively in the GUI; refer to Setting Simulator Control Variables With The GUI.

- **Value Range:** 0,1
- **Default:** off (0)

IgnoreNote

This variable instructs the tool to ignore VHDL assertion notes.

You can set this variable interactively in the GUI; refer to Setting Simulator Control Variables With The GUI.

- **Value Range:** 0,1
- **Default:** off (0)

IgnoreWarning

This variable instructs the tool to ignore VHDL assertion warnings.

You can set this variable interactively in the GUI; refer to Setting Simulator Control Variables With The GUI.

- **Value Range:** 0,1
- **Default:** off (0)

IterationLimit

This variable specifies a limit on simulation kernel iterations allowed without advancing time.

You can set this variable interactively in the GUI; refer to Setting Simulator Control Variables With The GUI.

- **Value Range:** positive integer
- **Default:** 5000

License

This variable controls the license file search.

- **Value Range:** one ore more of the following <license_option>, separated by spaces if using multiple entries. Refer also to the vsim <license_option>.

Table A-3. License Variable: License Options

license_option	Description
lnlonly	only use msimhdlsim and hdlsim
mixedonly	exclude single language licenses
nomgc	exclude MGC licenses
nolnl	exclude language neutral licenses
nomix	exclude msimhdlmix and hdlmix
nomti	exclude MTI licenses
noqueue	do not wait in license queue if no licenses are available
noslvhdl	exclude qhsimvh and vsim
noslvlog	exclude qhsimvl and vsimvlog
plus	only use PLUS license
vlog	only use VLOG license
vhdl	only use VHDL license

- **Default:** search all licenses

LockedMemory

For HP-UX 10.2 use only. This variable enables memory locking to speed up large designs (> 500mb memory footprint)

- **Value Range:** positive integer in units of MB.
- **Default:** disabled

MaxReportRhsCrossProducts

This variable specifies a limit on number of Cross (bin) products which are listed against a Cross when a XML or UCDB report is generated. The warning reports when any instance of unusually high number of Cross (bin) product and truncation of Cross (bin) product list for a Cross.

- **Value Range:** positive integer

- **Default:** 1000

NumericStdNoWarnings

This variable disables warnings generated within the accelerated numeric_std and numeric_bit packages.

You can set this variable interactively in the GUI; refer to Setting Simulator Control Variables With The GUI.

- **Value Range:** 0, 1
- **Default:** off (0)

OnFinish

This variable controls the behavior of the tool when it encounters $finish in the design code.

- **Value Range:**
    - **ask** —
        - In batch mode, the simulation exits.
        - In GUI mode, a dialog box pops up and asks for user confirmation on whether to quit the simulation.
    - **stop** — Causes the simulation to stay loaded in memory. This can make some post-simulation tasks easier.
    - **exit** — The simulation exits without asking for any confirmation.
- **Default: ask** — Exits in batch mode; prompts user in GUI mode.

PathSeparator

This variable specifies the character used for hierarchical boundaries of HDL modules. This variable does not affect file system paths. The argument to PathSeparator must not be the same character as DatasetSeparator.

- **Value Range:** any character except those with special meaning, such as \, {, }, etc.
- **Default:** /

PrintSimStats

This variable instructs the simulator to print the output of the simstats command upon exit. You can set this variable interactively with the -printsimstats argument to the vsim command.

- **Value Range:** 0, 1

- **Default:** 0

Resolution

This variable specifies the simulator resolution. The argument must be less than or equal to the UserTimeUnit and must not contain a space between value and units, for example:

```
Resoultion = 10fs
```

You can override this value with the -t argument to vsim. You should set a smaller resolution if your delays get truncated.

- **Value Range:** fs, ps, ns, us, ms, or sec with optional prefix of 1, 10, or 100
- **Default:** ps

RunLength

This variable specifies the default simulation length in units specified by the UserTimeUnit variable

You can set this variable interactively in the GUI; refer to Setting Simulator Control Variables With The GUI.

- **Value Range:** positive integer
- **Default:** 100

ShowFunctions

This variable sets the format for Breakpoint and Fatal error messages. When set to 1 (the default value), messages will display the name of the function, task, subprogram, module, or architecture where the condition occurred, in addition to the file and line number. Set to 0 to revert messages to previous format.

- **Value Range:** 0, 1
- **Default:** 1

SignalSpyPathSeparator

This variable specifies a unique path separator for the Signal Spy functions. The argument to SignalSpyPathSeparator must not be the same character as DatasetSeparator.

- **Value Range:** any character except those with special meaning, such as \, {, }, etc.
- **Default:** /

Startup

This variable specifies a simulation startup macro. Refer to the do command

- **Value Range:** = do <DO filename>; any valid macro (do) file
- **Default:** commented out (;)

StdArithNoWarnings

This variable suppresses warnings generated within the accelerated Synopsys std_arith packages.

You can set this variable interactively in the GUI; refer to Setting Simulator Control Variables With The GUI.

- **Value Range:** 0, 1
- **Default:** off (0)

ToggleMaxIntValues

This variable sets the maximum number of VHDL integer values to record with toggle coverage.

- **Value Range:** positive integer
- **Default:** 100

TranscriptFile

This variable specifies a file for saving command transcript. You can specify environment variables in the pathname.

- **Value Range:** any valid filename
- **Default:** transcript

UnbufferedOutput

This variable controls VHDL and Verilog files open for write.

- **Value Range:** 0 (buffered), 1 (unbuffered)
- **Default:** 0

UseCsupV2

Applies only to HP-UX 11.00 and when you compiled FLI/PLI/VPI C++ code with the -AA option for aCC.

This variable instructs **vsim** to use */usr/lib/libCsup_v2.sl* for shared object loading.

- **Value Range:** 0, 1
- **Default:** off (0)

UserTimeUnit

This variable specifies scaling for the Wave window and the default time units to use for commands such as force and run. You should generally set this variable to default, in which case it takes the value of the Resolution variable.

- **Value Range:** fs, ps, ns, us, ms, sec, or default
- **Default:** default

Veriuser

This variable specifies a list of dynamically loadable objects for Verilog PLI/VPI applications.

- **Value Range:** one or more valid shared object names
- **Default:** commented out (;)

WarnConstantChange

This variable controls whether a warning is issued when the change command changes the value of a VHDL constant or generic.

- **Value Range:** 0, 1
- **Default:** on (1)

WaveSignalNameWidth

This variable controls the number of visible hierarchical regions of a signal name shown in the Wave Window.

- **Value Range:** 0 (display full name), positive integer (display corresponding level of hierarchy)
- **Default:** 0

WLFCacheSize

This variable sets the number of megabytes for the WLF reader cache; WLF reader caching caches blocks of the WLF file to reduce redundant file I/O

- **Value Range:** positive integer

- **Default:** 0

WLFCollapseMode

This variable controls when the WLF file records values.

- **Value Range:** 0 (every change of logged object), 1 (end of each delta step), 2 (end of simulator time step)
- **Default:** 1

WLFCompress

This variable enables WLF file compression.

- **Value Range:** 0, 1
- **Default:** 1 (on)

WLFDeleteOnQuit

This variable specifies whether a WLF file should be deleted when the simulation ends.

- **Value Range:** 0, 1
- **Default:** 0 (do not delete)

WLFFilename

This variable specifies the default WLF file name.

- **Value Range:** 0, 1
- **Default:** vsim.wlf

WLFOptimize

This variable specifies whether the viewing of waveforms is optimized.

- **Value Range:** 0, 1
- **Default:** 1 (on)

WLFSaveAllRegions

This variable specifies the regions to save in the WLF file.

- **Value Range:** 0 (only regions containing logged signals), 1 (all design hierarchy)
- **Default:** 0

WLFSizeLimit

This variable limits the WLF file by size (as closely as possible) to the specified number of megabytes; if both size and time limits are specified the most restrictive is used.

You can set this variable interactively in the GUI; refer to Setting Simulator Control Variables With The GUI.

- **Value Range:** any positive integer in units of MB or 0 (unlimited)
- **Default:** 0 (unlimited)

WLFTimeLimit

This variable limits the WLF file by time (as closely as possible) to the specified amount of time. If both time and size limits are specified the most restrictive is used.

You can set this variable interactively in the GUI; refer to Setting Simulator Control Variables With The GUI.

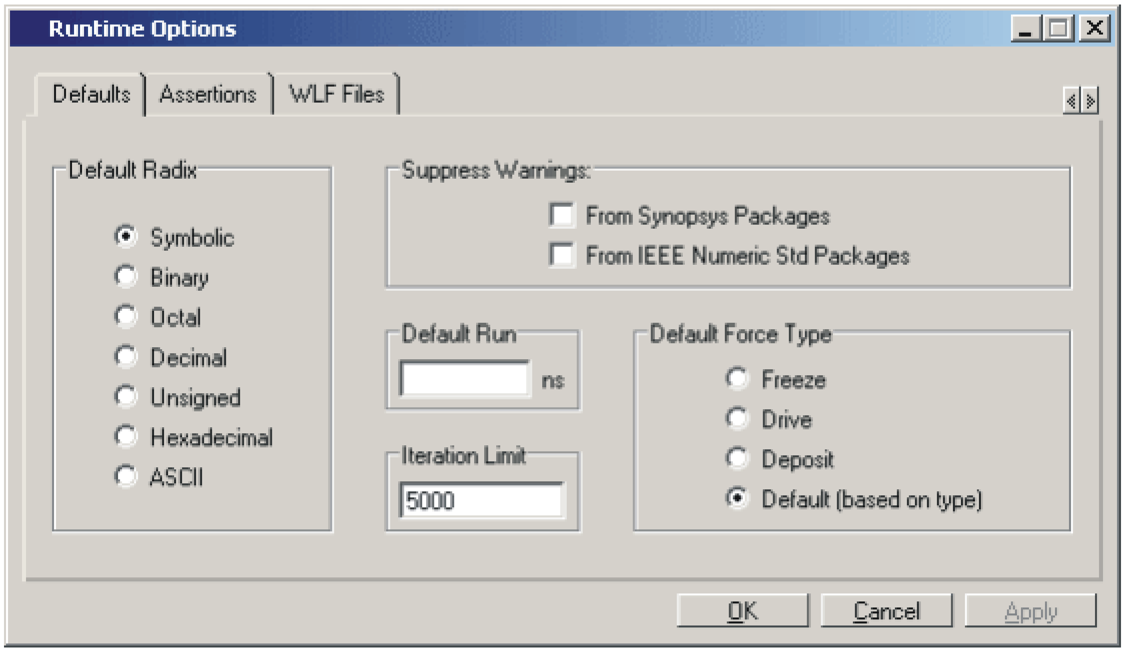
- **Value Range:** any positive integer or 0 (unlimited)
- **Default:** 0 (unlimited)

Setting Simulator Control Variables With The GUI

Changes made in the **Runtime Options** dialog are written to the active *modelsim.ini* file, if it is writable, and affect the current session as well as all future sessions. If the file is read-only, the changes affect only the current session. The **Runtime Options** dialog is accessible by selecting **Simulate > Runtime Options** in the Main window. The dialog contains three tabs - Defaults, Assertions, and WLF Files.

The Defaults tab includes these options:

Figure A-1. Runtime Options Dialog: Defaults Tab



Default Radix
[✓ Symbolic]
Binary
Octal
Decimal
Unsigned
Hexadecimal
ASCII

Suppress Warnings:
☐ From Synopsys Packages
☐ From IEEE Numeric Std Packages

Default Run
ns

Iteration Limit
5000

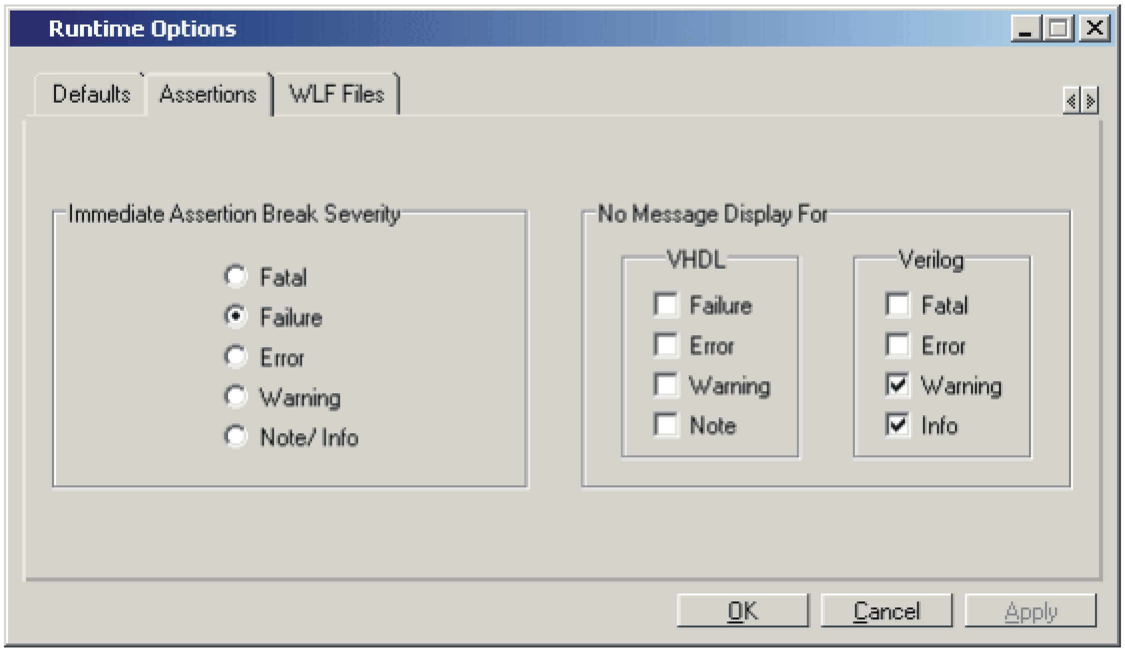
Default Force Type
Freeze
Drive
Deposit
[✓ Default (based on type)]

- **Default Radix** — Sets the default radix for the current simulation run. You can also use the radix command to set the same temporary default. The chosen radix is used for all commands (force, examine, change are examples) and for displayed values in the Objects, Locals, Dataflow, List, and Wave windows. The corresponding *modelsim.ini* variable is DefaultRadix.
- **Suppress Warnings**
  - o Selecting **From Synopsys Packages** suppresses warnings generated within the accelerated Synopsys std_arith packages. The corresponding *modelsim.ini* variable is StdArithNoWarnings.
  - o Selecting **From IEEE Numeric Std Packages** suppresses warnings generated within the accelerated numeric_std and numeric_bit packages. The corresponding *modelsim.ini* variable is NumericStdNoWarnings.
- **Default Run —** Sets the default run length for the current simulation. The corresponding *modelsim.ini* variable is RunLength.

- **Iteration Limit —** Sets a limit on the number of deltas within the same simulation time unit to prevent infinite looping. The corresponding *modelsim.ini* variable is IterationLimit.
- **Default Force Type** — Selects the default force type for the current simulation. The corresponding *modelsim.ini* variable is DefaultForceKind.

The Assertions tab includes these options:

Figure A-2. Runtime Options Dialog Box: Assertions Tab



Immediate Assertion Break Severity
- ○ Fatal
- ● Failure
- ○ Error
- ○ Warning
- ○ Note/ Info

No Message Display For

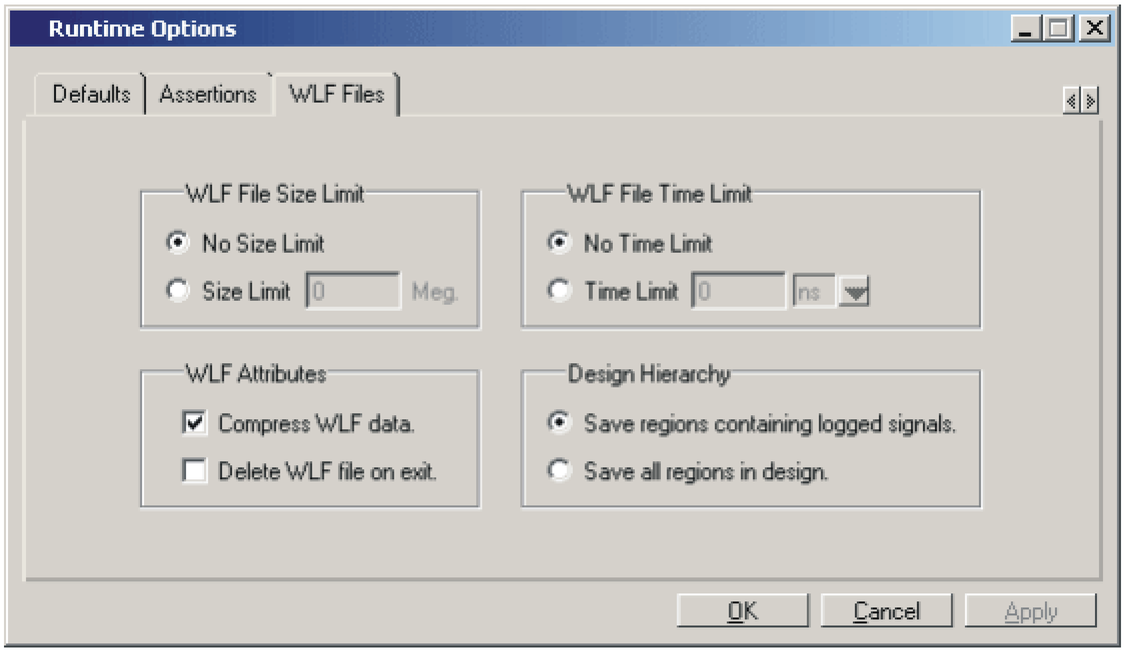
VHDL
- ☐ Failure
- ☐ Error
- ☐ Warning
- ☐ Note

Verilog
- ☐ Fatal
- ☐ Error
- ☑ Warning
- ☑ Info

- **No Message Display For -VHDL** — Selects the VHDL assertion severity for which messages will not be displayed (even if break on assertion is set for that severity). Multiple selections are possible. The corresponding *modelsim.ini* variables are IgnoreFailure, IgnoreError, IgnoreWarning, and IgnoreNote.

The WLF Files tab includes these options:

Figure A-3. Runtime Options Dialog Box, WLF Files Tab



WLF File Size Limit
[✓ No Size Limit]
Size Limit 0 Meg.

WLF File Time Limit
[✓ No Time Limit]
Time Limit 0 ns

WLF Attributes
[✓ Compress WLF data.]
Delete WLF file on exit.

Design Hierarchy
[✓ Save regions containing logged signals.]
Save all regions in design.

- **WLF File Size Limit** — Limits the WLF file by size (as closely as possible) to the specified number of megabytes. If both size and time limits are specified, the most restrictive is used. Setting it to 0 results in no limit. The corresponding *modelsim.ini* variable is WLFSizeLimit.
- **WLF File Time Limit** — Limits the WLF file by size (as closely as possible) to the specified amount of time. If both time and size limits are specified, the most restrictive is used. Setting it to 0 results in no limit. The corresponding *modelsim.ini* variable is WLFTimeLimit.
- **WLF Attributes** — Specifies whether to compress WLF files and whether to delete the WLF file when the simulation ends. You would typically only disable compression for troubleshooting purposes. The corresponding *modelsim.ini* variables are WLFCompress for compression and WLFDeleteOnQuit for WLF file deletion.
- **Design Hierarchy** — Specifies whether to save all design hierarchy in the WLF file or only regions containing logged signals. The corresponding *modelsim.ini* variable is WLFSaveAllRegions.

Message System Variables

The message system variables (located under the [msg_system] heading) help you identify and troubleshoot problems while using the application. See also Message System.

error

This variable changes the severity of the listed message numbers to "error". Refer to Changing Message Severity Level for more information.

- **Value Range:** list of message numbers
- **Default:** none

fatal

This variable changes the severity of the listed message numbers to "fatal". Refer to Changing Message Severity Level for more information.

- **Value Range:** list of message numbers
- **Default:** none

note

This variable changes the severity of the listed message numbers to "note". Refer to Changing Message Severity Level for more information

- **Value Range:** list of message numbers
- **Default:** none

suppress

This variable suppresses the listed message numbers. Refer to Changing Message Severity Level for more information

- **Value Range:** list of message numbers
- **Default:** none

warning

This variable changes the severity of the listed message numbers to "warning". Refer to Changing Message Severity Level for more information

- **Value Range:** list of message numbers
- **Default:** none

msgmode

This variable controls where the simulator outputs elaboration and runtime messages. Refer to the section “Message Viewer” for more information.

- Value Range: tran (transcript only), wlf (wlf file only), both
- Default: both

Commonly Used INI Variables

Several of the more commonly used *modelsim.ini* variables are further explained below.

Common Environment Variables

You can use environment variables in your initialization files. Use a dollar sign ($) before the environment variable name. For example:

```
[Library]
work = $HOME/work_lib
test_lib = ./$TESTNUM/work
...
[vsim]
IgnoreNote = $IGNORE_ASSERTS
IgnoreWarning = $IGNORE_ASSERTS
IgnoreError = 0
IgnoreFailure = 0
```

There is one environment variable, MODEL_TECH, that you cannot — and should not — set. MODEL_TECH is a special variable set by Model Technology software. Its value is the name of the directory from which the VCOM or VLOG compilers or VSIM simulator was invoked. MODEL_TECH is used by the other Model Technology tools to find the libraries.

Hierarchical Library Mapping

By adding an "others" clause to your *modelsim.ini* file, you can have a hierarchy of library mappings. If the ModelSim tools don't find a mapping in the *modelsim.ini* file, then they will search only the library section of the initialization file specified by the "others" clause. For example:

```
[Library]
asic_lib = /cae/asic_lib
work = my_work
others = /install_dir/modeltech/modelsim.ini
```

Since the file referred to by the "others" clause may itself contain an "others" clause, you can use this feature to chain a set of hierarchical INI files for library mappings.

Creating a Transcript File

A feature in the system initialization file allows you to keep a record of everything that occurs in the transcript: error messages, assertions, commands, command outputs, etc. To do this, set the value for the TranscriptFile line in the *modelsim.ini* file to the name of the file in which you would like to record the ModelSim history.

```
; Save the command window contents to this file
TranscriptFile = trnscrpt
```

You can disable the creation of the transcript file by using the following ModelSim command immediately after ModelSim starts:

```
transcript file ""
```

Using a Startup File

The system initialization file allows you to specify a command or a *do* file that is to be executed after the design is loaded. For example:

```
; VSIM Startup command
Startup = do mystartup.do
```

The line shown above instructs ModelSim to execute the commands in the macro file named *mystartup.do*.

```
; VSIM Startup command
Startup = run -all
```

The line shown above instructs VSIM to run until there are no events scheduled.

See the do command for additional information on creating do files.

Turning Off Assertion Messages

You can turn off assertion messages from your VHDL code by setting a switch in the *modelsim.ini* file. This option was added because some utility packages print a huge number of warnings.

```
[vsim]
IgnoreNote = 1
IgnoreWarning = 1
IgnoreError = 1
IgnoreFailure = 1
```

Turning off Warnings from Arithmetic Packages

You can disable warnings from the Synopsys and numeric standard packages by adding the following lines to the [vsim] section of the *modelsim.ini* file.

```
[vsim]
NumericStdNoWarnings = 1
StdArithNoWarnings = 1
```

Force Command Defaults

The **force** command has **-freeze**, **-drive**, and **-deposit** options. When none of these is specified, then **-freeze** is assumed for unresolved signals and **-drive** is assumed for resolved signals. But if you prefer **-freeze** as the default for both resolved and unresolved signals, you can change the defaults in the *modelsim.ini* file.

```
[vsim]
; Default Force Kind
; The choices are freeze, drive, or deposit
DefaultForceKind = freeze
```

Restart Command Defaults

The **restart** command has **-force**, **-nobreakpoint**, **-nofcovers**, **-nolist**, **-nolog**, and **-nowave** options. You can set any of these as defaults by entering the following line in the *modelsim.ini* file:

```
DefaultRestartOptions = <options>
```

where <options> can be one or more of -force, -nobreakpoint, -nofcovers, -nolist, -nolog, and -nowave.

Example:

```
DefaultRestartOptions = -nolog -force
```

VHDL Standard

You can specify which version of the 1076 Std ModelSim follows by default using the VHDL93 variable:

```
[vcom]
; VHDL93 variable selects language version as the default.
; Default is VHDL-2002.
; Value of 0 or 1987 for VHDL-1987.
; Value of 1 or 1993 for VHDL-1993.
; Default or value of 2 or 2002 for VHDL-2002.
VHDL93 = 2002
```

Opening VHDL Files

You can delay the opening of VHDL files with an entry in the *INI* file if you wish. Normally VHDL files are opened when the file declaration is elaborated. If the **DelayFileOpen** option is enabled, then the file is not opened until the first read or write to that file.

```
[vsim]
DelayFileOpen = 1
```

Variable Precedence

Note that some variables can be set in a *.modelsim* file (Registry in Windows) or a .ini file. A variable set in the *.modelsim* file takes precedence over the same variable set in a .ini file. For example, assume you have the following line in your *modelsim.ini* file:

```
TranscriptFile = transcript
```

And assume you have the following line in your *.modelsim* file:

```
set PrefMain(file) {}
```

In this case the setting in the *.modelsim* file overrides that in the *modelsim.ini* file, and a transcript file will not be produced.

Simulator State Variables

Unlike other variables that must be explicitly set, simulator state variables return a value relative to the current simulation. Simulator state variables can be useful in commands, especially when used within ModelSim DO files (macros). The variables are referenced in commands by prefixing the name with a dollar sign ($).

argc

This variable returns the total number of parameters passed to the current macro.

architecture

This variable returns the name of the top-level architecture currently being simulated; for a configuration or Verilog module, this variable returns an empty string.

configuration

This variable returns the name of the top-level configuration currently being simulated; returns an empty string if no configuration.

delta

This variable returns the number of the current simulator iteration.

entity

This variable returns the name of the top-level VHDL entity or Verilog module currently being simulated.

library

This variable returns the library name for the current region.

MacroNestingLevel

This variable returns the current depth of macro call nesting.

n

This variable represents a macro parameter, where n can be an integer in the range 1-9.

Now

This variable always returns the current simulation time with time units (e.g., 110,000 ns) Note: will return a comma between thousands.

now

This variable when time resolution is a unary unit (i.e., 1ns, 1ps, 1fs): returns the current simulation time without time units (e.g., 100000) when time resolution is a multiple of the unary unit (i.e., 10ns, 100ps, 10fs): returns the current simulation time with time units (e.g. 110000 ns) Note: will not return comma between thousands.

resolution

This variable returns the current simulation time resolution.

Referencing Simulator State Variables

Variable values may be referenced in simulator commands by preceding the variable name with a dollar sign ($). For example, to use the **now** and **resolution** variables in an **echo** command type:

```
echo "The time is $now $resolution."
```

Depending on the current simulator state, this command could result in:

```
The time is 12390 ps 10ps.
```

If you do not want the dollar sign to denote a simulator variable, precede it with a "\". For example, \$now will not be interpreted as the current simulator time.

Special Considerations for the now Variable

For the when command, special processing is performed on comparisons involving the **now** variable. If you specify "when {$now=100}...", the simulator will stop at time 100 regardless of the multiplier applied to the time resolution.

You must use 64-bit time operators if the time value of **now** will exceed 2147483647 (the limit of 32-bit numbers). For example:

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

See Simulator Tcl Time Commands for details on 64-bit time operators.

Appendix B
Location Mapping

Pathnames to source files are recorded in libraries by storing the working directory from which the compile is invoked and the pathname to the file as specified in the invocation of the compiler. The pathname may be either a complete pathname or a relative pathname.

Referencing Source Files with Location Maps

ModelSim tools that reference source files from the library locate a source file as follows:

- If the pathname stored in the library is complete, then this is the path used to reference the file.
- If the pathname is relative, then the tool looks for the file relative to the current working directory. If this file does not exist, then the path relative to the working directory stored in the library is used.

This method of referencing source files generally works fine if the libraries are created and used on a single system. However, when multiple systems access a library across a network, the physical pathnames are not always the same and the source file reference rules do not always work.

Using Location Mapping

Location maps are used to replace prefixes of physical pathnames in the library with environment variables. The location map defines a mapping between physical pathname prefixes and environment variables.

ModelSim tools open the location map file on invocation if the MGC_LOCATION_MAP environment variable is set. If MGC_LOCATION_MAP is not set, ModelSim will look for a file named *"mgc_location_map"* in the following locations, in order:

- the current directory
- your home directory
- the directory containing the ModelSim binaries
- the ModelSim installation directory

Use these two steps to map your files:

1. Set the environment variable MGC_LOCATION_MAP to the path to your location map file.
2. Specify the mappings from physical pathnames to logical pathnames:

```
$SRC
/home/vhdl/src
/usr/vhdl/src

$IEEE
/usr/modeltech/ieee
```

Pathname Syntax

The logical pathnames must begin with *$* and the physical pathnames must begin with */*. The logical pathname is followed by one or more equivalent physical pathnames. Physical pathnames are equivalent if they refer to the same physical directory (they just have different pathnames on different systems).

How Location Mapping Works

When a pathname is stored, an attempt is made to map the physical pathname to a path relative to a logical pathname. This is done by searching the location map file for the first physical pathname that is a prefix to the pathname in question. The logical pathname is then substituted for the prefix. For example, "/usr/vhdl/src/test.vhd" is mapped to "$SRC/test.vhd". If a mapping can be made to a logical pathname, then this is the pathname that is saved. The path to a source file entry for a design unit in a library is a good example of a typical mapping.

For mapping from a logical pathname back to the physical pathname, ModelSim expects an environment variable to be set for each logical pathname (with the same name). ModelSim reads the location map file when a tool is invoked. If the environment variables corresponding to logical pathnames have not been set in your shell, ModelSim sets the variables to the first physical pathname following the logical pathname in the location map. For example, if you don't set the SRC environment variable, ModelSim will automatically set it to "/home/vhdl/src".

Mapping with TCL Variables

Two Tcl variables may also be used to specify alternative source-file paths; SourceDir and SourceMap. You would define these variables in a *modelsim.tcl* file. See the The modelsim.tcl File for details.

Appendix C
Error and Warning Messages

Message System

The ModelSim message system helps you identify and troubleshoot problems while using the application. The messages display in a standard format in the Transcript pane. Accordingly, you can also access them from a saved transcript file (see Saving the Transcript File for more details).

Message Format

The format for the messages is:

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

- **SEVERITY LEVEL** — may be one of the following:

Table C-1. Severity Level Types

severity level	meaning
Note	This is an informational message.
Warning	There may be a problem that will affect the accuracy of your results.
Error	The tool cannot complete the operation.
Fatal	The tool cannot complete execution.

- **Tool** — indicates which ModelSim tool was being executed when the message was generated. For example tool could be vcom, vdel, vsim, etc.
- Group — indicates the topic to which the problem is related. For example group could be FLI, PLI, VCD, etc.

Example

```
# ** Error: (vsim-PLI-3071) ./src/19/testfile(77): $fdumplimit : Too few
arguments.
```

Getting More Information

Each message is identified by a unique MsgNum id. You can access additional information about a message using the unique id and the verror command. For example:

```
% verror 3071
Message # 3071:
Not enough arguments are being passed to the specified system task or
function.
```

Changing Message Severity Level

You can suppress or change the severity of notes, warnings, and errors that come from **vcom**, **vlog**, and **vsim**. You cannot change the severity of or suppress Fatal or Internal messages.

There are two ways to modify the severity of or suppress notes, warnings, and errors:

- Use the -error, -fatal, -note, -suppress, and -warning arguments to vcom, vlog, or vsim. See the command descriptions in the Reference Manual for details on those arguments.
- Set a permanent default in the [msg_system] section of the *modelsim.ini* file. See Simulator Control Variables for more information.

Suppressing Warning Messages

You can suppress some warning messages. For example, you may receive warning messages about unbound components about which you are not concerned.

Suppressing VCOM Warning Messages

Use the **-nowarn <number>** argument to **vcom** to suppress a specific warning message. For example:

```
vcom -nowarn 1
```

suppresses unbound component warning messages.

Alternatively, warnings may be disabled for all compiles via the *modelsim.ini* file (see Verilog Compiler Control Variables).

The warning message numbers are:

```
1 = unbound component
2 = process without a wait statement
3 = null range
4 = no space in time literal
5 = multiple drivers on unresolved signal
6 = compliance checks
7 = optimization messages
8 = lint checks
9 = signal value dependency at elaboration
10 = VHDL93 constructs in VHDL87 code
14 = locally static error deferred until simulation run
```

These numbers are category-of-warning message numbers. They are unrelated to vcom arguments that are specified by numbers, such as **vcom -87** – which disables support for VHDL-1993 and 2002.

Suppressing VLOG Warning Messages

Use the **+nowarn<CODE>** argument to **vlog** to suppress a specific warning message. Warnings that can be disabled include the <CODE> name in square brackets in the warning message. For example:

```
vlog +nowarnDECAY
```

suppresses decay warning messages.

Suppressing VSIM Warning Messages

Use the **+nowarn<CODE>** argument to **vsim** to suppress a specific warning message. Warnings that can be disabled include the <CODE> name in square brackets in the warning message. For example:

```
vsim +nowarnTFMPC
```

suppresses warning messages about too few port connections.

Exit Codes

The table below describes exit codes used by ModelSim tools.

Table C-2. Exit Codes

Exit code	Description
0	Normal (non-error) return
1	Incorrect invocation of tool
2	Previous errors prevent continuing
3	Cannot create a system process (execv, fork, spawn, etc.)
4	Licensing problem
5	Cannot create/open/find/read/write a design library
6	Cannot create/open/find/read/write a design unit
7	Cannot open/read/write/dup a file (open, lseek, write, mmap, munmap, fopen, fdopen, fread, dup2, etc.)
8	File is corrupted or incorrect type, version, or format of file
9	Memory allocation error

Table C-2. Exit Codes

Exit code	Description
10	General language semantics error
11	General language syntax error
12	Problem during load or elaboration
13	Problem during restore
14	Problem during refresh
15	Communication problem (Cannot create/read/write/close pipe/socket)
16	Version incompatibility
19	License manager not found/unreadable/unexecutable (vlm/mgvlm)
42	Lost license
43	License read/write failure
44	Modeltech daemon license checkout failure #44
45	Modeltech daemon license checkout failure #45
90	Assertion failure (SEVERITY_QUIT)
99	Unexpected error in tool
100	GUI Tcl initialization failure
101	GUI Tk initialization failure
102	GUI IncrTk initialization failure
111	X11 display error
202	Interrupt (SIGINT)
204	Illegal instruction (SIGILL)
205	Trace trap (SIGTRAP)
206	Abort (SIGABRT)
208	Floating point exception (SIGFPE)
210	Bus error (SIGBUS)
211	Segmentation violation (SIGSEGV)
213	Write on a pipe with no reader (SIGPIPE)
214	Alarm clock (SIGALRM)
215	Software termination signal from kill (SIGTERM)

Table C-2. Exit Codes

Exit code	Description
216	User-defined signal 1 (SIGUSR1)
217	User-defined signal 2 (SIGUSR2)
218	Child status change (SIGCHLD)
230	Exceeded CPU limit (SIGXCPU)
231	Exceeded file size limit (SIGXFSZ)

Miscellaneous Messages

This section describes miscellaneous messages which may be associated with ModelSim.

Compilation of DPI Export TFs Error

```
# ** Fatal: (vsim-3740) Can't locate a C compiler for compilation of
                                       DPI export tasks/functions.
```

- Description — ModelSim was unable to locate a C compiler to compile the DPI exported tasks or functions in your design.
- Suggested Action —Make sure that a C compiler is visible from where you are running the simulation.

Empty port name warning

```
# ** WARNING: [8] <path/file_name>: empty port name in port list.
```

- Description — ModelSim reports these warnings if you use the **-lint** argument to vlog. It reports the warning for any NULL module ports.
- Suggested action — If you wish to ignore this warning, do not use the **-lint** argument.

Lock message

```
waiting for lock by user@user. Lockfile is <library_path>/_lock
```

- Description — The *_lock* file is created in a library when you begin a compilation into that library, and it is removed when the compilation completes. This prevents simultaneous updates to the library. If a previous compile did not terminate properly, ModelSim may fail to remove the *_lock* file.
- Suggested action — Manually remove the *_lock* file after making sure that no one else is actually using that library.

Metavalue detected warning

```
Warning: NUMERIC_STD.">": metavalue detected, returning FALSE
```

- Description — This warning is an assertion being issued by the IEEE **numeric_std** package. It indicates that there is an 'X' in the comparison.
- Suggested action — The message does not indicate which comparison is reporting the problem since the assertion is coming from a standard package. To track the problem, note the time the warning occurs, restart the simulation, and run to one time unit before the noted time. At this point, start stepping the simulator until the warning appears. The location of the blue arrow in a Source window will be pointing at the line following the line with the comparison.

  These messages can be turned off by setting the **NumericStdNoWarnings** variable to 1 from the command line or in the *modelsim.ini* file.

Sensitivity list warning

```
signal is read by the process but is not in the sensitivity list
```

- Description — ModelSim outputs this message when you use the **-check_synthesis** argument to vcom. It reports the warning for any signal that is read by the process but is not in the sensitivity list.
- Suggested action — There are cases where you may purposely omit signals from the sensitivity list even though they are read by the process. For example, in a strictly sequential process, you may prefer to include only the clock and reset in the sensitivity list because it would be a design error if any other signal triggered the process. In such cases, your only option is to not use the **-check_synthesis** argument.

Tcl Initialization error 2

```
Tcl_Init Error 2 : Can't find a usable Init.tcl in the following
      directories :
   ./../tcl/tcl8.3 .
```

- Description — This message typically occurs when the base file was not included in a Unix installation. When you install ModelSim, you need to download and install 3 files from the ftp site. These files are:

  ```
  modeltech-base.tar.gz
  modeltech-docs.tar.gz
  modeltech-<platform>.exe.gz
  ```

  If you install only the <platform> file, you will not get the Tcl files that are located in the base file.

  This message could also occur if the file or directory was deleted or corrupted.

- Suggested action — Reinstall ModelSim with all three files.

Too few port connections

```
# ** Warning (vsim-3017): foo.v(1422): [TFMPC] - Too few port
                                 connections. Expected 2, found 1.
# Region: /foo/tb
```

- Description — This warning occurs when an instantiation has fewer port connections than the corresponding module definition. The warning doesn't necessarily mean anything is wrong; it is legal in Verilog to have an instantiation that doesn't connect all of the pins. However, someone that expects all pins to be connected would like to see such a warning.

  Here are some examples of legal instantiations that will and will not cause the warning message.

  Module definition:

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

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

  ```
  foo inst1(e, f, g, ); // positional association
  foo inst1(.a(e), .b(f), .c(g), .d()); // named association
  ```

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

  ```
  foo inst1(e, f, g); // positional association
  foo inst1(.a(e), .b(f), .c(g)); // named association
  ```

  Any instantiation above will leave pin *d* unconnected but the first example has a placeholder for the connection. Here's another example:

  ```
  foo inst1(e, , g, h);
  foo inst1(.a(e), .b(), .c(g), .d(h));
  ```

- Suggested actions —
  - Check that there is not an extra comma at the end of the port list. (e.g., model(a,b,)). The extra comma is legal Verilog and implies that there is a third port connection that is unnamed.
  - If you are purposefully leaving pins unconnected, you can disable these messages using the **+nowarnTFMPC** argument to vsim.

VSIM license lost

```
Console output:
Signal 0 caught... Closing vsim vlm child.
vsim is exiting with code 4
FATAL ERROR in license manager

transcript/vsim output:
# ** Error: VSIM license lost; attempting to re-establish.
#    Time: 5027 ns  Iteration: 2
# ** Fatal: Unable to kill and restart license process.
#    Time: 5027 ns  Iteration: 2
```

- Description — ModelSim queries the license server for a license at regular intervals. Usually these "License Lost" error messages indicate that network traffic is high, and communication with the license server times out.
- Suggested action — Anything you can do to improve network communication with the license server will probably solve or decrease the frequency of this problem.

Enforcing Strict 1076 Compliance

The optional **-pedanticerrors** argument to vcom enforces strict compliance to the IEEE 1076 LRM in the cases listed below. The default behavior for these cases is to issue an insuppressible warning message. If you compile with **-pedanticerrors**, the warnings change to an error, unless otherwise noted. Descriptions in quotes are actual warning/error messages emitted by **vcom**. As noted, in some cases you can suppress the warning using **-nowarn [level]**.

- Type conversion between array types, where the element subtypes of the arrays do not have identical constraints.
- "Extended identifier terminates at newline character (0xa)."
- "Extended identifier contains non-graphic character 0x%x."
- "Extended identifier \"%s\" contains no graphic characters."
- "Extended identifier \"%s\" did not terminate with backslash character."
- "An abstract literal and an identifier must have a separator between them."

  This is for forming physical literals, which comprise an optional numeric literal, followed by a separator, followed by an identifier (the unit name). Warning is level 4, which means "-nowarn 4" will suppress it.
- In VHDL 1993 or 2002, a subprogram parameter was declared using VHDL 1987 syntax (which means that it was a class VARIABLE parameter of a file type, which is the only way to do it in VHDL 1987 and is illegal in later VHDLs). Warning is level 10.
- "Shared variables must be of a protected type." Applies to VHDL 2002 only.

- Expressions evaluated during elaboration cannot depend on signal values. Warning is level 9.
- "Non-standard use of output port '%s' in PSL expression." Warning is level 11.
- "Non-standard use of linkage port '%s' in PSL expression." Warning is level 11.
- Type mark of type conversion expression must be a named type or subtype, it can't have a constraint on it.
- When the actual in a PORT MAP association is an expression, it must be a (globally) static expression. The port must also be of mode IN.
- The expression in the CASE and selected signal assignment statements must follow the rules given in 8.8 of the LRM. In certain cases we can relax these rules, but **-pedanticerrors** forces strict compliance.
- A CASE choice expression must be a locally static expression. We allow it to be only globally static, but **-pedanticerrors** will check that it is locally static. Same rule for selected signal assignment statement choices. Warning level is 8.
- When making a default binding for a component instantiation, ModelSim's non-standard search rules found a matching entity. VHDL 2002 LRM Section 5.2.2 spells out the standard search rules. Warning level is 1.
- Both FOR GENERATE and IF GENERATE expressions must be globally static. We allow non-static expressions unless **-pedanticerrors** is present.
- When the actual part of an association element is in the form of a conversion function call [or a type conversion], and the formal is of an unconstrained array type, the return type of the conversion function [type mark of the type conversion] must be of a constrained array subtype. We relax this (with a warning) unless **-pedanticerrors** is present when it becomes an error.
- OTHERS choice in a record aggregate must refer to at least one record element.
- In an array aggregate of an array type whose element subtype is itself an array, all expressions in the array aggregate must have the same index constraint, which is the element's index constraint. No warning is issued; the presence of **-pedanticerrors** will produce an error.
- Non-static choice in an array aggregate must be the only choice in the only element association of the aggregate.
- The range constraint of a scalar subtype indication must have bounds both of the same type as the type mark of the subtype indication.
- The index constraint of an array subtype indication must have index ranges each of whose both bounds must be of the same type as the corresponding index subtype.
- When compiling VHDL 1987, various VHDL 1993 and 2002 syntax is allowed. Use **-pedanticerrors** to force strict compliance. Warnings are all level 10.

Appendix D
Verilog PLI/VPI/DPI

This appendix describes the ModelSim implementation of the Verilog PLI (Programming Language Interface), VPI (Verilog Procedural Interface) and SystemVerilog DPI (Direct Programming Interface). These three interfaces provide a mechanism for defining tasks and functions that communicate with the simulator through a C procedural interface. There are many third party applications available that interface to Verilog simulators through the PLI (see Third Party PLI Applications). In addition, you may write your own PLI/VPI/DPI applications.

Implementation Information

ModelSim Verilog implements the PLI as defined in the IEEE Std 1364-2001, with the exception of the **acc_handle_datapath()** routine. We did not implement the **acc_handle_datapath()** routine because the information it returns is more appropriate for a static timing analysis tool.

The VPI is partially implemented as defined in the IEEE Std 1364-2005. The list of currently supported functionality can be found in the following file:

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

ModelSim SystemVerilog implements DPI as defined in IEEE Std P1800-2005.

The IEEE Std 1364 is the reference that defines the usage of the PLI/VPI routines, and the IEEE Std P1800-2005 Language Reference Manual (LRM) defines the usage of DPI routines. This manual describes only the details of using the PLI/VPI/DPI with ModelSim Verilog and SystemVerilog.

g++ Compiler Support for use with PLI/VPI/DPI

We strongly encourage that unless you have a reason to do otherwise, you should use the built-in g++ compiler that is shipped with the ModelSim compiler to compile your C++ code. This is the version that has been tested and is supported for any given release.

Specifying Your Own g++ Compiler

If you must use a different g++ compiler, other than that shipped with ModelSim, you need to set a variable in your modelsim.ini file, as follows:

```
CppPath = /usr/bin/g++
```

to point to the desired g++ version.

Registering PLI Applications

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

```
typedef int (*p_tffn)();
typedef struct t_tfcell {
   short type;/* USERTASK, USERFUNCTION, or USERREALFUNCTION */
   short data;/* passed as data argument of callback function */
   p_tffn checktf;  /* argument checking callback function */
   p_tffn sizetf;   /* function return size callback function */
   p_tffn calltf;   /* task or function call callback function */
   p_tffn misctf;   /* miscellaneous reason callback function */
   char *tfname;/* name of system task or function */
      /* The following fields are ignored by ModelSim Verilog */
   int forwref;
   char *tfveritool;
   char *tferrmessage;
   int hash;
   struct t_tfcell *left_p;
   struct t_tfcell *right_p;
   char *namecell_p;
   int warning_printed;
} s_tfcell, *p_tfcell;
```

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

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

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

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

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

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

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

It is an error if a PLI shared library does not contain a veriusertfs array or an init_usertfs function.

Since PLI applications are dynamically loaded by the simulator, you must specify which applications to load (each application must be a dynamically loadable library, see Compiling and Linking C Applications for PLI/VPI/DPI). The PLI applications are specified as follows (note that on a Windows platform the file extension would be .dll):

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

  Veriuser = pliapp1.so pliapp2.so pliappn.so

- As a list in the PLIOBJS environment variable:

  % setenv PLIOBJS "pliapp1.so pliapp2.so pliappn.so"

- As a -pli argument to the simulator (multiple arguments are allowed):

  -pli pliapp1.so -pli pliapp2.so -pli pliappn.so

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

Registering VPI Applications

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

Example D-1. VPI Application Registration

```
PLI_INT32 MyFuncCalltf( PLI_BYTE8 *user_data )
{ ... }
PLI_INT32 MyFuncCompiletf( PLI_BYTE8 *user_data )
{ ... }
PLI_INT32 MyFuncSizetf( PLI_BYTE8 *user_data )
{ ... }
PLI_INT32 MyEndOfCompCB( p_cb_data cb_data_p )
{ ... }
PLI_INT32 MyStartOfSimCB( p_cb_data cb_data_p )
{ ... }
void RegisterMySystfs( void )
  {

      vpiHandle tmpH;
      s_cb_data callback;
      s_vpi_systf_data systf_data;

      systf_data.type        = vpiSysFunc;
      systf_data.sysfunctype = vpiSizedFunc;
      systf_data.tfname      = "$myfunc";
      systf_data.calltf      = MyFuncCalltf;
      systf_data.compiletf   = MyFuncCompiletf;
      systf_data.sizetf      = MyFuncSizetf;
      systf_data.user_data   = 0;
      tmpH = vpi_register_systf( &systf_data );
      vpi_free_object(tmpH);

      callback.reason    = cbEndOfCompile;
      callback.cb_rtn    = MyEndOfCompCB;
      callback.user_data = 0;
      tmpH = vpi_register_cb( &callback );
      vpi_free_object(tmpH);

      callback.reason    = cbStartOfSimulation;
      callback.cb_rtn    = MyStartOfSimCB;
      callback.user_data = 0;
      tmpH = vpi_register_cb( &callback );
      vpi_free_object(tmpH);
  }

void (*vlog_startup_routines[ ] ) () = {
   RegisterMySystfs,
      0    /* last entry must be 0 */
};
```

Loading VPI applications into the simulator is the same as described in Registering PLI Applications.

Using PLI and VPI Together

PLI and VPI applications can co-exist in the same application object file. In such cases, the applications are loaded at startup as follows:

- If an init_usertfs() function exists, then it is executed and only those system tasks and functions registered by calls to mti_RegisterUserTF() will be defined.
- If an init_usertfs() function does not exist but a veriusertfs table does exist, then only those system tasks and functions listed in the veriusertfs table will be defined.
- If an init_usertfs() function does not exist and a veriusertfs table does not exist, but a vlog_startup_routines table does exist, then only those system tasks and functions and callbacks registered by functions in the vlog_startup_routines table will be defined.

As a result, when PLI and VPI applications exist in the same application object file, they must be registered in the same manner. VPI registration functions that would normally be listed in a vlog_startup_routines table can be called from an init_usertfs() function instead.

Registering DPI Applications

DPI applications do not need to be registered. However, each DPI imported or exported task or function must be identified using SystemVerilog ‘import “DPI-C”’ or ‘export “DPI-C”’syntax. Examples of the syntax follow:

```
export "DPI-C" task t1;
task t1(input int i, output int o);
.
.
.
end task
import "DPI-C" function void f1(input int i, output int o);
```

Your code must provide imported functions or tasks, compiled with an external compiler. An imported task must return an int value, "1" indicating that it is returning due to a disable, or "0" indicating otherwise.

These imported functions or objects may then be loaded as a shared library into the simulator with either the command line option **-sv_lib <lib>** or **-sv_liblist <bootstrap_file>**. For example,

```
vlog dut.v
gcc -shared -Bsymbolic -o imports.so imports.c
vsim -sv_lib imports top -do <do_file>
```

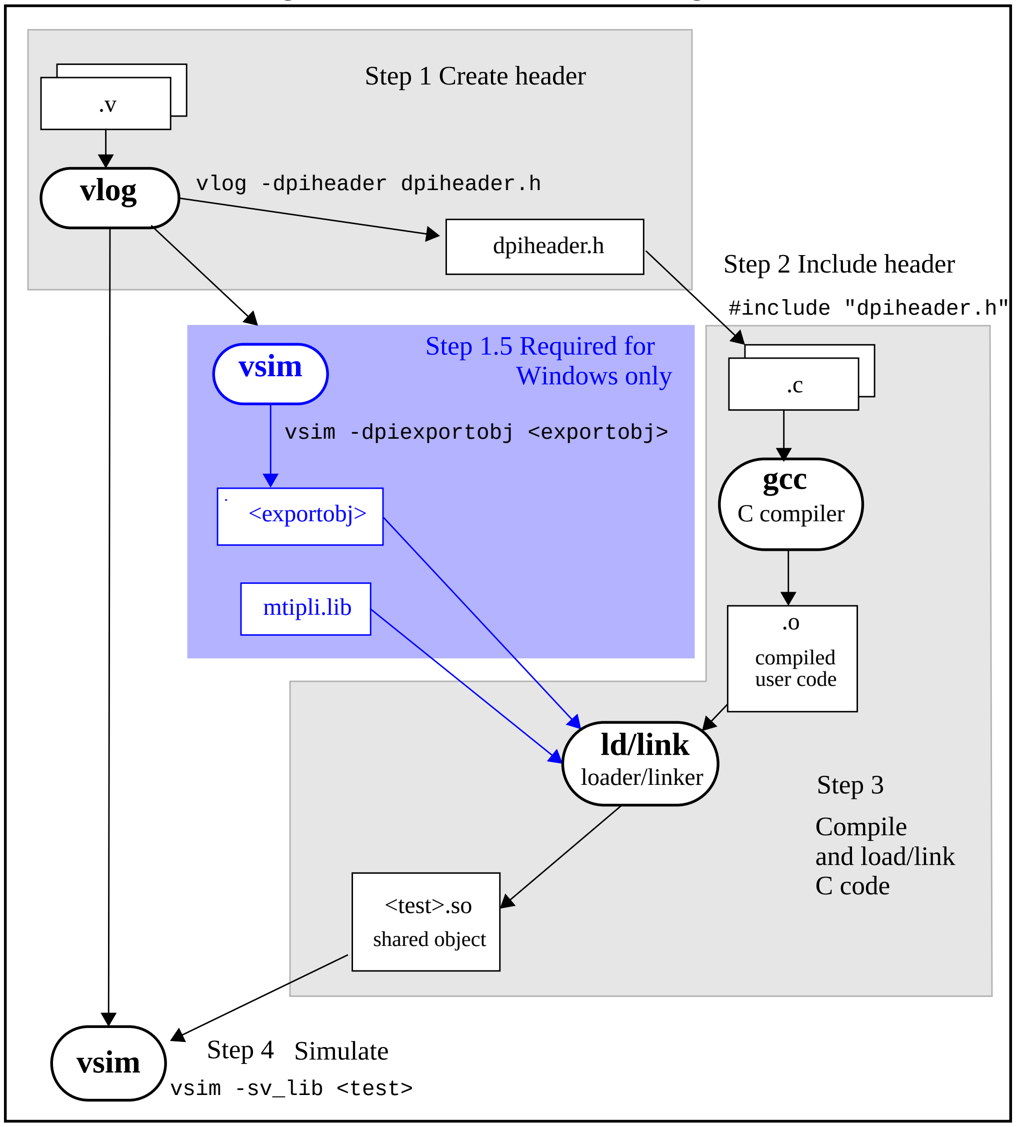
The **-sv_lib** option specifies the shared library name, without an extension. A file extension is added by the tool, as appropriate to your platform. For a list of file extensions accepted by platform, see DPI File Loading.

You can also use the command line options **-sv_root** and **-sv_liblist** to control the process for loading imported functions and tasks. These options are defined in the IEEE Std P1800-2005 LRM.

DPI Use Flow

Correct use of ModelSim DPI depends on the flow presented in this section.

Figure D-1. DPI Use Flow Diagram



1. Run vlog to generate a *dpiheader.h* file.

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

```
vlog -dpiheader <dpiheader>.h files.v
```

2. **Required for Windows only;** Run a preliminary invocation of vsim with the **-dpiexportobj** argument.

   Because of limitations with the linker/loader provided on Windows, this additional step is required. You must create the exported task/function compiled object file (*exportobj*) by running a preliminary vsim command, such as:

   ```
   vsim -dpiexportobj exportobj top
   ```

3. Include the *dpiheader.h* file in your C code.

   ModelSim recommends that any user DPI C code that accesses exported tasks/functions, or defines imported tasks/functions, will include the *dpiheader.h* file. This allows the C compiler to verify the interface between C and ModelSim.

4. Compile the C code into a shared object.

   Compile your code, providing any *.a* or other *.o* files required.

   For Windows users — In this step, the object file needs to be bound together with the *.obj* that you created using the **-dpiexportobj** argument, into a single *.dll* file.

5. Simulate the design.

   When simulating, specify the name of the imported DPI C shared object (according to the SystemVerilog LRM). For example:

   ```
   vsim -sv_lib <test> top
   ```

When Your DPI Export Function is Not Getting Called

This issue can arise in your C code due to the way the C linker resolves symbols. It happens if a name you choose for a SystemVerilog export function happens to match a function name in a custom, or even standard C library. In this case, your C compiler will bind calls to the function in that C library, rather than to the export function in the SystemVerilog simulator.

The symptoms of such a misbinding can be difficult to detect. Generally, the misbound function silently returns an unexpected or incorrect value.

To determine if you have this type of name aliasing problem, consult the C library documentation (either the online help or man pages).

Simplified Import of FLI / PLI / C Library Functions

In addition to the traditional method of importing FLI / PLI / C library functions, a simplified method can be used: you can declare VPI and FLI functions as DPI-C imports. When you declare VPI and FLI functions as DPI-C imports, the DPI shared object is loaded at runtime automatically. Neither the C implementation of the import tf, nor the **-sv_lib** argument is required.

Also, on most platforms (see Platform Specific Information), you can declare most standard C library functions as DPI-C imports.

The following example is processed directly, without DPI C code:

```
package cmath;
    import "DPI-C" function real sin(input real x);
    import "DPI-C" function real sqrt(input real x);
endpackage

package fli;
    import "DPI-C" function mti_Cmd(input string cmd);
endpackage

module top;
    import cmath::*;
    import fli::*;
    int status, A;
    initial begin
        $display("sin(0.98) = %f", sin(0.98));
        $display("sqrt(0.98) = %f", sqrt(0.98));
        status = mti_Cmd("change A 123");
        $display("A = %1d, status = %1d", A, status);
    end
endmodule
```

To simulate, you would simply enter a command such as: **vsim top**.

Platform Specific Information

This feature is not supported on AIX.

On Windows, only FLI and PLI commands may be imported in this fashion. C library functions are not automatically importable. They must be wrapped in user DPI C functions, which are brought into the simulator using the **-sv_lib** argument.

Use Model for Read-Only Work Libraries

You may want to create the work library as a read-only entity, which enables multiple users to simultaneously share the design library at runtime. The steps are as follows:

- Windows and RS6000/RS64

  On these platforms, simply change the permissions on the design library to read only by issuing a command such as "chmod -R a-w <libname>". Do this after you have finished compiling with vlog/vcom and vopt.

- All Other Platforms

  If a design contains no DPI export tasks or functions, the work library can be changed by simply changing the permissions, as shown for win32 and rs6000/rs64 above.

  For designs that contain DPI export tasks and functions, and are not run on Windows or RS6000/RS64, by default vsim creates a shared object in directory *<libname>/_dpi*. This shared object is called *exportwrapper.so* (Linux and Solaris) or *exportwrapper.sl (*hp700, hppa64, and hpux_ia64). If you are using a read-only library, **vsim** must not create any objects in the library.

To prevent **vsim** from creating objects in the library at runtime, the **vsim -dpiexportobj** flow is available on all platforms. Use this flow after compilation, but before you start simulation using the design library.

An example command sequence on Linux would be:

```
vlib work
vlog -dpiheader dpiheader.h test.sv
gcc -shared -Bsymbolic -o test.so test.c
vsim -c -dpiexportobj work/_dpi/exportwrapper top
chmod -R a-w work
```

The library is now ready for simulation by multiple simultaneous users, as follows:

```
vsim top -sv_lib test
```

The work/_dpi/exportwrapper argument provides a basename for the shared object.

At runtime, **vsim** automatically checks to see if the file *work/_dpi/exportwrapper.so* is up-to-date with respect to its C source code. If it is out of date, an error message is issued and elaboration stops.

Compiling and Linking C Applications for PLI/VPI/DPI

The following platform-specific instructions show you how to compile and link your PLI/VPI/DPI C applications so that they can be loaded by ModelSim. Various native C/C++ compilers are supported on different platforms. The gcc compiler is supported on all platforms.

The following PLI/VPI/DPI routines are declared in the include files located in the ModelSim *<install_dir>/modeltech/include* directory:

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

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

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

For all UNIX Platforms

The information in this section applies to all UNIX platforms.

app.so

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

- Add a path before *app.so* in the command line option or control variable (The path may include environment variables.)
- Put the path in a UNIX shell environment variable:

  LD_LIBRARY_PATH_32= <*library path without filename*> (for Solaris/Linux 32-bit)

  or

  LD_LIBRARY_PATH_64= <*library path without filename*> (for Solaris 64-bit)

  or

  SHLIB_PATH= <*library path without filename*> (for HP-UX)

Correct Linking of Shared Libraries with -Bsymbolic

In the examples shown throughout this appendix, the **-Bsymbolic** linker option is used with the compilation (**gcc** or **g++**) or link (**ld**) commands to correctly resolve symbols. This option instructs the linker to search for the symbol within the local shared library and bind to that symbol if it exists. If the symbol is not found within the library, the linker searches for the symbol within the vsimk executable and binds to that symbol, if it exists.

When using the **-Bsymbolic** option, the linker may warn about symbol references that are not resolved within the local shared library. It is safe to ignore these warnings, provided the symbols are present in other shared libraries or the vsimk executable. (An example of such a warning would be a reference to a common API call such as `vpi_printf()`).

Windows Platforms

- Microsoft Visual C 4.1 or Later

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

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

When executing **cl** commands in a DO file, use the **/NOLOGO** switch to prevent the Microsoft C compiler from writing the logo banner to stderr. Writing the logo causes Tcl to think an error occurred.

- MinGW gcc 3.2.3

```
gcc -c -I<install_dir>\include app.c
gcc -shared -Bsymbolic -o app.dll app.o -L<install_dir>\win32 -lmtipli
```

The ModelSim tool requires the use of MinGW gcc compiler rather than the Cygwin gcc compiler. MinGW gcc is available on the ModelSim FTP site. Remember to add the path to your gcc executable in the Windows environment variables.

DPI Imports on Windows Platforms

When linking the shared objects, be sure to specify one export option for each DPI imported task or function in your linking command line. You can use the **-isymfile** argument from the vlog command to obtain a complete list of all imported tasks/functions expected by ModelSim.

As an alternative to specifying one -export option for each imported task or function, you can make use of the __declspec (dllexport) macro supported by Visual C. You can place this macro before every DPI import task or function declaration in your C source. All the marked functions will be available for use by **vsim** as DPI import tasks and functions.

DPI Flow for Exported Tasks and Functions on Windows Platforms

Since the Windows platform lacks the necessary runtime linking capabilities, you must perform an additional manual step in order to prepare shared objects containing calls to exported SystemVerilog tasks or functions. You need to invoke a special run of vsim. The command is as follows:

```
vsim <top du list> -dpiexportobj <objname> <other args>
```

The **-dpiexportobj** generates an object file <objname>.obj that contains "glue" code for exported tasks and functions. You must add that object file to the link line for your *.dll*, listed after the other object files. For example, a link line for MinGW would be:

```
gcc -shared -Bsymbolic -o app.dll app.obj <objname>.obj
        -L<install_dir>\modeltech\win32 -lmtipli
```

and a link line for Visual C would be:

```
link -dll -export:<init_function> app.obj <objname>.obj\
        <install_dir>\modeltech\win32\mtipli.lib -out:app.dll
```

32-bit Linux Platform

If your PLI/VPI/DPI application uses anything from a system library, you will need to specify that library when you link your PLI/VPI/DPI application. For example, to use the standard C library, specify '-lc' to the 'ld' command.

- gcc compiler

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

  If you are using ModelSim with RedHat version 7.1 or below, you also need to add the **-noinhibit-exec** switch when you specify **-Bsymbolic**.

  The compiler switch -freg-struct-return must be used when compiling any FLI application code that contains foreign functions that return real or time values.

64-bit Linux for IA64 Platform

64-bit Linux is supported on RedHat Linux Advanced Workstation 2.1 for Itanium 2.

- gcc compiler (gcc 3.2 or later)

  ```
  gcc -c -fPIC -I/<install_dir>/modeltech/include app.c
  ld -shared -Bsymbolic -E --allow-shlib-undefined -o app.so app.o
  ```

  If your PLI/VPI/DPI application requires a user or vendor-supplied C library, or an additional system library, you will need to specify that library when you link your PLI/VPI/DPI application. For example, to use the system math library libm, specify -lm to the ld command:

  ```
  gcc -c -fPIC -I/<install_dir>/modeltech/include math_app.c
  ld -shared -Bsymbolic -E --allow-shlib-undefined -o math_app.so math_app.o -lm
  ```

64-bit Linux for Opteron/Athlon 64 and EM64T Platforms

64-bit Linux is supported on RedHat Linux EWS 3.0 for Opteron/Athlon 64 and EM64T.

- gcc compiler (gcc 3.2 or later)

  ```
  gcc -c -fPIC -I/<install_dir>/modeltech/include app.c
  ld -shared -Bsymbolic -E --allow-shlib-undefined -o app.so app.o
  ```

  To compile for 32-bit operation, specify the -m32 argument on the gcc command line.

  If your PLI/VPI/DPI application requires a user or vendor-supplied C library, or an additional system library, you will need to specify that library when you link your

PLI/VPI/DPI application. For example, to use the system math library libm, specify -lm to the ld command:

```
gcc -c -fPIC -I/<install_dir>/modeltech/include math_app.c
ld -shared -Bsymbolic -E --allow-shlib-undefined -o math_app.so math_app.o -lm
```

32-bit Solaris Platform

If your PLI/VPI/DPI application uses anything from a system library, you will need to specify that library when you link your PLI/VPI/DPI application. For example, to use the standard C library, specify '-lc' to the 'ld' command.

- gcc compiler

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

- cc compiler

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

64-bit Solaris Platform

- gcc compiler

```
gcc -c -I<install_dir>/modeltech/include -m64 -fPIC app.c
gcc -shared -Bsymbolic -o app.so -m64 app.o
```

  This was tested with gcc 3.2.2. You may need to add the location of *libgcc_s.so.1* to the LD_LIBRARY_PATH_64 environment variable.

- cc compiler

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

32-bit HP700 Platform

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

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

- gcc compiler

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

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

- cc compiler

```
cc -c +z +DD32 -I/<install_dir>/modeltech/include app.c
ld -b -o app.sl app.o -lc
```

64-bit HP Platform

- cc compiler

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

64-bit HP for IA64 Platform

- cc compiler (/opt/ansic/bin/cc, /usr/ccs/bin/ld)

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

If your PLI/VPI/DPI application requires a user or vendor-supplied C library, or an additional system library, you will need to specify that library when you link your PLI/VPI/DPI application. For example, to use the system math library, specify '-lm' to the 'ld' command:

```
cc -c +DD64 -I/<install_dir>/modeltech/include math_app.c
ld -b -o math_app.sl math_app.o -lm
```

32-bit IBM RS/6000 Platform

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

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

- gcc compiler

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

- cc compiler

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

The *app.exp* file must export the PLI/VPI initialization function or table. For the PLI, the exported symbol should be "init_usertfs". Alternatively, if there is no init_usertfs function, then the exported symbol should be "veriusertfs". For the VPI, the exported symbol should be "vlog_startup_routines". These requirements ensure that the appropriate symbol is exported, and thus ModelSim can find the symbol when it dynamically loads the shared object.

DPI Imports on 32-bit IBM RS/6000 Platform

When linking the shared objects, be sure to specify **-bE:<isymfile>** option on the link command line. <isymfile> is the name of the file generated by the**-isymfile** argument to the vlog command. Once you have created the <isymfile>, it contains a complete list of all imported tasks and functions expected by ModelSim.

DPI Flow for Exported Tasks and Functions on 32-bit IBM RS/6000 Platform

Since the RS6000 platform lacks the necessary runtime linking capabilities, you must perform an additional manual step in order to prepare shared objects containing calls to exported SystemVerilog tasks or functions shared object file. You need to invoke a special run of vsim. The command is as follows:

```
vsim <top du list> -dpiexportobj <objname> <other args>
```

The **-dpiexportobj** generates the object file <objname>.o that contains "glue" code for exported tasks and functions. You must add that object file to the link line, listed after the other object files. For example, a link line would be:

```
ld -o app.so app.o <objname>.o
        -bE:<isymfile> -bI:/<install_dir>/modeltech/rs6000/mti_exports -bM:SRE -bnoentry -lc
```

64-bit IBM RS/6000 Platform

Only versions 5.1 and later of AIX support the 64-bit platform. A gcc 64-bit compiler is not available at this time.

- VisualAge cc compiler

```
cc -c -q64 -I/<install_dir>/modeltech/include app.c
ld -o app.s1 app.o -b64 -bE:app.exports \
    -bI:/<install_dir>/modeltech/rs64/mti_exports -bM:SRE -bnoentry -lc
```

DPI Imports on 64-bit IBM RS/6000 Platform

When linking the shared objects, be sure to specify **-bE:<isymfile>** option on the link command line. <isymfile> is the name of the file generated by the**-isymfile** argument to the vlog command. Once you have created the <isymfile>, it contains a complete list of all imported tasks and functions expected by ModelSim.

DPI Flow for Exported Tasks and Functions on 64-bit IBM RS/6000 Platform

Since the RS6000 platform lacks the necessary runtime linking capabilities, you must perform an additional manual step in order to prepare shared objects containing calls to exported SystemVerilog tasks or functions shared object file. You need to invoke a special run of vsim. The command is as follows:

```
vsim <top du list> -dpiexportobj <objname> <other args>
```

The **-dpiexportobj** generates the object file <objname>.o that contains "glue" code for exported tasks and functions. You must add that object file to the link line, listed after the other object files. For example, a link line would be:

```
ld -o app.dll app.o <objname>.o
      -bE:<isymfile> -bI:/<install_dir>/modeltech/rs6000/mti_exports -bM:SRE
      -bnoentry -lc
```

Compiling and Linking C++ Applications for PLI/VPI/DPI

ModelSim does not have direct support for any language other than standard C; however, C++ code can be loaded and executed under certain conditions.

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

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

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

The following platform-specific instructions show you how to compile and link your PLI/VPI/DPI C++ applications so that they can be loaded by ModelSim.

Although compilation and simulation switches are platform-specific, loading shared libraries is the same for all platforms. For information on loading libraries, see DPI File Loading.

For PLI/VPI only

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

- Add a path before *app.so* in the foreign attribute specification. (The path may include environment variables.)
- Put the path in a UNIX shell environment variable:
  LD_LIBRARY_PATH_32= *<library path without filename> (32-bit)*
  or
  LD_LIBRARY_PATH_64= *<library path without filename> (64-bit)*

Windows Platforms

- Microsoft Visual C++ 4.1 or Later

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

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

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

  When executing **cl** commands in a DO file, use the **/NOLOGO** switch to prevent the Microsoft C compiler from writing the logo banner to stderr. Writing the logo causes Tcl to think an error occurred.

- MinGW C++ Version 3.2.3

```
g++ -c -I<install_dir>\modeltech\include app.cpp
g++ -shared -Bsymbolic -o app.dll app.o -L<install_dir>\modeltech\win32 -lmtipli
```

  ModelSim requires the use of MinGW gcc compiler rather than the Cygwin gcc compiler.

DPI Imports on Windows Platforms

When linking the shared objects, be sure to specify one -export option for each DPI imported task or function in your linking command line. You can use Verilog’s **-isymfile** option to obtain a complete list of all imported tasks and functions expected by ModelSim.

DPI Special Flow for Exported Tasks and Functions

Since the Windows platform lacks the necessary runtime linking complexity, you must perform an additional manual step in order to compile the HDL source files into the shared object file. You need to invoke a special run of vsim. The command is as follows:

```
vsim <top du list> -dpiexportobj <objname> <other args>
```

The -dpiexportobj generates the object file <objname>.obj that contains "glue" code for exported tasks and functions. You must add that object file to the link line, listed after the other object files. For example, if the object name was *dpi1*, the link line for MinGW would be:

```
g++ -shared -Bsymbolic -o app.dll app.obj <objname>.obj
        -L<install_dir>\modeltech\win32 -lmtipli
```

32-bit Linux Platform

- GNU C++ Version 2.95.3 or Later

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

64-bit Linux for IA64 Platform

64-bit Linux is supported on RedHat Linux Advanced Workstation 2.1 for Itanium 2.

- GNU C++ compiler version gcc 3.2 or later

```
g++ -c -fPIC -I/<install_dir>/modeltech/include app.cpp
ld -shared -Bsymbolic -E --allow-shlib-undefined -o app.so app.o
```

If your PLI/VPI application requires a user or vendor-supplied C library, or an additional system library, you will need to specify that library when you link your PLI/VPI application. For example, to use the system math library libm, specify '-lm' to the 'ld' command:

```
g++ -c -fPIC -I/<install_dir>/modeltech/include math_app.cpp
ld -shared -Bsymbolic -E --allow-shlib-undefined -o math_app.so math_app.o -lm
```

64-bit Linux for Opteron/Athlon 64 and EM64T Platforms

64-bit Linux is supported on RedHat Linux EWS 3.0 for Opteron/Athlon 64 and EM64T.

- GNU C++ compiler version gcc 3.2 or later

```
g++ -c -fPIC -I/<install_dir>/modeltech/include app.cpp
ld -shared -Bsymbolic -E --allow-shlib-undefined -o app.so app.o
```

To compile for 32-bit operation, specify the -m32 argument on the gcc command line.

If your PLI/VPI/DPI application requires a user or vendor-supplied C library, or an additional system library, you will need to specify that library when you link your PLI/VPI/DPI application. For example, to use the system math library libm, specify -lm to the ld command:

```
g++ -c -fPIC -I/<install_dir>/modeltech/include math_app.cpp
ld -shared -Bsymbolic -E --allow-shlib-undefined -o math_app.so math_app.o -lm
```

32-bit Solaris Platform

If your PLI/VPI application uses anything from a system library, you will need to specify that library when you link your PLI/VPI application. For example, to use the standard C library, specify '-lc' to the 'ld' command.

- GNU C++ compiler version gcc 3.2 or later

```
g++ -c -I/<install_dir>/modeltech/include app.cpp
ld -G -Bsymbolic -o app.so app.o -lc
```

- Sun Forte C++ Compiler

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

64-bit Solaris Platform

- GNU C++ compiler version gcc 3.2 or later

```
g++ -c -I<install_dir>/modeltech/include -m64 -fPIC app.cpp
g++ -shared -Bsymbolic -o app.so -m64 app.o
```

  This was tested with gcc 3.2.2. You may need to add the location of *libgcc_s.so.1* to the LD_LIBRARY_PATH_64 environment variable.

- cc compiler

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

32-bit HP700 Platform

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

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

- GNU C++ compiler

```
g++ -c -fPIC -I/<install_dir>/modeltech/include app.cpp
ld -b -o app.sl app.o -lc
```

- cc compiler

```
cc -c +z +DD32 -I/<install_dir>/modeltech/include app.cpp
ld -b -o app.sl app.o -lc
```

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

64-bit HP Platform

- cc Compiler

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

64-bit HP for IA64 Platform

- HP ANSI C++ Compiler (/opt/ansic/bin/cc, /usr/ccs/bin/ld)

```
cc -c +DD64 -I/<install_dir>/modeltech/include app.cpp
ld -b -o app.sl app.o
```

  If your PLI/VPI application requires a user or vendor-supplied C library, or an additional system library, you will need to specify that library when you link your PLI/VPI application. For example, to use the system math library, specify '-lm' to the 'ld' command:

```
cc -c +DD64 -I/<install_dir>/modeltech/include math_app.c
ld -b -o math_app.sl math_app.o -lm
```

32-bit IBM RS/6000 Platform

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

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

- GNU C++ compiler version gcc 3.2 or later

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

- VisualAge C++ Compiler

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

The *app.exp* file must export the PLI/VPI initialization function or table. For the PLI, the exported symbol should be "init_usertfs". Alternatively, if there is no init_usertfs function, then the exported symbol should be "veriusertfs". For the VPI, the exported symbol should be "vlog_startup_routines". These requirements ensure that the appropriate symbol is exported, and thus ModelSim can find the symbol when it dynamically loads the shared object.

For DPI Imports

When linking the shared objects, be sure to specify **-bE:<isymfile>** option on the link command line. <isymfile> is the name of the file generated by the**-isymfile** argument to the vlog command. Once you have created the <isymfile>, it contains a complete list of all imported tasks and functions expected by ModelSim.

DPI Special Flow for Exported Tasks and Functions

Since the RS6000 platform lacks the necessary runtime linking capabilities, you must perform an additional manual step in order to prepare shared objects containing calls to exported SystemVerilog tasks or functions shared object file. You need to invoke a special run of vsim. The command is as follows:

```
vsim <top du list> -dpiexportobj <objname> <other args>
```

The **-dpiexportobj** generates the object file <objname>.o that contains "glue" code for exported tasks and functions. You must add that object file to the link line, listed after the other object files. For example, a link line would be:

```
ld -o app.dll app.o <objname>.o
        -bE:<isymfile> -bI:/<install_dir>/modeltech/rs6000/mti_exports -bM:SRE
        -bnoentry -lc
```

64-bit IBM RS/6000 Platform

Only version 5.1 and later of AIX supports the 64-bit platform. A gcc 64-bit compiler is not available at this time.

- VisualAge C++ Compiler

```
cc -c -q64 -I/<install_dir>/modeltech/include app.cpp
ld -o app.s1 app.o -b64 -bE:app.exports \
    -bI:/<install_dir>/modeltech/rs64/mti_exports -bM:SRE -bnoentry -lc
```

For DPI Imports

When linking the shared objects, be sure to specify **-bE:<isymfile>** option on the link command line. <isymfile> is the name of the file generated by the**-isymfile** argument to the vlog command. Once you have created the <isymfile>, it contains a complete list of all imported tasks and functions expected by ModelSim.

DPI Special Flow for Exported Tasks and Functions

Since the RS6000 platform lacks the necessary runtime linking capabilities, you must perform an additional manual step in order to prepare shared objects containing calls to exported SystemVerilog tasks or functions shared object file. You need to invoke a special run of vsim. The command is as follows:

```
vsim <top du list> -dpiexportobj <objname> <other args>
```

The **-dpiexportobj** generates the object file <objname>.o that contains "glue" code for exported tasks and functions. You must add that object file to the link line, listed after the other object files. For example, a link line would be:

```
ld -o app.so app.o <objname>.o
        -bE:<isymfile> -bI:/<install_dir>/modeltech/rs6000/mti_exports -bM:SRE
        -bnoentry -lc
```

Specifying Application Files to Load

PLI and VPI file loading is identical. DPI file loading uses switches to the **vsim** command.

PLI/VPI file loading

The PLI/VPI applications are specified as follows:

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

  ```
  Veriuser = pliapp1.so pliapp2.so pliappn.so
  ```

- As a list in the PLIOBJS environment variable:

  ```
  % setenv PLIOBJS "pliapp1.so pliapp2.so pliappn.so"
  ```

- As a **-pli** argument to the simulator (multiple arguments are allowed):

  ```
  -pli pliapp1.so -pli pliapp2.so -pli pliappn.so
  ```

Note

On Windows platforms, the file names shown above should end with *.dll* rather than *.so*.

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

See also Simulator Variables for more information on the *modelsim.ini* file.

DPI File Loading

DPI applications are specified to vsim using the following SystemVerilog arguments:

Table D-1. vsim Arguments for DPI Application

Argument	Description
-sv_lib <name>	specifies a library name to be searched and used. No filename extensions must be specified. (The extensions ModelSim expects are: *.sl* for HP, *.dll* for Win32, *.so* for all other platforms.)
-sv_root <name>	specifies a new prefix for shared objects as specified by -sv_lib
-sv_liblist	specifies a "bootstrap file" to use

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

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

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

It is a mistake to specify DPI import tasks and functions (tf) inside PLI/VPI shared objects. However, a DPI import tf can make calls to PLI/VPI C code, providing that **vsim -gblso** was used to mark the PLI/VPI shared object with global symbol visibility. See Loading Shared Objects with Global Symbol Visibility.

Loading Shared Objects with Global Symbol Visibility

On Unix platforms you can load shared objects such that all symbols in the object have global visibility. To do this, use the **-gblso** argument to **vsim** when you load your PLI/VPI application. For example:

```
vsim -pli obj1.so -pli obj2.so -gblso obj1.so top
```

The **-gblso** argument works in conjunction with the GlobalSharedObjectList variable in the *modelsim.ini* file. This variable allows user C code in other shared objects to refer to symbols in a shared object that has been marked as global. All shared objects marked as global are loaded by the simulator earlier than any non-global shared objects.

PLI Example

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

```
hello.c:
      #include "veriuser.h"
      static PLI_INT32 hello()
      {
         io_printf("Hi there\n");
         return 0;
      }
      s_tfcell veriusertfs[] = {
         {usertask, 0, 0, 0, hello, 0, "$hello"},
         {0}  /* last entry must be 0 */
      };
   hello.v:
      module hello;
         initial $hello;
      endmodule
   Compile the PLI code for the Solaris operating system:
      % cc -c -I<install_dir>/modeltech/include hello.c
      % ld -G -Bsymbolic -o hello.sl hello.o
   Compile the Verilog code:
      % vlib work
      % vlog hello.v
   Simulate the design:
      % vsim -c -pli hello.sl hello
      # Loading work.hello
      # Loading ./hello.sl
      VSIM 1> run -all
      # Hi there
      VSIM 2> quit
```

VPI Example

The following example is a trivial, but complete VPI application. A general VPI example can be found in *<install_dir>/modeltech/examples/verilog/vpi*.

hello.c:

```
#include "vpi_user.h"
static PLI_INT32 hello(PLI_BYTE8 * param)
{
   vpi_printf( "Hello world!\n" );
   return 0;
}
void RegisterMyTfs( void )
{
   s_vpi_systf_data systf_data;
   vpiHandle systf_handle;
   systf_data.type        = vpiSysTask;
   systf_data.sysfunctype = vpiSysTask;
   systf_data.tfname      = "$hello";
   systf_data.calltf      = hello;
   systf_data.compiletf   = 0;
   systf_data.sizetf      = 0;
   systf_data.user_data   = 0;
   systf_handle = vpi_register_systf( &systf_data );
   vpi_free_object( systf_handle );
}
```

```
      void (*vlog_startup_routines[])() = {
         RegisterMyTfs,
         0
      };
   hello.v:
      module hello;
         initial $hello;
      endmodule
   Compile the VPI code for the Solaris operating system:
      % gcc -c -I<install_dir>/include hello.c
      % ld -G -Bsymbolic -o hello.sl hello.o
   Compile the Verilog code:
      % vlib work
      % vlog hello.v
   Simulate the design:
      % vsim -c -pli hello.sl hello
      # Loading work.hello
      # Loading ./hello.sl
      VSIM 1> run -all
      # Hello world!
      VSIM 2> quit
```

DPI Example

The following example is a trivial but complete DPI application. For win32 and RS6000 platforms, an additional step is required. For additional examples, see the *<install_dir>/modeltech/examples/systemverilog/dpi* directory.

```
hello_c.c:
#include "svdpi.h"
#include "dpiheader.h"
int c_task(int i, int *o)
{
   printf("Hello from c_task()\n");
   verilog_task(i, o); /* Call back into Verilog */
   *o = i;
   return(0); /* Return success (required by tasks) */
}
hello.v:
module hello_top;
   int ret;
   export "DPI-C" task verilog_task;
   task verilog_task(input int i, output int o);
      #10;
      $display("Hello from verilog_task()");
   endtask
   import "DPI-C" context task c_task(input int i, output int o);
   initial
   begin
      c_task(1, ret);  // Call the c task named 'c_task()'
   end
endmodule
Compile the Verilog code:
   % vlib work
   % vlog -sv -dpiheader dpiheader.h hello.v
```

```
Compile the DPI code for the Solaris operating system:
   % gcc -c -g -I<install_dir>/modeltech/include hello_c.c
   % ld -G -Bsymbolic -o hello_c.so hello_c.o
Simulate the design:
   % vsim -c -sv_lib hello_c hello_top
   # Loading work.hello_c
   # Loading ./hello_c.so
   VSIM 1> run -all
   # Hello from c_task()
   # Hello from verilog_task()
   VSIM 2> quit
```

The PLI Callback reason Argument

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

`reason_endofcompile`
: For the completion of loading the design.

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

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

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

`reason_startofrestart`
: For the start of execution of the **restore** command, but before any of the simulation state has been restored. This allows the PLI application to prepare for the restore, but it shouldn't restore its state with calls to tf_read_restart() until it is called with reason_restart. The reason_startofrestart value is passed only for a restore command, and not in the case that the simulator is invoked with -restore.

`reason_restart`
: For the execution of the **restore** command. This is when the PLI application must restore its state with calls to tf_read_restart().

`reason_reset`
: For the execution of the **restart** command. This is when the PLI application should free its memory and reset its state. We recommend that all PLI applications reset their internal state during a restart as the shared library containing the PLI code might not be

reloaded. (See the **-keeploaded** and **-keeploadedrestart** arguments to **vsim** for related information.)

`reason_endofreset`
For the completion of the **restart** command, after the simulation state has been reset but before the design has been reloaded.

`reason_interactive`
For the execution of the $stop system task or any other time the simulation is interrupted and waiting for user input.

`reason_scope`
For the execution of the **environment** command or selecting a scope in the structure window. Also for the call to acc_set_interactive_scope() if the callback_flag argument is non-zero.

`reason_paramvc`
For the change of value on the system task or function argument.

`reason_synch`
For the end of time step event scheduled by tf_synchronize().

`reason_rosynch`
For the end of time step event scheduled by tf_rosynchronize().

`reason_reactivate`
For the simulation event scheduled by tf_setdelay().

`reason_paramdrc`
Not supported in ModelSim Verilog.

`reason_force`
Not supported in ModelSim Verilog.

`reason_release`
Not supported in ModelSim Verilog.

`reason_disable`
Not supported in ModelSim Verilog.

The sizetf Callback Function

A user-defined system function specifies the width of its return value with the sizetf callback function, and the simulator calls this function while loading the design. The following details on the sizetf callback function are not found in the IEEE Std 1364:

- If you omit the sizetf function, then a return width of 32 is assumed.
- The sizetf function should return 0 if the system function return value is of Verilog type "real".
- The sizetf function should return -32 if the system function return value is of Verilog type "integer".

PLI Object Handles

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

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

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

If your PLI application places value change callbacks on accRegBit or accTerminal objects, *do not* call acc_close() while these callbacks are in effect.

Third Party PLI Applications

Many third party PLI applications come with instructions on using them with ModelSim Verilog. Even without the instructions, it is still likely that you can get it to work with ModelSim Verilog as long as the application uses standard PLI routines. The following guidelines are for preparing a Verilog-XL PLI application to work with ModelSim Verilog.

Generally, a Verilog-XL PLI application comes with a collection of object files and a veriuser.c file. The veriuser.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 C Applications for PLI/VPI/DPI). For example, if you have a *veriuser.c* file and a library archive *libapp.a* file that contains the application's object files, then the following commands should be used to create a dynamically loadable object for the Solaris operating system:

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

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

Note

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

Support for VHDL Objects

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

Table D-2. Supported VHDL Objects

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

The type and fulltype constants for VHDL objects are defined in the *acc_vhdl.h* include file. All of these objects (except signals) are scope objects that define levels of hierarchy in the structure window. Currently, the PLI ACC interface has no provision for obtaining handles to generics, types, constants, variables, attributes, subprograms, and processes.

IEEE Std 1364 ACC Routines

ModelSim Verilog supports the following ACC routines:

Table D-3. Supported ACC Routines

Routines		
acc_append_delays	acc_free	acc_next
acc_append_pulsere	acc_handle_by_name	acc_next_bit
acc_close	acc_handle_calling_mod_m	acc_next_cell
acc_collect	acc_handle_condition	acc_next_cell_load
acc_compare_handles	acc_handle_conn	acc_next_child
acc_configure	acc_handle_hiconn	acc_next_driver
acc_count	acc_handle_interactive_scope	acc_next_hiconn
acc_fetch_argc	acc_handle_loconn	acc_next_input
acc_fetch_argv	acc_handle_modpath	acc_next_load
acc_fetch_attribute	acc_handle_notifier	acc_next_loconn
acc_fetch_attribute_int	acc_handle_object	acc_next_modpath
acc_fetch_attribute_str	acc_handle_parent	acc_next_net
acc_fetch_defname	acc_handle_path	acc_next_output
acc_fetch_delay_mode	acc_handle_pathin	acc_next_parameter
acc_fetch_delays	acc_handle_pathout	acc_next_port
acc_fetch_direction	acc_handle_port	acc_next_portout
acc_fetch_edge	acc_handle_scope	acc_next_primitive
acc_fetch_fullname	acc_handle_simulated_net	acc_next_scope
acc_fetch_fulltype	acc_handle_tchk	acc_next_specparam
acc_fetch_index	acc_handle_tchkarg1	acc_next_tchk
acc_fetch_location	acc_handle_tchkarg2	acc_next_terminal
acc_fetch_name	acc_handle_terminal	acc_next_topmod
acc_fetch_paramtype	acc_handle_tfarg	acc_object_in_typelist
acc_fetch_paramval	acc_handle_itfarg	acc_object_of_type
acc_fetch_polarity	acc_handle_tfinst	acc_product_type
acc_fetch_precision	acc_initialize	acc_product_version
acc_fetch_pulsere		acc_release_object
acc_fetch_range		acc_replace_delays
acc_fetch_size		acc_replace_pulsere
acc_fetch_tfarg		acc_reset_buffer
acc_fetch_itfarg		acc_set_interactive_scope
acc_fetch_tfarg_int		acc_set_pulsere
acc_fetch_itfarg_int		acc_set_scope
acc_fetch_tfarg_str		acc_set_value
acc_fetch_itfarg_str		acc_vcl_add
acc_fetch_timescale_info		acc_vcl_delete
acc_fetch_type		acc_version
acc_fetch_type_str		
acc_fetch_value		

acc_fetch_paramval() cannot be used on 64-bit platforms to fetch a string value of a parameter. Because of this, the function acc_fetch_paramval_str() has been added to the PLI for this use. acc_fetch_paramval_str() is declared in acc_user.h. It functions in a manner similar to acc_fetch_paramval() except that it returns a char *. acc_fetch_paramval_str() can be used on all platforms.

IEEE Std 1364 TF Routines

ModelSim Verilog supports the following TF (task and function) routines;

Table D-4. Supported TF Routines

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

SystemVerilog DPI Access Routines

ModelSim SystemVerilog supports all routines defined in the "svdpi.h" file defined in P1800-2005.

Verilog-XL Compatible Routines

The following PLI routines are not defined in IEEE Std 1364, but ModelSim Verilog provides them for compatibility with Verilog-XL.

```
char *acc_decompile_exp(handle condition)
```

This routine provides similar functionality to the Verilog-XL **acc_decompile_expr** routine. The condition argument must be a handle obtained from the **acc_handle_condition** routine. The value returned by **acc_decompile_exp** is the string representation of the condition expression.

```
char *tf_dumpfilename(void)
```

This routine returns the name of the VCD file.

```
void tf_dumpflush(void)
```

A call to this routine flushes the VCD file buffer (same effect as calling **$dumpflush** in the Verilog code).

```
int tf_getlongsimtime(int *aof_hightime)
```

This routine gets the current simulation time as a 64-bit integer. The low-order bits are returned by the routine, while the high-order bits are stored in the **aof_hightime** argument.

64-bit Support for PLI

The PLI function acc_fetch_paramval() cannot be used on 64-bit platforms to fetch a string value of a parameter. Because of this, the function acc_fetch_paramval_str() has been added to the PLI for this use. acc_fetch_paramval_str() is declared in acc_user.h. It functions in a manner similar to acc_fetch_paramval() except that it returns a char *. acc_fetch_paramval_str() can be used on all platforms.

Using 64-bit ModelSim with 32-bit Applications

If you have 32-bit PLI/VPI/DPI applications and wish to use 64-bit ModelSim, you will need to port your code to 64 bits by moving from the ILP32 data model to the LP64 data model. We strongly recommend that you consult the 64-bit porting guides for Sun and HP.

PLI/VPI Tracing

The foreign interface tracing feature is available for tracing PLI and VPI function calls. Foreign interface tracing creates two kinds of traces: a human-readable log of what functions were called, the value of the arguments, and the results returned; and a set of C-language files that can be used to replay what the foreign interface code did.

The Purpose of Tracing Files

The purpose of the logfile is to aid you in debugging PLI or VPI code. The primary purpose of the replay facility is to send the replay files to support for debugging co-simulation problems, or debugging PLI/VPI problems for which it is impractical to send the PLI/VPI code. We still need you to send the VHDL/Verilog part of the design to actually execute a replay, but many problems can be resolved with the trace only.

Invoking a Trace

To invoke the trace, call vsim with the **-trace_foreign** argument:

Syntax

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

Arguments

`<action>`
Can be either the value 1, 2, or 3. Specifies one of the following actions:

Table D-5. Values for <action> Argument

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

`-tag <name>`
Used to give distinct file names for multiple traces. Optional.

Examples

```
vsim -trace_foreign 1 mydesign
```
Creates a logfile.

```
vsim -trace_foreign 3 mydesign
```
Creates both a logfile and a set of replay files.

```
vsim -trace_foreign 1 -tag 2 mydesign
```
Creates a logfile with a tag of "2".

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

Debugging PLI/VPI/DPI Application Code

In order to debug your PLI/VPI/DPI application code in a debugger, you must first:

1. Compile the application code with debugging information (using the **-g** option) and without optimizations (for example, don’t use the **-O** option).
2. Load **vsim** into a debugger.

   Even though **vsim** is stripped, most debuggers will still execute it. You can invoke the debugger directly on **vsimk**, the simulation kernel where your application code is loaded (for example, "ddd `which vsimk`"), or you can attach the debugger to an already running **vsim** process. In the second case, you must attach to the PID for **vsimk**, and you must specify the full path to the **vsimk** executable (for example, "gdb $MTI_HOME/sunos5/vsimk 1234").

   On Solaris, AIX, and Linux systems you can use either **gdb** or **ddd**. On HP-UX systems you can use the **wdb** debugger from HP. You will need version 1.2 or later.
3. Set an entry point using breakpoint.

   Since initially the debugger recognizes only **vsim's** PLI/VPI/DPI function symbols, when invoking the debugger directly on **vsim** you need to place a breakpoint in the first PLI/VPI/DPI function that is called by your application code. An easy way to set an entry point is to put a call to acc_product_version() as the first executable statement in your application code. Then, after **vsim** has been loaded into the debugger, set a breakpoint in this function. Once you have set the breakpoint, run **vsim** with the usual arguments.

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

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

Troubleshooting a Missing DPI Import Function

DPI uses C function linkage. If your DPI application is written in C++, it is important to remember to use extern "C" declaration syntax appropriately. Otherwise the C++ compiler will produce a mangled C++ name for the function, and the simulator is not able to locate and bind the DPI call to that function.

Also, if you do not use the **-Bsymbolic** argument on the command line for specifying a link, the system may bind to an incorrect function, resulting in unexpected behavior. For more information, see Correct Linking of Shared Libraries with -Bsymbolic.

HP-UX Specific Warnings

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

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

Appendix E
Command and Keyboard Shortcuts

This appendix is a collection of the keyboard and command shortcuts available in the ModelSim GUI.

Command Shortcuts

- You may abbreviate command syntax, but there's a catch — the minimum number of characters required to execute a command are those that make it unique. Remember, as we add new commands some of the old shortcuts may not work. For this reason ModelSim does not allow command name abbreviations in macro files. This minimizes your need to update macro files as new commands are added.
- Multiple commands may be entered on one line if they are separated by semi-colons (;). For example:

  ```
  vlog -nodebug=ports level3.v level2.v ; vlog -nodebug top.v
  ```

  The return value of the last function executed is the only one printed to the transcript. This may cause some unexpected behavior in certain circumstances. Consider this example:

  ```
  vsim -c -do "run 20 ; simstats ; quit -f" top
  ```

  You probably expect the **simstats** results to display in the Transcript window, but they will not, because the last command is **quit -f**. To see the return values of intermediate commands, you must explicitly print the results. For example:

  ```
  vsim -do "run 20 ; echo [simstats]; quit -f" -c top
  ```

Command History Shortcuts

You can review the simulator command history, or reuse previously entered commands with the following shortcuts at the ModelSim/VSIM prompt:

Table E-1. Command History Shortcuts

Shortcut	Description
!!	repeats the last command
!n	repeats command number n; n is the VSIM prompt number (e.g., for this prompt: VSIM 12>, n =12)
!abc	repeats the most recent command starting with "abc"

Table E-1. Command History Shortcuts (cont.)

Shortcut	Description
^xyz^ab^	replaces "xyz" in the last command with "ab"
up arrow and down arrow keys	scrolls through the command history
click on prompt	left-click once on a previous ModelSim or VSIM prompt in the transcript to copy the command typed at that prompt to the active cursor
his or history	shows the last few commands (up to 50 are kept)

Main and Source Window Mouse and Keyboard Shortcuts

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

Table E-2. Mouse Shortcuts

Mouse - UNIX and Windows	Result
Click the left mouse button	relocate the cursor
Click and drag the left mouse button	select an area
Shift-click the left mouse button	extend selection
Double-click the left mouse button	select a word
Double-click and drag the left mouse button	select a group of words
Ctrl-click the left mouse button	move insertion cursor without changing the selection
Click the left mouse button on a previous ModelSim or VSIM prompt	copy and paste previous command string to current prompt
Click the middle mouse button	paste selection to the clipboard
Click and drag the middle mouse button	scroll the window

Table E-3. Keyboard Shortcuts

Keystrokes - UNIX and Windows	Result
Left Arrow Right Arrow	move cursor left or right one character

Table E-3. Keyboard Shortcuts (cont.)

Keystrokes - UNIX and Windows	**Result**
Ctrl + Left Arrow Ctrl + Right Arrow	move cursor left or right one word
Shift + Any Arrow	extend text selection
Ctrl + Shift + Left Arrow Ctrl + Shift + Right Arrow	extend text selection by one word
Up Arrow Down Arrow	Transcript Pane: scroll through command history Source Window: move cursor one line up or down
Ctrl + Up Arrow Ctrl + Down Arrow	Transcript Pane: moves cursor to first or last line Source Window: moves cursor up or down one paragraph
Ctrl + Home	move cursor to the beginning of the text
Ctrl + End	move cursor to the end of the text
Backspace Ctrl + h (UNIX only)	delete character to the left
Delete Ctrl + d (UNIX only)	delete character to the right
Esc (Windows only)	cancel
Alt	activate or inactivate menu bar mode
Alt-F4	close active window
Home Ctrl + a (UNIX only)	move cursor to the beginning of the line
Ctrl + b	move cursor left
Ctrl + d	delete character to the right
End Ctrl + e	move cursor to the end of the line
Ctrl + f (UNIX) Right Arrow (Windows)	move cursor right one character
Ctrl + k	delete to the end of line
Ctrl + n	move cursor one line down (Source window only under Windows)
Ctrl + o (UNIX only)	insert a new line character at the cursor
Ctrl + p	move cursor one line up (Source window only under Windows)

Table E-3. Keyboard Shortcuts (cont.)

Keystrokes - UNIX and Windows	**Result**
Ctrl + s (UNIX) Ctrl + f (Windows)	find
Ctrl + t	reverse the order of the two characters on either side of the cursor
Ctrl + u	delete line
Page Down Ctrl + v (UNIX only)	move cursor down one screen
Ctrl + w (UNIX) Ctrl + x (Windows)	cut the selection
Ctrl + s Ctrl + x (UNIX Only)	save
Ctrl + y (UNIX) Ctrl + v (Windows)	paste the selection
Ctrl + a (Windows Only)	select the entire contents of the widget
Ctrl + \	clear any selection in the widget
Ctrl + - (UNIX) Ctrl + / (UNIX) Ctrl + z (Windows)	undoes previous edits in the Source window
Meta + < (UNIX only)	move cursor to the beginning of the file
Meta + > (UNIX only)	move cursor to the end of the file
Page Up Meta + v (UNIX only)	move cursor up one screen
Meta + w (UNIX) Ctrl + c (Windows)	copy selection
F3	Peforms a Find Next action in the Source Window.
F4 Shift+F4	Change focus to next pane in main window Change focus to previous pane in main window
F5 Shift+F5	Toggle between expanding and restoring size of pane to fit the entire main window Toggle on/off the pane headers.
F8	search for the most recent command that matches the characters typed (Main window only)
F9	run simulation
F10	continue simulation
F11 (Windows only)	single-step

Table E-3. Keyboard Shortcuts (cont.)

Keystrokes - UNIX and Windows	Result
F12 (Windows only)	step-over

The Main window allows insertions or pastes only after the prompt; therefore, you don't need to set the cursor when copying strings to the command line.

List Window Keyboard Shortcuts

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

Table E-4. List Window Keyboard Shortcuts

Key - UNIX and Windows	Action
Left Arrow	scroll listing left (selects and highlights the item to the left of the currently selected item)
Right Arrow	scroll listing right (selects and highlights the item to the right of the currently selected item)
Up Arrow	scroll listing up
Down Arrow	scroll listing down
Page Up Ctrl + Up Arrow	scroll listing up by page
Page Down Ctrl + Down Arrow	scroll listing down by page
Tab	searches forward (down) to the next transition on the selected signal
Shift + Tab	searches backward (up) to the previous transition on the selected signal (does not function on HP workstations)
Shift + Left Arrow Shift + Right Arrow	extends selection left/right
Ctrl + f (Windows) Ctrl + s (UNIX)	opens the Find dialog box to find the specified item label within the list display

Wave Window Mouse and Keyboard Shortcuts

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

Table E-5. Wave Window Mouse Shortcuts

Mouse action	Result
Ctrl + Click left mouse button and drag[1] ↘	zoom area (in)
Ctrl + Click left mouse button and drag ↗	zoom out
Ctrl + Click left mouse button and drag ↖	zoom fit
Click left mouse button and drag	moves closest cursor
Ctrl + Click left mouse button on a scroll bar arrow	scrolls window to very top or bottom (vertical scroll) or far left or right (horizontal scroll)
Click middle mouse button in scroll bar (UNIX only)	scrolls window to position of click

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

Table E-6. Wave Window Keyboard Shortcuts

Keystroke	Action
s	bring into view and center the currently active cursor
i Shift + i +	zoom in (mouse pointer must be over the cursor or waveform panes)
o Shift + o -	zoom out (mouse pointer must be over the cursor or waveform panes)
f Shift + f	zoom full (mouse pointer must be over the cursor or waveform panes)
l Shift + l	zoom last (mouse pointer must be over the cursor or waveform panes)
r Shift + r	zoom range (mouse pointer must be over the cursor or waveform panes)

Table E-6. Wave Window Keyboard Shortcuts

Keystroke	Action
Up Arrow Down Arrow	scrolls entire window up or down one line, when mouse pointer is over waveform pane scrolls highlight up or down one line, when mouse pointer is over pathname or values pane
Left Arrow	scroll pathname, values, or waveform pane left
Right Arrow	scroll pathname, values, or waveform pane right
Page Up	scroll waveform pane up by a page
Page Down	scroll waveform pane down by a page
Tab	search forward (right) to the next transition on the selected signal - finds the next edge
Shift + Tab	search backward (left) to the previous transition on the selected signal - finds the previous edge
Ctrl + f (Windows) Ctrl + s (UNIX)	open the find dialog box; searches within the specified field in the pathname pane for text strings
Ctrl + Left Arrow Ctrl + Right Arrow	scroll pathname, values, or waveform pane left or right by a page

Appendix F
Setting GUI Preferences

The ModelSim GUI is programmed using Tcl/Tk. It is highly customizable. You can control everything from window size, position, and color to the text of window prompts, default output filenames, and so forth.

Most user GUI preferences are stored as Tcl variables in the *.modelsim* file on Unix/Linux platforms or the Registry on Windows platforms. The variable values save automatically when you exit ModelSim. Some of the variables are modified by actions you take with menus or windows (e.g., resizing a window changes its geometry variable). Or, you can edit the variables directly either from the ModelSim > prompt or the Edit Preferences dialog.

Customizing the Simulator GUI Layout

You can customize the layout of panes, windows, toolbars, etc. This section discusses layouts and how they are used in ModelSim.

Layouts and Modes of Operation

ModelSim ships with three default layouts that correspond to three modes of operation.

Table F-1. Predefined GUI Layouts

Layout	Mode
NoDesign	a design is not yet loaded
Simulate	a design is loaded
Coverage	a design is loaded with code coverage enabled

As you load and unload designs, ModelSim switches between the layouts.

Custom Layouts

You can create custom layouts or modify the three default layouts.

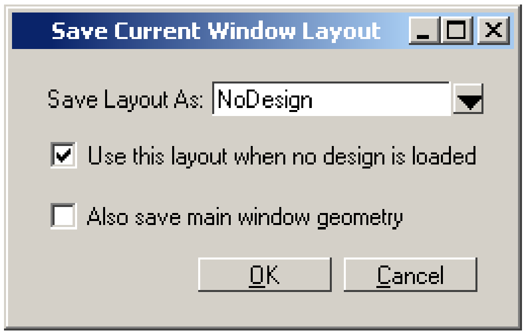
Creating Custom Layouts

To create a custom layout or modify one of the default layouts, follow these steps:

1. Rearrange the GUI as you see fit (see Navigating the Graphic User Interface for details).

2. Select **Layout** > **Save**.

Figure F-1. Save Current Window Layout Dialog Box



Save Current Window Layout

Save Layout As: NoDesign

[✓] Use this layout when no design is loaded

[ ] Also save main window geometry

3. Specify a new name or use an existing name to overwrite that layout.
4. Click OK.

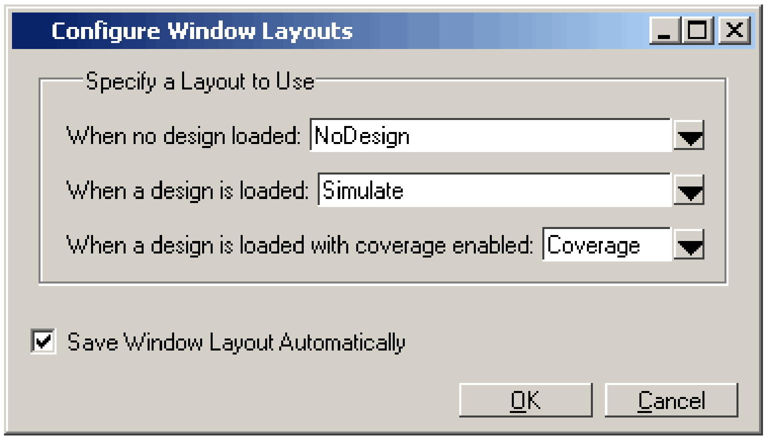
The layout is saved to the *.modelsim* file (or Registry on Windows).

Assigning Layouts to Modes

You can assign which layout appears in each mode (no design loaded, design loaded, design loaded with coverage). Follow these steps:

1. Create your custom layouts as described above.
2. Select **Layout** > **Configure**.

Example F-1. Configure Window Layouts Dialog Box



Configure Window Layouts

Specify a Layout to Use

When no design loaded: NoDesign

When a design is loaded: Simulate

When a design is loaded with coverage enabled: Coverage

[✓] Save Window Layout Automatically

3. Select a layout for each mode.
4. Click OK.

The layout assignment is saved to the *.modelsim* file (Registry on Windows).

Automatic Saving of Layouts

By default any changes you make to a layout are saved automatically when you exit the tool or when you change modes. For example, if you load a design with code coverage, rearrange some windows, and then quit the simulation, the changes are saved to whatever layout was assigned to the "load with coverage" mode.

To disable automatic saving of layouts, select **Layout** > **Configure** and uncheck **Save Window Layout Automatically**.

Resetting Layouts to Their Defaults

You can reset the layouts for the three modes to their original defaults. Select **Layout** > **Reset**. This command does not delete custom layouts.

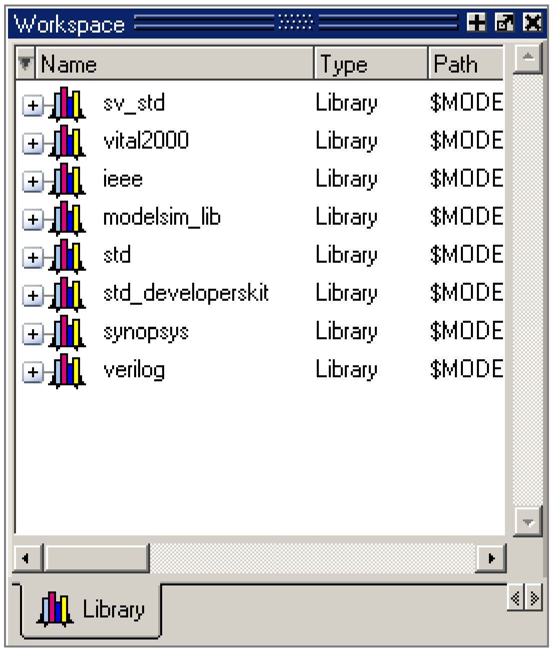
Navigating the Graphic User Interface

This section discusses how to rearrange various elements of the GUI.

Manipulating Panes

Window panes (e.g., Workspace) can be positioned at various places within the parent window or they can be dragged out ("undocked") of the parent window altogether.

Figure F-2. GUI: Window Pane

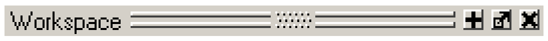


Name	Type	Path
sv_std	Library	$MODE
vital2000	Library	$MODE
ieee	Library	$MODE
modelsim_lib	Library	$MODE
std	Library	$MODE
std_developerskit	Library	$MODE
synopsys	Library	$MODE
verilog	Library	$MODE

Moving Panes

When you see a double bar at the top edge of a pane, it means you can modify the pane position.

Figure F-3. GUI: Double Bar



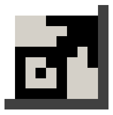
Click-and drag the pane handle in the middle of a double bar (your mouse pointer will change to a four-headed arrow when it is in the correct location) to reposition the pane inside the parent window. As you move the mouse to various parts of the main window, a gray outline will show you valid locations to drop the pane.

Or, drag the pane outside of the parent window, and when you let go of the mouse button, the pane becomes a free-floating window.

Docking and Undocking Panes

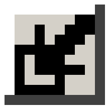
You can undock a pane by clicking the undock button in the heading of a pane.

Figure F-4. GUI: Undock Button



To redock a floating pane, click on the pane handle at the top of the window and drag it back into the parent window, or click the dock icon.

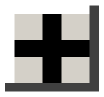
Figure F-5. GUI: Dock Button



Zooming Panes

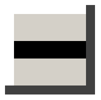
You can expand panes to fill the entire Main window by clicking the zoom icon in the heading of the pane.

Figure F-6. GUI: Zoom Button



To restore the pane to its original size and position click the unzoom button in the heading of the pane.

Figure F-7. GUI: Zoom Button



Columnar Information Display

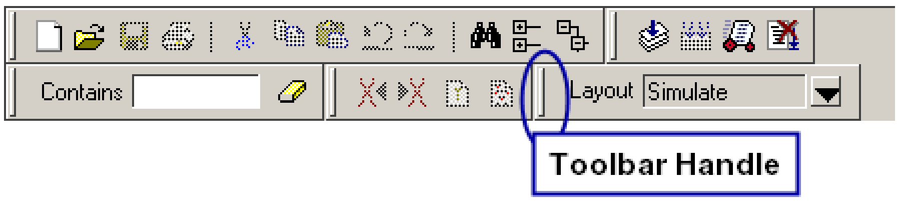
Many panes (e.g., Objects, Workspace, etc.) display information in a columnar format. You can perform a number of operations on columnar formats:

- Click and drag on a column heading to rearrange columns
- Click and drag on a border between column names to increase/decrease column size
- Sort columns by clicking once on the column heading to sort in ascending order; clicking twice to sort in descending order; and clicking three times to sort in default order.
- Hide or show columns by either right-clicking a column heading and selecting an object from the context menu or by clicking the column-list drop down arrow and selecting an object.

Quick Access Toolbars

Toolbar buttons provide access to commonly used commands and functions. Toolbars can be docked and undocked (moved to or from the main toolbar area) by clicking and dragging on the toolbar handle at the left-edge of a toolbar.

Figure F-8. Toolbar Manipulation



You can also hide/show the various toolbars. To hide or show a toolbar, right-click on a blank spot of the main toolbar area and select a toolbar from the list.

To reset toolbars to their original state, right-click on a blank spot of the main toolbar area and select **Reset**.

Simulator GUI Preferences

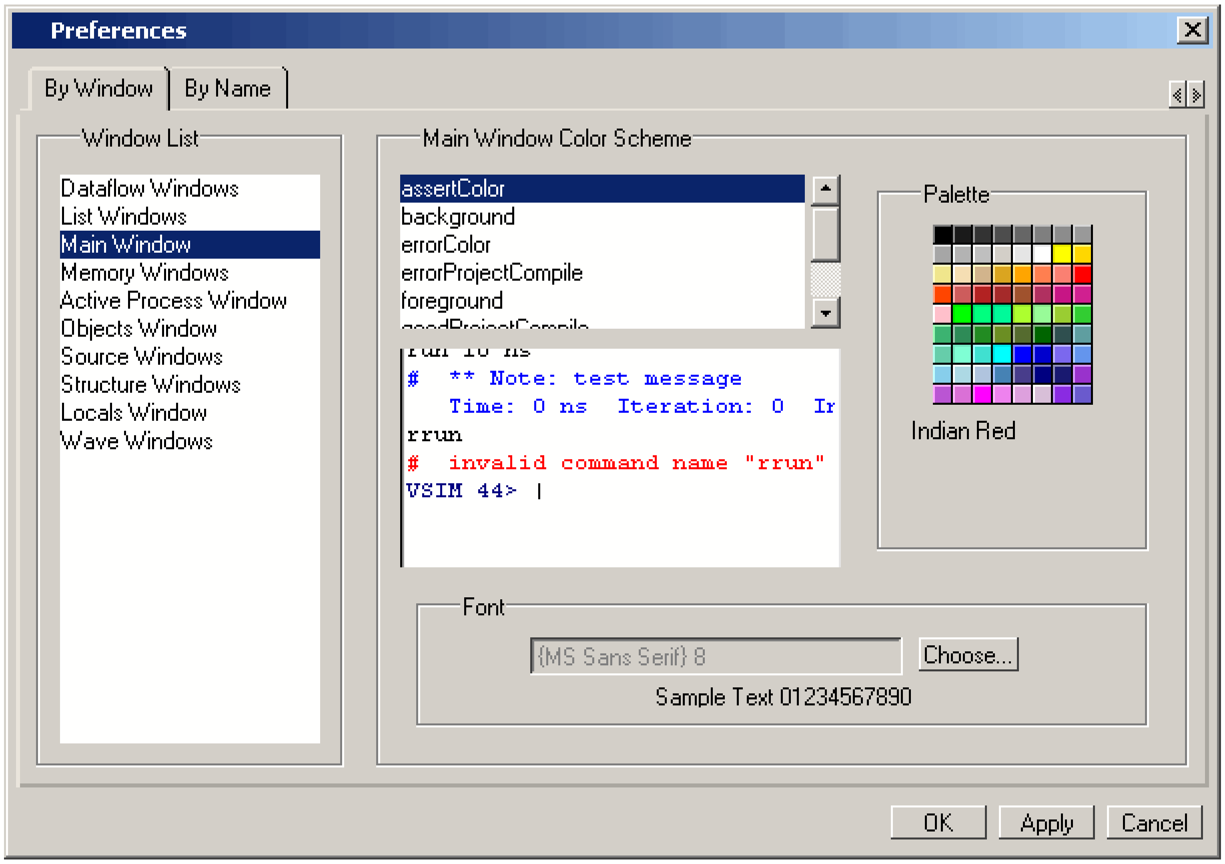
Simulator GUI preferences are stored by default either in the *.modelsim* file in your HOME directory on UNIX/Linux platforms or the Registry on Windows platforms.

Setting Preference Variables from the GUI

To edit a variable value from the GUI, select **Tools > Edit Preferences**.

The dialog organizes preferences by window and by name. The By Window tab primarily allows you to change colors and fonts for various GUI objects. For example, if you want to change the color of assertion messages in the Main window, you would select "Main window" in the first column, select "assertColor" in the second column, and click a color on the palette. Clicking OK or Apply changes the variable, and the change is saved when you exit ModelSim.

Figure F-9. Preferences Dialog Box: By Window Tab



Dataflow Windows
List Windows
Main Window
Memory Windows
Active Process Window
Objects Window
Source Windows
Structure Windows
Locals Window
Wave Windows

assertColor
background
errorColor
errorProjectCompile
foreground

```
# ** Note: test message
   Time: 0 ns  Iteration: 0  In
rrun
# invalid command name "rrun"
VSIM 44> |
```

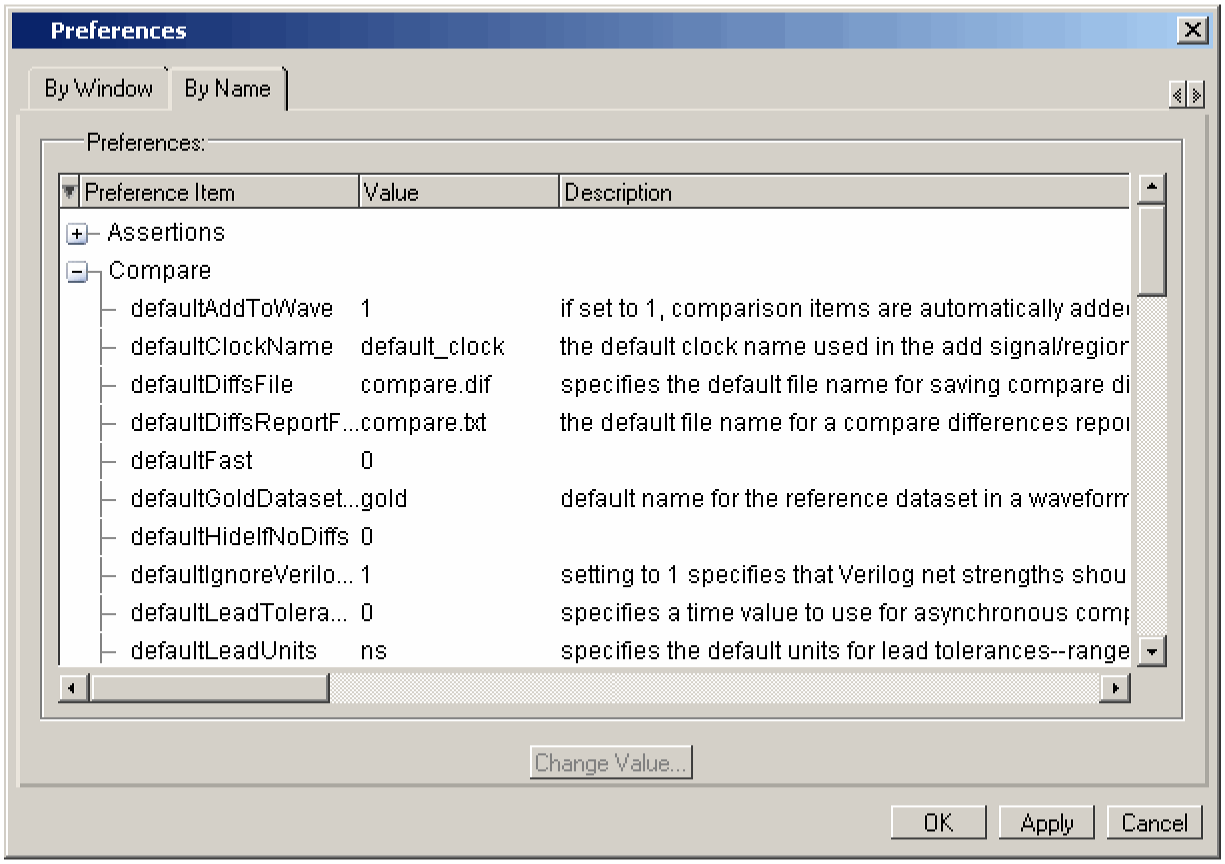
Indian Red

{MS Sans Serif} 8

Sample Text 01234567890

The By Name tab lists every Tcl variable in a tree structure. Expand the tree, highlight a variable, and click Change Value to edit the current value.

Figure F-10. Preferences Dialog Box: By Name Tab



Preference Item	Value	Description
Assertions		
Compare		
defaultAddToWave	1	if set to 1, comparison items are automatically adde
defaultClockName	default_clock	the default clock name used in the add signal/regior
defaultDiffsFile	compare.dif	specifies the default file name for saving compare di
defaultDiffsReportF...	compare.txt	the default file name for a compare differences repoi
defaultFast	0	
defaultGoldDataset...	gold	default name for the reference dataset in a waveforrr
defaultHideIfNoDiffs	0	
defaultIgnoreVerilo...	1	setting to 1 specifies that Verilog net strengths shou
defaultLeadTolera...	0	specifies a time value to use for asynchronous comp
defaultLeadUnits	ns	specifies the default units for lead tolerances--range

Saving GUI Preferences

GUI preferences are saved automatically when you exit the tool.

If you prefer to store GUI preferences elsewhere, set the MODELSIM_PREFERENCES environment variable to designate where these preferences are stored. Setting this variable causes ModelSim to use a specified path and file instead of the default location. Here are some additional points to keep in mind about this variable setting:

- The file does not need to exist before setting the variable as ModelSim will initialize it.
- If the file is read-only, ModelSim will not update or otherwise modify the file.
- This variable may contain a relative pathname, in which case the file is relative to the working directory at the time the tool is started.

The modelsim.tcl File

Previous versions saved user GUI preferences into a *modelsim.tcl* file. Current versions will still read in a *modelsim.tcl* file if it exists. ModelSim searches for the file as follows:

- use MODELSIM_TCL environment variable if it exists (if MODELSIM_TCL is a list of files, each file is loaded in the order that it appears in the list); else
- use *./modelsim.tcl*; else
- use $(HOME)/modelsim.tcl if it exists

Note that in versions 6.1 and later, ModelSim will save to the *.modelsim* file any variables it reads in from a *modelsim.tcl* file. The values from the *modelsim.tcl* file will override like variables in the *.modelsim* file.

Appendix G
System Initialization

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

Files Accessed During Startup

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

Table G-1. Files Accessed During Startup

File	**Purpose**
modelsim.ini	contains initial tool settings; see Simulator Control Variables for specific details on the *modelsim.ini* file
location map file	used by ModelSim tools to find source files based on easily reallocated "soft" paths; default file name is mgc_location_map
pref.tcl	contains defaults for fonts, colors, prompts, window positions, and other simulator window characteristics
.modelsim (UNIX) or Windows registry	contains last working directory, project file, printer defaults, and other user-customized GUI characteristics
modelsim.tcl	contains user-customized settings for fonts, colors, prompts, other GUI characteristics; maintained for backwards compatibility with older versions (see The modelsim.tcl File)
<project_name>.mpf	if available, loads last project file which is specified in the registry (Windows) or *$(HOME)/.modelsim* (UNIX); see What are Projects? for details on project settings

Environment Variables Accessed During Startup

The table below describes the environment variables that are read during startup. They are listed in the order in which they are accessed. For more information on environment variables, see Environment Variables.

Table G-2. Environment Variables Accessed During Startup

Environment variable	**Purpose**
MODEL_TECH	set by ModelSim to the directory in which the binary executables reside (e.g., *../modeltech/<platform>/*)
MODEL_TECH_OVERRIDE	provides an alternative directory for the binary executables; MODEL_TECH is set to this path
MODELSIM	identifies the pathname of the *modelsim.ini* file
MGC_WD	identifies the Mentor Graphics working directory
MGC_LOCATION_MAP	identifies the pathname of the location map file; set by ModelSim if not defined
MODEL_TECH_TCL	identifies the pathname of all Tcl libraries installed with ModelSim
HOME	identifies your login directory (UNIX only)
MGC_HOME	identifies the pathname of the MGC tool suite
TCL_LIBRARY	identifies the pathname of the Tcl library; set by ModelSim to the same pathname as MODEL_TECH_TCL; must point to libraries supplied by Model Technology
TK_LIBRARY	identifies the pathname of the Tk library; set by ModelSim to the same pathname as MODEL_TECH_TCL; must point to libraries supplied by Model Technology
ITCL_LIBRARY	identifies the pathname of the [incr]Tcl library; set by ModelSim to the same path as MODEL_TECH_TCL; must point to libraries supplied by Model Technology
ITK_LIBRARY	identifies the pathname of the [incr]Tk library; set by ModelSim to the same pathname as MODEL_TECH_TCL; must point to libraries supplied by Model Technology
VSIM_LIBRARY	identifies the pathname of the Tcl files that are used by ModelSim; set by ModelSim to the same pathname as MODEL_TECH_TCL; must point to libraries supplied by Model Technology
MTI_COSIM_TRACE	creates an *mti_trace_cosim* file containing debugging information about FLI/PLI/VPI function calls; set to any value before invoking the simulator
MTI_LIB_DIR	identifies the path to all Tcl libraries installed with ModelSim

Table G-2. Environment Variables Accessed During Startup

Environment variable	Purpose
MTI_VCO_MODE	determines which version of ModelSim to use on platforms that support both 32- and 64-bit versions when ModelSim executables are invoked from the *modeltech/bin* directory by a Unix shell command (using full path specification or PATH search)
MODELSIM_TCL	identifies the pathname to a user preference file (e.g., *C:\modeltech\modelsim.tcl*); can be a list of file pathnames, separated by semicolons (Windows) or colons (UNIX); note that user preferences are now stored in the *.modelsim* file (Unix) or registry (Windows); ModelSim will still read this environment variable but it will then save all the settings to the *.modelsim* file when you exit the tool

Initialization Sequence

The following list describes in detail ModelSim's initialization sequence. The sequence includes a number of conditional structures, the results of which are determined by the existence of certain files and the current settings of environment variables.

In the steps below, names in uppercase denote environment variables (except MTI_LIB_DIR which is a Tcl variable). Instances of *$(NAME)* denote paths that are determined by an environment variable (except *$(MTI_LIB_DIR)* which is determined by a Tcl variable).

1. Determines the path to the executable directory (*../modeltech/<platform>*). Sets MODEL_TECH to this path, *unless* MODEL_TECH_OVERRIDE exists, in which case MODEL_TECH is set to the same value as MODEL_TECH_OVERRIDE.
2. Finds the *modelsim.ini* file by evaluating the following conditions:
   - use *$(MODELSIM)/modelsim.ini* if it exists; else
   - use *$(MGC_PWD)/modelsim.ini*; else
   - use *./modelsim.ini*; else
   - use *$(MODEL_TECH)/modelsim.ini*; else
   - use *$(MODEL_TECH)/../modelsim.ini*; else
   - use *$(MGC_HOME)/lib/modelsim.ini*; else
   - set path to *./modelsim.ini* even though the file doesn't exist
3. Finds the location map file by evaluating the following conditions:
   - use MGC_LOCATION_MAP if it exists (if this variable is set to "no_map", ModelSim skips initialization of the location map); else

- use *mgc_location_map* if it exists; else
- use *$(HOME)/mgc/mgc_location_map*; else
- use *$(HOME)/mgc_location_map*; else
- use *$(MGC_HOME)/etc/mgc_location_map*; else
- use *$(MGC_HOME)/shared/etc/mgc_location_map*; else
- use *$(MODEL_TECH)/mgc_location_map*; else
- use *$(MODEL_TECH)/../mgc_location_map*; else
- use no map

4. Reads various variables from the [vsim] section of the *modelsim.ini* file. See Simulation Control Variables for more details.
5. Parses any command line arguments that were included when you started ModelSim and reports any problems.
6. Defines the following environment variables:
   - use MODEL_TECH_TCL if it exists; else
   - set MODEL_TECH_TCL=*$(MODEL_TECH)/../tcl*
   - set TCL_LIBRARY=*$(MODEL_TECH_TCL)/tcl8.3*
   - set TK_LIBRARY=*$(MODEL_TECH_TCL)/tk8.3*
   - set ITCL_LIBRARY=*$(MODEL_TECH_TCL)/itcl3.0*
   - set ITK_LIBRARY=*$(MODEL_TECH_TCL)/itk3.0*
   - set VSIM_LIBRARY=*$(MODEL_TECH_TCL)/vsim*
7. Initializes the simulator's Tcl interpreter.
8. Checks for a valid license (a license is not checked out unless specified by a *modelsim.ini* setting or command line option).
9. The next four steps relate to initializing the graphical user interface.
10. Sets Tcl variable MTI_LIB_DIR=$(MODEL_TECH_TCL)
11. Loads *$(MTI_LIB_DIR)/vsim/pref.tcl*.
12. Loads GUI preferences, project file, etc. from the registry (Windows) or *$(HOME)/.modelsim* (UNIX).
13. Searches for the *modelsim.tcl* file by evaluating the following conditions:
    - use MODELSIM_TCL environment variable if it exists (if MODELSIM_TCL is a list of files, each file is loaded in the order that it appears in the list); else

- use *./modelsim.tcl*; else
- use *$(HOME)/modelsim.tcl* if it exists

That completes the initialization sequence. Also note the following about the *modelsim.ini* file:

- When you change the working directory within ModelSim, the tool reads the [library], [vcom], and [vlog] sections of the local *modelsim.ini* file. When you make changes in the compiler or simulator options dialog or use the **vmap** command, the tool updates the appropriate sections of the file.
- The *pref.tcl* file references the default .ini file via the [GetPrivateProfileString] Tcl command. The .ini file that is read will be the default file defined at the time *pref.tcl* is loaded.

Index

Third-Party Information

This section provides information on third-party software that may be included in the ModelSim product, including any additional license terms.

- This product may include Valgrind third-party software.

- This product may use MinGW GCC third-party software.

  Refer to the license file in your install directory:

  <install_directory>/docs/legal/mingw_gcc.pdf

- This software application may include GNU GCC third-party software.

  Refer to the license file in your install directory:

End-User License Agreement

The latest version of the End-User License Agreement is available on-line at:
www.mentor.com/terms_conditions/enduser.cfm

IMPORTANT INFORMATION

USE OF THIS SOFTWARE IS SUBJECT TO LICENSE RESTRICTIONS. CAREFULLY READ THIS LICENSE AGREEMENT BEFORE USING THE SOFTWARE. USE OF SOFTWARE INDICATES YOUR COMPLETE AND UNCONDITIONAL ACCEPTANCE OF THE TERMS AND CONDITIONS SET FORTH IN THIS AGREEMENT. ANY ADDITIONAL OR DIFFERENT PURCHASE ORDER TERMS AND CONDITIONS SHALL NOT APPLY.

END-USER LICENSE AGREEMENT ("Agreement")

This is a legal agreement concerning the use of Software between you, the end user, as an authorized representative of the company acquiring the license, and Mentor Graphics Corporation and Mentor Graphics (Ireland) Limited acting directly or through their subsidiaries (collectively "Mentor Graphics"). Except for license agreements related to the subject matter of this license agreement which are physically signed by you and an authorized representative of Mentor Graphics, this Agreement and the applicable quotation contain the parties' entire understanding relating to the subject matter and supersede all prior or contemporaneous agreements. If you do not agree to these terms and conditions, promptly return or, if received electronically, certify destruction of Software and all accompanying items within five days after receipt of Software and receive a full refund of any license fee paid.

1. **GRANT OF LICENSE.** The software programs, including any updates, modifications, revisions, copies, documentation and design data ("Software"), are copyrighted, trade secret and confidential information of Mentor Graphics or its licensors who maintain exclusive title to all Software and retain all rights not expressly granted by this Agreement. Mentor Graphics grants to you, subject to payment of appropriate license fees, a nontransferable, nonexclusive license to use Software solely: (a) in machine-readable, object-code form; (b) for your internal business purposes; (c) for the license term; and (d) on the computer hardware and at the site authorized by Mentor Graphics. A site is restricted to a one-half mile (800 meter) radius. Mentor Graphics' standard policies and programs, which vary depending on Software, license fees paid or services purchased, apply to the following: (a) relocation of Software; (b) use of Software, which may be limited, for example, to execution of a single session by a single user on the authorized hardware or for a restricted period of time (such limitations may be technically implemented through the use of authorization codes or similar devices); and (c) support services provided, including eligibility to receive telephone support, updates, modifications, and revisions.

2. **EMBEDDED SOFTWARE.** If you purchased a license to use embedded software development ("ESD") Software, if applicable, Mentor Graphics grants to you a nontransferable, nonexclusive license to reproduce and distribute executable files created using ESD compilers, including the ESD run-time libraries distributed with ESD C and C++ compiler Software that are linked into a composite program as an integral part of your compiled computer program, provided that you distribute these files only in conjunction with your compiled computer program. Mentor Graphics does NOT grant you any right to duplicate, incorporate or embed copies of Mentor Graphics' real-time operating systems or other embedded software products into your products or applications without first signing or otherwise agreeing to a separate agreement with Mentor Graphics for such purpose.

3. **BETA CODE.** Software may contain code for experimental testing and evaluation ("Beta Code"), which may not be used without Mentor Graphics' explicit authorization. Upon Mentor Graphics' authorization, Mentor Graphics grants to you a temporary, nontransferable, nonexclusive license for experimental use to test and evaluate the Beta Code without charge for a limited period of time specified by Mentor Graphics. This grant and your use of the Beta Code shall not be construed as marketing or offering to sell a license to the Beta Code, which Mentor Graphics may choose not to release commercially in any form. If Mentor Graphics authorizes you to use the Beta Code, you agree to evaluate and test the Beta Code under normal conditions as directed by Mentor Graphics. You will contact Mentor Graphics periodically during your use of the Beta Code to discuss any malfunctions or suggested improvements. Upon completion of your evaluation and testing, you will send to Mentor Graphics a written evaluation of the Beta Code, including its strengths, weaknesses and recommended improvements. You agree that any written evaluations and all inventions, product improvements, modifications or developments that Mentor Graphics conceived or made during or subsequent to this Agreement, including those based partly or wholly on your feedback, will be the exclusive property of Mentor Graphics. Mentor Graphics will have exclusive rights, title and interest in all such property. The provisions of this section 3 shall survive the termination or expiration of this Agreement.

4. **RESTRICTIONS ON USE.** You may copy Software only as reasonably necessary to support the authorized use. Each copy must include all notices and legends embedded in Software and affixed to its medium and container as received from Mentor Graphics. All copies shall remain the property of Mentor Graphics or its licensors. You shall maintain a record of the number and primary location of all copies of Software, including copies merged with other software, and shall make those records available to Mentor Graphics upon request. You shall not make Software available in any form to any person other than employees and on-site contractors, excluding Mentor Graphics' competitors, whose job performance requires access and who are under obligations of confidentiality. You shall take appropriate action to protect the confidentiality of Software and ensure that any person permitted access to Software does not disclose it or use it except as permitted by this Agreement. Except as otherwise permitted for purposes of interoperability as specified by applicable and mandatory local law, you shall not reverse-assemble, reverse-compile, reverse-engineer or in any way derive from Software any source code. You may not sublicense, assign or otherwise transfer Software, this Agreement or the rights under it, whether by operation of law or otherwise ("attempted transfer"), without Mentor Graphics' prior written consent and payment of Mentor Graphics' then-current applicable transfer charges. Any attempted transfer without Mentor Graphics' prior written consent shall be a material breach of this Agreement and may, at Mentor Graphics' option, result in the immediate termination of the Agreement and licenses granted under this Agreement. The terms of this Agreement, including without limitation, the licensing and assignment provisions shall be binding upon your successors in interest and assigns. The provisions of this section 4 shall survive the termination or expiration of this Agreement.

5. **LIMITED WARRANTY.**

   5.1. Mentor Graphics warrants that during the warranty period Software, when properly installed, will substantially conform to the functional specifications set forth in the applicable user manual. Mentor Graphics does not warrant that Software will meet your requirements or that operation of Software will be uninterrupted or error free. The warranty period is 90 days starting on the 15th day after delivery or upon installation, whichever first occurs. You must notify Mentor Graphics in writing of any nonconformity within the warranty period. This warranty shall not be valid if Software has been subject to misuse, unauthorized modification or improper installation. MENTOR GRAPHICS' ENTIRE LIABILITY AND YOUR EXCLUSIVE REMEDY SHALL BE, AT MENTOR GRAPHICS' OPTION, EITHER (A) REFUND OF THE PRICE PAID UPON RETURN OF SOFTWARE TO MENTOR GRAPHICS OR (B) MODIFICATION OR REPLACEMENT OF SOFTWARE THAT DOES NOT MEET THIS LIMITED WARRANTY, PROVIDED YOU HAVE OTHERWISE COMPLIED WITH THIS AGREEMENT. MENTOR GRAPHICS MAKES NO WARRANTIES WITH RESPECT TO: (A) SERVICES; (B) SOFTWARE WHICH IS LICENSED TO YOU FOR A LIMITED TERM OR LICENSED AT NO COST; OR (C) EXPERIMENTAL BETA CODE; ALL OF WHICH ARE PROVIDED "AS IS."

   5.2. THE WARRANTIES SET FORTH IN THIS SECTION 5 ARE EXCLUSIVE. NEITHER MENTOR GRAPHICS NOR ITS LICENSORS MAKE ANY OTHER WARRANTIES, EXPRESS, IMPLIED OR STATUTORY, WITH RESPECT TO SOFTWARE OR OTHER MATERIAL PROVIDED UNDER THIS AGREEMENT. MENTOR GRAPHICS AND ITS LICENSORS SPECIFICALLY DISCLAIM ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE AND NON-INFRINGEMENT OF INTELLECTUAL PROPERTY.

6. **LIMITATION OF LIABILITY.** EXCEPT WHERE THIS EXCLUSION OR RESTRICTION OF LIABILITY WOULD BE VOID OR INEFFECTIVE UNDER APPLICABLE LAW, IN NO EVENT SHALL MENTOR GRAPHICS OR ITS LICENSORS BE LIABLE FOR INDIRECT, SPECIAL, INCIDENTAL, OR CONSEQUENTIAL DAMAGES (INCLUDING LOST PROFITS OR SAVINGS) WHETHER BASED ON CONTRACT, TORT OR ANY OTHER LEGAL THEORY, EVEN IF MENTOR GRAPHICS OR ITS LICENSORS HAVE BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES. IN NO EVENT SHALL MENTOR GRAPHICS' OR ITS LICENSORS' LIABILITY UNDER THIS AGREEMENT EXCEED THE AMOUNT PAID BY YOU FOR THE SOFTWARE OR SERVICE GIVING RISE TO THE CLAIM. IN THE CASE WHERE NO AMOUNT WAS PAID, MENTOR GRAPHICS AND ITS LICENSORS SHALL HAVE NO LIABILITY FOR ANY DAMAGES WHATSOEVER. THE PROVISIONS OF THIS SECTION 6 SHALL SURVIVE THE EXPIRATION OR TERMINATION OF THIS AGREEMENT.

7. **LIFE ENDANGERING ACTIVITIES.** NEITHER MENTOR GRAPHICS NOR ITS LICENSORS SHALL BE LIABLE FOR ANY DAMAGES RESULTING FROM OR IN CONNECTION WITH THE USE OF SOFTWARE IN ANY APPLICATION WHERE THE FAILURE OR INACCURACY OF THE SOFTWARE MIGHT RESULT IN DEATH OR PERSONAL INJURY. THE PROVISIONS OF THIS SECTION 7 SHALL SURVIVE THE EXPIRATION OR TERMINATION OF THIS AGREEMENT.

8. **INDEMNIFICATION.** YOU AGREE TO INDEMNIFY AND HOLD HARMLESS MENTOR GRAPHICS AND ITS LICENSORS FROM ANY CLAIMS, LOSS, COST, DAMAGE, EXPENSE, OR LIABILITY, INCLUDING ATTORNEYS' FEES, ARISING OUT OF OR IN CONNECTION WITH YOUR USE OF SOFTWARE AS

DESCRIBED IN SECTION 7. THE PROVISIONS OF THIS SECTION 8 SHALL SURVIVE THE EXPIRATION OR TERMINATION OF THIS AGREEMENT.

9. **INFRINGEMENT.**

9.1. Mentor Graphics will defend or settle, at its option and expense, any action brought against you alleging that Software infringes a patent or copyright or misappropriates a trade secret in the United States, Canada, Japan, or member state of the European Patent Office. Mentor Graphics will pay any costs and damages finally awarded against you that are attributable to the infringement action. You understand and agree that as conditions to Mentor Graphics' obligations under this section you must: (a) notify Mentor Graphics promptly in writing of the action; (b) provide Mentor Graphics all reasonable information and assistance to defend or settle the action; and (c) grant Mentor Graphics sole authority and control of the defense or settlement of the action.

9.2. If an infringement claim is made, Mentor Graphics may, at its option and expense: (a) replace or modify Software so that it becomes noninfringing; (b) procure for you the right to continue using Software; or (c) require the return of Software and refund to you any license fee paid, less a reasonable allowance for use.

9.3. Mentor Graphics has no liability to you if infringement is based upon: (a) the combination of Software with any product not furnished by Mentor Graphics; (b) the modification of Software other than by Mentor Graphics; (c) the use of other than a current unaltered release of Software; (d) the use of Software as part of an infringing process; (e) a product that you make, use or sell; (f) any Beta Code contained in Software; (g) any Software provided by Mentor Graphics' licensors who do not provide such indemnification to Mentor Graphics' customers; or (h) infringement by you that is deemed willful. In the case of (h) you shall reimburse Mentor Graphics for its attorney fees and other costs related to the action upon a final judgment.

9.4. THIS SECTION IS SUBJECT TO SECTION 6 ABOVE AND STATES THE ENTIRE LIABILITY OF MENTOR GRAPHICS AND ITS LICENSORS AND YOUR SOLE AND EXCLUSIVE REMEDY WITH RESPECT TO ANY ALLEGED PATENT OR COPYRIGHT INFRINGEMENT OR TRADE SECRET MISAPPROPRIATION BY ANY SOFTWARE LICENSED UNDER THIS AGREEMENT.

10. **TERM.** This Agreement remains effective until expiration or termination. This Agreement will immediately terminate upon notice if you exceed the scope of license granted or otherwise fail to comply with the provisions of Sections 1, 2, or 4. For any other material breach under this Agreement, Mentor Graphics may terminate this Agreement upon 30 days written notice if you are in material breach and fail to cure such breach within the 30 day notice period. If Software was provided for limited term use, this Agreement will automatically expire at the end of the authorized term. Upon any termination or expiration, you agree to cease all use of Software and return it to Mentor Graphics or certify deletion and destruction of Software, including all copies, to Mentor Graphics' reasonable satisfaction.

11. **EXPORT.** Software is subject to regulation by local laws and United States government agencies, which prohibit export or diversion of certain products, information about the products, and direct products of the products to certain countries and certain persons. You agree that you will not export any Software or direct product of Software in any manner without first obtaining all necessary approval from appropriate local and United States government agencies.

12. **RESTRICTED RIGHTS NOTICE.** Software was developed entirely at private expense and is commercial computer software provided with RESTRICTED RIGHTS. Use, duplication or disclosure by the U.S. Government or a U.S. Government subcontractor is subject to the restrictions set forth in the license agreement under which Software was obtained pursuant to DFARS 227.7202-3(a) or as set forth in subparagraphs (c)(1) and (2) of the Commercial Computer Software - Restricted Rights clause at FAR 52.227-19, as applicable. Contractor/manufacturer is Mentor Graphics Corporation, 8005 SW Boeckman Road, Wilsonville, Oregon 97070-7777 USA.

13. **THIRD PARTY BENEFICIARY.** For any Software under this Agreement licensed by Mentor Graphics from Microsoft or other licensors, Microsoft or the applicable licensor is a third party beneficiary of this Agreement with the right to enforce the obligations set forth herein.

14. **AUDIT RIGHTS.** You will monitor access to, location and use of Software. With reasonable prior notice and during your normal business hours, Mentor Graphics shall have the right to review your software monitoring system and reasonably relevant records to confirm your compliance with the terms of this Agreement, an addendum to this Agreement or U.S. or other local export laws. Such review may include FLEXlm or FLEXnet report log files that you shall capture and provide at Mentor Graphics' request. Mentor Graphics shall treat as confidential information all of your information gained as a result of any request or review and shall only use or disclose such information as required by law or to enforce its rights under this Agreement or addendum to this Agreement. The provisions of this section 14 shall survive the expiration or termination of this Agreement.

15. **CONTROLLING LAW, JURISDICTION AND DISPUTE RESOLUTION.** THIS AGREEMENT SHALL BE GOVERNED BY AND CONSTRUED UNDER THE LAWS OF THE STATE OF OREGON, USA, IF YOU ARE LOCATED IN NORTH OR SOUTH AMERICA, AND THE LAWS OF IRELAND IF YOU ARE LOCATED OUTSIDE OF NORTH OR SOUTH AMERICA. All disputes arising out of or in relation to this Agreement shall be submitted to the exclusive jurisdiction of Portland, Oregon when the laws of Oregon apply, or Dublin, Ireland when the laws of Ireland apply. Notwithstanding the foregoing, all disputes in Asia (except for Japan) arising out of or in relation to this Agreement shall be resolved by arbitration in Singapore before a single arbitrator to be appointed by the Chairman of the Singapore International Arbitration Centre ("SIAC") to be conducted in the English language, in accordance with the Arbitration Rules of the SIAC in effect at the time of the dispute, which rules are deemed to be incorporated by reference in this section 15. This section shall not restrict Mentor Graphics' right to bring an action against you in the jurisdiction where your place of business is located. The United Nations Convention on Contracts for the International Sale of Goods does not apply to this Agreement.

16. **SEVERABILITY.** If any provision of this Agreement is held by a court of competent jurisdiction to be void, invalid, unenforceable or illegal, such provision shall be severed from this Agreement and the remaining provisions will remain in full force and effect.

17. **PAYMENT TERMS AND MISCELLANEOUS.** You will pay amounts invoiced, in the currency specified on the applicable invoice, within 30 days from the date of such invoice. Any past due invoices will be subject to the imposition of interest charges in the amount of one and one-half percent per month or the applicable legal rate currently in effect, whichever is lower. Some Software may contain code distributed under a third party license agreement that may provide additional rights to you. Please see the applicable Software documentation for details. This Agreement may only be modified in writing by authorized representatives of the parties. Waiver of terms or excuse of breach must be in writing and shall not constitute subsequent consent, waiver or excuse.

    Rev. 060210, Part No. 227900