SYNOPSYS®

Simulator Configuration Guide for Synopsys Models

To search the entire manual set, press this toolbar button. For help, refer to intro.pdf.

July 31, 2001

Contents

Figures

Tables

Preface

About This Manual

This manual contains procedures for using Synopsys models with the most widely used simulators. The scope includes the following types of models:

- SmartModels (including FlexModels)
- MemPro Models
- Hardware Models

Note that this manual contains illustrations of third-party software files solely to demonstrate the end user modifications needed to get Synopsys models working with these tools. Third-party software changes frequently. Refer to the third-party tool vendor's documentation for definitive information about their licensed software.

Related Documents

For more information about SmartModels (including FlexModels), or to navigate to a related online document, refer to the *Guide to SmartModel Documentation*. For information on supported platforms and simulators, refer to *SmartModel Library Supported Simulators and Platforms*.

For detailed information about specific SmartModels (including FlexModels), use the Browser tool ($LMC_HOME/bin/sl_browser) to access the online model datasheets.

For more information about MemPro, or to navigate to a related online document, refer to the *Guide to MemPro Documentation*.

For more information about hardware models, or to navigate to a related online document, refer to the *Guide to Hardware Model Documents*.

Some Hyperlinks May Not Work

Because this manual is included with multiple product documentation sets, some hyperlinks do not work properly in all cases. For example, hyperlinks from this manual to other books in the hardware model documentation set will only work from a hardware model installation tree. Similarly, hyperlinks to other books installed in $LMC_HOME will only work in that location.

To work around this limitation, you can visit the Synopsys Web site and navigate to the latest documentation for all Synopsys models:

http://www.synopsys.com/products/lm/doc

Manual Overview

This manual contains the following chapters:

Preface	Describes the manual and lists the typographical conventions and symbols used in it. Tells how to get technical assistance.
Chapter 1 Using Synopsys Models with Simulators	Basic information for configuring and instantiating SmartModels, FlexModels, MemPro models, and hardware models for use in hardware simulators.
Chapter 2 Using VCS with Synopsys Models	How to configure SmartModels, FlexModels, MemPro models, and hardware models for use with VCS. Includes a script that you can use to run FlexModel example testbenches in VCS.
Chapter 3 Using Verilog-XL with Synopsys Models	How to configure SmartModels, FlexModels, MemPro models, and hardware models for use with Verilog-XL.
Chapter 4 Using NC-Verilog with Synopsys Models	How to configure SmartModels, FlexModels, MemPro models, and hardware models for use with NC-Verilog.
Chapter 5 Using MTI Verilog with Synopsys Models	How to configure SmartModels, FlexModels, MemPro models, and hardware models for use with MTI Verilog.
Chapter 6 Using Scirocco with Synopsys Models	How to configure SmartModels, FlexModels, MemPro models, and hardware models for use with Scirocco. Includes a script that you can use to run FlexModel example testbenches in Scirocco.
Chapter 7 Using VSS with Synopsys Models	How to configure SmartModels, FlexModels, MemPro models, and hardware models for use with VSS.

Typographical and Symbol Conventions

- **Default UNIX prompt**

  Represented by a percent sign (`%`).

- **User input** (text entered by the user)

  Shown in **`bold`** type, as in the following command line example:

  ```
  % cd $LMC_HOME/hdl
  ```

- **System-generated text** (prompts, messages, files, reports)

  Shown as in the following system message:

  ```
  No Mismatches: 66 Vectors processed: 66 Possible
  ```

- **Variables** for which you supply a specific value

  Shown in italic type, as in the following command line example:

      % **setenv LMC_HOME** *prod_dir*

  In this example, you substitute a specific name for *prod_dir* when you enter the command.

- Command syntax

  Choice among alternatives is shown with a vertical bar (|) as in the following syntax example:

      -effort_level low | medium | high

  In this example, you must choose one of the three possibilities: low, medium, or high.

  Optional parameters are enclosed in square brackets ([ ]) as in the following syntax example:

      pin1 [*pin2 ... pinN*]

  In this example, you must enter at least one pin name (*pin1*), but others are optional ([*pin2 ... pinN*]).

Getting Help

If you have a question while using Synopsys products, use the following resources:

1. Start with the available product documentation installed on your network or located at the root level of your Synopsys CD-ROM. Every documentation set contains overview information in the intro.pdf file.

   Additional Synopsys documentation is available at this URL:

       http://www.synopsys.com/products/lm/doc

   Datasheets for models are available using the Model Directory:

       http://www.synopsys.com/products/lm/modelDir.html

2. Visit the online Support Center at this URL:

       http://www.synopsys.com/support/lm/support.html

This site gives you access to the following resources:

- ❍ SOLV-IT!, the Synopsys automated problem resolution system

- ❍ product-specific FAQs (frequently asked questions)

- ❍ lists of supported simulators and platforms

- ❍ the ability to open a support help call

- ❍ the ability to submit a delivery request for some product lines

3. If you still have questions, you can call the Support Center:

North American customers:
Call the Synopsys Eagle*i* and Logic Modeling Products Support Center hotline at 1-800-445-1888 (or 1-503-748-6920) from 6:30 AM to 5 PM Pacific Time, Monday through Friday.

International customers:
Call your local sales office.

The Synopsys Website

General information about Synopsys and its products is available at this URL:

http://www.synopsys.com

Synopsys Common Licensing (SCL) Document Set

Synopsys common licensing (SCL) software is delivered on a CD that is separate from the tools that use this software to authorize their use. The SCL documentation set includes the following publications, which are located in (root)/docs/scl on the SCL CD and also available on the Synopsys FTP server (ftp://ftp.synopsys.com):

- *Licensing QuickStart*—(142K PDF file)
  This booklet provides instructions for obtaining an electronic copy of your license key file and for installing and configuring SCL on UNIX and Windows NT.

- *Licensing Installation and Administration Guide*—(2.08M PDF file)
  This guide provides information about installation and configuration, key concepts, examples of license key files, migration to SCL, maintenance, and troubleshooting.

You can find general SCL information on the Web at:

http://www.synopsys.com/keys

Comments?

To report errors or make suggestions, please send e-mail to:

doc@synopsys.com

To report an error on a specific page, select the entire page (including headers and footers), and copy to the buffer. Then paste the buffer to the body of your e-mail message. This will provide us with the information we need to correct the problem.

1
Using Synopsys Models with Simulators

Overview

There are a variety of different types of models used in the verification process. This manual covers the following kinds:

- SmartModels (including FlexModels)

- MemPro models

- Hardware models

SmartModels and FlexModels are binary behavioral models that connect to over 30 commercial simulators through the SWIFT interface. If you are using a SWIFT simulator that does not have a separate chapter devoted to it in this manual, refer to this chapter for the basic information needed to get the models working on your simulator. For information on SmartModel/FlexModel supported simulators, refer to the *SmartModel Library Supported Platforms and Simulators Manual.*

MemPro models are produced in Verilog or VHDL and do not use the SWIFT interface. They do require simulator-specific PLI/CLI/FLI code that needs to be bound in to the supported simulator executable. MemPro is supported on the simulators listed in Table 5.

The hardware modeler uses real silicon in combination with specialized hardware and software to represent the full functionality of modeled devices in your simulation. It does not have a standard interface comparable to SWIFT. Hardware models are a combination of hardware and software, as follows:

- The hardware consists of the actual silicon of the device being modeled, installed on a special-purpose Device Adapter and inserted into the hardware modeling system.

● The software consists of a series of ASCII files containing Shell Software that describes the device interface and initialization, along with optional information such as timing delays, state tracking, and timing checks.

For simulator-specific information about using hardware models, refer first to "Using Hardware Models with Different Simulators" on page 35 for an overview and then consult the appropriate simulator-specific chapter in this manual for detailed setup procedures.

The procedures in this chapter are organized into the following major sections:

● "Using SmartModels with SWIFT Simulators" on page 18

● "Using FlexModels with SWIFT Simulators" on page 24

● "Using MemPro Models with VHDL and Verilog Simulators" on page 29

● "Using Hardware Models with Different Simulators" on page 35

Using SmartModels with SWIFT Simulators

SWIFT is a standard EDA event-level simulation interface developed by Synopsys. The SWIFT interface enables multiple simulators with different requirements to use models from the same SmartModel Library. Each simulator provides a standard model interface, specified by SWIFT, that allows it to load the same SmartModel Library.

When the simulator encounters a SmartModel during simulation, it uses a set of SWIFT functions to create and configure the model, map to its ports, initialize it, and set its time units. The SWIFT interface also allows participating simulators to integrate the SmartModel Library into their particular framework, including application-specific menus. For more information, refer to the documentation provided by your simulator vendor.

SmartModel SWIFT Parameters

SmartModel attributes or parameters are model-specific values needed by the simulator to configure a model. You configure SmartModels when you instantiate them in your design using these SWIFT parameters. This could take the form of Verilog defparams, VHDL generics, or symbol properties, depending on the simulator you are using. For details, refer to the documentation for your simulator.

Table 1 lists the SmartModel configuration parameters. All SmartModels require an InstanceName, TimingVersion, and DelayRange. In addition, some SmartModels need a MemoryFile, JEDECFile, SCFFile, or PCLFile attribute. FlexModels use a slightly different set of attributes for configuration, described in "FlexModel SWIFT Parameters" on page 24.

Table 1: SmartModel SWIFT Parameters

Parameter Name	Used By	Description
InstanceName	All SmartModels	Specifies an instance name for a particular instance of a SmartModel. Used in messages to indicate which instance is issuing the message; also used in user-defined timing. Can be set by the simulator from the hierarchical name in the HDL description; or can be set using the InstanceName property on the symbol.
TimingVersion	All SmartModels	Specifies the timing version a SmartModel instance should use when scheduling changes on its outputs or checking setup and hold times on its inputs. The allowed values are "min," "typ," and "max."
DelayRange	All SmartModels	Specifies a propagation delay range for a particular instance of a SmartModel.
MemoryFile	SmartModels with internal memory such as RAMs, ROMs, and processors and controllers that have on-chip RAM or ROM.	Specifies a memory image file (MIF) to load for a particular instance of a SmartModel.
JEDECFile	JEDEC-based PAL and PLD models	Specifies a JEDEC file to load for a particular instance of a SmartModel.
SCFFile	FPGAs and CPLDs	Specifies a model command file (MCF) to load for a particular instance of a SmartModel.

Table 1: SmartModel SWIFT Parameters (Continued)

Parameter Name	Used By	Description
PCLFile	Processor models (for example, microprocessors and microcontrollers); these are usually hardware verification models.	Specifies a compiled PCL program file to load for a particular instance of a SmartModel.

☞ **Note**

To determine the required configuration file (i.e., MemoryFile, JEDECFile, SCFFile, or PCLFile) for any SmartModel, refer to the model's datasheet.

Instantiating SmartModels

If you are using an HDL-based simulator, generate a model wrapper file (*model*.v or *model*.vhd) using your simulator vendor's procedure. Use the model wrapper to instantiate the model in your design. The model wrapper must map the model's ports to signals in your design. Modify SWIFT parameters in the model wrapper as needed. Here are some parameter examples for a SmartModel memory model:

VHDL:

```
U1: cyc7150
    GENERIC MAP(
    TimingVersion => "cy7c150",
    DelayRange => "MAX",
    MemoryFile => "mem1";
```

Verilog:

```
defparam
    u1.TimingVersion = "cy7c150",
    u1.DelayRange = "MAX",
    u1.MemoryFile = "mem1";
```

You can also instantiate SmartModels in schematic-capture based systems by using model symbols and attaching values to symbol properties. For details on instantiating SmartModels using symbols with QuickSim II, refer to "Using QuickSim II with Synopsys Models" on page 217.

The SWIFT Command Channel

The SWIFT interface specification requires that simulator vendors include a minimal command interface to the SmartModel Library. This interface is called the command channel. The command channel supports several types of commands:

Model Commands

Model commands affect only a selected model instance. Following is a list of the model commands:

DumpMemory *output_file*
> Dumps the current memory image of a model to the specified output file. If *output_file* exists, it is overwritten; otherwise, a new file is created.

ReportStatus
> Prints a message that describes the configuration status of a model.

SetConstraints ON | OFF
> Enables or disables timing constraint checks for a model. By default, models check for and warn of timing constraints.

👉 **Note**
> Some simulator vendors supply additional interfaces to the DumpMemory, ReportStatus, and SetConstraints capabilities.

SmartBrowser Commands for SmartCircuit Models

In addition to the model commands which apply to all SmartModels, the command channel also supports the following SmartBrowser commands for SmartCircuit models:

- Analyze Commands
- Assign Commands
- Examine Commands
- List Commands
- Set and Show Commands
- Trace Commands
- General Commands

For more information about these SmartBrowser commands, refer to the *SmartModel Library Users Manual*.

Session Commands

Session commands act on all models in the simulation session. You can enable session commands by setting the LMC_COMMAND environment variable. Here is an example that enables tracing of timing files and model versions, followed by a list of all the session commands.

```
% setenv LMC_COMMAND "TraceTimeFile on;TracePath ON"
```

☞ **Note**
> The session command strings are case-insensitive, as illustrated above
> (ON and on are equivalent).

TraceTimeFile ON | OFF
> Enables or disables trace messages that list the timing files loaded at simulation startup. The default is OFF.

TracePath ON | OFF
> Enables or disables tracing of paths to files used to determine versions of models. The default is OFF.

Verbose ON | OFF
> Enables or disables the generation of error messages when a SmartModel instance cannot be created. The default is OFF.

NoLicenseFatal ON | OFF
> When set to ON, causes the SWIFT session to send a fatal error message to the simulator and terminate if any SmartModel in the simulation fails to authorize. The default is OFF.

⏰ **Attention**
> You must invoke the TraceTimeFile, TracePath, and NoLicenseFatal commands before the start of the simulation run if you want them to take effect for that session.

Fault Simulations

The SmartModel Library fault simulation capability is folded into the logic simulation SmartModel Library so that only one set of directories and utilities need to be installed and maintained. Fault simulation availability depends on the:

- **Model**—Note that the following types of models are incompatible with fault simulation:

  - Hardware verification (HV) models are driven by PCL commands rather than machine instructions and do not respond adequately to propagated faults. Fault simulation results may not be as accurate when HV models are present in the circuit.

  - FlexModels do not support fault simulation.

  - SmartCircuit models do not support fault simulation.

- **Simulator**—Fault analysis is supported by Mentor's QuickFault II, VEDA's VerdictFault, and Teradyne's LASAR. For more information about fault simulation support, refer to your simulator documentation.

- **Authorization**—You need one of the following license features:

  - simmodel-ultra

  - simmodel-prem

  - simmodel-sw-all

  - simmodel-std

  - simmodel-base

  - simmodel-sw-model-*model_name* (e.g., simmodel-sw-ttl00)

In most cases you can use the same circuit description for both logic and fault simulation. However, you may need to supply different circuit stimuli for each type of simulation. All model messages except version, copyright, and configuration error messages are suppressed in fault simulation. Usage and timing messages are suspended because they are meaningless in a fault simulation. In order to work efficiently during a fault simulation, each model manages its own diverge and converge operations.

Using FlexModels with SWIFT Simulators

Regardless of which simulator you are using, you must configure FlexModels by defining the required SWIFT parameters or attributes shown in Table 2 for each FlexModel instance in your design. You configure FlexModels when you instantiate them in your design using these SWIFT parameters. This could take the form of Verilog defparams, VHDL generics, or symbol properties, depending on the simulator you are using.

Table 2: FlexModel SWIFT Parameters

Parameter[a]	Data Type	Description
FlexTimingMode	FLEX_TIMING_MODE_OFF (default) FLEX_TIMING_MODE_ON FLEX_TIMING_MODE_CYCLE	Disables/enables timing simulation. (For Verilog, prepend a back quote (') to the constant.) ***Note***: Direct C Control users can set this parameter to: - "0" for timing mode off - "1" for timing mode on - "2" for cycle-based timing
TimingVersion	Model timing version	The FlexModel timing version. Refer to the individual FlexModel datasheets for available timing versions.
DelayRange	"MIN", "TYP", "MAX" (default)	If you set FlexTimingMode to on, you can select MIN, TYP, or MAX delay values with this parameter.
FlexModelId	"*instance_name*"	A unique name that identifies each FlexModel instance. This name is also used by the flex_get_inst_handle command to get an integer instance handle. ***Note***: Used only with _fx models
FlexModelId_*cmd_stream*	"*instance_name*"	A unique name that identifies each FlexModel instance or command stream. This name is also used by the flex_get_inst_handle command to get an integer instance handle. For information on *cmd_stream* names, refer to the individual FlexModel datasheets. ***Note***: Used only with _fz models.

Table 2: FlexModel SWIFT Parameters (Continued)

Parameter[a]	Data Type	Description
FlexCFile	"*path_to_C_file* -u | -c"	Specifies the path to an executable C program and whether to start up in coupled (-c) or uncoupled (-u) mode. Uncoupled mode is the default. *Note***:** Used only with _fx models for Direct C Control.
FlexModelSrc_*cmd_stream*	"*path_to_C_file* -u | -c"	If you want to control a FlexModel using Direct C Control, change the default value for *cmd_stream* (HDL) to the name of the command stream defined in the individual FlexModel datashee*ts*. Use this parameter to specify the path to an executable C program and whether to start up in coupled (-c) or uncoupled (-u) mode. Uncoupled mode is the default. *Note***:** Used only with _fz models for Direct C Control.

a. Some FlexModels have additional SWIFT parameters that need to be specified to configure internal memory (for example, the usbhost_fz). For details, refer to the individual FlexModel datasheets.

flexm_setup Command Reference

In addition to specifying SWIFT parameters, you must run the flexm_setup utility each time you install a new or updated FlexModel into your $LMC_HOME tree. This ensures that you pick up the latest package files for that version of the model.

Syntax

flexm_setup [-help] [-dir *path*] *model*

Argument

model   Pathname to the FlexModel you want to set up.

Switches

-help   Prints help information.

-d[ir] *path*   Copies the contents of the FlexModel's versioned src/verilog and src/vhd directories into *path*/src/verilog and *path*/src/vhdl. The directory specified by *path* must already exist.

Examples

When run without the -dir switch, flexm_setup just prints the name of the versioned directory of the selected model's source files

```
# Lists name of versioned directory containing source files
% flexm_setup mpc860_fx
```

When run with the -dir switch pointing to your working directory, flexm_setup copies over all the versioned package files you need to that working directory.

```
# Creates copy in 'flexmodel' directory of model source files
% mkdir workdir
% flexm_setup -dir workdir mpc860_fx
```

Instantiating FlexModels with Direct C Control

Direct C Control is how you use FlexModels on SWIFT simulators with standard FlexModel integrations. With Direct C Control, all model commands come from an external compiled C program that you point to using the FlexCFile SWIFT parameter. For users familiar with Synopsys Hardware Verification models, this is similar to setting the PCLFile parameter to point to the location of a compiled PCL program. In addition, you must also set the FlexModelId parameter, which does not have a default value. To generate model wrappers and instantiate models, you use the same simulator-specific procedures as you would for traditional SmartModels.

Note that the individual FlexModel datasheets document the command syntax and examples for issuing model commands from Verilog, VHDL, VERA, or C, but HDL control and switching between different command sources from the testbench are only available to the simulators with custom integrations. Other SWIFT simulators must stick to Direct C Control for issuing commands to FlexModels.

To use Direct C Control, follow these steps:

1. If you are using an HDL-based simulator, generate a model wrapper file (*model*_fx.v or *model*_fx.vhd) using your simulator vendor's procedure. Use the model wrapper to instantiate the model in your design. Add the FlexCFile parameter to the model instantiation and point it to the location of *your_compiled_C_file* that you create to drive commands into the model. Modify other SWIFT parameters in the model wrapper as needed. Here are some examples for how to instantiate a model for use with Direct C Control:

VHDL:

```
U1: pcimaster
      GENERIC MAP(
      FlexModelId => "modelId_1",
      FlexCFile => "./tb.o",
      FlexTimingMode => "1",
      TimingVersion => "pcimaster",
      DelayRange => "MAX"
```

Verilog:

```
defparam
      u1.FlexModelId = "modelId_1",
      u1.FlexCFile = "./tb.o",
      u1.FlexTimingMode = "1",
      u1.TimingVersion = "pcimaster",
      u1.DelayRange = "MAX";
```

For both of these examples, the C testbench file must have the same instance name, as follows:

```
int Id_1, status;
      char *sInstName = "modelId_1";

/* Get the instance handle */

      flex_get_inst_handle(sInstName,&Id_1, &status);
```

2. Create a working directory and run flexm_setup to make a copy of the model's C object file there, as shown in the following example:

```
% $LMC_HOME/bin/flexm_setup -dir workdir model_fx
```

You must run flexm_setup every time you update your FlexModel installation with a new model version. Table 3 lists the files that flexm_setup copies to your working directory.

Table 3: FlexModel Direct C Control Files

File Name	Description	Location
*model*_pkg.o	Model-specific functions for UNIX.	*workdir*/src/C/
*model*_pkg.obj	Model-specific functions for NT.	*workdir*/src/C/

3. Compile the C object files in with the C program that you write to drive commands into the model (represented in the following examples as *your_C_file.c*). Note that these examples include creation of a working directory (*workdir*) and running flexm_setup, as explained in the previous step. The compile line differs based on your platform:

    a. On HP-UX, you need to link in the -LBSD library as shown in the following example:

```
% mkdir workdir
% flexm_setup -dir workdir model_fx
% /bin/c89 -o executable_name \
your_C_file.c \
workdir/src/C/hp700/model_pkg.o \
$LMC_HOME/lib/hp700.lib/flexmodel_pkg.o \
-I$LMC_HOME/sim/C/src \
-Iworkdir/src/C \
-lBSD
```

    b. On Solaris, you need to link in the -lsocket library as shown in the following example:

```
% mkdir workdir
% flexm_setup -dir workdir model_fx
% cc -o executable_name \
your_C_file.c \
workdir/src/C/solaris/model_pkg.o \
$LMC_HOME/lib/sun4Solaris.lib/flexmodel_pkg.o \
-I$LMC_HOME/sim/C/src \
-Iworkdir/src/C \
-lsocket
```

    c. AIX:

```
% mkdir workdir
% flexm_setup -dir workdir model_fx
% /bin/cc -o executable_name \
your_C_file.c \
workdir/src/C/ibmrs/model_pkg.o \
${LMC_HOME}/lib/ibmrs.lib/flexmodel_pkg.o \
-Iworkdir/src/C \
-I${LMC_HOME}/sim/C/src \
-ldl
```

d. Linux:

```
% mkdir workdir
% flexm_setup -dir workdir model_fx
% egcs -o executable_name \
your_C_file.c \
workdir/src/C/x86_linux/model_pkg.o \
${LMC_HOME}/lib/x86_linux.lib/flexmodel_pkg.o \
-Iworkdir/src/C \
-I${LMC_HOME}/sim/C/src
```

e. On NT, you need to link in a Windows socket library as shown in the following
   example.

```
> md workdir
> flexm_setup -dir workdir model_fx
> cl -O2 -MD -DMSC -DWIN32 -Feexecutable_name
your_C_file.c
workdir\src\C\pcnt\model_pkg.obj
%LMC_HOME%\lib\pcnt.lib\flexmodel_pkg.obj
-I%LMC_HOME%\sim\C\src
-Iworkdir\src\C
wsock32.lib
```

Note

The entire compilation expression must appear on the same line. The NT
example was tested using Microsoft's Visual C++ compiler v5.0.

The C executable file that you created in this step is the program that you point to using
the FlexCFile SWIFT attribute for the model instance in your design.

Using MemPro Models with VHDL and Verilog Simulators

Regardless of which simulator you are using, you must configure MemPro models by
defining the required parameters or attributes shown in Table 4 for each MemPro model
instance in your design. You configure MemPro models when you instantiate them in
your design using these generics or parameters.

Table 4: MemPro Generic/Parameter Descriptions

Name	Data Type	Description
model_id	Integer	Either the model_id or model_alias generic or parameter specifies a unique user handle for a specified model instance. This user handle is used to address a memory model using testbench commands.
model_alias	String	
		Note: You do not have to assign all MemPro model instances a model_id or model_alias, only those instances on which you wish to use the testbench interface. However, each model with a model_id or model_alias must be assigned a unique handle.
memoryfile	String	Specifies the file name of the memory image file to preload during model initialization. If memoryfile is set to a null string (memoryfile = ""), memory image preloading during initialization is disabled. Supported files formats are SmartModel Memory Image, Motorola S-Record, Intel Hex, and Verilog $readmemh. Memory models can also be loaded using the mem_load command.
default_data	String	Specifies the default data returned from all uninitialized memory addresses. *Note*: Models in non-volatile memory classes may not have their Default Memory Value set to anything except all ones. Any other setting is ignored and MemGen issues an warning.
message_level	Integer	Specifies the type or types of messages returned by the model. For a detailed description of message types, refer to "Controlling MemPro Model Messages" on page 33

MemPro models are supported on the simulators listed in Table 5.

Table 5: MemPro Supported Simulators

Verilog Simulators	VHDL Simulators
VCS	Scirocco
Verilog-XL	VSS
NC-Verilog	Cyclone
MTI Verilog	Leapfrog
	MTI VHDL
	NC-VHDL

Each of the simulators in Table 5 has its own chapter in this manual that explains the simulator-specific procedure for using MemPro models in those environments.

Using MemPro Models with VHDL Simulators

This section describes how to include MemPro memory models and testbench interface commands in your design. The MemPro VHDL interface code is contained in the following files:

slm_hdlc.vhd   Simulator-specific HDL-to-C interface code.

mempro_pkg.vhd   MemPro-specific module containing the VHDL
                          implementation of the MemPro testbench interface.

rdramd_pkg.vhd   RDRAM-specific module.

All of these files are located in the $LMC_HOME/sim/*simulator*/src directory.

Using MemPro Models with Verilog Simulators

This section describes how to include MemPro memory models and testbench interface commands in your design. The following files define MemPro PLI routines and interface commands:

slm_pli.o   PLI routines. This file is located in the
                       $LMC_HOME/lib/*platform*.lib directory.

mempro_pkg.v   Verilog testbench task definitions for MemPro interface
                       commands. This file is located in the
                       $LMC_HOME/sim/pli/src directory.

mempro_c_tb.h   C testbench function definitions for MemPro interface commands. This is located in the $LMC_HOME/include directory.

Instantiating MemPro Models

You instantiate MemPro models just like any other HDL models, as shown in the following DRAM examples.

MemPro Verilog Instantiation

```
dram1x64 bank1
        ( .ras  ( rasr ),
        .ucas  ( ucasr ),
        .lcas  ( lcasr ),
        .we    ( wer ),
        .oe    ( oer ),
        .a     ( adrr ),
        .dq    ( dataw ));

defparam bank1.model_id     = "tbench.bank1",
         bank1.memoryfile   = "dram.dat",
         bank1.message_level = `SLM_XHANDLING | `SLM_TIMING | `SLM_WARNING,
         bank1.default_data = 64'hxxx;
```

MemPro VHDL Instantiation

```
U1 : dram1x64
 generic map ( model_id      => 10,
               memoryfile    => "dram.dat",
               message_level => (SLM_TIMING + SLM_XHANDLING + SLM_WARNING),
               default_data  => "XXXX");
 port map
              ( a             => adrw,
                dq            => dataw,
                ras           => rasw,
                lcas          => lcasw,
                ucas          => ucasw,
                we            => wew,
                oe            => oew );
```

Controlling MemPro Model Messages

MemPro model messages are grouped into categories that you can individually enable or disable for each model instance. Several message categories are applicable to all models; additional categories may be defined for specific models or model types. The general categories are:

Fatal

Fatal messages are always enabled. When a fatal error is detected, the simulation stops immediately after reporting the message. For example, referencing an unknown MemPro model instance handle causes a fatal error.

Error

Error messages apply to incorrect situations from which the model is able to recover, allowing simulation to continue. For example, MemPro generates an error message when the model receives a command that would put it in an illegal state.

Warning

Warning messages apply to situations that users may want to check, but are not obviously wrong. For example, MemPro generates a warning message when significant bits of an address are ignored.

Info

Info messages inform you of the status or behavior of the model. MemPro generates info messages infrequently. For example, when a memory model is initialized from a file, MemPro issues an info message.

Timing

MemPro uses timing messages to report timing constraint violations. Typical situations that cause timing messages are setup or pulse-width violations.

X-Handling

MemPro generates X-handling messages if a model samples unknowns on input ports when valid data was expected.

Controlling MemPro Message Output

There are three ways to control messaging for MemPro models:

1. Set individual Message Settings when you specify the model message categories (except Fatal).

2. Use the message_level generic or parameter. For more information, refer to .

3. Use a command stream or testbench command.

By default, MemPro models display all the general message categories (Fatal, Error, Warning, Info, Timing, and X-handling). If you set a generic or parameter for a model instance, that setting overrides the default behavior. In turn, if the command stream or testbench interface is used, it overrides the generic or parameter value.

Message Level Constants

MemPro provides constants for setting message levels on each instantiated model. The constants described in Table 6 are defined in mempro_pkg.v (for Verilog simulators) and mempro_pkg.vhd (for VHDL simulators).

Table 6: MemPro Message Constant Descriptions

Constant	Value[a]	Description
SLM_ERROR	1	Fatal and error messages generated.
SLM_WARNING	2	Fatal and warning messages generated.
SLM_TIMING	4	Fatal and timing messages generated.
SLM_XHANDLING	8	Fatal and X-handling messages generated.
SLM_INFO	16	Fatal and info messages generated.
SLM_ALL_MSGS	$2^{28}-1$	All message types generated.
SLM_NO_MSGS	0	Only fatal messages generated.

a. Note that bits 5 through 27 are unused but reserved.

You can combine these constants to get any combination of messages you desire. The following Verilog and VHDL code fragments define a model instantiation having timing, X-handling, and warning (as well as fatal) messages enabled.

Verilog

```
bank1.message_level = `SLM_XHANDLING | `SLM_TIMING | `SLM_WARNING,
```

VHDL

```
message_level => (SLM_TIMING + SLM_XHANDLING + SLM_WARNING),
```

Using Hardware Models with Different Simulators

After you install your hardware modeling system, the final task is to link your simulator with the Synopsys Simulator Function Interface (SFI). Procedures for linking the simulator with the SFI are specific to the particular simulator.

Synopsys provides four ModelAccess products, supporting QuickSim II, Cyclone, Verilog-XL, and NC-Verilog. For usage information, refer to the following sections in this book:

- "Using Hardware Models with QuickSim II" on page 244

- "Using Hardware Models with Cyclone" on page 174

- "Using Hardware Models with Verilog-XL" on page 83

- "Using Hardware Models with NC-Verilog" on page 108

Linking Other Supported Simulators

Because many hardware modeling features are provided through the SFI software, the functionality of your environment is determined by the version of the SFI that is integrated with your simulator. Some simulators can be dynamically or statically linked on site with the most recent SFI. For the current list of simulators and versions that are supported for dynamic or static linking on site with the SFI, refer to *Hardware Modeling Supported Platforms and Simulators*.

If you use one of the simulators on this list, you can link your simulator with the most recent version of the SFI libraries on the distribution media, allowing you to take advantage of the latest hardware modeling system software enhancements and bug fixes. Some simulators have additional requirements. For information, refer to your simulator vendor's documentation.

If you use a simulator that is not on the list, consult your simulator vendor about which version of the SFI has been integrated with your simulator. Depending on the version of the SFI, you should be able to install and use the most recent Runtime Modeler Software, although you may not be able to take advantage of all hardware modeling system software enhancements and fixes.

IKOS Voyager

For information on this interface, refer to the Voyager/LM Hardware Interface chapter of the *Voyager Series User's Guide, Volume 4*.

Do not install the hardware modeling system software under the $VOYAGER_HOME directory, or files could be overwritten and the installation corrupted. The IKOS-created sms directory (under $VOYAGER_HOME) and the Synopsys-created sms directory must be kept separate.

Teradyne LASAR

You can dynamically link the SFI with LASAR. For complete Teradyne-specific installation information, refer to Teradyne's *LASAR Manager Guide for UNIX Systems*.

VEDA Vulcan

You can dynamically link the SFI with Vulcan at simulator runtime. For current linking information, please contact VEDA technical support directly.

Viewlogic Fusion ViewSim

You can statically link the SFI on-site with ViewSim. For information, refer to the Viewlogic Fusion ViewSim manual or contact Viewlogic technical support directly at 1-800-223-8439. In addition, Synopsys provides a SOLV-IT! article with some information. For instructions on accessing SOLV-IT!, refer to "Getting Help" on page 14.

2

Using VCS with Synopsys Models

Overview

This chapter explains how to use SmartModels, FlexModels, MemPro models, and hardware models with VCS. These procedures are centered on VCS v5.1, but contain notes about other versions of VCS as well. The procedures are organized into the following major sections:

- "Setting Environment Variables" on page 38

- "Using SmartModels with VCS" on page 39

- "Using FlexModels with VCS" on page 42

- "Using MemPro Models with VCS" on page 51

- "Using Hardware Models with VCS" on page 55

🔆 **Hint**

This chapter includes a script that you can use to run any FlexModel examples testbench with minimal setup required. You can cut-and-paste the script right out of this PDF file. Refer to "Script for Running FlexModel Examples in VCS" on page 49.

Setting Environment Variables

First, set the basic environment variables. If you are not using one of the model types, skip that step. In some cases the procedures that follow in this chapter include steps for setting additional environment variables.

1. Set the LMC_HOME variable to the location of your SmartModel. FlexModel, and MemPro model installation tree, as shown in the following example:

   ```
   % setenv LMC_HOME path_to_models_installation
   ```

2. Set the LM_LICENSE_FILE or SNPSLMD_LICENSE_FILE environment variable to point to the product authorization file, as shown in the following example:

   ```
   % setenv LM_LICENSE_FILE path_to_product_authorization_file
   ```

   ```
   % setenv SNPSLMD_LICENSE_FILE path_to_product_authorization_file
   ```

   You can put license keys for multiple products (for example, SmartModels and hardware models) into the same authorization file. If you need to keep separate authorization files for different products, use a colon-separated list (UNIX) or semicolon-separated list (NT) to specify the search path in your variable setting.

   Caution

   Do not include la_dmon-based authorizations in the same file with snpslmd-based authorizations. If you have authorizations that use la_dmon, keep them in a separate license file that uses a different license server (lmgrd) process than the one you use for snpslmd-based authorizations.

3. If you are using the hardware modeler, set the LM_DIR and LM_LIB environment variables, as shown in the following examples:

   ```
   % setenv LM_DIR hardware_model_install_path/sms/lm_dir
   % setenv LM_LIB hardware_model_install_path/sms/models: \
    hardware_model_install_path/sms/maps
   ```

   If you put your models in a directory other than the default of /sms/models, modify the above variable setting accordingly.

4. Depending on your platform, set your load library variable to point to the platform-specific directory in $LMC_HOME, as shown in the following examples:

   Solaris:

   ```
   % setenv LD_LIBRARY_PATH $LMC_HOME/lib/sun4Solaris.lib:$LD_LIBRARY_PATH
   ```

   Linux:

   ```
   % setenv LD_LIBRARY_PATH $LMC_HOME/lib/x86_linux.lib:$LD_LIBRARY_PATH
   ```

AIX:

```
% setenv LIBPATH $LMC_HOME/lib/ibmrs.lib:$LIBPATH
```

HP-UX:

```
% setenv SHLIB_PATH $LMC_HOME/lib/hp700.lib:$SHLIB_PATH
```

NT:

Make sure that %LMC_HOME%\lib\pcnt.lib is in the Path user variable.

5. Set the VCS_HOME variable to the location of your VCS installation tree, as shown in the following example, and make sure that VCS is set up properly in your environment:

```
% setenv VCS_HOME VCS_install_path
```

6. Set the VCS_SWIFT_NOTES variable to 1, as shown in the following example:

```
% setenv VCS_SWIFT_NOTES 1
```

Using SmartModels with VCS

To use SmartModels with VCS, follow this procedure:

1. Generate a Verilog model wrapper file by running VCS with the -lmc-swift-template switch, as shown in the following example:

```
% $VCS_HOME/bin/vcs -lmc-swift-template model
```

where *model* is the name of the model you want to use.

This will create a *model*.swift.v file. Note that VCS version 5.1 and up generates bused Verilog wrapper files by default. If you want bit-blasted wrappers, use the -old switch.

Synopsys provides a tool, vcs_sg, that allows you to generate multiple model wrapper files. You must select VCS as your Verilog simulator during the SmartModel installation in order to have vcs_sg available. It will be installed as

```
$LMC_HOME/bin/vcs_sg
```

The vcs_sg tool also extends the usefulness of the model wrapper files generated by VCS in two ways:

❍ it adds statements that allow the DelayRange to be controlled by the VCS command line +define parameters (or a defparam in your testbench)

❍ it adds a check for the VCS command line +define+SwiftChecksOff parameter that turns constraints off.

You can change the default name of the generated wrapper files (*<model>*.swift.v), as well as the location that the generated wrappers are written to. Invoke

$LMC_HOME/bin/vcs_sg -h

to return the usage message for the vcs_sg tool:

```
HP-UX B.10.20 /u/mikehu [3]==> $LMC_HOME/bin/vcs_sg -h
Copyright(C)2001 Logic Modeling, Synopsys Inc. ALL RIGHTS RESERVED
@(#) vcs_sg.pl $Revision: /main/5 $  $Date: 2001/03/02 13:42:03 $
################################################################
Generate .v files for VCS for Smartmodels

 Command line options:
   -h this message
   -l <path> to specify location of created files
       default is current directory
   -m <model_name> to specify which model to process
       default is entire library if no -m switch
       multiple -m <model_name> switches may be used
   -u convert file name to uppercase
   -t truncate name to <model>.v
   -a add mode, generate only models without .v file
      in specified destination
   -o <file_name> create log file file_name
   -b bit only i/f (no bus i/f if supported)
   -v verbose mode to see all messages.
   -d set debug mode.


################################################################
```

2. Instantiate SmartModels in your design, defining the ports and defparams as required. For details on the required SWIFT parameters and SmartModel instantiation examples, refer to "SmartModel SWIFT Parameters" on page 18.

3. Invoke the VCS simulator as shown in the following examples:

Solaris:

```
% $VCS_HOME/bin/vcs -lmc-swift model.swift.v model_tb.v \
-l vcs_sim.log \
-Mupdate \
-RI
```

HP-UX:

```
% $VCS_HOME/bin/vcs -lmc-swift model.swift.v model_tb.v \
-l vcs_sim.log \
-Mupdate \
-RI \
-LDFLAGS "-a shared -lm -lc -a archive"
```

AIX:

```
% $VCS_HOME/bin/vcs -lmc-swift model.swift.v model_tb.v \
-l vcs_sim.log \
-Mupdate \
-RI \
-LDFLAGS -lld
```

Linux:

```
% $VCS_HOME/bin/vcs -lmc-swift model.swift.v model_tb.v \
-l vcs_sim.log \
-Mupdate \
-RI \
-LDFLAGS -rdynamic
```

where *model*.swift.v is the template you created in the previous step and *model*_tb.v is the testbench where the model is instantiated. Each model instantiated in the testbench must have a *model*.swift.v wrapper file listed on the VCS invocation line.

VCS SmartModel Explanation

Table 7 lists each line in the invocation examples above, along with explanations for what each one does.

Table 7: VCS SmartModel Explanation

Line Reference	Description
`$VCS_HOME/bin/vcs` `-lmc-swift` *model*`.swift.v` *model*`_tb.v`	Path to the file that starts the VCS simulator, a switch that causes VCS to load the SWIFT interface, and then the specified model wrapper and Verilog testbench files.
`-l vcs_sim.log`	Specifies a log file where VCS writes compilation and simulation messages.
`-Mupdate`	This specifies incremental compilation, which causes VCS to compile only the modules that have changed since the last run.
`-RI`	This makes VCS run interactively. VCS invokes the XVCS GUI after compilation and pauses the simulator at time zero.
`-LDFLAGS` *switches*	Additional platform-specific switches that may be needed.

Using FlexModels with VCS

To use FlexModels with VCS, follow this procedure. VCS links the external PLI routines that contain the custom FlexModel integration code during compilation of your design. This procedure covers users on UNIX and NT. If you are on NT, substitute the appropriate NT syntax for any UNIX command line examples (percent signs around variables and backslashes in paths).

1. If you want the improved performance that comes with bused wrappers, generate a Verilog model wrapper file by running VCS with the -lmc-swift-template switch, as shown in the following example:

       % **$VCS_HOME/bin/vcs -lmc-swift-template** *model*

   where *model* is the name of the model you want to use.

   This creates a *model*.swift.v file. Note that VCS version 5.1 and up generates bused Verilog wrapper files by default. For bit-blasted wrappers, use the -old switch.

👉 **Note** ──

   The bused wrappers enable improved performance but do not work with the examples testbench shipped with the model. To exercise the examples testbench, use the wrappers shipped with the model (see Table 8), as explained in the rest of this procedure. If you are using the bused wrappers, adjust accordingly.

──

2. Create a working directory and run flexm_setup to make copies of the model's interface and example files there, as shown in the following example:

       % **$LMC_HOME/bin/flexm_setup –dir** *workdir model_***fx**

   You must run flexm_setup every time you update your FlexModel installation with a new model version. Table 8 lists the files that flexm_setup copies to your working directory.

Table 8: FlexModel VCS Verilog Files

File Name	Description	Location
*model*_pkg.inc	Verilog task definitions for FlexModel interface commands. This file also references the flexmodel_pkg.inc and *model*_user_pkg.inc files.	*workdir*/src/verilog/
*model*_user_pkg.inc	Clock frequency setup and user customizations.	*workdir*/src/verilog/
*model*_fx_vcs.v	A SWIFT wrapper that you can use to instantiate the model.	*workdir*/examples/verilog/

Table 8: FlexModel VCS Verilog Files (Continued)

File Name	Description	Location
model.v	A bus-level wrapper around the SWIFT model. This allows you to use vectored ports for the model in your testbench.	*workdir*/examples/verilog/
*model*_tst.v	A testbench that instantiates the model and shows how to use basic model commands.	*workdir*/examples/verilog/

3. Update the clock frequency supplied in the *model*_user_pkg.inc file to correspond to the CLK period you want for the model. This file is located in:

   > *workdir*/**src/verilog/***model***_user_pkg.inc**

   where *workdir* is your working directory.

4. Add the following line to your Verilog testbench to include FlexModel testbench interface commands in your design:

   > **`include "***model***_pkg.inc"**

☞ **Note**

   Be sure to add *model*_pkg.inc within the module from which you will be issuing FlexModel commands.

   Because the *model*_pkg.inc file includes references to flexmodel_pkg.inc and *model*_user_pkg.inc, you don't need to add flexmodel_pkg.inc or *model*_user_pkg.inc to your testbench.

5. Instantiate FlexModels in your design, defining the ports and defparams as required (refer to the example testbench supplied with the model). You use the supplied bus-level wrapper (*model*.v) in the top-level of your design to instantiate the supplied bit-blasted wrapper (*model*_fx_vcs.v).

   Example using bus-level wrapper (*model*.v) without timing:

   ```
   model U1 ( model ports )
   defparam
           U1.FlexModelId = "TMS_INST1";
   ```

   Example using bus-level wrapper (*model*.v) with timing:

   ```
   model U1 ( model ports )
   defparam
           U1.FlexTimingMode = `FLEX_TIMING_MODE_ON,
           U1.TimingVersion = "timingversion",
           U1.DelayRange = "range",
           U1.FlexModelId= "TMS_INST1";
   ```

6. Invoke VCS to compile and simulate your design as shown in the following examples:

Solaris

```
% vcs -o simv workdir/examples/verilog/model.v \
  workdir/examples/verilog/model_fx_vcs.v \
  $LMC_HOME/lib/sun4Solaris.lib/slm_pli.o \
  testbench.v \
  -P $LMC_HOME/sim/pli/src/slm_pli.tab \
  -lmc-swift \
  +incdir+$LMC_HOME/sim/pli/src \
  +incdir+workdir/src/verilog
% simv
```

HP-UX

```
% vcs -o simv workdir/examples/verilog/model.v \
  workdir/examples/verilog/model_fx_vcs.v \
  $LMC_HOME/lib/hp700.lib/slm_pli.o \
  testbench.v \
  -P $LMC_HOME/sim/pli/src/slm_pli.tab \
  -lmc-swift \
  +incdir+$LMC_HOME/sim/pli/src \
  +incdir+workdir/src/verilog \
  -LDFLAGS "-a shared -lm -lc -a archive"
% simv
```

AIX

```
% vcs -o simv workdir/examples/verilog/model.v \
  workdir/examples/verilog/model_fx_vcs.v \
  $LMC_HOME/lib/ibmrs.lib/slm_pli.o \
  testbench.v \
  -P $LMC_HOME/sim/pli/src/slm_pli.tab \
  -lmc-swift \
  +incdir+$LMC_HOME/sim/pli/src \
  +incdir+workdir/src/verilog \
  -LDFLAGS -lld
% simv
```

Linux

```
% vcs -o simv workdir/examples/verilog/model.v \
  workdir/examples/verilog/model_fx_vcs.v \
  $LMC_HOME/lib/x86_linux.lib/slm_pli.o \
  testbench.v \
  -P $LMC_HOME/sim/pli/src/slm_pli.tab \
  -lmc-swift \
  +incdir+$LMC_HOME/sim/pli/src \
  +incdir+workdir/src/verilog \
  -LDFLAGS -rdynamic
% simv
```

NT

```
> vcs -o simv .\examples\verilog\model.v
workdir\examples\verilog\model_fx_vcs.v
+incdir+%LMC_HOME%\sim\pli\src
+incdir+workdir\src\verilog
testbench.v
-lmc-swift -P
%LMC_HOME%\sim\pli\src\slm_pli.tab
%LMC_HOME%\lib\pcnt.lib\slm_pli_vcs.lib

> simv.exe
```

Note

The entire compilation expression must appear on the same line. The NT example was tested using Microsoft's Visual C++ compiler v5.0.

VCS FlexModel Examples

First we present a basic one-model example and then show you how to use more than one FlexModel in the same simulation in the following sections:

- "One FlexModel on Solaris" on page 46
- "Two FlexModels on Solaris" on page 48
- "Three FlexModels on HP-UX" on page 48

One FlexModel on Solaris

To use one FlexModel with VCS on Solaris, invoke the simulator as shown in the following example:

```
% $VCS_HOME/bin/vcs \
`$LMC_HOME/bin/flexm_setup model_fx`/examples/verilog/model_tst.v \
`$LMC_HOME/bin/flexm_setup model_fx`/examples/verilog/model.v \
`$LMC_HOME/bin/flexm_setup model_fx`/examples/verilog/model_fx_vcs.v \
+incdir+$LMC_HOME/sim/pli/src \
+incdir+`$LMC_HOME/bin/flexm_setup model_fx`/src/verilog \
-P $LMC_HOME/sim/pli/src/slm_pli.tab \
$LMC_HOME/lib/sun4Solaris.lib/slm_pli.o \
+incdir+$LMC_HOME/sim/pli/src \
-l vcs_sim.log \
-Mupdate \
-RI \
-lmc-swift
```

where *model* is the name of the FlexModel you are using.

Table 10 lists each line in the invocation example above, along with explanations for what each one does.

Table 9: VCS With One FlexModel On Solaris Model Explanation

Line Reference	Description
`$VCS_HOME/bin/vcs`	Path to the file that starts the VCS simulator.
`` `$LMC_HOME/bin/flexm_setup `` *model_fx* `` ` `` `/examples/verilog/`*model*`_tst.v`	Specifies the path to the model testbench file.
`` `$LMC_HOME/bin/flexm_setup `` *model_fx* `` ` `` `/examples/verilog/`*model*`.v`	Specifies the path to the model Verilog wrapper file.
`` `$LMC_HOME/bin/flexm_setup `` *model_fx* `` ` `` `/examples/verilog/`*model*`_fx_vcs.v`	Specifies the path to the model VCS template file.
`+incdir+$LMC_HOME/sim/pli/src`	Includes the path to the flexmodel_pkg.inc file, which contains Verilog task definitions for general FlexModel interface commands.
`` +incdir+`$LMC_HOME/bin/flexm_setup `` *model*`_fx`` ` ``/src/verilog`	Includes the path to the model-specific Verilog task files, including *model*_pkg.inc.
`-P $LMC_HOME/sim/pli/src/slm_pli.tab`	Specifies the FlexModel/MemPro PLI table entry file.
`$LMC_HOME/lib/sun4Solaris.lib/slm_pli.o`	Specifies the platform-specific PLI object file.
`-l vcs_sim.log`	Specifies a log file where VCS writes compilation and simulation messages.
`-Mupdate`	This specifies incremental compilation, which causes VCS to compile only the modules that have changed since the last run.
`-RI`	This makes VCS run interactively. VCS invokes the XVCS GUI after compilation and pauses the simulator at time zero.
`-lmc-swift`	This switch causes VCS to load the SWIFT interface.

Two FlexModels on Solaris

This example shows how to use the mpc740_fx and mpc750_l2_fx FlexModels together with VCS on Solaris. Invoke the simulator as shown in the following example:

```
% $VCS_HOME/bin/vcs \
`$LMC_HOME/bin/flexm_setup mpc750_l2_fx`/examples/verilog/mpc750_l2_tst.v \
`$LMC_HOME/bin/flexm_setup mpc750_l2_fx`/examples/verilog/mpc750_l2.v \
`$LMC_HOME/bin/flexm_setup mpc750_l2_fx`/examples/verilog/mpc750_l2_fx_vcs.v \
`$LMC_HOME/bin/flexm_setup mpc740_fx`/examples/verilog/mpc740.v \
`$LMC_HOME/bin/flexm_setup mpc740_fx`/examples/verilog/mpc740_fx_vcs.v \
+incdir+$LMC_HOME/sim/pli/src \
+incdir+`$LMC_HOME/bin/flexm_setup mpc750_l2_fx`/src/verilog \
+incdir+`$LMC_HOME/bin/flexm_setup mpc740_fx`/src/verilog \
-P $LMC_HOME/sim/pli/src/slm_pli.tab \
$LMC_HOME/lib/sun4Solaris.lib/slm_pli.o \
+incdir+$LMC_HOME/sim/pli/src \
-l vcs_sim.log \
-Mupdate \
-RI \
-lmc-swift
```

Three FlexModels on HP-UX

This next example shows how to use the PCI system testbench and the pcimaster_fx, pcislave_fx, and pcimonitor_fx FlexModels together with VCS on HP-UX. Follow these steps:

1. Set up the PCI system testbench as shown in the following example:

```
% mkdir pci_tb
% cp -rf `$LMC_HOME/bin/flexm_setup pcimaster_fx`/* pci_tb
% cp -rf `$LMC_HOME/bin/flexm_setup pcimonitor_fx`/* pci_tb
% cp -rf `$LMC_HOME/bin/flexm_setup pcislave_fx`/* pci_tb
```

2. Invoke the VCS simulator as shown in the following example:

```
% $VCS_HOME/bin/vcs \
./pci_tb/examples/verilog/pcisys_tst.v \
./pci_tb/examples/verilog/pcimaster.v \
./pci_tb/examples/verilog/pcimaster_fx_vcs.v \
./pci_tb/examples/verilog/pcislave.v \
./pci_tb/examples/verilog/pcislave_fx_vcs.v \
./pci_tb/examples/verilog/pcimonitor.v \
./pci_tb/examples/verilog/pcimonitor_fx_vcs.v \
+incdir+./pci_tb/src/verilog \
+incdir+$LMC_HOME/sim/pli/src \
$LMC_HOME/lib/hp700.lib/slm_pli.o \
-P $LMC_HOME/sim/pli/src/slm_pli.tab \
-l vcs_sim.log \
-Mupdate \
-RI \
```

```
-lmc-swift \
-LDFLGS "-a shared -lm -lc -a archive"
```

Script for Running FlexModel Examples in VCS

On page 50 is a Perl script (Figure 1) that you can use to run VCS on a FlexModel examples testbench. You can use this script on any installed FlexModel because each one comes with a prebuilt testbench example that shows how to use the model commands and all the Verilog wrapper and task definition files that you need. This script runs on HP-UX, Solaris, and NT.

To invoke VCS on a FlexModel and its example testbench, follow these steps:

1. Use the Acrobat Reader's text selection tool to select the script shown in Figure 1 and copy the contents to a local file named run_flex_examples_in_vcs.pl.

2. Save the file and change the permissions so that the file is executable (chmod 775 in UNIX).

3. If you are on NT, you also need to copy the following line to a file named run_flex_examples_in_vcs.cmd and put it in your working directory:

   ```
   %LMC_HOME%\lib\pcnt.lib\sl_perl.exe run_flex_example_in_vcs.pl %*
   ```

   On NT you invoke this cmd wrapper, which subinvokes the Perl script.

4. Invoke the script as shown in the following examples:

   UNIX

   ```
   % run_flex_examples_in_vcs.pl model_fx
   ```

   NT

   ```
   > run_flex_examples_in_vcs.cmd model_fx
   ```

   where *model_fx* is the name of the FlexModel you want to run.

Note

This script was developed for internal use and is made available for user convenience. It is not actively maintained as part of the licensed software.

```perl
#!/usr/local/bin/perl
# $Revision$
$output_file = "Example_Simulator_Run_Script";
die "\nERROR running $0: ", "No FlexModel name given\n\n", unless($ARGV[0] );
$LmcHome = $ENV{ LMC_HOME }; die "ERROR running $0: ", "The LMC_HOME environment
variable must be set.\n" unless( $LmcHome );
$VcsHome = $ENV{ VCS_HOME };
die "ERROR running $0: ", "The VCS_HOME environment variable must be set.\n" unless(
$VcsHome );$VcsSwiftNotes = $ENV{ VCS_SWIFT_NOTES };
die "ERROR running $0: ", "The VCS_SWIFT_NOTES environment variable must be set.\n",
"\nSet VCS_SWIFT_NOTES to the value 1\n\n", unless( $VcsSwiftNotes );
require "$LmcHome/lib/bin/libmdl01003.pl";
$Platform = GetPlatform();
$Platform_lib = PlatformToLibDir($Platform);
%platform_suffix = ( hp700 => "o", solaris => "o", pcnt => "lib");
$suffix = $platform_suffix{ $Platform };$flexmodel_name = $ARGV[0];
$model_path = $LmcHome . "/models/" . $flexmodel_name;
if ( -e $model_path ) {}
else { die "\nERROR running $0: ", "FlexModel $flexmodel_name Does not Exist in
Library\n\n"; }$version_path = `$LmcHome/bin/flexm_setup $flexmodel_name`;
chomp($version_path);
if ( $flexmodel_name =~ /_fx/ ) { $flexmodel_name =~ s/_fx//g; $flex_or_c = "_fx";}
elsif ($flexmodel_name =~ /_fz/ ) {
    $flexmodel_name =~ s/_fz//g;
    $flex_or_c = "_fz";}
else { die "\nERROR running $0: ", "$flexmodel_name is not a FlexModel. Model must
have an _fx or _fz to be a FlexModel\n\n";}
$execute_command = $VcsHome . "/bin/vcs -Mupdate -RI -l vcs_sim.log "
                 . $version_path . "/examples/verilog/"
                 . $flexmodel_name . "_tst.v +incdir+"
                 . $version_path . "/src/verilog +libext+.inc "
                 . $version_path . "/examples/verilog/" . $flexmodel_name . ".v "
                 . $version_path . "/examples/verilog/"
                 . $flexmodel_name . $flex_or_c . "_vcs.v ";
if ( $Platform eq "pcnt" ) {
    $execute_command = $execute_command . $LmcHome . $Platform_lib . "slm_pli_vcs." .
$suffix;}
else {
  $execute_command = $execute_command . $LmcHome . $Platform_lib . "slm_pli.o" .
$suffix;}
$execute_command = $execute_command . " -P " . $LmcHome . "/sim/pli/src/slm_pli.tab"
. " -lmc-swift +incdir+" . $LmcHome . "/sim/pli/src"; print "$execute_command\n";
open(OFILE,"> $output_file") || die " Could not create file : $output_file\n";
print OFILE ("# This is an example of VCS command line to run the\n");
print OFILE ("# supplied FlexModel testbench.\n");
print OFILE ("# Note: The model version was calculated using the flexm_setup
command\n");print OFILE ("\n$execute_command\n");
close(OFILE); system($execute_command);
```

Figure 1: run_flex_examples_in_vcs.pl Script

Example Simulator Run Script

The run_flex_examples_in_vcs.pl script also creates an example simulator run script in your current working directory for the specified model. You can use this run script to invoke VCS after running the run_flex_examples_in_vcs.pl script. The following example shows the contents of the "Example_Simulator_Run_Script" after running the run_flex_examples_in_vcs.pl script using the mpc860_fx model.

```
# This is an example of VCS command line to run the supplied FlexModel
testbench.
# Note: The model version was calculated using the flexm_setup command

/d/vcs501/vcs5.0.1A/bin/vcs -Mupdate -RI -l vcs_sim.log
/d/lmgqa2/install/lmc_home/models/mpc860_fx/mpc860_fx02009/examples/veri
log/mpc860_tst.v
+incdir+/d/lmgqa2/install/lmc_home/models/mpc860_fx/mpc860_fx02009/src/v
erilog +libext+.inc
/d/lmgqa2/install/lmc_home/models/mpc860_fx/mpc860_fx02009/examples/veri
log/mpc860.v
/d/lmgqa2/install/lmc_home/models/mpc860_fx/mpc860_fx02009/examples/veri
log/mpc860_fx_vcs.v /d/lmgqa2/install/lmc_home/lib/hp700.lib/slm_pli.o -
P /d/lmgqa2/install/lmc_home/sim/pli/src/slm_pli.tab -lmc-swift
+incdir+/d/lmgqa2/install/lmc_home/sim/pli/src
```

Using MemPro Models with VCS

To use MemPro models with VCS, follow this procedure. VCS links external PLI routines during compilation of your design. You do not need to rebuild the VCS simulator.

1. To use the MemPro C testbench, add the following line to your local copy of slm_pli.tab:

   $_your_task_name_ **call=**_your_fn_name_

2. If you are using MemPro testbench interface commands in your design, perform step 2a or step 2b. If you are not using HDL or C testbenches, skip to step 3.

   a. If you are using Verilog testbench calls, add the following line to your Verilog testbench.

      `include "mempro_pkg.v"

   b. If you are using the MemPro C testbench, add the following line to your C testbench code.

      #include "mempro_c_tb.h"

For more information on using the MemPro testbench interfaces, refer to the "HDL Testbench Interface" and "C Testbench Interface" chapters in the *MemPro User's Manual*.

3. Invoke VCS to compile and simulate your design:

HP-UX

If you *are not* using the MemPro C testbench:

```
% vcs Verilog_modules MemPro_model_files \
      $LMC_HOME/lib/hp700.lib/slm_pli.o \
      -P $LMC_HOME/sim/pli/src/slm_pli.tab \
      +incdir+$LMC_HOME/sim/pli/src -RI \
      -LDFLAGS "-Wl,-a,default -ldld -lc -lm -lBSD"
```

If you *are* using the MemPro C testbench:

```
% vcs Verilog_modules MemPro_model_files \
      $LMC_HOME/lib/hp700.lib/slm_pli.o \
      -P local_slm_pli.tab your_testbench.c \
      -CFLAGS "-I$LMC_HOME/include" \
      +incdir+$LMC_HOME/sim/pli/src -RI \
      -LDFLAGS "-Wl,-a,default -ldld -lc -lm -lBSD"
```

Solaris

If you *are not* using the MemPro C testbench:

```
% vcs Verilog_modules MemPro_model_files \
      $LMC_HOME/lib/sun4Solaris.lib/slm_pli.o \
      -P $LMC_HOME/sim/pli/src/slm_pli.tab \
      +incdir+$LMC_HOME/sim/pli/src -RI
```

If you *are* using the MemPro C testbench:

```
% vcs Verilog_modules MemPro_model_files \
      $LMC_HOME/lib/sun4Solaris.lib/slm_pli.o \
      -P local_slm_pli.tab your_testbench.c \
      -CFLAGS "-I$LMC_HOME/include" \
      +incdir+$LMC_HOME/sim/pli/src -RI
```

Example:

```
% vcs tbench.v mydram.v mysram.v \
      $LMC_HOME/lib/sun4Solaris.lib/slm_pli.o \
      -P $LMC_HOME/sim/pli/src/slm_pli.tab \
      +incdir+$LMC_HOME/sim/pli/src -RI
```

NT

If you *are not* using the MemPro C testbench:

```
> vcs Verilog_modules MemPro_model_files
     %LMC_HOME%\lib\pcnt.lib\slm_pli_vcs.lib
     -p %LMC_HOME%\sim\pli\src\slm_pli.tab
     +incdir+%LMC_HOME%\sim\pli\src -RI
```

If you *are* using the MemPro C testbench:

```
> vcs Verilog_modules MemPro_model_files
     %LMC_HOME%\lib\pcnt.lib\slm_pli_vcs.lib
     -P local_slm_pli.tab your_testbench.c
     -CFLAGS "-I%LMC_HOME%\include"
     +incdir+%LMC_HOME%\sim\pli\src -RI
```

Attention

If you are using VCS 5.0 or earlier, add the "-Zp4" switch to your VCS command and replace the "slm_pli_vcs.lib" library with the "slm_pli_v4vcs.lib" library. If you are using VCS 5.1 or later, add the "-ldl" switch to your VCS command.

Linux

If you *are not* using the MemPro C testbench:

```
% vcs Verilog_modules MemPro_model_files \
     $LMC_HOME/lib/x86_linux.lib/slm_pli.o \
     -P $LMC_HOME/sim/pli/src/slm_pli.tab \
     +incdir+$LMC_HOME/sim/pli/src -RI \
     -LDFLAGS -rdynamic
```

If you *are* using the MemPro C testbench:

```
% vcs Verilog_modules MemPro_model_files \
     $LMC_HOME/lib/x86_linux.lib/slm_pli.o \
     -P local_slm_pli.tab your_testbench.c \
     -CFLAGS "-I$LMC_HOME/include" \
     +incdir+$LMC_HOME/sim/pli/src -RI \
     -LDFLAGS -rdynamic
```

4. Instantiate MemPro models in your design. Define ports and generics as required. For information on generics used with MemPro models, refer to "Instantiating MemPro Models" on page 32.

5. Compile your design:

If you *are* using the MemScope Dynamic Data Exchange feature:

```
% vcs +vcs+lic+wait +incdir+$LMC_HOME/sim/pli/src \
      design_name $LMC_HOME/lib/platform.lib/slm_pli.o \
      -Xstrict -P $LMC_HOME/sim/pli/src/slm_pli.tab -cc $cc_path
```

If you *are not* using the MemScope Dynamic Data Exchange feature:

```
% vcs +vcs+lic+wait +incdir+$LMC_HOME/sim/pli/src \
      design_name $LMC_HOME/lib/platform.lib/slm_pli.o \
      -P $LMC_HOME/sim/pli/src/slm_pli.tab -cc $cc_path
```

6. Invoke VCS on your design:

```
% simv
```

VCS MemPro Model Explanation

Table 10 lists each line in the invocation examples above, along with explanations for what each one does.

Table 10: VCS MemPro Model Explanation

Line Reference	Description
`$VCS_HOME/bin/vcs` *mempro_model*`.v` *mempro_model*`_tb.v`	Path to the file that starts the VCS simulator, followed by the specified model and testbench Verilog files.
`-P $LMC_HOME/sim/pli/src/slm_pli.tab`	Specifies the MemPro PLI table entry file.
`$LMC_HOME/lib/sun4Solaris.lib/slm_pli.o`	Specifies the platform-specific MemPro PLI object file.
`-Mupdate`	This specifies incremental compilation, which causes VCS to compile only the modules that have changed since the last run.
`-RI`	This makes VCS run interactively. VCS invokes the XVCS GUI after compilation and pauses the simulator at time zero.
`-LDFLAGS` *switches*	Additional platform-specific switches that may be needed.

Using Hardware Models with VCS

To use hardware models with VCS, follow this procedure:

1. Add the hardware model install tree to your path variable, as shown in the following example:

   ```
   % set path=(/install/sms/bin/platform/ $path)
   ```

2. Set the VCS_LMC environment variable to the hm directory in the VCS install, as shown in the following example:

   ```
   % setenv VCS_LMC $VCS_HOME/platform/lmc/hm
   ```

3. Set the LM_SFI environment variable to the SFI directory in the hardware modeling tree, as shown in the following example:

   ```
   % setenv LM_SFI hardware_modeler_install_path/sms/lib/platform/
   ```

4. Set the VCS_LMC_HM_ARCH environment variable so that you can later use the -lmc-hm switch. This variable must be set to find the SFI directory in the sms/lib tree, as shown in the following examples:

   Solaris

   ```
   % setenv VCS_LMC_HM_ARCH sun4.solaris
   ```

   HP-UX

   ```
   % setenv VCS_LMC_HM_ARCH pa_hp102
   ```

5. Create a Verilog HDL template for the hardware model using the lmvc_template script provided by VCS, as shown in the following example:

   ```
   % lmvc_template model_file
   ```

   where *model_file* is the name of the hardware model's .MDL file.

   For example, if your model is the TILS299, enter:

   ```
   % lmvc_template TILS299.MDL
   ```

   This step produces a TILS299.lmvc.v file that contains the module definition with all the calls, declarations, and assignments necessary to make the file a valid VCS module.

☞ **Note**

The lmvc_template program looks for Shell Software files in the directories indicated by the LM_LIB environment variable. You can modify the port list generated by the lmvc_template to match the existing model instantiations by editing the .NAM file.

6. Compile your description. Make sure to include the hardware model template and supporting PLI and library files. To interface the hardware modeler to VCS, add the -lmc-hm switch to the VCS command line, as shown in the following example:

```
% vcs +plusarg_save -RI test.v TILS299.lmvc.v -lmc-hm -o simv \
+override_model_delays +maxdelays -l vcs.log &
```

You can optionally invoke VCS without the -lmc-hm switch by using the -P switch to point to the $VCS_LMC/lmvc.tab file and including the $VCS_LMC/lmvc.o object file and $LM_SFI/lm_sfi.a library, as shown in the following example:

```
% vcs +plusarg_save -RI test.v TILS299.lmvc.v -P $VCS_LMC/lmvc.tab \
$VCS_LMC/lmvc.o $LM_SFI/lm_sfi.a -o simv \
+override_model_delays +maxdelays -l vcs.log &
```

where:

- vcs is the compiler

- test.v is the file that is part of the top level system source files

- TILS299.lmvc.v is the vcs template for the HW model

- lmvc.tab is the VCS file for lmvc calls for vector logging, and timing measurement

- lmvc.o is the object code for LMC C file (lmvc.c), which contains the definitions for all the lmvc tasks/functions.

- lm_sfi.a is the simulator function interface software that links the VCS simulator to the hardware modeler.

- +override_model_delays is a switch that allows you to specify timing other than typical.

☞ **Note**

the -RI option is not required to generate the simv file. It is used to have the simulator automatically execute after compilation and to use the xvcs debugger.

For more information on using the .tab/.c files and options with VCS, refer to the *VCS Users's Guide*.

Note that in the previous VCS releases, the hardware model could only be simulated with typical delays. The VCS 5.2 release has removed this restriction, so you can now either use a runtime option on the command line or make the change in the delayrange parameter. Note that the runtime option does override any delayrange parameter specification. The following excerpt is from the VCS 5.2 Release Notes:

VCS 5.2 has a new runtime option, +override_model_delays that enables you to use the +mindelays, typdelays, or +maxdelays runtime option to specify timing in SWIFT SmartModels or Synopsys hardware models and in so doing override the DelayRange parameter in the template files for these models that otherwise specifies the timing for the model.

Example Using Runtime Option

Here is an example using the runtime option:

```
% vcs +plusarg_save -RI test.v TILS299.lmvc.v -P $VCS_LMC/lmvc.tab \
$VCS_LMC/lmvc.o $LM_SFI/lm_sfi.a -o simv +override_model_delays \
+maxdelays -l vcs.log &
```

Example Using DelayRange Parameter

Here is an example using the DelayRange parameter:

```
        TILS299 hwm_1 ( .\SL (shift_left), .\CLK (clock), .\SR
(shift_right), .\NCLR (clear),
                        .\-G2 (g2), .\-G1 (g1), .\S1 (select_1), .\S0
(select_0),
                        .\D/QD (bit_4), .\F/QF (bit_6), .\B/QB (bit_2),
.\C/QC (bit_3),
                        .\A/QA (bit_1), .\G/QG (bit_7), .\E/QE (bit_5),
.\H/QH (bit_8),
                        .\QA (high_bit), .\QH (low_bit) );
`ifdef MAX
defparam hwm_1.DelayRange = "MAX" ;
`endif
```

Run your simulation as usual. After running the vcs compiler, you should see a compiled simv file. To run your simulation, type in simv.

You need an additional passcode to use the hardware model interface. If you do not have a passcode, contact VCS Simulation Support at 800-837-4564.

VCS Utilities

If you want to turn on test vector logging or timing measurement, you can invoke tasks 'lm_log', 'lm_log_off', 'lm_measure_time', or 'lm_measure_time_off'.

```
instance_name.lm_measure_time;
instance_name.lm_measure_time_off;
```

where *instance_name* is a string that is the hierarchical path name of the instance for the hardware model. For example, assuming that our instance is top.hwm_1 with these features, it would look like the following example:

```
module top;
      TILS299 hwm_1();
 initial begin
         top.hwm_1.lm_measure_time;
         top.hwm_1.lm_log ("file_name");
         #7000;

         top.hwm_1.lm_measure_time_off;
         top.hwm_1.lm_log_off;
  end
```

3

Using Verilog-XL with Synopsys Models

Overview

This chapter explains how to use SmartModels, FlexModels, MemPro models, and hardware models with Verilog-XL. The procedures are organized into the following major sections:

- "Setting Environment Variables" on page 59
- "Using SmartModels with Verilog-XL" on page 61
- "Using FlexModels with Verilog-XL" on page 79
- "Using MemPro Models with Verilog-XL" on page 81
- "Using Hardware Models with Verilog-XL" on page 83

Setting Environment Variables

First, set the basic environment variables. If you are not using one of the model types, skip that step. In some cases the procedures that follow in this chapter include steps for setting additional environment variables.

1. Set the LMC_HOME variable to the location of your SmartModel. FlexModel, and MemPro model installation tree, as shown in the following example:

   ```
   % setenv LMC_HOME path_to_models_installation
   ```

2. Set the LM_LICENSE_FILE or SNPSLMD_LICENSE_FILE environment variable to point to the product authorization file, as shown in the following example:

```
% setenv LM_LICENSE_FILE path_to_product_authorization_file
```

```
% setenv SNPSLMD_LICENSE_FILE path_to_product_authorization_file
```

You can put license keys for multiple products (for example, SmartModels and hardware models) into the same authorization file. If you need to keep separate authorization files for different products, use a colon-separated list (UNIX) or semicolon-separated list (NT) to specify the search path in your variable setting.

Caution

Do not include la_dmon-based authorizations in the same file with snpslmd-based authorizations. If you have authorizations that use la_dmon, keep them in a separate license file that uses a different license server (lmgrd) process than the one you use for snpslmd-based authorizations.

3. If you are using the hardware modeler, set the LM_DIR and LM_LIB environment variables, as shown in the following examples:

```
% setenv LM_DIR hardware_model_install_path/sms/lm_dir
% setenv LM_LIB hardware_model_install_path/sms/models: \
 hardware_model_install_path/sms/maps
```

If you put your models in a directory other than the default of /sms/models, modify the above variable setting accordingly.

Note

On NT, these hardware modeler environment variables are set automatically when you install the software.

4. Set the CDS_INST_DIR variable to the location of your Cadence installation tree, as shown in the following example, and make sure that Verilog-XL is set up properly in your environment:

```
% setenv CDS_INST_DIR path_to_Cadence_installation
```

5. Depending on your platform, set your load library variable to point to the platform-specific directory in $LMC_HOME, as shown in the following examples:

Solaris:

```
% setenv LD_LIBRARY_PATH $LMC_HOME/lib/sun4Solaris.lib:$LD_LIBRARY_PATH
```

Linux:

```
% setenv LD_LIBRARY_PATH $LMC_HOME/lib/x86_linux.lib:$LD_LIBRARY_PATH
```

AIX:

```
% setenv LIBPATH $LMC_HOME/lib/ibmrs.lib:$LIBPATH
```

HP-UX:

```
% setenv SHLIB_PATH $LMC_HOME/lib/hp700.lib:$SHLIB_PATH
```

NT:

Make sure that %LMC_HOME%\lib\pcnt.lib is in the Path user variable.

Using SmartModels with Verilog-XL

SmartModels work with Verilog-XL using a PLI application called LMTV that is delivered in the form of a swiftpli shared library in $LMC_HOME/lib/*platform*.lib. If you cannot use the swiftpli, refer to "Using SmartModels with Verilog-XL on the IBMRS AIX Platform" on page 62.

To use the prebuilt swiftpli, follow this procedure:

1. Instantiate the SmartModels in your design, defining the ports and defparams as required. For details on required SmartModel SWIFT parameters and a model instantiation example, refer to "Using SmartModels with SWIFT Simulators" on page 18.

2. There is no need to build a Verilog executable. You can use the one that Cadence provides at $CDS_INST_DIR/tools/bin by adding it to your path variable.

3. To use the swiftpli shared library, invoke the Verilog simulator to compile and simulate your design as shown in the examples below:

   UNIX

   ```
   % verilog testbench model.v +loadpli1=swiftpli:swift_boot \
       +incdir+$LMC_HOME/sim/pli/src
   ```

   NT

   ```
   > verilog testbench model.v +loadpli1=swiftpli:swift_boot
       +incdir+%LMC_HOME%\sim\pli\src
   ```

☞ **Note**

For information on LMTV commands that you can use with SmartModels on Verilog-XL, refer to "LMTV Commands" on page 281.

Using SmartModels with Verilog-XL on the IBMRS AIX Platform

To use SmartModels and FlexModels on IBM with Verilog-XL, follow this procedure. The instructions provided here are based on the assumption that you are already using Verilog-XL.

1. The following lines are already in the copy of veriuser.c at $LMC_HOME/sim/pli/src. If you are using another veriuser.c file, copy these lines into it. They are needed to make the SmartModel C-Pipe interface.

```
/* ------------- cut here - section 1 ------------- */
/* C-pipe interface */
extern int pli_slm_post();
extern int slm_mempro_handle();
extern int slm_mempro_width_info();
extern int slm_transport();
extern int slm_transport_checktf();
extern int slm_inertial();
extern int slm_inertial_checktf();
/* ------------- end section 1 ------------- */
/* ------------- cut here - section 2 ------------- */
        { usertask, 0, 0, 0, pli_slm_post, 0, "$slm_post_hdl", true },
        { usertask, 0, 0, 0, pli_slm_post, 0, "$slm_post", true },
        { usertask, 0, 0, 0, slm_mempro_handle, 0, "$slm_mempro_handle",/
        true },
        { usertask, 0, 0, 0, slm_mempro_width_info, 0,/
        "$slm_mempro_width_info", true },
        { usertask, 0, slm_transport_checktf, 0, slm_transport, 0,/
        "$slm_transport", true },
        { usertask, 0, slm_inertial_checktf, 0, slm_inertial, 0,/
        "$slm_inertial", true },
/* ------------- end section 2 ------------- */
```

2. You will need to edit the veriuser.c file to pick up the LMTV header files as follows:

```
        a. After #include "vxl_veriuser.h" add:
        #include "ccl_lmtv_include.h"
        b. After "/*** add user entries here ***/" add:
        #include "ccl_lmtv_include_code.h"
```

3. Invoke the Cadence PLI Wizard utility.

   The instructions that follow summarize the procedure found in the *Cadence PLI Wizard User Guide* but are not intended to replace the Cadence procedure.

   The PLI Wizard utility will present a series of forms for you to fill out. After you have completed each window as described in the following steps, click the Next button at the bottom to invoke the next window.

4. In the Select Config Session Name and Directory form,

   ❍ Enter the name of your session file in the **Config Session Name** field.

   ❍ Enter the path to the session file in the **Config Session Directory** field.

5. Select **Verilog-XL** as the Verilog simulator.

6. Select **Stand Alone** as the Verilog target.

7. Select **PLI 1.0 Application** and **SWIFT Interface** as your components. Specify your LMC_HOME path in the Root Path field next to **SWIFT Interface**.

8. Select **Static Linking** for the PLI Application Linking Mode.

9. Select **Existing VERIUSER (source/object file)**, and enter the path to your edited veriuser.c source file.

10. Enter the path to the $LMC_HOME/lib/*platform*.lib/slm_pli.o object file in the **PLI Object Files** field.

11. Leave the **Select Compiler & Options** form as is.

12. When the Session Setup Summary form appears, click Finish.

13. Click Yes when asked whether or not to build targets now.

    The new Verilog-XL executable that includes LMTV & the C-Pipe interface will then be built in the session directory. Add this directory to your PATH statement to use the new Verilog-XL executable.

Verilog-XL Usage Notes for SmartModels

This section describes the Synopsys Logic Models To Verilog (LMTV) interface.   You can use LMTV to instantiate and work with SmartModels in Verilog-XL, as described in the following sections:

● LMTV Modes of Operation

● Capturing and Simulating the Design

● Using SmartModel Windows with Verilog-XL

● Customizing Model Timing

● Simulating an Older Design Using LMTV

● Using FlexModels with Verilog-XL

LMTV Modes of Operation

To take advantage of the SWIFT SmartModel Library while maintaining compatibility with the older Verilog-XL-specific SmartModel Library, the LMTV interface has two modes of operation, SWIFT SmartModel Mode and Historic SmartModel Mode.

SWIFT SmartModel Mode

In SWIFT SmartModel mode, the models you instantiate are SWIFT SmartModels. This is the mode intended for primary use. Use this mode if you are implementing a new design using the SWIFT SmartModel Library, if you are a new Verilog-XL user, or if you want to transition your existing design into this mode.

Two sets of v shells support SWIFT SmartModel mode: swift and swift-uc. With swift-uc, module names and attribute names are provided in all uppercase. The two sets of v shells provide compatibility with most third-party tools.

Historic SmartModel Mode

In Historic SmartModel mode, the models you instantiate have the characteristics of the Verilog-XL-specific SmartModels. Historic SmartModel mode is provided only for backward-compatibility for designs that use models from the Verilog-XL-specific SmartModel Library.   Use the Historic SmartModel mode only if you are continuing with an older design that was captured using the Verilog-XL-specific SmartModel Library.

👉 Note

You must use the same mode throughout a design. You cannot mix modes within a design.

Table 11 lists the different characteristics of the SWIFT and Historic SmartModel modes.

Table 11: Characteristics of Historic and SWIFT SmartModel Modes

Differences	SWIFT SmartModel Mode		Historic SmartModel Mode
	swift	**swift-uc**	
Model Attributes	TimingVersion	TIMINGVERSION	COMPONENT
	ModelType	MODELTYPE	MODELTYPE
	DelayRange	DELAYRANGE	RANGE
	MemoryFile	MEMORYFILE	MEMORYFILE
	JEDECFile	JEDECFILE	JEDECFILE
	SCFFile	SCFFILE	CGAFILE
	PCLFile	PCLFILE	PCLFILE

Table 11: Characteristics of Historic and SWIFT SmartModel Modes

Differences	SWIFT SmartModel Mode		Historic SmartModel Mode
	swift	swift-uc	
Module Names	Alphabetic characters are lowercase	Alphabetic characters are uppercase	Alphabetic characters are uppercase
Port Ordering	Numeric—for example, ports of bus A[0:11] are in this order: A0, A1, A2, A3, ..., A9, A10, A11.		Alphanumeric—for example, ports of bus A[0:11] are in this order: A0, A1, A10, A11, ..., A8, A9)
Command Names	Begin with $lm_		Begin with $lai_
Switch Names	Begin with +lm		Begin with +lai
Message Format	Refers to model names and timing version names. Timing units are in nanoseconds (ns).		Refers only to timing version names. No timing units specified.
Ignored	+laiobj ignored LAI_OBJ ignored		+laiobj ignored LAI_OBJ ignored
User-defined Windows	*model*.v files do not have to be modified		*model*.v files must be modified
Resistive Strength	Reports true resistive strength of outputs		Maps resistive strength of outputs to "strong"
Memory Windows	Supports memory windows		Does not support memory windows.

Implementing SWIFT Mode or Historic SmartModel Mode

Both the SWIFT and the Historic SmartModel modes reference the SWIFT SmartModel Library, but they use different sets of *model*.v files (vshells) to invoke the models. For each mode, there is a specific directory, shown in Table 12, that contains *model*.v files to be referenced by that mode. You determine the mode that will be used both during design capture and when you invoke Verilog-XL, as follows:

1. During design capture, use the appropriate model attributes, port ordering, command channel, memory access, user-defined windows, module names, and resistive strength output expectations shown in Table 11.

2. When you invoke Verilog-XL, reference the appropriate *model*.v directory using the -y switch, as described in .

Table 12: model.v Directories

Mode	Directory
SWIFT SmartModel Mode	$LMC_HOME/special/cds/verilog/swift
SWIFT SmartModel Mode - Uppercase	$LMC_HOME/special/cds/verilog/swift-uc
Historic SmartModel Mode	$LMC_HOME/special/cds/verilog/historic

Capturing and Simulating the Design

Capturing and simulating the design in Verilog-XL involves the following steps, each of which is described in detail in this section:

-
-
-
-
-

Verilog-XL Design Flow

Figure 2 shows the Verilog-XL design flow, with two paths. You choose one path or the other based on the task at hand:

- LMTV SWIFT SmartModel mode—recommended for new designs
- LMTV Historic SmartModel mode—recommended for older designs that use the Verilog-XL-specific SmartModel Library

You can create a design file (*design*.v) textually with an HDL description or graphically using Concept. FlexModel users should also use the appropriate Verilog wrapper file from the *model*_fx/examples/verilog directory. This file is copied to your working directory using the flexm_setup tool. For information on flexm_setup, refer to

When Verilog-XL simulates the design, it references either the SWIFT SmartModel mode *model*.v files or the Historic SmartModel mode *model*.v files. You must specify one of these directories when you invoke Verilog-XL. A design cannot reference both directories.
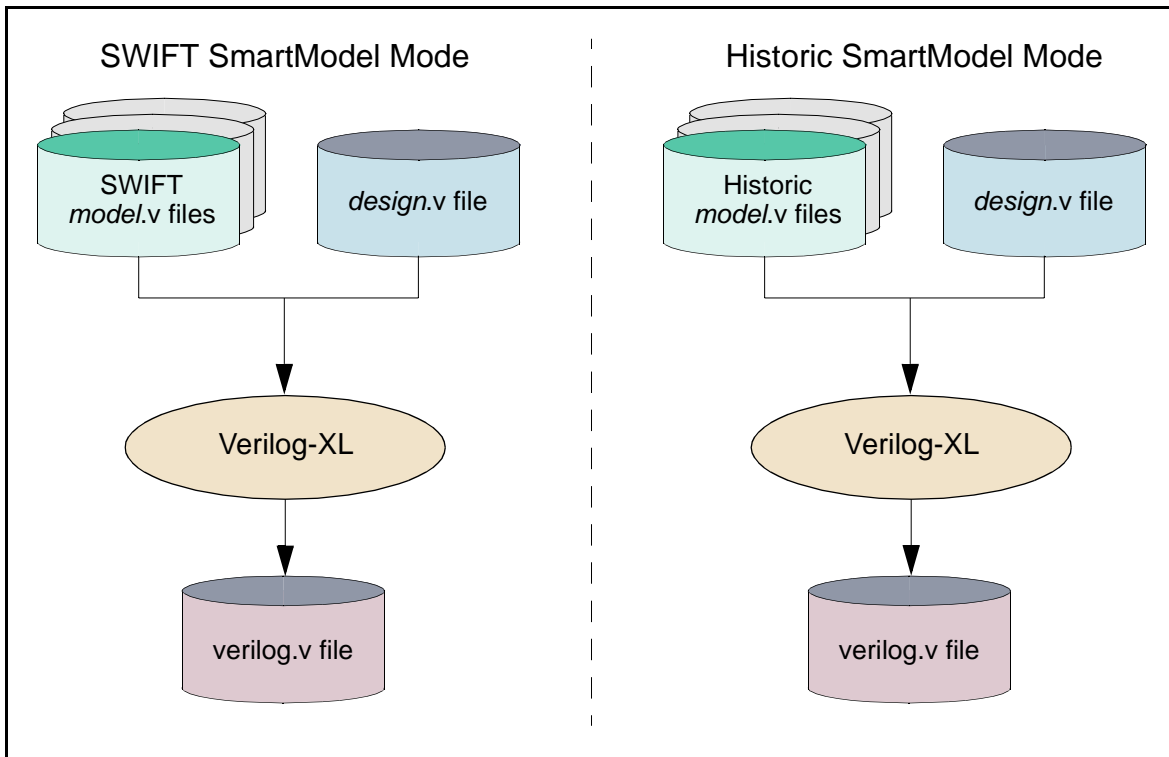
Figure 2: Verilog-XL Design Flow

Preparing to Use Verilog-XL

Before you use Verilog-XL in either mode, make sure that your executable search path points to the Verilog-XL executable that contains the LMTV interface.

Verilog-XL Design Capture

You instantiate models from the SWIFT SmartModel Library by creating HDL descriptions for Verilog-XL. The following example shows the Verilog-XL code for instantiating a simple NAND gate (ttl00) in a *design*.v file for the LMTV SWIFT SmartModel mode. For instance U1, the TimingVersion and DelayRange parameter values have both been changed from the defaults. For instance U2, only the DelayRange attribute value has been changed from the default.

```
top_mod contains:
module TOP_MOD;
defparam
  U1.TimingVersion = "54F00-FAI",
  U1.DelayRange  = "min",
  U2.DelayRange  = "typ";
```

```
    ttl00 U1(.I1(clk), .I2(enable), .O1(output));
    ttl00 U2(.I1(clk), .I2(enable), .O1(output));
  endmodule
```

The following example shows the Verilog-XL code for the same instantiation, but for the LMTV Historic SmartModel mode. Notice that the attribute names are different and that the alphabetic characters in the model name are upper case.

```
  top_mod contains:
  module TOP_MOD;
   defparam
   U1.COMPONENT= "54F00-FAI",
   U1.RANGE= "min",
   U2.RANGE= "typ";

  TTL00 U1(.I1(clk), .I2(enable), .O1(output));
  TTL00 U2(.I1(clk), .I2(enable), .O1(output));
  endmodule
```

Concept Design Capture

As an alternative, you can capture a design using the Concept design flow (refer to Figure 3). First, diagram the design in Concept using a custom symbol library.

Next, execute vloglink to generate the vloglink.v file. Finally, for the SWIFT SmartModel mode only, execute the mod_param utility provided by Synopsys to convert model instance parameter names to SWIFT-compliant names in the vloglink.v file.

```
          ┌─────────────────────┐
          │    Create Design    │
          │ Diagram Using Concept│
          └─────────────────────┘
                     │
                     ▼
          ┌─────────────────────┐
          │      Execute        │
          │      vloglink       │
          └─────────────────────┘
                     │
                     ▼
              ⬭ design.v file ⬭

                     │
                     ▼
                  ◇ SWIFT         ──────▶  ┌─────────────────┐
                  SmartModel                │     Execute     │
                  Mode? ◇                    │    mod_param    │
                     │                       └─────────────────┘
                     ▼                              │
                  ( Done )                          ▼
                                          ⬭ SWIFT-compliant
                                             vloglink.v file ⬭
```
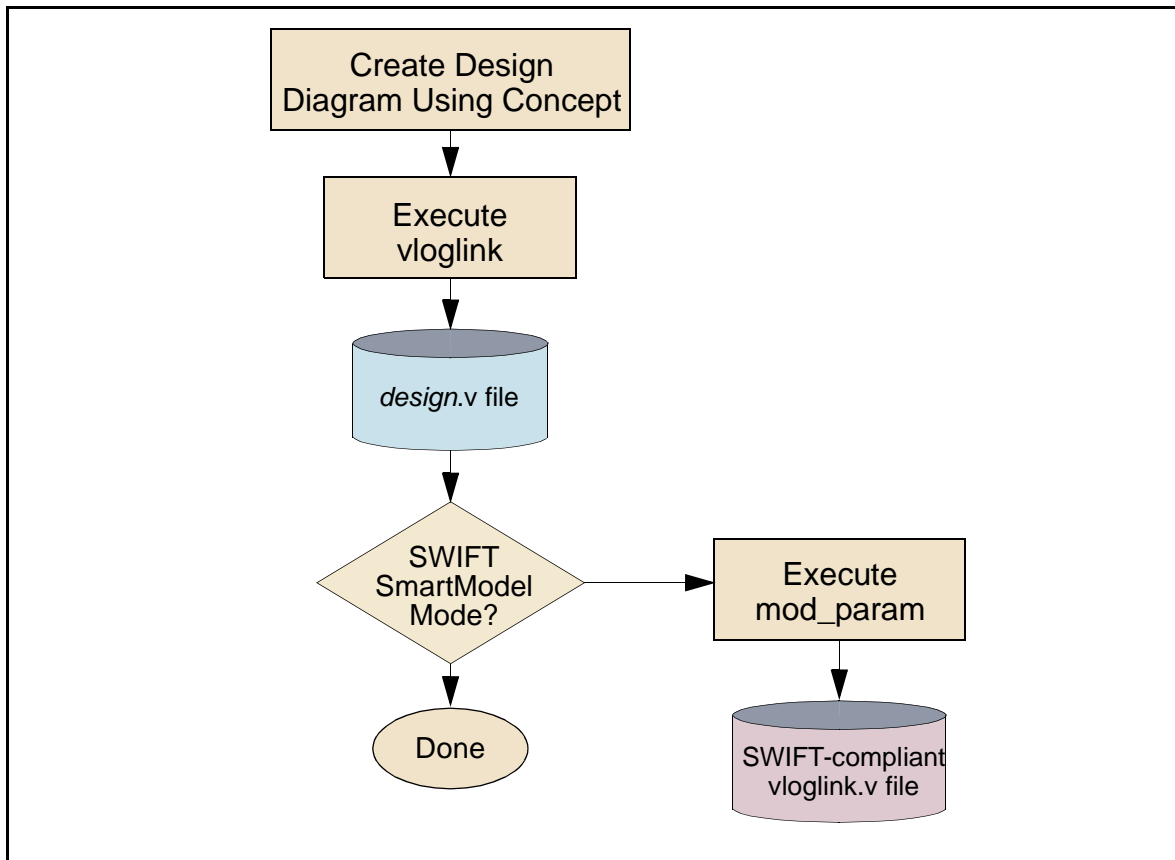
Figure 3: Concept Design Flow

Concept Procedure

To create a design file graphically using Concept, follow these steps:

1. Merge the file into your master.local file.

2. Invoke Concept, instantiate the symbols, and write the schematic.

3. Execute vloglink.

4. If you are using the SWIFT SmartModel mode, use one of these methods to prepare vloglink.v for simulation:

   ❍ Run verilog with the -u switch

   --or--

   ❍ Run mod_param on the vloglink.v file

This converts parameter names to SWIFT-compliant form. (For more information about the mod_param utility, run mod_param with the -h to display the usage message.)

Using SmartModel Windows with Verilog-XL

SmartModel Windows, also referred to as "Windows," is a SmartModel Library feature that allows you to view and change the contents of internal registers during simulation, for models and simulators (including Verilog-XL) that support this feature. For general information about SmartModel Windows, refer to the *SmartModel Library User's Manual*. This section provides information about using SmartModel windows with Verilog-XL.

LMTV SmartModel Windows Commands

The following commands allow you to work with SmartModel Windows during Verilog-XL simulation. Commands are instance-specific, which means that they must be issued once for each instance. These commands are most often placed in the testbench, but can also be issued at the command line. For details and examples, refer to the specific command descriptions.

$lm_monitor_enable(), $lai_enable_monitor()

Enables SmartModel Windows for one or more window elements in a specified model instance. Can be used in both Historic and SWIFT SmartModel modes, but is recommended for use only in Historic SmartModel mode.

$lm_monitor_disable(), $lai_disable_monitor()

Disables SmartModel Windows for one or more window elements in a specified model instance. Can be used in both Historic and SWIFT SmartModel modes, but is recommended for use only in Historic SmartModel mode.

$lm_monitor_vec_map(), $lm_monitor_vec_unmap()

Enables or disables a direct mapping between a user-defined variable and a window element in a specified model instance. The window element can be part of an array. Can be used only in SWIFT SmartModel mode.

$lm_status(), $lai_status()

Displays the names and values of internal windows for a specified model instance. Can be used in both Historic and SWIFT SmartModel modes.

Creating User-Defined Window Elements

You can create user-defined window elements only for SmartCircuit FPGA or CPLD models. The way you create these window elements depends on whether you will access the window elements using $lm_monitor_enable(), which can be used in either the Historic or the SWIFT SmartModel mode; or $lm_monitor_vec_map(), which can be used only in SWIFT SmartModel mode.

Both $lm_monitor_enable() and $lm_monitor_vec_map() need to be given the names of the model's window elements. However, these two commands receive the window element names differently, as follows:

- The $lm_monitor_enable() command expects to find the window element names in the *model*.v files. Therefore, before invoking $lm_monitor_enable(), you must use ccn_report to modify the *model*.v files so that they contain the window element names. For details on how to use the ccn_report tool, refer to the *SmartModel Library User's Manual*.

- The $lm_monitor_vec_map(), on the other hand, expects to be passed the window element names through its own window_element argument, and does not look in the *model*.v files. Therefore, you do not need to create modified *model*.v files before executing the $lm_monitor_vec_map() command.

For more information about creating window elements using auto windows, refer to the *SmartModel Library User's Manual.*

In Historic SmartModel Mode

In Historic SmartModel mode, you can access user-defined windows only by using the $lm_monitor_enable() command. This means that you must create user-defined windows for SmartCircuit FPGA and CPLD models by creating modified *model*.v files.

1. If you do not already have a compiled configuration netlist (CCN) file, generate one by executing smartccn on your design. For details on how to use the smartccn tool, refer to the *SmartModel Library User's Manual.*

2. Generate a windows definition file by executing ccn_report on your CCN file, as shown in the following example.

   ```
   % ccn_report  ccn_filename -m model_name -A1 windows_file
   ```

3. Generate a modified *model*.v file that contains the window information by executing ccn_report again, as shown in the following example.

   ```
   % ccn_report  ccn_filename -m model_name -v -w windows_file \
   -y $LMC_HOME/special/cds/verilog/historic \
   -mn module_name -o modified_model.v
   ```

4. Add the windows definition file to your Model Command File (MCF) in the form of a do command statement, as follows:

   ```
   do windows_file
   ```

5. Make sure that your design references the *modified_model.v* file.

In SWIFT SmartModel Mode

In SWIFT SmartModel mode, you can access user-defined windows either through $lm_monitor_enable() or $lm_monitor_vec_map(). The $lm_monitor_enable() command is provided in SWIFT SmartModel mode for backward compatibility. We recommend that you use this command only for existing designs. The $lm_monitor_vec_map() command is intended for use with new designs.

Follow the same procedure as described in except that in Step 3, when invoking ccn_report, use this value for the -y argument:

```
-y $LMC_HOME/special/cds/verilog/swift
```

Accessing Window Elements

The way you access SmartModel window elements depends on whether you are running in SWIFT mode or Historic SmartModel mode. The following sections provide instructions for both modes.

In Historic SmartModel Mode

In Historic SmartModel mode, you can access only scalar window elements. You cannot access the vectored memory window elements available in SWIFT SmartModel mode. You read and write to predefined window elements using the $lm_monitor_enable() or $lai_enable_monitor() commands. To access window elements in Historic SmartModel mode, follow these steps.

1. Enable SmartModel Windows for the model instance, either for specific window elements or for all window elements. For example, to enable SmartModel Windows for instance U4 for all window elements, use this command:

   ```
   $lm_monitor_enable (U4);
   ```

2. To enable only window elements A_REG and D_REG for U4, use this command:

   ```
   $lm_monitor_enable (U1, "A_REG", "D_REG");
   ```

3. To read from a specific window element, use the $monitor, $strobe, $write, and $display   Verilog commands. For more information, refer to the Cadence documentation.

4. To display the contents of all window elements, use $lm_status (or $lai_status). For example, to display all window elements for instance U1, use this command:

   ```
   $lm_status("U1");
   ```

The information is returned in the following format:

```
Note:  SmartModel Windows Status:
    INREG "Input Register":000
    OUTREG "Output Register"000
    IO_INREG "I/O Input Register":000
    HREG "Hidden Register":0
```

5. To write to a window element, assign a value to the window element, in this format:

`instance.window_element=value`

For example, to clear Bit 3 of the window element HREG in the instance U1 (assuming that HREG has write access), use this command:

```
U1.HREG[3]=0;
```

☞ **Note**

Refer to the individual model datasheets for information about the read/write capabilities of model window elements.

In SWIFT SmartModel Mode

In SWIFT SmartModel mode, you can use the $lm_monitor_enable() or $lai_enable_monitor() commands to monitor scalar windows, in exactly the same way as described in "In Historic SmartModel Mode" on page 72.   However, as before, you cannot use these commands to access vectored memory windows.

To use memory windows, available in SWIFT but not in Historic SmartModel mode, you must use the $lm_monitor_vec_map() command. This command works for both scalar and vectored windows.

💡 **Hint**

For simplicity, when implementing new designs in SWIFT SmartModel mode, use the $lm_monitor_vec_map() command for both scalar and vectored windows applications. It is best not to use the $lm_monitor_enable() or $lai_enable_monitor() commands at all.

To access window elements using the $lm_monitor_vec_map() command, follow these steps:

1. Define a register for the window element. You can give the register the same name as the window element, or a different name. For example, to define the register MY_A_REG to map to the 32-bit register A_REG, you could use this definition:

`reg [31:0] MY_A_REG;`

2. Enable the window element and map it to the register. For example, to enable the window element A_REG for instance U1 and map A_REG to the register MY_A_REG, you could use this command:

```
$lm_monitor_vec_map (MY_A_REG, U1, "A_REG");
```

3. To read from a window element, use any appropriate Verilog command to examine the contents of the register mapped to that window element.   For example, to read the contents of window element A_REG, you could use this command:

```
$display ("Address is %b", MY_A_REG);
```

4. To write to a window element, assign a value to the register mapped to that window element. For example, to set Bit 4 of the window element A_REG you could use this command:

```
MY_A_REG[4]=1;
```

Example 1

The following example shows the predefined scalar window elements w0 and w2, as they might appear in a typical testbench.

```
reg MY_VAR_W0;  // users can choose descriptive variable
reg MY_VAR_W2;  // names to fit their applications.
// the next two lines map the variable names to the
// window elements and enable the window elements.
$lm_monitor_vec_map( MY_VAR_W0, "U1", "w0", 0 );
$lm_monitor_vec_map( MY_VAR_W2, "U1", "w2", 0 );
```

Once these window elements are set up in your testbench, you can use the graphical or monitoring capabilities of Verilog-XL to read, write, or trace the variables MY_VAR_W0 and MY_VAR_W2, which now hold the values of the window elements w0 and w2.

Example 2

The following example illustrates the use of the $lm_monitor_vec_map() command to use memory window elements to track transactions on a memory device. In the example, a 4K x 8 bit memory model with instance name U1 has these predefined memory window elements:

- Memory array window element: MEM 4K x 8 bits

- Memory address window element: Mem_addr 12 bits

- Memory read/write window element: Mem_rw 2 bits

For more information about memory windows, refer to the *SmartModel Library User's Manual.*

The following example code monitors the memory device. If there is a memory write, the code checks to see if the write was to any location in the address range between 'h100 and 'h200 (not allowed) and if so, issues an error message.

If the Mem_addr window element (which contains the most recently accessed address) contains unknown values, no test is performed and another error message is issued.

```
ram_model U1();
// Declare registers for the address, rw, and memory data window
elements
 reg [11:0] ADDR;
 reg [1:0] RW;
 reg [7:0] DATA;
 initial
// Map the ADDR and RW registers to the MEM_addr and MEM_rw windows
begin
 $lm_monitor_vec_map (ADDR, U1, "MEM_addr");
 $lm_monitor_vec_map (RW, U1, "MEM_rw");
end
// Whenever there is a memory transaction, check the address
// and the direction for an illegal write, and the address for
// unknown values.
always @(RW)
 begin
  if (ADDR >= 'h100 && ADDR <= 'h200 && RW[1] == 0 && ADDR != 'hx)
  begin
// There was an illegal write.
// Temporarily map DATA register to address pointed to by ADDR
// and enable MEM array window element to get value of data
    $lm_monitor_vec_map (DATA, U1, "MEM", ADDR);
     $display("Illegal write of value %h at address %h", DATA, ADDR);
// Turn off enabling of memory array
    $lm_monitor_vec_unmap (DATA, U1);
  end
  if (ADDR=='hx)
     $display("Warning!  Multiple simultaneous transactions.");
  end
```

Customizing Model Timing

You can customize the timing of SmartModels by changing the timescale or creating custom timing files.

Changing the Timescale

As with the Verilog-XL-specific SmartModel Library, you can change the timescale of SmartModels from the default of 100 ps (picoseconds) time_units and 100 ps precision, defined in the module definitions. These values are specified by the first line of the module definition, as shown in the following example.

```
`timescale 100 ps / 100 ps
```

For both the SWIFT and Historic SmartModel modes, to change the timescale, you must copy the affected *model*.v files into a separate directory. Then modify each 'timescale compiler directive to the desired value. When invoking the simulator, use the -y switch to indicate the path to the directory that contains your modified *model*.v files.

Creating Custom Timing Files

You can create and use custom timing files in both the SWIFT and Historic SmartModel modes. The procedure is the same for both. For more information on User-Defined Timing, refer to the *SmartModel Library User's Manual.*

Simulating an Older Design Using LMTV

If you have an older design that was created using the Verilog-XL-specific SmartModel Library, you can simulate it in either mode:

● Historic SmartModel mode—in this case you do not have to modify the design

● SWIFT SmartModel mode—in this case you must modify the design

In both cases, you must make some modifications to the simulation environment as described in the following sections.

LMTV/SWIFT and Verilog-XL-Specific SmartModel Libraries

Table 13 lists the differences between the LMTV/SWIFT and Verilog-XL-specific SmartModel Libraries.

Table 13: LMTV/SWIFT and Verilog-XL-specific Libraries

LMTV/SWIFT SmartModel Library	Verilog-XL-specific SmartModel Library
Uses simplified search algorithm for user-defined timing files.	Uses complex search algorithm for user-defined timing files.

Table 13: LMTV/SWIFT and Verilog-XL-specific Libraries (Continued)

LMTV/SWIFT SmartModel Library	Verilog-XL-specific SmartModel Library
In SWIFT SmartModel mode, reports true resistive strength of outputs; a switch optionally maps all to strong. In Historic SmartModel mode, mimics Verilog-XL-specific SmartModel Library.	Always maps resistive strength of outputs to strong.
Supports the Verilog $reset and $restart commands.	Does not support the Verilog $reset and $restart commands.
Always uses only $LMC_HOME to find models; uses no other switches/variables.	Uses +laiobj, $LAI_OBJ, +lai_lib, $LAI_LIB as well as $LMC_VLOG to find models.
LMTV interface does not support Cadence fault simulation.	Supports Cadence fault simulation.

Environment Modifications

This section describes environment modifications you need to make if you want to simulate an existing design in either the LMTV SWIFT or Historic SmartModel mode.

Environment Variables

For both modes, set the LMC_PATH and LMC_HOME environment variables instead of the LMC_VLOG, LAI_OBJ, and LAI_LIB environment variables, which are ignored by the LMTV interface. For more information about user configuration, refer to the *SmartModel Library Administrator's Manual*.

Command Line Switches

The LMTV interface ignores the +laiobj and +lailib switches, regardless of which mode you are using. However, LMTV does recognize the +laiudtmsg and +lmudtmsg switches, which are equivalent.

Resistive Strength

For SWIFT SmartModel mode only, the SWIFT interface reports the true resistive strength of output pins, instead of mapping them all to "strong" as is done in the Verilog-XL-specific SmartModel Library. You might want to modify your expected output accordingly. However, if you want only a quick comparison and do not want to modify your expected output, you can revert to the Verilog-XL-specific behavior by using the +lmoldstr command switch. For more information about setting switches, refer to .

Edits to Design File

This section describes edits you need to make to your design file if you want to simulate an existing design in the LMTV SWIFT SmartModel mode.

☞ **Note**

Note that lai_* commands are recognized by the LMTV interface, so you do not have to change command names for either mode.

Model Parameter Names

Change the parameter names to the corresponding SWIFT SmartModel mode entries, as shown in Table 11 on page 64.

Model Names

Change the alphabetic parts of model names to all lower case. (If you use SWIFT-UC mode, this step is not required.)

Port Names

If you have used explicit port naming in your module instantiations (that is, if you have explicitly mapped each net name to the corresponding port name in the model instantiation statement), you do not need to do anything about port names.

If, on the other hand, you have used implicit port naming (that is, if you have listed the nets in the model instantiation statement in the same order as the ports were declared in the .v file), you need to ensure that your port names conform to the ordering scheme used in the SWIFT SmartModel mode, as described in Table 11 on page 64.

Using FlexModels with Verilog-XL

FlexModels work with Verilog-XL using a PLI application called LMTV that is delivered in the form of a swiftpli shared library in $LMC_HOME/lib/*platform*.lib. If you cannot use the swiftpli, refer to "FlexModels:PLI Static Linking with LMTV" on page 81.

To use the prebuilt swiftpli, follow this procedure:

1. Create a working directory and run flexm_setup to make copies of the model's interface and example files there, as shown in the following example:

   ```
   % $LMC_HOME/bin/flexm_setup -dir workdir model_fx
   ```

   You must run flexm_setup every time you update your FlexModel installation with a new model version. Table 14 lists the files that flexm_setup copies to your working directory.

Table 14: FlexModel Verilog-XL Files

File Name	Description	Location
*model*_pkg.inc	Verilog task definitions for FlexModel interface commands. This file also references the flexmodel_pkg.inc and *model*_user_pkg.inc files.	*workdir*/src/verilog/
*model*_user_pkg.inc	Clock frequency setup and user customizations.	*workdir*/src/verilog/
*model*_fx_vxl.v	A SWIFT wrapper that you can use to instantiate the model.	*workdir*/examples/verilog/
model.v	A bus-level wrapper around the SWIFT model. This allows you to use vectored ports for the model in your testbench.	*workdir*/examples/verilog/
*model*_tst.v	A testbench that instantiates the model and shows how to use basic model commands.	*workdir*/examples/verilog/

2. There is no need to build a Verilog executable. You can use the one from $CDS_INST_DIR/tools/bin by adding $CDS_INST_DIR/tools/bin to your path statement.

3. Update the clock frequency supplied in the *model*_user_pkg.inc file to correspond to the CLK period you want for the model. This file is located in:

   ```
   workdir/src/verilog/model_user_pkg.inc
   ```

   where *workdir* is your working directory.

4. Add the following line to your Verilog testbench to include FlexModel testbench interface commands in your design:

```
`include "model_pkg.inc"
```

👉 **Note**

Be sure to add *model*_pkg.inc within the module from which you will be issuing FlexModel commands.

Because the *model*_pkg.inc file includes references to flexmodel_pkg.inc and *model*_user_pkg.inc, you don't need to add flexmodel_pkg.inc or *model*_user_pkg.inc to your testbench.

5. Instantiate the FlexModel in your design, defining the ports and defparams as required (refer to the example testbench supplied with the model). You use the supplied bus-level wrapper (*model*.v) in the top-level of your design to instantiate the supplied bit-blasted wrapper (*model*_fx_vxl.v).

Example using bus-level wrapper (*model*.v) without timing:

```
model U1 ( model ports )
defparam
      U1.FlexModelId = "TMS_INST1";
```

Example using supplied bus-level wrapper (*model*.v) with timing:

```
model U1 ( model ports )
defparam
      U1.FlexTimingMode = `FLEX_TIMING_MODE_ON,
      U1.TimingVersion = "timingversion",
      U1.DelayRange = "range,"
      U1.FlexModelId= "TMS_INST1";
```

6. Invoke the Verilog-XL simulator to compile and simulate your design as shown in the examples below:

UNIX

```
% verilog testbench +loadpli1=swiftpli:swift_boot \
   ./workdir/examples/verilog/model.v \
   ./workdir/examples/verilog/model_fx_vxl.v \
   +incdir+$LMC_HOME/sim/pli/src \
   +incdir+workdir/src/verilog
```

NT

```
> verilog testbench +loadpli1=swiftpli:swift_boot
   workdir\examples\verilog\model.v
   workdir\examples\verilog\model_fx_vxl.v
   +incdir+%LMC_HOME%\sim\pli\src
   +incdir+workdir\src\verilog
```

☞ **Note** ───

For information on LMTV commands that you can use with FlexModels on Verilog-XL, refer to "LMTV Commands" on page 281.

───

FlexModels:PLI Static Linking with LMTV

To use FlexModels on IBM with Verilog-XL, follow the procedure in "Using SmartModels with Verilog-XL on the IBMRS AIX Platform" on page 62.

Using MemPro Models with Verilog-XL

MemPro models work with Verilog-XL using a PLI application called LMTV that is delivered in the form of a swiftpli shared library in $LMC_HOME/lib/*platform*.lib. If you cannot use the swiftpli, refer to "Static Linking with LMTV" on page 82.

To use the prebuilt swiftpli, follow this procedure:

1. If you are on NT, make sure %LMC_HOME%\bin is in your Path variable.

2. To include MemPro testbench interface commands in your design, add one of the following lines to your testbench:

   Verilog testbench:

   ```
   `include "mempro_pkg.v"
   ```

   C testbench:

   ```
   #include "mempro_c_tb.h"
   ```

   For more information on using the MemPro testbench interfaces, refer to the "HDL Testbench Interface" and "C Testbench Interface" chapters in the *MemPro User's Manual*.

3. Instantiate MemPro models in your design. Define ports and generics as required. For information on generics used with MemPro models, refer to "Instantiating MemPro Models" on page 32. For information on message levels and message level constants, refer to "Controlling MemPro Model Messages" on page 33.

4. There is no need to build a Verilog executable. You can use the one from $CDS_INST_DIR/tools/bin by adding $CDS_INST_DIR/tools/bin to your path statement.

5. Invoke the Verilog-XL simulator to compile and simulate your design as shown in the examples below:

UNIX

```
% verilog testbench Verilog_modules MemPro_model_files \
+incdir+$LMC_HOME/sim/pli/src \
+loadpli1=swiftpli:swift_boot
```

NT

```
> verilog testbench Verilog_modules MemPro_model_files
+incdir+%LMC_HOME%\sim\pli\src
+loadpli1=swiftpli:swift_boot
```

☞ **Note**

If you are also using SmartModels or FlexModels in your design, you do not
need to load the swiftpli again, since the same library is used to enable all
three types of models in Verilog-XL.

Static Linking with LMTV

If you cannot use the Synopsys-supplied swiftpli shared library, refer to the Cadence
documentation for information on using the PLIWizard to build your own PLI library.
Synopsys still ships the files needed to build your own PLI. These include:

● edited copy of veriuser.c in the $LMC_HOME/sim/pli/src directory

● LMTV object (lmtv.o) in the $LMC_HOME/lib/*platform*.lib directory

● C-Pipe shared library (slm_pli_dyn.*ext*), in the $LMC_HOME/lib/*platform*.lib
directory

If you build your own PLI, you will need to edit the veriuser.c file to pick up the LMTV
header files as follows:

```
a. After #include "vxl_veriuser.h" add:
#include "ccl_lmtv_include.h"
b. After "/*** add user entries here ***/" add:
#include "ccl_lmtv_include_code.h"
```

Using Hardware Models with Verilog-XL

This section describes how to configure Release 3.5a of ModelAccess for Verilog. ModelAccess is the software you use to interface hardware models with the simulator. To dynamically link the SFI with Verilog-XL, you must have version 2.8 or later of Verilog-XL on UNIX and version 3.0 on NT. You also need Release 3.5a of ModelAccess for Verilog. The hardware modeling information is presented in the following sections of this chapter:

- "Prerequisites" on page 83
- "The ma_verilog Software Tree" on page 84
- "Using Hardware Models" on page 85

Prerequisites

If you have not already done so, perform these tasks:

- Install the Verilog-XL simulator according to instructions provided by Cadence Design Systems, Inc.

- Perform the complete installation and configuration of the hardware modeling system, including hardware and software (R3.5a or later) as outlined in the Quick Reference in Chapter 1 of either the *ModelSource Hardware Installation Guide* or the *LM-family Hardware Installation Guide.*

- Boot the modeler if it is not already booted.

The ma_verilog Software Tree

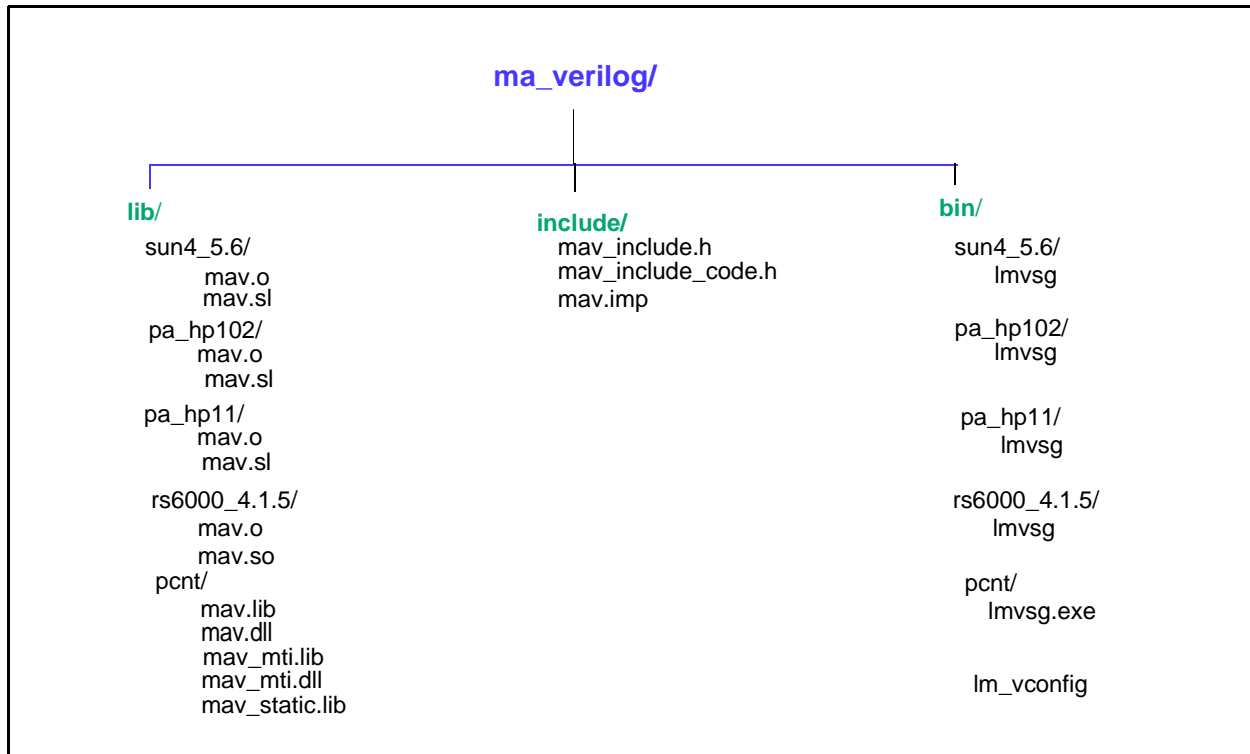The ModelAccess for Verilog (ma_verilog) directory structure is illustrated in Figure 4.

```
                            ma_verilog/
                                |
        ┌───────────────────────┼───────────────────────┐
        │                       │                       │
      lib/                   include/                  bin/
      sun4_5.6/              mav_include.h             sun4_5.6/
          mav.o              mav_include_code.h            lmvsg
          mav.sl             mav.imp
      pa_hp102/                                        pa_hp102/
          mav.o                                            lmvsg
          mav.sl
      pa_hp11/                                         pa_hp11/
          mav.o                                            lmvsg
          mav.sl
      rs6000_4.1.5/                                    rs6000_4.1.5/
          mav.o                                            lmvsg
          mav.so
      pcnt/                                            pcnt/
          mav.lib                                          lmvsg.exe
          mav.dll
          mav_mti.lib
          mav_mti.dll                                  lm_vconfig
          mav_static.lib
```

Figure 4: The ma_verilog Software Tree

Generating the Verilog-XL Model Shell

You must use the Logic Modeling Verilog Shell Generator (lmvsg) to generate new Verilog HDL shells (*model*.v files) for the hardware models you are using. Note that you cannot use any *model*.v files that might have existed prior to your use of ModelAccess for Verilog. All *model*.v files must be newly generated.

For each hardware model, both UNIX and Windows NT users issue this command at the operating system prompt:

```
% lmvsg -d destination_model.MDL
```

The complete syntax of the lmvsg command is provided in "lmvsg Command Reference" on page 99.

Using Hardware Models

To instantiate hardware models in Verilog-XL, ModelAccess for Verilog maps the Cadence PLI to the Simulator Function Interface (SFI), as shown in Figure 5. For information about the SFI, refer to the *Simulator Integration Manual.*
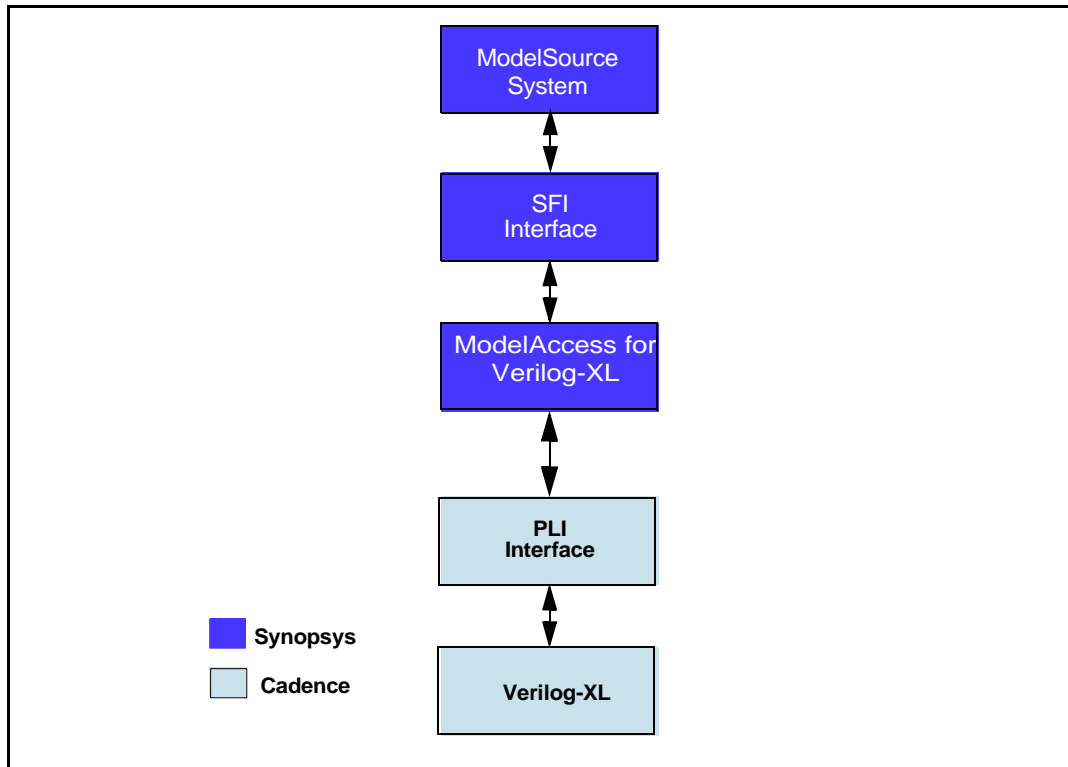

Figure 5: SFI Communication with PLI

ModelAccess for Verilog Methodology

To simulate with hardware models using ModelAccess for Verilog consists of these tasks:

- "Simulation Example" on page 86
- "Creating the Model Shell" on page 86
- "Instantiating the Hardware Model" on page 88
- "Performance Monitoring" on page 89
- "Compiling and Simulating" on page 89
- "Examining the Output verilog.log File" on page 90

Simulation Example

This simulation example illustrates how to use hardware models in a Verilog-XL simulation using ModelAccess for Verilog. This example assumes that all ModelAccess for Verilog configuration tasks have been accomplished.

Creating the Model Shell

If you use ModelAccess for Verilog, you cannot use existing *model*.v files that were generated by crshell**;** you must regenerate the *model*.v files as described.

The task of creating a model shell should have been accomplished by executing lmvsg , as described in "Generating the Verilog-XL Model Shell" on page 84. For example, to create the model shell (*model*.v file) for the TILS299 hardware model (an 8-bit universal shift/storage register with 3-state outputs) in the current working directory, execute the following:

```
% lmvsg TILS299.MDL
```

(By default, if no destination is specified, the current working directory is the destination directory for the TILS299.v file. For complete syntax of the lmvsg script, refer to "lmvsg Command Reference" on page 99.)

The following illustration shows the TILS299.v file that contains a listing of the model's pin names, pin declarations, parameter declarations, and the model invocation, which references the *model*.MDL file (in this case, TILS299.MDL).

```
// Generated by lmvsg 1.000
// Copyright (c) 1984-1996 Synopsys Inc. ALL RIGHTS RESERVED

`timescale 1 ns / 1 ns
`expand_vectornets

module TILS299(
  CLK , CLR , G1 , G2 , S0 , S1 , SL , SR , QA , QH , A , B ,
  C , D , E , F , G , H );

  // Pin declarations
  input   CLK ;
  input   CLR ;
  input   G1 ;
  input   G2 ;
  input   S0 ;
  input   S1 ;
  input   SL ;
  input   SR ;
  output  QA ;
  reg     QA__PULL ;
  reg     QA__STRONG ;
  assign  (pull0, pull1) QA =  QA__PULL ;
```

```
        assign  QA = QA__STRONG ;
        output  QH ;
        reg     QH__PULL ;
        reg     QH__STRONG ;
        assign  (pull0, pull1) QH =  QH__PULL ;
        assign  QH = QH__STRONG ;
        inout   A ;
        reg     A__PULL ;
        reg     A__STRONG ;
        assign  (pull0, pull1) A =  A__PULL ;
        assign   A = A__STRONG ;
        inout   B ;
        reg     B__PULL ;
        reg     B__STRONG ;
        assign  (pull0, pull1) B =  B__PULL ;
        assign   B = B__STRONG ;
        inout   C ;
        reg     C__PULL ;
        reg     C__STRONG ;
        assign  (pull0, pull1) C =  C__PULL ;
        assign   C = C__STRONG ;
        inout   D ;
        reg     D__PULL ;
        reg     D__STRONG ;
        assign  (pull0, pull1) D =  D__PULL ;
        assign   D = D__STRONG ;
        inout   E ;
        reg     E__PULL ;
        reg     E__STRONG ;
        assign  (pull0, pull1) E =  E__PULL ;
        assign   E = E__STRONG ;
        inout   F ;
        reg     F__PULL ;
        reg     F__STRONG ;
        assign  (pull0, pull1) F =  F__PULL ;
        assign   F = F__STRONG ;
        inout   G ;
        reg     G__PULL ;
        reg     G__STRONG ;
        assign  (pull0, pull1) G =  G__PULL ;
        assign   G = G__STRONG ;
        inout   H ;
        reg     H__PULL ;
        reg     H__STRONG ;
        assign  (pull0, pull1) H =  H__PULL ;
        assign   H = H__STRONG ;

    // Parameter declarations
      parameter  ModelType = "HARDWARE";
```

```
      parameter  TimingVersion = "TILS299.MDL";
      parameter  DelayRange = "Max";

  // Invoke the model
    initial
      begin
      $lmhw_model(
         "TILS299.MDL",
         ModelType,
         "attr", "timingversion", TimingVersion,
         "attr", "delayrange", DelayRange ,
         "in", CLK ,
         "in", CLR ,
         "in", G1 ,
         "in", G2 ,
         "in", S0 ,
         "in", S1 ,
         "in", SL ,
         "in", SR ,
         "out", QA , QA__STRONG , QA__PULL ,
         "out", QH , QH__STRONG , QH__PULL ,
         "io", A , A__STRONG , A__PULL ,
         "io", B , B__STRONG , B__PULL ,
         "io", C , C__STRONG , C__PULL ,
         "io", D , D__STRONG , D__PULL ,
         "io", E , E__STRONG , E__PULL ,
         "io", F , F__STRONG , F__PULL ,
         "io", G , G__STRONG , G__PULL ,
         "io", H , H__STRONG , H__PULL );
      end
   endmodule

   `autoexpand_vectornets
```

Instantiating the Hardware Model

Before instantiating a hardware model, you first examine the *model*.v file you created, to get the port names to use in the instantiation, and also to see whether you want to change any of the model's default parameters. The *model*.v files contain default values for the model parameters, which you can override using the "defparam" statement in the model instantiation.

The following example shows how to instantiation a hardware model (TILS299 in this case) in a testbench. Notice the two "defparam" statements; the definitions of the TimingVersion (TILS299A.MDL) and DelayRange (MIN) parameters in the instantiation override the default definitions in the model.v file (TILS299.MDL and MAX, respectively). In this example, TILS299A.MDL represents a custom timing version that the designer wants to use instead of the default timing version TILS299.MDL.

```
/ Instantiate UUT : ModelSource TILS299 hardware model : U1
  defparam    U1.TimingVersion="TILS299A.MDL";
  defparam    U1.DelayRange = "MIN";
 TILS299 U1(.CLK (clkw),
            .CLR (clrw),
            .A   (io1w[0]),
            .B   (io1w[1]),
            .C   (io1w[2]),
            .D   (io1w[3]),
            .E   (io1w[4]),
            .F   (io1w[5]),
            .G   (io1w[6]),
            .H   (io1w[7]),
            .G1  (g1w),
            .G2  (g2w),
            .QA  (qa1w),
            .QH  (qh1w),
            .S0  (s0w),
            .S1  (s1w),
            .SL  (slw),
            .SR  (srw));
```

Performance Monitoring

You can monitor the performance of a hardware model and append the results to the simulator log file after simulation. To enable performance monitoring, in the window where you are running the simulator, enter the following:

```
% setenv LM_OPTION "monitor_performance"
```

For more information, refer to "Performance Monitoring" in the *ModelSource User's Manual.*

Compiling and Simulating

UNIX users accomplish this task by executing the Verilog-XL executable previously built, referencing the testbench and the *model*.v file, as in the following example:

```
% verilog TILS299.v tbench.v +loadpli1=mav:mav_boot
```

As Verilog executes, it outputs progress, status, and error messages to the screen and saves the transcript to a file named verilog.log, which you can examine if necessary for troubleshooting.

Examining the Output verilog.log File

After echoing the command that invoked Verilog, and the copyright and source information, Verilog announces its progress as it compiles the input files. When the prompt C1> is issued, the simulator is waiting at time 0 for you to enter a command. Typing a period (.), which means "continue", starts the simulation run. Typing "$finish;" at the prompt terminates the simulation session.

Notice in particular these lines, which state the release numbers of ModelAccess for Verilog and SFI:

```
Runtime, ModelAccess for Verilog-XL R3.5a
SFI Copyright 1988-2000 Synopsys, Incorporated.; 08/30/00; R3.5a
```

If you are troubleshooting and call Synopsys Technical Support for help, you will be asked for the SFI release number (in this case, R3.5a). (For instructions on contacting Synopsys Technical Support, refer to "Getting Help" on page 14.) The following illustration shows an example verilog.log file without errors.

```
Host command: verilog.lmv
Command arguments:
    -s
    TILS299.v
    tbench.v

VERILOG-XL 2.2.1 log file created Jan  8, 1997  14:14:00
VERILOG-XL 2.2.1   Jan  8, 1997  14:14:00

...
...
Compiling source file "TILS299.v"
Compiling source file "tbench.v"

    Runtime, ModelAccess for Verilog R2.0
    SFI Copyright 1988-1996 Synopsys, Incorporated.; 05 Sep 1996;
R3.3a
Type ? for help
C1 > .
...
...
L47 "tbench.v": $stop at simulation time 4200
C1 > $finish;
C1: $finish at simulation time 4200
54 simulation events + 10 accelerated events
```

```
CPU time: 0.4 secs to compile + 0.2 secs to link + 0.5 secs in
simulation
End of VERILOG-XL 2.2.1   Jan  8, 1997  14:15:09
```

Optional Capabilities During Simulation

During simulation, you can optionally enable timing measurement and test vector logging.

Timing Measurement

LM-family modeling systems can measure input-to-output propagation delays on a hardware model. You enable timing measurement using the command $lm_timing_measurements(), described in "$lm_timing_measurements Command Reference" on page 96.

☞ **Note** ───
    Timing measurement is not supported for ModelSource 3200 and 3400.
───

The following illustration shows an example of timing measurement for the TILS299 model. The six lines of code following "SIMULATION run time duration" turn on timing measurement, measure for 4200 timing units, then turn off timing measurement. The timing information is saved in the file TILS299.TIM.

Test Vector Logging

ModelSource and LM-family modeling systems can capture and write to a file the input stimuli presented to a hardware model, as well as the resulting sampled output values.

Test vectors are useful for debugging a simulation and for verifying the functionality of a hardware model.   You enable test vector logging by using the command $lm_log_test_vectors(), described in "$lm_log_test_vectors Command Reference" on page 94.

The following illustration also shows an example of test vector logging for the model TILS299. The six lines of code following the timing measurement enable test vector logging, implement the logging for 4200 time units (the duration of the simulation), and then disable the logging. The test vectors are saved in a file named hwm299.vec.

```
// Instantiate UUT : ModelSource TILS299 hardware model : U1
TILS299 U1(.CLK (clkw),
           .CLR (clrw),
           .A   (io1w[0]),
           .B   (io1w[1]),
           .C   (io1w[2]),
           .D   (io1w[3]),
           .E   (io1w[4]),
```

```
                .F   (io1w[5]),
                .G   (io1w[6]),
                .H   (io1w[7]),
                .G1  (g1w),
                .G2  (g2w),
                .QA  (qa1w),
                .QH  (qh1w),
                .S0  (s0w),
                .S1  (s1w),
                .SL  (slw),
                .SR  (srw));
   // SIMULATION run time duration
   initial
    begin
    $lm_timing_measurements ("tbench.U1, 1, "TILS299.TIM");
    #4200
    $lm_timing_measurements ("tbench.U1, 0, "TILS299.TIM");
    end
   initial
     begin
     $lm_log_test_vectors("tbench.U1",1,"hwm299.vec");
     #4200 $stop;
     $lm_log_test_vectors("tbench.U1",0,"hwm299.vec");
     end
```

The Test Vector Log File

This next illustration shows part of a test vector log file, hwm299.vec.

```
# test_vector_format 2
# test TILS299
# time stamp = 1 nanosecond
# runtime_modeler_software R3.3a
# simulator_function_interface R3.3a
SR                          1       I
SL                          2       I
...
#patterns  { IIIIIIIIBBBBBBBBOO }
#          {      ---           }
#          { SSSSCGGCHGFEDCBAQQ }
#          { RL10L21L////////HA }
#          {      K  RQQQQQQQQ  }
#          {          HGFEDCBA  }
        INIT   DDDDDUUDTTTTTTTTT
                      ZZZZZZZZLL
           0   DDDDUDDDTTTTTTTTT
                      LLLLLLLLLL
          30   DDDDUDDDDDDDDDDDD
          50   DDDDDDDDDDDDDDDDD
         100   DDDDUDDDDDDDDDDDD
```

```
                       LLLLLLLLLL
         150   DDDDDDDUDDDDDDDD
                       LLLLLLLLLL
         200   DDDDUDDUDDDDDDDD
                       LLLLLLLLLL
         250   DDDDDDDUDDDDDDDD
         300   DDDDUDDUDDDDDDDD
                       LLLLLLLLLL
         350   DDUUDDDUDDDDDDDD
                         zzzzzzzzLL
         355   DDUUDUUUDDDDDDDD
                         zzzzzzzzLL
     . . .
     . . .
```

Understanding the Test Vector File

The test vector file is written in Logic Modeling test vector format. Symbols for input and output values are defined in Table 15.

Table 15: Test Vector Symbols

Symbol	Input/Output	Definition
U	Input	Drive hard 1
D	Input	Drive hard 0
u	Input	Drive soft (resistive) 1
d	Input	Drive soft (resistive) 0
T	Input	Drive floating level
N	Input	Drive unknown level
H	Output	Sense hard 1
L	Output	Sense hard 0
h	Output	Sense soft (resistive) 1
l	Output	Sense soft (resistive) 0
Z	Output	Sense floating level. Used for an I/O pin in the input state whose last driven value was 1 (either U or u)
z	Output	Sense floating level. Used in two cases: for an I/O pin in the input state whose last driven value was 0 (either D or d), or for an output pin that is not driving.

Table 15: Test Vector Symbols (Continued)

Symbol	Input/Output	Definition
X	Output	Sense unknown value. Unknowns on outputs are generated by unknown propagation, value forcing, voltage unknowns, or inconsistent unknowns.
?	Output	Sense any level ("don't care").

Saving and Restarting the Simulation State

The Verilog-XL $save() task saves the complete simulation data structure into a specified file. The saved data structure includes the pattern memory for each hardware model simulation instance.

The $restart() task restores the complete Verilog-XL simulation from the specified file. The pattern memory for each hardware model simulation instance is restored into the hardware modeler's pattern memory.

Linking the SFI Debug Library

By default, ModelAccess for Verilog dynamically links the non-debug version of the SFI library. If you want to use the SFI library's debug version for troubleshooting, define the environment variable HOSTDEBUG. For information about setting and using HOSTDEBUG, refer to the *Simulator Integration Manual.* For troubleshooting assistance, contact Synopsys Technical Support (for instructions, refer to "Getting Help" on page 14).

$lm_log_test_vectors Command Reference

The $lm_log_test_vectors command enables test vector logging for a specified instance, and specifies a file name for the test vector log.

Syntax

$lm_log_test_vectors (*"instance_path"*, *on_off*, *"filename"*)

Arguments

instance_path   Specifies the pathname of the model instance for which test vector logging is to be enabled or disabled.

on_off   Indicates whether test vector logging is to be enabled or disabled. Allowed values are 1 to enable logging, or 0 (the default) to disable logging.

filename   Specifies the file name to be used for the test vector log.

Description

Enables test vector logging for a specified instance and specifies a filename for the test vector log. By default, test vector logging is not performed. When test vector logging is on, the pin value information created during the simulation for the specified device instance is written in test vector file format to *filename.*

For detailed information about test vector logging, refer to the *ModelSource User's Manual* or the *LM-family Modeler Manual.*

Example

The following example enables test vector logging for the instance "U1", and saves the test vector log in the file "U1.log".

```
$lm_log_test_vectors ("Tbench.U1", 1, "U1.log");
```

$lm_loop_instance Command Reference

The $lm_loop_instance command enables the loop mode for a specified model instance.

Syntax

$**lm_loop_instance** ("*instance_path"*)

Arguments

instance_path   Specifies the pathname of the model instance for which the
                          loop mode is to be enabled.

Description

Enables the loop mode for a specified instance. In loop mode, the hardware modeler repeatedly plays to the physical device the pattern history of the specified device instance. This command is most often used to analyze the behavior of a device and its pattern history with an oscilloscope or logic analyzer connected to the device.

Once in loop mode, the interface prompts you to press the Return key to exit the loop mode.

Examples

The following example turns on loop mode for the "U1" model instance.

```
$lm_loop_instance ( "U1" );
```

The following message is displayed while the instance is in loop mode.

```
Entering loop mode for hardware model instance U1
Press Return to terminate loop mode.
```

$lm_timing_information Command Reference

The $lm_timing_information command lets you override the hardware modeler's default handling of timing information for a specified model instance.

Syntax

$lm_timing_information (*"instance_path"*, *"timing_option"*);

Arguments

instance_path   Specifies the Verilog pathname of the instance whose timing information is to be modified.

timing_option   Allowed values are "nodelay" to ignore all delay information, "delay" to process all delay information, "notimingchecks" to ignore all timing checks, and "timingchecks" to apply all timing checks. The defaults are "delay" and "timingchecks".

Description

The $lm_timing_information command allows you to override the hardware modeler's default handling of timing information for a specified model instance. By default, the hardware modeler processes all delay information and applies all timing checks. You can decrease model evaluation time by disabling these activities. The hardware modeler does not process information that is not needed by the simulator.

Example

The following example disables timing checks for the "U1" model instance.

```
$lm_timing_information ("U1", "notimingchecks");
```

$lm_timing_measurements Command Reference

The $lm_timing_measurements command enables timing measurements for a specified model instance. It is not supported for ModelSource 3200 and 3400.

Syntax

$lm_timing_measurements (*"instance_path"*, *on_off*, *"filename"*)

Arguments

instance_path	Specifies the pathname of the model instance for which timing measurement is to be enabled or disabled.
on_off	Indicates whether test vector logging is to be enabled or disabled. Allowed values are 1 to enable logging, or 0 (the default) to disable logging.
filename	Specifies the file name to be used for the test vector log.

Description

The $lm_timing_measurements command enables timing measurement for a specified model instance. By default, timing measurement is not performed. Instead, the hardware modeler uses the delay values provided in the .DLY file in the Shell Software. When timing measurement is enabled, the hardware modeler returns to the simulator and logs to the specified file the actual delays measured from the device.

Example

The following example enables timing measurements for the"U1" model instance and saves the timing measurement log in the "U1.log" file.

```
$lm_timing_measurements ( "Tbench.U1", 1, "U1.log");
```

$lm_unknowns Command Reference

The $lm_unknowns command lets you override the hardware modeler's default handling of unknown values for specified instances or pins, or for all hardware model instances in the simulation.

Syntax

$lm_unknowns ("*option=value*" [,"*option=value*",...] [, "*device_or_pin*"])

Arguments

You can use the following values for "option=value":

propagate=<u>yes</u>	no	When "yes" (the default), enables the "on_unknown propagate" statement, if there is one, in the model's options file (for example, TILS299.OPT) for the specified instance or pin, or for all hardware model instances in the simulation. Set

propagate=no if you want to disable or override the "on_unknown propagate" statement in the .OPT file for a specific instance or pin.

👉 **Note**

If there is no "on_unknown propagate" statement in the model's .OPT file, unknown propagation is disabled even if you use" $lm_unknowns propagate=yes". For the "propagate=yes" option to have an effect, there must be an "on_unknown propagate" statement in the model's .OPT file. For more information about the on_unknown statement, refer to the *Shell Software Reference Manual.*

value=<u>previous</u> | high | low | float

Specifies the value to be passed to the device when an unknown value is passed to the modeler. The default is "previous", meaning that if the simulator sets an input pin to "unknown", the modeler drives the input to its previous value. For more information, refer to the description of set_previous in the on_unknown reference pages in the *Shell Software Reference Manual.*

sequence_count=*num_sequences*

Specifies the number of random sequences to propagate unknowns through the hardware model. The *num_sequences* setting is an integer of value 0 (the default) through 20. The default value, 0, is usually sufficient; setting a higher value ensures that unknowns will be propagated, but uses more pattern memory.

random_seed=*seed_value*

Specifies the initial seed for the random sequence generator. *seed_value* is an integer of value 0 (the default) through 65535.

device_or_pin   Specifies the Verilog pathname of a device or pin whose unknown values are to be translated into the value specified by *value*. The default is to apply the statement to all hardware model instances in the simulation.

Description

The $lm_unknowns command lets you override the hardware modeler's default handling of unknown values for specified model instances or pins, or for all hardware model instances in the simulation. By default, the hardware modeler translates all unknown values to "previous" before passing them to the device. Using this command, you can specify values of high (1), low (0) or float (?), or disable unknown propagation, for a specified instance or pin, or for all hardware model instances in the simulation if no instance or pin is given. For detailed information about unknown handling, refer to the *Shell Software Reference Manual*.

Examples

The following example disables unknown propagation and causes a low value to be passed to the device when an unknown value is passed to the hardware modeler for the instance "Tbench.U1."

```
$lm_unknowns ("propagate=no", "value=low", "Tbench.U1")
```

The following example disables unknown propagation, causes a high value to be passed to the device when an unknown value is passed to the modeler, specifies 20 random sequences to propagate unknowns through the hardware model, and specifies 200 as the seed for the random sequence generator, for all hardware model instances in the simulation.

```
$lm_unknowns ("propagate=yes", "value=high", "sequence_count=20",
   "random_seed=200")
```

lmvsg Command Reference

For a specified hardware model the lmsvg script creates a *model*.v file and places it in the specified destination directory.

Syntax

lmvsg [-d *destination*] [-i] [-w] [-v *vector_path*] [-h] *model*.MDL

Arguments

-d *destination*	Specifies the destination directory in which to store the generated *model*.v file. The default is the current directory.
-i	Generates a warning if a pin name is an illegal Verilog identifier. By default, no warning is issued.

-w	Specifies pullup and pulldown signal strength of `weak1` and `weak0` instead of the default `pull1` and `pull0`, respectively. This will provide compatibility with Cadence's hardware modeler interface.
-v *vector_path*	Specifies the pathname to the file containing a list of vectors.
-h	Displays the online help for this command.
*model***.**MDL	Specifies the name of the MDL file of the hardware model whose *model*.v file is to be generated.

Description

The lmvsg script creates a *model*.v, in the destination directory. The model's pin names may not be legal Verilog identifiers. If a pin name is found that is not a legal Verilog identifier, lmvsg escapes the illegal name (for example, the pin name "-CLR" becomes "\-CLR".) and, if the -i switch was issued, displays a warning message.

If a pin alias is defined in the *model*.NAM file, the pin alias is used as the pin name. For information about editing the *model*.NAM file, refer to the *ModelSource User's Manual*.

By default, lmvsg generates a module that contains a port for each logical pin. If you want the module to use vectors for buses, you can provide a file containing a list of the vectors. For example, if a device contains a 32-bit address bus, the default behavior of lmvsg is to generate a module with a port list containing the ports A0, A1, ..., A31. You can use the -v switch to name a file containing the statement "A[31:0]". lmvsg then generates the module using a 32-bit vector for the address bus.

4

Using NC-Verilog with Synopsys Models

Overview

This chapter explains how to use SmartModels, FlexModels, MemPro models, and hardware models with NC-Verilog. The procedures are organized into the following major sections:

Setting Environment Variables

First, set the basic environment variables. If you are not using one of the model types, skip that step. In some cases the procedures that follow in this chapter include steps for setting additional environment variables.

1. Set the LMC_HOME variable to the location of your SmartModel. FlexModel, and MemPro model installation tree, as shown in the following example:

   ```
   % setenv LMC_HOME path_to_models_installation
   ```

2. Set the LM_LICENSE_FILE or SNPSLMD_LICENSE_FILE environment variable
   to point to the product authorization file, as shown in the following example:

   % **setenv LM_LICENSE_FILE** *path_to_product_authorization_file*

   % **setenv SNPSLMD_LICENSE_FILE** *path_to_product_authorization_file*

   You can put license keys for multiple products (for example, SmartModels and
   hardware models) into the same authorization file. If you need to keep separate
   authorization files for different products, use a colon-separated list (UNIX) or
   semicolon-separated list (NT) to specify the search path in your variable setting.

   Caution

   Do not include la_dmon-based authorizations in the same file with snpslmd-
   based authorizations. If you have authorizations that use la_dmon, keep
   them in a separate license file that uses a different license server (lmgrd)
   process than the one you use for snpslmd-based authorizations.

3. If you are using the hardware modeler, set the LM_DIR and LM_LIB environment
   variables, as shown in the following examples:

   % **setenv LM_DIR** *hardware_model_install_path***/sms/lm_dir**
   % **setenv LM_LIB** *hardware_model_install_path*/**sms/models: **
    *hardware_model_install_path***/sms/maps**

   If you put your models in a directory other than the default of /sms/models, modify
   the above variable setting accordingly.

4. Set the CDS_INST_DIR variable to the location of your Cadence installation tree,
   as shown in the following example, and make sure that NC-Verilog is set up
   properly in your environment:

   % **setenv CDS_INST_DIR** *path_to_Cadence_installation*

5. Depending on your platform, set your load library variable to point to the platform-
   specific directory in $LMC_HOME, as shown in the following examples:

   Solaris:

   % **setenv LD_LIBRARY_PATH $LMC_HOME/lib/sun4Solaris.lib:$LD_LIBRARY_PATH**

   Linux:

   % **setenv LD_LIBRARY_PATH $LMC_HOME/lib/x86_linux.lib:$LD_LIBRARY_PATH**

   AIX:

   % **setenv LIBPATH $LMC_HOME/lib/ibmrs.lib:$LIBPATH**

HP-UX:

```
 % setenv SHLIB_PATH $LMC_HOME/lib/hp700.lib:$SHLIB_PATH
```

NT:

Make sure that %LMC_HOME%\lib\pcnt.lib is in the Path user variable.

Using SmartModels with NC-Verilog

SmartModels work with NC-Verilog using a PLI application called LMTV that is delivered in the form of a swiftpli shared library in $LMC_HOME/lib/*platform*.lib. If you cannot use the swiftpli, refer to "Static Linking with LMTV" on page 104.

To use the prebuilt swiftpli, follow this procedure:

1. Instantiate SmartModels in your design, defining the ports and defparams as required. For details on required SmartModel SWIFT parameters and model instantiation examples, refer to "Using SmartModels with SWIFT Simulators" on page 18.

2. There is no need to build a Verilog executable. You can use the one from $CDS_INST_DIR/tools/bin by adding it to your path statement.

3. To use the swiftpli shared library, invoke the NC-Verilog simulator to compile and simulate your design as shown in the following examples:

   UNIX

   ```
   % ncverilog testbench model.v +loadpli1=swiftpli:swift_boot \
       +incdir+$LMC_HOME/sim/pli/src
   ```

   NT

   ```
   > ncverilog testbench model.v +loadpli1=swiftpli:swift_boot
       +incdir+%LMC_HOME%\sim\pli\src
   ```

☞ **Note**

If you are using ncelab and ncsim, use the -loadpli1 switch instead of the +loadpli1 switch.

For information on LMTV commands that you can use with SmartModels on NC-Verilog, refer to "LMTV Command Reference" on page 279.

Static Linking with LMTV

If you cannot use the Synopsys-supplied swiftpli shared library, refer to the Cadence documentation for information on using the PLIWizard to build your own PLI library. Synopsys still ships the files needed to build your own PLI. These include:

● edited copy of veriuser.c in the $LMC_HOME/sim/pli/src directory

● LMTV object (lmtv.o) in the $LMC_HOME/lib/*platform*.lib directory

If you build your own PLI, you will need to edit the veriuser.c file to pick up the LMTV header files as follows:

```
a. After #include "vxl_veriuser.h" add:
#include "ccl_lmtv_include.h"
b. After "/*** add user entries here ***/" add:
#include "ccl_lmtv_include_code.h"
```

Using FlexModels with NC-Verilog

FlexModels work with NC-Verilog using a PLI application called LMTV that is delivered in the form of a swiftpli shared library in $LMC_HOME/lib/*platform*.lib. If you cannot use the swiftpli, refer to "Static Linking with LMTV" on page 106.

To use the prebuilt swiftpli, follow this procedure:

1. Create a working directory and run flexm_setup to make copies of the model's interface and example files there, as shown in the following example:

```
% $LMC_HOME/bin/flexm_setup -dir workdir model_fx
```

You must run flexm_setup every time you update your FlexModel installation with a new model version. Table 16 lists the files that flexm_setup copies to your working directory.

Table 16: FlexModel NC-Verilog Files

File Name	Description	Location
*model*_pkg.inc	Verilog task definitions for FlexModel interface commands. This file also references the flexmodel_pkg.inc and *model*_user_pkg.inc files.	*workdir*/src/verilog/
*model*_user_pkg.inc	Clock frequency setup and user customizations.	*workdir*/src/verilog/
*model*_fx_vxl.v	A SWIFT wrapper that you can use to instantiate the model.	*workdir*/examples/verilog/

Table 16: FlexModel NC-Verilog Files (Continued)

File Name	Description	Location
model.v	A bus-level wrapper around the SWIFT model. This allows you to use vectored ports for the model in your testbench.	*workdir*/examples/verilog/
*model*_tst.v	A testbench that instantiates the model and shows how to use basic model commands.	*workdir*/examples/verilog/

2. Update the clock frequency supplied in the *model*_user_pkg.inc file to correspond to the CLK period you want for the model. This file is located in:

   > *workdir*/**src/verilog/***model*_**user_pkg.inc**

   where *workdir* is your working directory.

3. Add the following line to your Verilog testbench to include FlexModel testbench interface commands in your design:

   > **`include "***model*_**pkg.inc"**

☞ **Note**

   Be sure to add *model*_pkg.inc within the module from which you will be issuing FlexModel commands.

   Because the *model*_pkg.inc file includes references to flexmodel_pkg.inc and *model*_user_pkg.inc, you don't need to add flexmodel_pkg.inc or *model*_user_pkg.inc to your testbench.

4. Instantiate FlexModels in your design, defining the ports and defparams as required (refer to the example testbench supplied with the model). You use the supplied bus-level wrapper (*model*.v) in the top-level of your design to instantiate the supplied bit-blasted wrapper (*model*_fx_vxl.v).

   Example using bus-level wrapper (*model*.v) without timing:

   ```
   model U1 ( model ports )
   defparam
         U1.FlexModelId = "TMS_INST1";
   ```

   Example using bus-level wrapper (*model*.v) with timing:

   ```
   model U1 ( model ports )
   defparam
         U1.FlexTimingMode = `FLEX_TIMING_MODE_ON,
         U1.TimingVersion = "timingversion",
         U1.DelayRange = "range",
         U1.FlexModelId= "TMS_INST1";
   ```

5. There is no need to build a Verilog executable. You can use the one from $CDS_INST_DIR/tools/bin by adding it to your path statement.

6. Invoke the NC-Verilog simulator to compile and simulate your design as shown in the following examples:

   UNIX

   ```
   % ncverilog testbench +loadpli1=swiftpli:swift_boot \
       ./workdir/examples/verilog/model.v \
       ./workdir/examples/verilog/model_fx_vxl.v \
       +incdir+$LMC_HOME/sim/pli/src \
       +incdir+workdir/src/verilog
   ```

   NT

   ```
   > ncverilog testbench +loadpli1=swiftpli:swift_boot
       workdir\examples\verilog\model.v
       workdir\examples\verilog\model_fx_vxl.v
       +incdir+%LMC_HOME%\sim\pli\src
       +incdir+workdir\src\verilog
   ```

☞ Note

If you are using ncelab and ncsim, use the -loadpli1 switch instead of the +loadpli1 switch.

For information on LMTV commands that you can use with FlexModels on NC-Verilog, refer to .

Static Linking with LMTV

If you cannot use the Synopsys-supplied swiftpli shared library, refer to the Cadence documentation for information on using the PLIWizard to build your own PLI library. Synopsys still ships the files needed to build your own PLI. These include:

● edited copy of veriuser.c in the $LMC_HOME/sim/pli/src directory

● LMTV object (lmtv.o) in the $LMC_HOME/lib/*platform*.lib directory

● C-Pipe shared library (slm_pli_dyn.*ext*), in the $LMC_HOME/lib/*platform*.lib directory

If you build your own PLI, you will need to edit the veriuser.c file to pick up the LMTV header files as follows:

```
a. After #include "vxl_veriuser.h" add:
#include "ccl_lmtv_include.h"
b. After "/*** add user entries here ***/" add:
#include "ccl_lmtv_include_code.h"
```

Using MemPro Models with NC-Verilog on UNIX

MemPro models work with NC-Verilog using a PLI application called LMTV that is delivered in the form of a swiftpli shared library in $LMC_HOME/lib/*platform*.lib. If you cannot use the swiftpli, refer to "Static Linking with LMTV" on page 108.

To use the prebuilt swiftpli, follow this procedure:

1. To include MemPro testbench interface commands in your design, add one of the following lines to your testbench:

   Verilog testbench:

   ```
   `include "mempro_pkg.v"
   ```

   C testbench:

   ```
   #include "mempro_c_tb.h"
   ```

   For more information on using the MemPro testbench interfaces, refer to the "HDL Testbench Interface" and "C Testbench Interface" chapters in the *MemPro User's Manual*.

2. Instantiate MemPro models in your design. Define ports and generics as required. For information on generics used with MemPro models, refer to "Instantiating MemPro Models" on page 32. For information on message levels and message level constants, refer to "Controlling MemPro Model Messages" on page 33.

3. There is no need to build a Verilog executable. You can use the one from $CDS_INST_DIR/tools/bin by adding it to your path statement.

4. Invoke the NC-Verilog simulator to compile and simulate your design as shown in the example below:

   ```
   % ncverilog testbench Verilog_modules MemPro_model_files \
   +incdir+$LMC_HOME/sim/pli/src \
   +loadpli1=swiftpli:swift_boot
   ```

☞ **Note**

If you are using ncelab and ncsim, use the -loadpli1 switch instead of the +loadpli1 switch.

Static Linking with LMTV

If you cannot use the Synopsys-supplied swiftpli shared library, refer to the Cadence documentation for information on using the PLIWizard to build your own PLI library. Synopsys still ships the files needed to build your own PLI. These include:

- edited copy of veriuser.c in the $LMC_HOME/sim/pli/src directory

- LMTV object (lmtv.o) in the $LMC_HOME/lib/*platform*.lib directory

- C-Pipe shared library (slm_pli_dyn.*ext*), in the $LMC_HOME/lib/*platform*.lib directory

If you build your own PLI, you will need to edit the veriuser.c file to pick up the LMTV header files as follows:

```
a. After #include "vxl_veriuser.h" add:
#include "ccl_lmtv_include.h"
b. After "/*** add user entries here ***/" add:
#include "ccl_lmtv_include_code.h"
```

Using Hardware Models with NC-Verilog

This section explains how to use Release 3.5a of ModelAccess for Verilog to interface hardware models with NC-Verilog. It is not necessary to edit and use the Makefile.nc to build a standalone version of the simulator to link to the hardware modeler. Note that dynamic linking is only supported on version 2.8 and above of NC-Verilog on HP-UX and Solaris, and version 3.0 on NT.

1. There is no need to build a Verilog executable. You can use the one from $CDS_INST_DIR/tools/bin by adding it to your path statement.

2. Set your SHLIB_PATH or LD_LIBRARY_PATH variable to point to the directories that contain the ModelAccess libraries. Solaris users also need to add the /usr/dt/lib and /usr/openwin/lib libraries.

   HP-UX

   ```
   % setenv SHLIB_PATH \
   hardware_model_iinstall_path/sms/ma_verilog/lib/pa_hp102:
   $CDS_INST_DIR/tools/lib
   ```

   Solaris

   ```
   % setenv LD_LIBRARY_PATH \
   hardware_model_install_path/sms/ma_verilog/lib/sun4.solaris:\
   $CDS_INST_DIR/tools/lib:/usr/dt/lib:/usr/openwin/lib
   ```

   For NT, add this path to the PATH user variable:

   ```
   hardware_model_install_path\sms\ma_verilog\lib\pcnt
   ```

3. Invoke the simulator as shown in the following example:

```
% ncverilog testbench.v model.v +loadpli1=mav:mav_boot
```

NC-Verilog Utilities

The following hardware model utilities are supported in NC-Verilog:

$**lm_log_test_vectors** (*"instance_path"*, *on_off*, *"filename"*)

The $lm_log_test_vectors command enables test vector logging for a specified instance, and specifies a file name for the test vector log. For a detailed syntax description, refer to "$lm_log_test_vectors Command Reference" on page 94.

$**lm_loop_instance** (*"instance_path"*)

The $lm_loop_instance command enables the loop mode for a specified model instance. For a detailed syntax description, refer to "$lm_loop_instance Command Reference" on page 95.

$lm_timing_information (*"instance_path"*, *"timing_option"*)

The $lm_timing_information command lets you override the hardware modeler's default handling of timing information for a specified model instance. For a detailed syntax description, refer to "$lm_timing_information Command Reference" on page 96.

$**lm_timing_measurements** (*"instance_path"*, *on_off*, *"filename"*)

The $lm_timing_measurements command enables timing measurements for a specified model instance. It is not supported for ModelSource 3200 and 3400. For a detailed syntax description, refer to "$lm_timing_measurements Command Reference" on page 96.

$lm_unknowns (*"option=value"* [,*"option=value"*,...] [, *"device_or_pin"*])

The $lm_unknowns command lets you override the hardware modeler's default handling of unknown values for specified instances or pins, or for all hardware model instances in the simulation. For a detailed syntax description, refer to "$lm_unknowns Command Reference" on page 97.

5

Using MTI Verilog with Synopsys Models

Overview

This chapter explains how to use SmartModels, FlexModels, MemPro models, and hardware models with MTI Verilog (ModelSim/VLOG). The procedures are organized into the following major sections:

- "Setting Environment Variables" on page 111

- "Using SmartModels with MTI Verilog" on page 113

- "Using FlexModels with MTI Verilog" on page 115

- "Using MemPro Models with MTI Verilog" on page 118

- "Using Hardware Models with MTI Verilog" on page 120

Setting Environment Variables

First, set the basic environment variables. If you are not using one of the model types, skip that step. In some cases the procedures that follow in this chapter include steps for setting additional environment variables.

1. Set the LMC_HOME variable to the location of your SmartModel. FlexModel, and MemPro model installation tree, as shown in the following example:

   ```
   % setenv LMC_HOME path_to_models_installation
   ```

2. Make sure MTI Verilog is set up properly in your environment.

3. Set the LM_LICENSE_FILE or SNPSLMD_LICENSE_FILE environment variable to point to the product authorization file, as shown in the following example:

    % **setenv LM_LICENSE_FILE** *path_to_product_authorization_file*

    % **setenv SNPSLMD_LICENSE_FILE** *path_to_product_authorization_file*

    You can put license keys for multiple products (for example, SmartModels and hardware models) into the same authorization file. If you need to keep separate authorization files for different products, use a colon-separated list (UNIX) or semicolon-separated list (NT) to specify the search path in your variable setting.

Caution

Do not include la_dmon-based authorizations in the same file with snpslmd-based authorizations. If you have authorizations that use la_dmon, keep them in a separate license file that uses a different license server (lmgrd) process than the one you use for snpslmd-based authorizations.

4. If you are using the hardware modeler, set the LM_DIR and LM_LIB environment variables, as shown in the following examples:

    % **setenv LM_DIR** *hardware_model_install_path***/sms/lm_dir**
    % **setenv LM_LIB** *hardware_model_install_path*/**sms/models: **
     *hardware_model_install_path***/sms/maps**

    If you put your models in a directory other than the default of /sms/models, modify the above variable setting accordingly.

5. Depending on your platform, set your load library variable to point to the platform-specific directory in $LMC_HOME, as shown in the following examples:

    Solaris:

     % **setenv LD_LIBRARY_PATH $LMC_HOME/lib/sun4Solaris.lib:$LD_LIBRARY_PATH**

    Linux:

     % **setenv LD_LIBRARY_PATH $LMC_HOME/lib/x86_linux.lib:$LD_LIBRARY_PATH**

    AIX:

     % **setenv LIBPATH $LMC_HOME/lib/ibmrs.lib:$LIBPATH**

    HP-UX:

     % **setenv SHLIB_PATH $LMC_HOME/lib/hp700.lib:$SHLIB_PATH**

    NT:

    Make sure that %LMC_HOME%\lib\pcnt.lib is in the Path user variable.

Using SmartModels with MTI Verilog

SmartModels work with MTI Verilog using a PLI application called LMTV that is delivered in the form of a swiftpli_mti shared library in $LMC_HOME/lib/*platform*.lib. If you cannot use the swiftpli_mti, refer to "Static Linking with LMTV" on page 114.

Synopsys does not currently ship a swiftpli_mti file for use on the IBMRS AIX platform. If you are using this platform, refer to "Using SmartModels with MTI Verilog on the IBMRS AIX Platform" on page 114.

To use SmartModels with the prebuilt swiftpli_mti, follow this procedure:

1. Instantiate SmartModels in your design, defining the ports and defparams as required. For details on the required SWIFT parameters and SmartModel instantiation examples, refer to "Using SmartModels with SWIFT Simulators" on page 18.

2. Compile your code as shown in the following examples:

   UNIX

   ```
   % vlog testbench model.v +incdir+$LMC_HOME/sim/pli/src
   ```

   NT

   ```
   > vlog testbench model.v +incdir+%LMC_HOME%\sim\pli\src
   ```

   where the *model*.v files are located at $LMC_HOME/special/cds/verilog/swift. These .v files are installed during the SmartModel installation if the customer selects either Cadence or MTI for an EDAV option.

3. Invoke the simulator as shown in the following examples:

   HP-UX

   ```
   % vsim -pli $LMC_HOME/lib/hp700.lib/swiftpli_mti.sl design
   ```

   Solaris

   ```
   % vsim -pli $LMC_HOME/lib/sun4Solaris.lib/swiftpli_mti.so design
   ```

   Linux

   ```
   % vsim -pli $LMC_HOME/lib/x86_linux.lib/swiftpli_mti.so design
   ```

   NT

   ```
   > vsim %LMC_HOME%\lib\pcnt.lib\swiftpli_mti.dll design
   ```

☞ **Note**

For information on LMTV commands that you can use with SmartModels on MTI Verilog, refer to "LMTV Command Reference" on page 279.

Static Linking with LMTV

If you cannot use the Synopsys-supplied swiftpli shared library, refer to the MTI documentation for information on building your own PLI and locating it for the simulator. Synopsys still ships the files needed to build your own PLI. These include:

- edited copy of veriuser.c in the $LMC_HOME/sim/pli/src directory

- LMTV object (lmtv.o) in the $LMC_HOME/lib/*platform*.lib directory

If you build your own PLI, you will need to edit the veriuser.c file to pick up the LMTV header files as follows:

```
a. After #include "vxl_veriuser.h" add:
#include "ccl_lmtv_include.h"
b. After "/*** add user entries here ***/" add:
#include "ccl_lmtv_include_code.h"
```

Using SmartModels with MTI Verilog on the IBMRS AIX Platform

1. Open the modelsim.ini file in a text editor and locate this line:

```
; Veriuser = veriuser.o
```

2. Replace the line with the following. The following expressions should all appear on one line with the PLI entry separated by a space.

```
Veriuser = $LMC_HOME/lib/hp700.lib/slm_pli_mti.sl \

$MODEL_TECH/libswiftpli.sl
```

3. Compile your code as shown in the following example:

```
% mti_path/bin/vlog testbench +incdir+$LMC_HOME/sim/pli/src \

+incdir+workdir/src/verilog
```

4. Invoke the simulator as shown in the following example:

```
% mti_path/bin/vsim design
```

Note that the -loadpli switch is not supported on this platform.

Using FlexModels with MTI Verilog

FlexModels work with Verilog-XL using a PLI application called LMTV that is delivered in the form of a swiftpli_mti shared library in $LMC_HOME/lib/*platform*.lib. If you cannot use the swiftpli_mti, refer to "Static Linking with LMTV" on page 117.

To use the prebuilt swiftpli_mti, follow this procedure:

1. Create a working directory and run flexm_setup to make copies of the model's interface and example files there, as shown in the following example:

   ```
   % $LMC_HOME/bin/flexm_setup -dir workdir model_fx
   ```

   You must run flexm_setup every time you update your FlexModel installation with a new model version. Table 17 lists the files that flexm_setup copies to your working directory.

Table 17: FlexModel MTI Verilog Files

File Name	Description	Location
*model*_pkg.inc	Verilog task definitions for FlexModel interface commands. This file also references the flexmodel_pkg.inc and *model*_user_pkg.inc files.	*workdir*/src/verilog/
*model*_user_pkg.inc	Clock frequency setup and user customizations.	*workdir*/src/verilog/
*model*_fx_mti.v	A SWIFT wrapper that you can use to instantiate the model.	*workdir*/examples/verilog/
model.v	A bus-level wrapper around the SWIFT model. This allows you to use vectored ports for the model in your testbench.	*workdir*/examples/verilog/
*model*_tst.v	A testbench that instantiates the model and shows how to use basic model commands.	*workdir*/examples/verilog/

2. Update the clock frequency supplied in the *model*_user_pkg.inc file to correspond to the CLK period you want for the model. This file is located in:

   ```
   workdir/src/verilog/model_user_pkg.inc
   ```

   where *workdir* is your working directory.

3. Add the following line to your Verilog testbench to include FlexModel testbench interface commands in your design:

   ```
   `include "model_pkg.inc"
   ```

👉 **Note** ───

Be sure to add *model*_pkg.inc within the module from which you will be issuing FlexModel commands.

──

Because the *model*_pkg.inc file includes references to flexmodel_pkg.inc and *model*_user_pkg.inc, you don't need to add flexmodel_pkg.inc or *model*_user_pkg.inc to your testbench.

4. Instantiate FlexModels in your design, defining the ports and defparams as required (refer to the example testbench supplied with the model). You use the supplied bus-level wrapper (*model*.v) in the top-level of your design to instantiate the supplied bit-blasted wrapper (*model*_fx_mti.v).

Example using bus-level wrapper (*model*.v) without timing:

```
model U1 ( model ports )
defparam
      U1.FlexModelId = "TMS_INST1";
```

Example using bus-level wrapper (*model*.v) with timing:

```
model U1 ( model ports )
defparam
      U1.FlexTimingMode = `FLEX_TIMING_MODE_ON,
      U1.TimingVersion = "timingversion",
      U1.DelayRange = "range",
      U1.FlexModelId= "TMS_INST1";
```

5. Compile your code as shown in the following examples:

UNIX

```
% vlog testbench \
    workdir/examples/verilog/model.v \
    workdir/examples/verilog/model_fx_mti.v \
    +incdir+$LMC_HOME/sim/pli/src \
    +incdir+workdir/src/verilog
```

NT

```
> vlog testbench
    workdir\examples\verilog\model.v
    workdir\examples\verilog\model_fx_mti.v
    +incdir+%LMC_HOME%\sim\pli\src
    +incdir+workdir\src\verilog
```

6. Invoke the simulator as shown in the following examples:

HP-UX

```
% vsim -pli $LMC_HOME/lib/hp700.lib/swiftpli_mti.sl design
```

Solaris

% **vsim -pli $LMC_HOME/lib/sun4Solaris.lib/swiftpli_mti.so** *design*

Linux

% **vsim -pli $LMC_HOME/lib/x86_linux.lib/swiftpli_mti.so** *design*

NT

> **vsim -pli %LMC_HOME%\lib\pcnt.lib\swiftpli_mti.dll** *design*

☞ **Note**

For information on LMTV commands that you can use with FlexModels on MTI-Verilog, refer to .

Static Linking with LMTV

If you cannot use the Synopsys-supplied swiftpli shared library, refer to the MTI documentation for information on building your own PLI and locating it for the simulator. Synopsys still ships the files needed to build your own PLI. These include:

● edited copy of veriuser.c in the $LMC_HOME/sim/pli/src directory

● LMTV object (lmtv.o) in the $LMC_HOME/lib/*platform*.lib directory

● C-Pipe shared library (slm_pli_dyn.*ext*), in the $LMC_HOME/lib/*platform*.lib directory

If you build your own PLI, you will need to edit the veriuser.c file to pick up the LMTV header files as follows:

```
a. After #include "vxl_veriuser.h" add:
#include "ccl_lmtv_include.h"
b. After "/*** add user entries here ***/" add:
#include "ccl_lmtv_include_code.h"
```

Using MemPro Models with MTI Verilog

MemPro models work with MTI Verilog (ModelSim) using a PLI application called LMTV that is delivered in the form of a swiftpli_mti shared library in $LMC_HOME/lib/*platform*.lib. If you cannot use the swiftpli_mti, refer to "Static Linking with LMTV" on page 119.

To use the prebuilt swiftpli_mti, follow this procedure:

1. To include MemPro testbench interface commands in your design, add one of the following lines to your testbench:

   Verilog testbench:

   ```
   `include "mempro_pkg.v"
   ```

   C testbench:

   ```
   #include "mempro_c_tb.h"
   ```

   For more information on using the MemPro testbench interfaces, refer to the "HDL Testbench Interface" and "C Testbench Interface" chapters in the *MemPro User's Manual*.

2. Instantiate MemPro models in your design. Define ports and generics as required. For information on generics used with MemPro models, refer to "Instantiating MemPro Models" on page 32. For information on message levels and message level constants, refer to "Controlling MemPro Model Messages" on page 33.

3. Compile your code as shown in the following examples:

   UNIX

   ```
   % vlog testbench model.v \
       +incdir+$LMC_HOME/sim/pli/src
   ```

   NT

   ```
   > vlog testbench model.v
       +incdir+%LMC_HOME%\sim\pli\src
   ```

4. Invoke the simulator as shown in the following examples:

   HP-UX

   ```
   % vsim -pli $LMC_HOME/lib/hp700.lib/swiftpli_mti.sl design
   ```

   Solaris

   ```
   % vsim -pli $LMC_HOME/lib/sun4Solaris.lib/swiftpli_mti.so design
   ```

Linux

```
% vsim -pli $LMC_HOME/lib/x86_linux.lib/swiftpli_mti.so design
```

NT

```
> vsim -pli %LMC_HOME%\lib\pcnt.lib\swiftpli_mti.dll design
```

☞ **Note**

If you are also using SmartModels or FlexModels in your design, you do not need to load the swiftpl_mti again, since the same library is used to enable all three types of models in MTI-Verilog.

Static Linking with LMTV

If you cannot use the Synopsys-supplied swiftpli shared library, refer to the MTI documentation for information on building your own PLI and locating it for the simulator. Synopsys still ships the files needed to build your own PLI. These include:

- edited copy of veriuser.c in the $LMC_HOME/sim/pli/src directory

- LMTV object (lmtv.o) in the $LMC_HOME/lib/*platform*.lib directory

- C-Pipe shared library (slm_pli_dyn.*ext*), in the $LMC_HOME/lib/*platform*.lib directory

If you build your own PLI, you will need to edit the veriuser.c file to pick up the LMTV header files as follows:

```
a. After #include "vxl_veriuser.h" add:
#include "ccl_lmtv_include.h"
b. After "/*** add user entries here ***/" add:
#include "ccl_lmtv_include_code.h"
```

Using Hardware Models with MTI Verilog

To use hardware models with MTI Verilog, follow this procedure. This procedure covers users on UNIX and NT. If you are on NT, substitute the appropriate NT syntax for any UNIX command line examples (percent signs around variables and backslashes in paths). Note that hardware models are supported on MTI Verilog v5.4c and up.

1. MTI Verilog only supports dynamic linking of PLI libraries. The three ways to specify the required ModelAccess shared library, and the order in which the simulator looks for PLI libraries, is listed below. Choose one of the following methods:

   a. Add the platform-specific shared library to the Veriuser entry in the modelsim.ini file:

      Solaris

      ```
      Veriuser = mav.so
      ```

      AIX

      ```
      Veriuser = mav.so
      ```

      HP-UX

      ```
      Veriuser = mav.sl
      ```

      NT

      ```
      Veriuser = mav_mti.dll
      ```

   b. Add an item in the PLIOBJS environment variable list:

      ```
      % setenv PLIOBJS "mav.ext"
      ```

   c. Use the -pli switch on the simulator invocation line:

      ```
      % vsim -pli mav.ext
      ```

☞ **Note** ──

For steps b and c, fill in the correct extension for your platform.

───

2. Regardless of the option you choose, you must locate the ModelAccess PLI library for the simulator using a platform-specific environment variable or by specifying the full path to the library in Step 1. Here are examples for setting the environment variables which show the full paths to the libraries:

   Solaris

   ```
   % setenv LD_LIBRARY_PATH \
   hardware_model_install_path/sms/ma_verilog/lib/sun4_5.6/mav.so
   ```

HP-UX

```
% setenv SHLIB_PATH \
hardware_model_install_path/sms/ma_verilog/lib/pa_hp102/mav.sl
```

AIX

```
% setenv LIBPATH \
hardware_model_install_path/sms/ma_verilog/lib/rs6000_4.1.5/mav.so
```

For NT, add this path to the PATH user variable:

```
hardware_model_install_path\sms\ma_verilog\lib\pcnt
```

3. Generate a Verilog module definition or shell for each hardware model that you want to use by running the Synopsys-provided lmvsg script, as shown in the following example:

```
% lmvsg destination_model.MDL
```

For this to work, the *hardware_model_install_path*/sms//ma_verilog/bin/*platform* directory must be in your PATH. For details on the complete syntax of the lmvsg command, refer to .

4. Use the Verilog module definitions to instantiate the hardware models in your testbench. The following example shows an example instantiation for the TILS299 hardware model Notice the two "defparam" statements; the definitions of the TimingVersion (TILS299A.MDL) and DelayRange (MIN) parameters in the instantiation override the default definitions in the model.v file (TILS299.MDL and MAX, respectively). In this example, TILS299A.MDL represents a custom timing version that the designer wants to use instead of the default timing version TILS299.MDL.

```
/ Instantiate UUT : ModelSource TILS299 hardware model : U1
  defparam    U1.TimingVersion="TILS299A.MDL";
  defparam    U1.DelayRange = "MIN";
 TILS299 U1(.CLK (clkw),
            .CLR (clrw),
            .A   (io1w[0]),
            .B   (io1w[1]),
            .C   (io1w[2]),
            .D   (io1w[3]),
            .E   (io1w[4]),
            .F   (io1w[5]),
            .G   (io1w[6]),
            .H   (io1w[7]),
            .G1  (g1w),
            .G2  (g2w),
            .QA  (qa1w),
            .QH  (qh1w),
            .S0  (s0w),
```

```
                        .S1  (slw),
                        .SL  (slw),
                        .SR  (srw));
```

5. Invoke the MTI Verilog simulator as shown in the following example, which
   illustrates the use of the -pli switch to specify the PLI library.

   ```
   % vsim -pli mav_library
   ```

MTI Verilog Utilities

The following hardware model utilities are supported in MTI Verilog:

$lm_log_test_vectors (“*instance_path”*, *on_off*, “*filename”*)

The $lm_log_test_vectors command enables test vector logging for a specified instance,
and specifies a file name for the test vector log. For a detailed syntax description, refer
to “$lm_log_test_vectors Command Reference” on page 94.

$**lm_loop_instance** (“*instance_path”*)

The $lm_loop_instance command enables the loop mode for a specified model instance.
For a detailed syntax description, refer to “$lm_loop_instance Command Reference” on
page 95.

$lm_timing_information (“*instance_path”*, “*timing_option”*)

The $lm_timing_information command lets you override the hardware modeler’s
default handling of timing information for a specified model instance. For a detailed
syntax description, refer to “$lm_timing_information Command Reference” on page 96.

$lm_timing_measurements (“*instance_path”*, *on_off*, “*filename”*)

The $lm_timing_measurements command enables timing measurements for a specified
model instance. It is not supported for ModelSource 3200 and 3400. For a detailed
syntax description, refer to “$lm_timing_measurements Command Reference” on
page 96.

$lm_unknowns (“*option=value”* [,”*option=value”*,...] [, “*device_or_pin”*])

The $lm_unknowns command lets you override the hardware modeler’s default
handling of unknown values for specified instances or pins, or for all hardware model
instances in the simulation. For a detailed syntax description, refer to “$lm_unknowns
Command Reference” on page 97.

6

Using Scirocco with Synopsys Models

Overview

This chapter explains how to use SmartModels, FlexModels, MemPro models, and hardware models with Scirocco. The procedures are organized into the following major sections:

- "Setting Environment Variables" on page 124

- "Using SmartModels with Scirocco" on page 125

- "Using FlexModels with Scirocco" on page 128

- "Using MemPro Models with Scirocco" on page 132

- "Using Hardware Models with Scirocco" on page 135

💡 **Hint**
This chapter includes a script that you can use to run any FlexModel examples testbench with minimal setup required. You can cut-and-paste the script right out of this PDF file. Refer to "Script for Running FlexModel Examples in Scirocco" on page 130 for details.

Setting Environment Variables

First, set the basic environment variables. If you are not using one of the model types, skip that step. In some cases the procedures that follow in this chapter include steps for setting additional environment variables.

1. Set the LMC_HOME variable to the location of your SmartModel. FlexModel, and MemPro model installation tree, as follows:

   ```
   % setenv LMC_HOME path_to_models_installation
   ```

2. Set the SYNOPSYS_SIM variable to point to the Scirocco installation directory as follows:

   ```
   % setenv SYNOPSYS_SIM Scirocco_installation_directory
   ```

3. Source the environ.csh Scirocco environment file.

   ```
   % source $SYNOPSYS_SIM/admin/install/setup/environ.csh
   ```

4. Set the LM_LICENSE_FILE or SNPSLMD_LICENSE_FILE environment variable to point to the product authorization file, as shown in the following example:

   ```
   % setenv LM_LICENSE_FILE path_to_product_authorization_file
   ```

   ```
   % setenv SNPSLMD_LICENSE_FILE path_to_product_authorization_file
   ```

   You can put license keys for multiple products (for example, SmartModels and hardware models) into the same authorization file. If you need to keep separate authorization files for different products, use a colon-separated list (UNIX) or semicolon-separated list (NT) to specify the search path in your variable setting.

   Caution
   Do not include la_dmon-based authorizations in the same file with snpslmd-based authorizations. If you have authorizations that use la_dmon, keep them in a separate license file that uses a different license server (lmgrd) process than the one you use for snpslmd-based authorizations.

5. If you are using the hardware modeler, set the LM_DIR and LM_LIB environment variables, as shown in the following examples:

   ```
   % setenv LM_DIR hardware_model_install_path/sms/lm_dir
   % setenv LM_LIB hardware_model_install_path/sms/models: \
    hardware_model_install_path/sms/maps
   ```

   If you put your models in a directory other than the default of /sms/models, modify the above variable setting accordingly.

6. Depending on your platform, set your load library variable to point to the platform-specific directory in $LMC_HOME, as shown in the following examples:

Solaris:

```
% setenv LD_LIBRARY_PATH $LMC_HOME/lib/sun4Solaris.lib:$LD_LIBRARY_PATH
```

Linux:

```
% setenv LD_LIBRARY_PATH $LMC_HOME/lib/x86_linux.lib:$LD_LIBRARY_PATH
```

AIX:

```
% setenv LIBPATH $LMC_HOME/lib/ibmrs.lib:$LIBPATH
```

HP-UX:

```
% setenv SHLIB_PATH $LMC_HOME/lib/hp700.lib:$SHLIB_PATH
```

NT:

Make sure that %LMC_HOME%\lib\pcnt.lib is in the Path user variable.

Using SmartModels with Scirocco

To use SmartModels with Scirocco, follow this procedure:

1. To create SmartModel VHDL templates, check to see if you have write permission for $LMC_HOME/synopsys/smartmodel; if so skip to Step 4. Otherwise, open the .synopsys_vss.setup file in your current working directory and search for the string SMARTMODEL. By default, the logical library name SMARTMODEL is mapped to $LMC_HOME/synopsys/smartmodel, as follows:

   ```
   SMARTMODEL : $LMC_HOME/synopsys/smartmodel
   ```

2. Change the directory to one for which you have write permission, as shown in the following example:

   ```
   SMARTMODEL : ~/smartmodel
   ```

3. Generate a VHDL model wrapper file by invoking create_smartmodel_lib with any optional arguments. For information on the syntax for this command, refer to "create_smartmodel_lib Command Reference" on page 126.

   ```
   % $SYNOPSYS_SIM/sim/bin/create_smartmodel_lib arguments
   ```

4. If you changed the SMARTMODEL mapping in Step 2, you must use the -srcdir option to specify that directory. Also, you can save time by using the -model or -modelfile option to specify the models you want. Otherwise, the script processes all installed SmartModels. For example, here is a recommended set of options to use for one SmartModel (ttl00 in this example).:

   ```
   % $SYNOPSYS_SIM/sim/bin/create_smartmodel_lib -model ttl00 \
   -srcdir ~/smartmodel
   ```

5. After create_smartmodel_lib has finished executing, verify that the VHDL template files have been created in the appropriate directory.

6. To use SmartModels in the VHDL source file of your design, specify the SMARTMODEL library and instantiate each SmartModel component. In the VHDL design file that uses SmartModel components, enter the following library and use clauses:

```
library SMARTMODEL;
use SMARTMODEL.components.all
```

The library logical name SMARTMODEL must be mapped to appropriate directories in your .synopsys_vss.setup file, as described on page 125.

7. Add the following line to your .synopsys_vss.setup file:

```
TIMEBASE :   PS
```

8. Instantiate SmartModels in your VHDL design. For information on required configuration parameters and instantiation examples, refer to "Using SmartModels with SWIFT Simulators" on page 18.

9. Compile your testbench as shown in the following example:

```
% vhdlan testbench
```

10. Invoke the Scirocco simulator as shown in the following example:

```
% scsim design
```

For information about scsim and the VHDL debugger, refer to the *Scirocco User's Guide*.

create_smartmodel_lib Command Reference

The command reference for create_smarmodel_lib is as follows:

Syntax

create_smartmodel_lib [--] [-nc] [-create] [-srcdir *dirpath*] [-analyze] [-nowarn] [-modelfile *file*] {-model *model_name*}

Arguments

--   Displays the usage message and lists the command line options.

-nc   Suppresses the Synopsys copyright message.

-create	Creates the VHDL source files (.vhd files) for the SMARTMODEL library and saves the source files in the $LMC_HOME/synopsys directory.
-src_dir *dirpath*	Lets you specify the location of the VHDL source files that you create. The default location is $LMC_HOME/synopsys.
-analyze	Analyzes the SMARTMODEL library source files (.vhd files) by invoking vhdlan. The analyzed files (.sim and .mra files) are saved in the $LMC_HOME/synopsys/smartmodel directory. This directory is specified by SMARTMODEL logical name mapping in the setup file.
-nowarn	Suppresses the generation of warning messages that notify you of any port name mappings.
-modelfile *file*	A list of SMARTMODEL component names is read from *file*. Names are separated by spaces. Only the specified component names are included in the SMARTMODEL component library.
-model *model_name*	The *model_name* is included in the resulting SMARTMODEL component library. Repeat this option to specify multiple models. Only specified component names are included in the SMARTMODEL component library.

Description

When issued without options, the create_smartmodel_lib command takes all of the files in the $LMC_HOME/models directory, creates and analyzes the VHDL template files, and saves them in the $LMC_HOME/synopsys/smartmodel directory. If you do not have write permission for $LMC_HOME/synopsys/smartmodel, the command terminates with an error message. In that case, you must use the -src_dir option to specify a writable directory in which to place the VHDL templates. You must also specify that directory through the SMARTMODEL library mapping in the .synopsys_vss.setup file in your current working directory.

Using FlexModels with Scirocco

To use FlexModels with Scirocco, follow this procedure:

1. If you want the improved performance that comes with bused wrappers, generate a VHDL model wrapper file by invoking create_smartmodel_lib with any optional arguments. For more information on the syntax for this command, refer to "create_smartmodel_lib Command Reference" on page 126.

   % **$SYNOPSYS_SIM/sim/bin/create_smartmodel_lib** *arguments*

☞ **Note**

The bused wrappers enable improved performance but do not work with the examples testbench shipped with the model. To exercise the examples testbench, use the wrappers shipped with the model (see Table 18), as explained in the rest of this procedure. If you are using the bused wrappers, adjust accordingly.

2. Create a working directory and run flexm_setup to make copies of the model's interface and example files there, as shown in the following example:

   % **$LMC_HOME/bin/flexm_setup -dir** *workdir model*_**fx**

You must run flexm_setup every time you update your FlexModel installation with a new model version. Table 18 describes the FlexModel Scirocco interface and example files that the flexm_setup tool copies.

Table 18: FlexModel Scirocco VHDL Files

File Name	Description	Location
*model*_pkg.vhd	Model command procedure calls for HDL Command Mode.	*workdir*/src/vhdl/
*model*_user_pkg.vhd	Clock frequency setup and user customizations.	*workdir*/src/vhdl/
*model*_fx_vss.vhd	A SWIFT wrapper for the model.	*workdir*/examples/vhdl/
*model*_fx_comp.vhd	Component definition for use with the *model* entity defined in the SWIFT wrapper file. This is put in a package named "COMPONENTS" when compiled.	*workdir*/examples/vhdl/
model.vhd	A bus-level wrapper around the SWIFT model. This allows you to use vectored ports for the model in your testbench. This file assumes that the "COMPONENTS" package has been installed in the logical library "slm_lib".	*workdir*/examples/vhdl/

Table 18: FlexModel Scirocco VHDL Files (Continued)

File Name	Description	Location
*model*_tst.vhd	A testbench that instantiates the model and shows how to use basic model commands.	*workdir*/examples/vhdl/

3. Update the clock frequency supplied in the *model*_user_pkg.vhd file in your working directory to correspond to the desired clock period for the model. After you run flexm_setup this file is located in:

> *workdir*/**src/vhdl/***model*_**user_pkg.vhd**

where *workdir* is your working directory.

4. Add the following line to your .synopsys_vss.setup file:

```
SLM_LIB   :   SLM_LIB_PATH
TIMEBASE :   PS
```

5. Compile the FlexModel VHDL files into logical library slm_lib as follows:

```
% vhdlan -noevent -w slm_lib $LMC_HOME/sim/vhpi/src/slm_hdlc.vhd
% vhdlan -event -w slm_lib $LMC_HOME/sim/vhpi/src/flexmodel_pkg.vhd
% vhdlan -event -w slm_lib workdir/src/vhdl/model_user_pkg.vhd
% vhdlan -event -w slm_lib workdir/src/vhdl/model_pkg.vhd
% vhdlan -event -w slm_lib workdir/src/vhdl/model_fx_comp.vhd
% vhdlan -event -w slm_lib workdir/src/vhdl/model_fx_vss.vhd
% vhdlan -event -w slm_lib workdir/src/vhdl/model.vhd
```

6. Add LIBRARY and USE statements to your testbench:

```
library slm_lib;
use slm_lib.flexmodel_pkg.all;
use slm_lib.model_pkg.all;
use slm_lib.model_user_pkg.all;
```

For example, you would use the following statement for the tms320c6201_fx model:

```
use slm_lib.tms320c6201_pkg.all;
use slm_lib.tms320c6201_user_pkg.all;
```

7. Instantiate FlexModels in your design, defining the ports and generics as required (refer to the example testbench supplied with the model). You use the supplied bus-level wrapper (*model*.vhd) in the top-level of your design to instantiate the supplied bit-blasted wrapper (*model*_fx_vss.vhd).

   Example using bus-level wrapper (*model*.vhd) without timing:

   ```
   U1: model
       generic map (FlexModelID => "TMS_INST1")
       port map ( model ports );
   ```

   Example using bus-level wrapper (*model*.vhd) with timing:

   ```
   U1: model
       generic map (FlexModelID    => "TMS_INST1",
       FlexTimingMode => FLEX_TIMING_MODE_ON,
       TimingVersion  => "timingversion",
       DelayRange     => "range")
        port map ( model ports );
   ```

8. Compile your testbench as shown in the following example:

   ```
   % vhdlan testbench
   ```

9. Invoke the Scirocco simulator as shown in the following example:

   ```
   % scsim -vhpi slm_vhpi:foreignINITelab:cpipe design
   ```

Script for Running FlexModel Examples in Scirocco

On is a Perl script (Figure 6) that you can use to run Scirocco on a FlexModel examples testbench. You can use this script on any installed FlexModel because each one comes with a prebuilt testbench example that shows how to use the model commands and all the VHDL wrapper and procedure definition files that you need. This script runs on HP-UX and Solaris.

To invoke Scirocco on a FlexModel and its example testbench, follow these steps:

1. Make a working directory and set the TMPDIR variable to that location as shown in the following example:

   ```
   % setenv TMPDIR path_to_workdir
   ```

2. Use the Acrobat Reader's text selection tool to select the script shown in Figure 6 and copy the contents to a local file named run_flex_examples_in_scirocco.pl.

3. Save the file and change the permissions so that the file is executable (chmod 775 in UNIX).

4. Invoke the script as shown in the following example:

   ```
   % run_flex_examples_in_scirocco.pl model_fx
   ```

where *model_**fx*** is the name of the FlexModel you want to run.

☞ **Note** ──

This script was developed for internal use and is made available for customer convenience. It is not actively maintained as part of the licensed software.

```perl
#!/usr/local/bin/perl
# $Revision: 1.4 $
die "\nERROR running $0: ", "No FlexModel name given\n\n", unless( $ARGV[0] );
$TmpDir = $ENV{ TMPDIR };die "ERROR running $0: ", "The TMPDIR environment variable must be set.\n"
    unless( $TmpDir );
$LmcHome = $ENV{ LMC_HOME };
die "ERROR running $0: ", "The LMC_HOME environment variable must be set.\n" unless( $LmcHome );
$SynopsysHome = $ENV{ SYNOPSYS_SIM };
die "ERROR running $0: ", "The SYNOPSYS_SIM environment variable must be set.\n" unless( $SynopsysHome );
require "$LmcHome/lib/bin/libmdl01003.pl";$Platform = GetPlatform();
$Platform_lib = PlatformToLibDir($Platform);
$flexmodel_name = $ARGV[0];$DirSep ="/";
$output_file = $TmpDir . $DirSep . "Example_Simulator_Run_Script_" . $flexmodel_name . ".txt";
$model_path = $LmcHome . "/models/" . $flexmodel_name;
if ( -e $model_path ) { }
else { die "\nERROR running $0: ", "FlexModel $flexmodel_name Does not Exist in Library\n\n"; }
$version_path = `$LmcHome/bin/flexm_setup $flexmodel_name`;
chomp($version_path);
if ( $flexmodel_name =~ /_fx/ ) { $flexmodel_name =~ s/_fx//g; $flex_or_c = "_fx";}elsif ($flexmodel_name =~ /_fz/
) { $flexmodel_name =~ s/_fz//g; $flex_or_c = "_fz";}else {
    die "\nERROR running $0: ", "$flexmodel_name is not a FlexModel. Model must have an _fx or _fz to be a
FlexModel\n\n";}
$setup_file = $TmpDir . $DirSep . ".synopsys_vss.setup";
open(OFILE,"> $setup_file") || die " Could not create file : $setup_file\n";
print OFILE ("-- Example Scirocco library setup file\n");
print OFILE ("work > DEFAULT\n");print OFILE ("DEFAULT : work\n");print OFILE ("SLM_LIB : slm_lib\n");print OFILE
("TIMEBASE = PS\n");print OFILE ("\n");close(OFILE);
$work_dir = $TmpDir . $DirSep . "work";$slm_lib_dir = $TmpDir . $DirSep . "slm_lib";
mkdir($work_dir, 0777);mkdir($slm_lib_dir, 0777);
if ( $Platform eq "pcnt" ) { $source_setup_file = "cmd /c " . $SynopsysHome . "/admin/setup/environ.bat"} else
{$source_setup_file = "source " . $SynopsysHome . "/admin/setup/environ.csh"}
$analyze_slm_hdlc = "vhdlan -noevent -w slm_lib " . $LmcHome . "/sim/vhpi/src/slm_hdlc.vhd";
$analyze_flexmodel_pkg = "vhdlan -event -w slm_lib " . $LmcHome . "/sim/vss/src/flexmodel_pkg.vhd";
$analyze_model_user_pkg = "vhdlan -event -w slm_lib " . $version_path . "/src/vhdl/" . $flexmodel_name .
"_user_pkg.vhd";
$analyze_model_pkg = "vhdlan -event -w slm_lib " . $version_path . "/src/vhdl/" . $flexmodel_name . "_pkg.vhd";
$analyze_model_prim = "vhdlan -event -w slm_lib " . $version_path . "/examples/vhdl/" . $flexmodel_name .
$flex_or_c . "_vss.vhd";
$analyze_model_comp = "vhdlan -event -w slm_lib " . $version_path . "/examples/vhdl/" . $flexmodel_name .
$flex_or_c . "_comp.vhd";
$analyze_model = "vhdlan -event -w slm_lib " . $version_path . "/examples/vhdl/" . $flexmodel_name . ".vhd";
$analyze_model_tst = "vhdlan -event " . $version_path . "/examples/vhdl/" . $flexmodel_name . "_tst.vhd";
if ( $Platform eq "pcnt" ) {$run_file = $TmpDir . $DirSep . "run_scirocco_" . $flexmodel_name . ".cmd";}else {
$run_file = $TmpDir . $DirSep . "run_scirocco_" . $flexmodel_name;}
open(OFILE,"> $run_file") || die " Could not create file : $run_file\n";
if ( $Platform eq "pcnt" ) { } else { print OFILE ("#!/bin/csh -f\n");}
print OFILE ("$source_setup_file\n");print OFILE ("$analyze_slm_hdlc\n");print OFILE
("$analyze_flexmodel_pkg\n");print OFILE ("$analyze_model_user_pkg\n");print OFILE ("$analyze_model_pkg\n");print
OFILE ("$analyze_model_prim\n");print OFILE ("$analyze_model_comp\n");print OFILE ("$analyze_model\n");print OFILE
("$analyze_model_tst\n");
$virsim_command = "virsim +simtype+Scirocco +simargs+\" -vhpi slm_vhpi:foreignINITelab:cpipe cfgtest\"
";print OFILE ("$virsim_command\n");
close(OFILE);if ( $Platform eq "pcnt" ) { } else { chmod(0777, $run_file);}
if ( $Platform eq "pcnt" ) { #$run_it = "wstart /d $TmpDir "; # Means nothing until Scirocco is on PCNT $run_it =
""; $run_it = $run_it . $run_file;}
else { $run_it = "cd " . $TmpDir . " ; "; $run_it = $run_it . $run_file;}
print("\n $run_it \n");system($run_it);
```

Figure 6: run_flex_examples_in_scirocco.pl Script

Example Simulator Run Script

The run_flex_examples_in_scirocco.pl script also creates an example C Shell simulator run script for the specified model in your $TMPDIR directory. You can use this run script to invoke Scirocco after running the run_flex_examples_in_scirocco.pl script. Here is what the script looks like for the mpc860_fx model.

```
#!/bin/csh -f
source /d/edavs/snps-2000.02PROD/admin/setup/environ.csh
vhdlan –noevent -w slm_lib /d/lmgqa2/dalew/flex/lmc_home/sim/vhpi/src/slm_hdlc.vhd
vhdlan -event -w slm_lib /d/lmgqa2/dalew/flex/lmc_home/sim/vss/src/flexmodel_pkg.vhd
vhdlan -event -w slm_lib
/d/lmgqa2/dalew/flex/lmc_home/models/mpc860_fx/mpc860_fx02012/src/vhdl/mpc860_user_pkg.v
hd
vhdlan -event -w slm_lib
/d/lmgqa2/dalew/flex/lmc_home/models/mpc860_fx/mpc860_fx02012/src/vhdl/mpc860_pkg.vhd
vhdlan -event -w slm_lib
/d/lmgqa2/dalew/flex/lmc_home/models/mpc860_fx/mpc860_fx02012/examples/vhdl/mpc860_fx_vs
s.vhd
vhdlan -event -w slm_lib
/d/lmgqa2/dalew/flex/lmc_home/models/mpc860_fx/mpc860_fx02012/examples/vhdl/mpc860_fx_co
mp.vhd
vhdlan -event -w slm_lib
/d/lmgqa2/dalew/flex/lmc_home/models/mpc860_fx/mpc860_fx02012/examples/vhdl/mpc860.vhd
vhdlan -event
/d/lmgqa2/dalew/flex/lmc_home/models/mpc860_fx/mpc860_fx02012/examples/vhdl/mpc860_tst.v
hd
virsim +simtype+Scirocco +simargs+" –vhpi slm_vhpi:foreignINITelab:cpipe cfgtest"
```

Using MemPro Models with Scirocco

You must have MemPro version 2000.04 or higher to use MemPro models with Scirocco. To use Scirocco with MemPro models, follow this procedure.

1. Add the Scirocco library path to your library path environment variable.

   HP-UX:

   ```
   % setenv SHLIB_PATH \
       $SYNOPSYS_SIM/hpux10/sim/lib:$SHLIB_PATH
   ```

   Solaris:

   ```
   % setenv LD_LIBRARY_PATH \
       $SYNOPSYS_SIM/sparcOS5/sim/lib:$LD_LIBRARY_PATH
   ```

2. Add the Scirocco executable to your search path:

   ```
   % set path = ($SYNOPSYS_SIM/platform/sim/bin $path)
   ```

   where *platform* is hpux10 or sparcOS5.

3. Create a logical library named slm_lib. The default physical library mapping for this is $LMC_HOME/sim/vhpi/lib. If you do not wish to install the library in the LMC_HOME tree, you can create it in any other location.

4. Add the following line to your .synopsys_vss.setup file:

```
SLM_LIB   :    $LMC_HOME/sim/vhpi/lib
TIMEBASE :   PS
```

5. Compile the MemPro VHDL files into logical library slm_lib. Note that rdramd_pkg.vhd is not required unless you are simulating RDRAM MemPro models.

```
% vhdlan -noevent -w slm_lib $LMC_HOME/sim/vhpi/src/slm_hdlc.vhd
% vhdlan -event -w slm_lib $LMC_HOME/sim/vhpi/src/mempro_pkg.vhd
% vhdlan -event -w slm_lib $LMC_HOME/sim/vhpi/src/rdramd_pkg.vhd
```

6. After generating a model using MemPro, compile the VHDL code for the model, as shown in the following example:

```
% vhdlan mymem.vhd
```

7. If you compiled the model to a library (SLM_MODELS in these examples), add a LIBRARY statement to your testbench:

```
LIBRARY SLM_MODELS
```

This makes models in SLM_MODELS accessible from your design. You can refer to the model using a standard VHDL convention, such as a CONFIGURATION statement:

```
CONFIGURATION mymem_a OF mymem_tst IS
    FOR atest
          FOR ALL : mymem use CONFIGURATION SLM_MODELS.mymem_behav;
          END FOR;
    END FOR;
END mymem_a;
```

8. Add LIBRARY and USE statements to your testbench:

```
LIBRARY SLM_LIB
USE SLM_LIB.mempro_pkg.all
```

This also provides access to MemPro testbench commands.

For more information on using the MemPro testbench interfaces, refer to the "HDL Testbench Interface" and "C Testbench Interface" chapters in the *MemPro User's Manual*.

9. Instantiate MemPro models in your design. Define ports and generics as required. For information on generics used with MemPro models, refer to "Instantiating MemPro Models" on page 32. For information on message levels and message level constants, refer to "Controlling MemPro Model Messages" on page 33.

10. Compile your design as shown in the following example

```
% vhdlan design_name
```

11. Invoke Scirocco on your design, making sure to include all libraries, as shown in the following example:

```
% scsim -debug_all -vhpi slm_vhpi:foreignINITelab:cpipe design_name
```

☞ **Note**

The -debug_all switch is only needed on Scirocco release 2000.02.

Using MemPro Models in a Testbench

To use a MemPro model in a testbench, follow this procedure. If you already created scsim when using a MemPro model or FlexModel, skip to Step 4.

1. Create a MemPro VHDL model using MemSpec. Generate code for Cyclone or VSS.

2. Instantiate the MemPro component in your testbench.

3. Set your UNIX search path so that it points to the new vhdlsim executable you just created.

4. Run the following commands to compile the required MemPro VHDL packages:

```
% mkdir slm_lib MY_WORK
% vhdlan -noevent -w slm_lib $LMC_HOME/sim/vhpi/src/slm_hdlc.vhd
% vhdlan -event -w slm_lib $LMC_HOME/sim/vhpi/src/mempro_pkg.vhd
```

5. Compile the model and testbench that you just created:

```
% vhdlan -event -w MY_WORK upd4516161.vhd
% vhdlan -event -w MY_WORK upd_testbench.vhd
```

6. Run the simulation:

```
% ./scsim -vhpi slm_vhpi:ForeignINITelab:cpipe your_testbench
```

Using Hardware Models with Scirocco

To use Scirocco with hardware models, follow this procedure. Note that your design can include a mix of event-based and cycle-based, but hardware models simulate only as event-based.

1. Make sure Scirocco is set up properly and all required environment variables are set, as explained in "Setting Environment Variables" on page 124.

2. Add the hardware model install tree to your path variable, as shown in the following example:

   ```
   % set path=(/install/sms/bin/your_platform/ $path)
   ```

3. Create the *model*.vhd wrapper file for your hardware model. You can use the nawk script provided in "Scirocco Template Generator Script for Hardware Models" on page 138 to generate this file. Copy the script and paste it into an executable file called hwm2vhdl.nawk.

4. If you generate the wrapper by hand, you must provide:

   ❍ an entity-architecture pair declaration so Scirocco can reference it in a later component instantiation statement.

   ❍ a package for defining constants, declaring components, and instantiating components.

Scirocco Example with TILS299 Hardware Model

The following example uses the TILS299 hardware model to show how to set up hardware models for use with Scirocco:

1. After creating the wrapper .vhd file, analyze the file using vhdlan, as shown in the following example:

   ```
   % vhldan -event TILS299.vhd
   ```

2. Place the hardware model in the testbench file and invoke the simulator, as follows:

   ```
   % scsim -t ns TB_TILS299
   ```

   The ns argument invokes the simulator with nanosecond time steps.

⏰ **Attention**──

When using hardware models with Scirocco, your design can include a mix of event-based and cycle-based, but hardware models simulate only as event-based.

──

Scirocco Utilities

The following hardware modeler simulator commands are supported in Scirocco.

#lmsi list devices | ids

You can use the lmsi list devices command to list all hardware model instances by device name, and the lmsi list ids command to list all hardware model instances by id name. For example:

```
# lmsi list devices
device name            id#    instance name          logging
TILS299                0      /TB_TILS299/U0         Off
# lmsi list ids
id#    device name            instance name          logging
0      TILS299                /TB_TILS299/U0         Off
```

You can also log test vectors for the hardware model. To log by ID number, specify an *id#* and a *filename*. The extension .TST is appended to the vector file name. If no file name is specified, VSS writes to a file named *device_name.id#*.TST. For example:

#lmsi logon *id# filename*

To log vectors by instance name, specify an *instance_name* and *filename*. The extension .TST is appended to the output file name. For example:

#lmsi logon *instance_name filename*

To log vectors for all hardware model device instances, specify **all**. A log file is created for each instance. The output files are named *device_name.id#*.TST. For example:

#lmsi logon all

To turn off vector logging, replace logon with logoff and omit the file name in the above examples.

VHDL Model Generics with Scirocco

You can also control hardware model behavior using VHDL generics in your hardware model instantiations. The nawk script on creates VHDL wrappers for hardware models with these VHDL generics set to values that are reasonable for most simulations. However, you can modify the values of the VHDL generics in your *model*.vhd files to suit your verification needs. For more information on supported VHDL generics, refer to the Synopsys *VHDL Simulation Interfaces Manual*. Following are descriptions of some of the most useful generics:

LMSI_TIMING_MEASUREMENT

You can use the LMSI_TIMING_MEASUREMENT generic to direct where timing values for your simulation session come from. There are two legal values:

ENABLED   The hardware modeler measures and records actual pin-to-pin timing values and passes them to the simulator.

DISABLED   The hardware modelers passes to the simulator the pin-to-pin timing values from the .TMG file. This is the default value.

LMSI_DELAY_TYPE

You can use the LMSI_DELAY_TYPE generic to specify whether the hardware modeler returns pin values to the simulator with minimum, typical, or maximum delays, as you can see in the following legal values:

MINIMUM   Return minimum delays for pin values to the simulator.

TYPICAL   Return typical delays for pin values to the simulator. This is the default.

MAXIMUM   Return maximum delays for pin values to the simulator.

LMSI_LOG

You can use the LMSI_LOG generic to specify whether the hardware modeler logs test vector or not. There are two legal values:

ENABLED   The hardware modeler logs test vectors.

DISABLED   The hardware modelers does not log test vectors. This is the default value.

Scirocco Template Generator Script for Hardware Models

Here is the nawk script that you can use to generate VHDL wrappers for the hardware models. Because of the length this script, you will have to cut-and-paste one page at a time from this PDF file to get the whole thing copied to your environment.

```
*********************************************
#  In your design directory type:
#
#  nawk -f hwm2vhdl.nawk $HWM/<model>.NAM > <outfile>.vhd
#
#  (where "$HWM" is the full path to your hardware modeling directory)
#  Instantiate .vhd into your design.
#
# THE SCRIPT:
#
# Script to generate a VSS/Scirocco VHDL shell for a hardware model
# using the <model>.NAM file

BEGIN {
   pin_type = 0
   is_it_a_vector = "No"
   data_type = ""
   prev_signal = ""
   prev_test = ""
   prev_number = ""
   prev_dir = ""
   ending = ";"

   printf "library SYNOPSYS;\n"
   printf "   use SYNOPSYS.ATTRIBUTES.all;\n"
   printf "library IEEE;\n"
   printf "   use IEEE.std_logic_1164.all;\n\n"

}

$2 ~ /generic_device_name/ {
   device = $3
   printf "entity " device " is\n"
   printf "   generic\n"
   printf "      (\n"
   printf "      timing : LMSI_TIMING_MEASUREMENT          := DISABLED;\n"
   printf "      delay_type : LMSI_DELAY_TYPE              := TYPICAL;\n"
   printf "      delay : LMSI_DELAY                        := ENABLED;\n"
   printf "      log : LMSI_LOG                            := DISABLED;\n"
   printf "      timing_violations : LMSI_TIMING_VIOLATIONS := DISABLED;\n"
   printf "      xprop : LMSI_XPROP                        := DISABLED;\n"
   printf "      xprop_method : LMSI_XPROP_METHOD          := HIGH\n"
   printf "      );\n\n"
   printf "   port\n"
   printf "      (\n"
}

$4 ~ /\(in_pin\)/ || $4 ~ /\(out_pin\)/ || $4 ~ /\(io_pin\)/ \
            || $4 ~ /\(power_pin\)/ {
   pin_type++
}

$2 ~ /\=/ || ($0 ~ /^$/ && pin_type ~ /3/) {
   if (pin_type == 1) {
      direction = "in   "
   }
   else if (pin_type == 2) {
      direction = "out  "
   }
   else if (pin_type == 3) {
      direction = "inout"
   }
   else {
      next
   }
   current_signal = $1 " "
   gsub(/\{/, "", current_signal)
   gsub(/\'/, "", current_signal)

   current_test = current_signal
```

```
        gsub(/[0-9]+ /, " ", current_test)

        n = split(current_signal, array_a, "[a-zA-Z]")
        current_number = array_a[n]
        gsub(/ /, "", current_number)

        if (prev_signal ~ /[0-9]+ /) {
            if (current_test == prev_test)  {
                if (is_it_a_vector == "No") {
                    data_start = prev_number
                }
                if ((current_number == prev_number - 1) || (current_number == prev_number + 1))
                {
                    is_it_a_vector = "Yes"
                }
                prev_signal = current_signal
                prev_test = current_test
                prev_number = current_number
                next
            }
            else {
                if (is_it_a_vector == "Yes") {
                    total = prev_number + data_start
                    if (prev_number > data_start) {
                        data_end = data_start
                        data_start = prev_number
                    }
                    else {
                        data_end = prev_number
                    }
                    data_type = "_vector (" data_start " " "downto " data_end ")"
                    prev_signal = prev_test
                }
            }
        }
        if (prev_signal != "") {
            gsub(/ /, "", prev_signal)
            n = split(prev_signal, array_c, "[a-zA-z0-9_]")
            y = 20 - n
            if (y > 0) {
                for (i = 1; i <= 20-n; i++) {
                    prev_signal = prev_signal " "
                }
            }
            if (($0 ~ /^$/) && (pin_type == 3)) {
                ending = ""
            }
            printf "      " prev_signal " : " prev_dir " std_logic" data_type ending "\n"
        }
        data_type = ""
        is_it_a_vector = "No"
        updown = ""
        prev_signal = current_signal
        prev_test = current_test
        prev_dir = direction
        prev_number = current_number
    }

    END {
        printf "      );\n"
        printf "end " device ";\n\n"
        printf "architecture LMSI of " device " is\n"
        printf "    attribute FOREIGN of LMSI : architecture is \"Synopsys:LMSI\";\n"
        printf "    begin\n"
        printf "end LMSI;\n\n"
    }
```

7

Using VSS with Synopsys Models

Overview

This chapter explains how to use SmartModels, FlexModels, MemPro models, and hardware models with VSS. The procedures are organized into the following major sections:

Setting Environment Variables

First, set the basic environment variables. If you are not using one of the model types, skip that step. In some cases the procedures that follow in this chapter include steps for setting additional environment variables.

1. Set the LMC_HOME variable to the location of your SmartModel. FlexModel, and MemPro model installation tree, as follows:

   `% setenv LMC_HOME `*`path_to_models_installation`*

2. Set the SYNOPSYS variable to point to the VSS installation directory as follows:

   `% setenv SYNOPSYS `*`VSS_installation_directory`*

3. Source the environ.csh VSS environment file.

   For VSS version 1998.08-1 and earlier, use this path:

   `% source $SYNOPSYS/admin/install/sin/environ.csh`

For VSS version 1999.05 and later, use this path:

```
% source $SYNOPSYS/admin/setup/environ.csh
```

4. Set the LM_LICENSE_FILE or SNPSLMD_LICENSE_FILE environment variable to point to the product authorization file, as shown in the following example:

```
% setenv LM_LICENSE_FILE path_to_product_authorization_file
```

```
% setenv SNPSLMD_LICENSE_FILE path_to_product_authorization_file
```

You can put license keys for multiple products (for example, SmartModels and hardware models) into the same authorization file. If you need to keep separate authorization files for different products, use a colon-separated list (UNIX) or semicolon-separated list (NT) to specify the search path in your variable setting.

Caution

Do not include la_dmon-based authorizations in the same file with snpslmd-based authorizations. If you have authorizations that use la_dmon, keep them in a separate license file that uses a different license server (lmgrd) process than the one you use for snpslmd-based authorizations.

5. If you are using the hardware modeler, set the LM_DIR and LM_LIB environment variables, as shown in the following examples:

```
% setenv LM_DIR hardware_model_install_path/sms/lm_dir
% setenv LM_LIB hardware_model_install_path/sms/models: \
 hardware_model_install_path/sms/maps
```

If you put your models in a directory other than the default of /sms/models, modify the above variable setting accordingly.

6. Depending on your platform, set your load library variable to point to the platform-specific directory in $LMC_HOME, as shown in the following examples:

Solaris:

```
% setenv LD_LIBRARY_PATH $LMC_HOME/lib/sun4Solaris.lib:$LD_LIBRARY_PATH
```

Linux:

```
% setenv LD_LIBRARY_PATH $LMC_HOME/lib/x86_linux.lib:$LD_LIBRARY_PATH
```

AIX:

```
% setenv LIBPATH $LMC_HOME/lib/ibmrs.lib:$LIBPATH
```

HP-UX:

```
% setenv SHLIB_PATH $LMC_HOME/lib/hp700.lib:$SHLIB_PATH
```

NT:

Make sure that %LMC_HOME%\lib\pcnt.lib is in the Path user variable.

Using SmartModels with VSS

To use SmartModels with VSS, follow this procedure:

1. To create SmartModel VHDL templates, check to see if you have write permission for $LMC_HOME/synopsys/smartmodel; if so skip to Step 3. Otherwise, open the .synopsys_vss.setup file in your current working directory and search for the string SMARTMODEL. By default, the logical library name SMARTMODEL is mapped to $LMC_HOME/synopsys/smartmodel, as follows:

   ```
   SMARTMODEL : $LMC_HOME/synopsys/smartmodel
   ```

2. Change the directory to one for which you have write permission, as in the following example:

   ```
   SMARTMODEL : ~/smartmodel
   ```

3. To generate VHDL model wrapper files, invoke create_smartmodel_lib with any optional arguments. For information on the syntax for this command, refer to "create_smartmodel_lib Command Reference" on page 144.

   ```
   % $SYNOPSYS_SIM/sim/bin/create_smartmodel_lib arguments
   ```

4. If you changed the SMARTMODEL mapping in Step 3, you must use the -srcdir option to specify that directory. Also, you can save time by using the -model or -modelfile option to specify the models you want. Otherwise, the script processes all installed SmartModels. For example, here is a recommended set of options to use for one SmartModel (ttl00 in this example).:

   ```
   % $SYNOPSYS_SIM/sim/bin/create_smartmodel_lib –model ttl00 \
   –srcdir ~/smartmodel
   ```

5. After create_smartmodel_lib has finished executing, verify that the VHDL template files have been created in the appropriate directory.

6. To use SmartModels in the VHDL source file of your design, specify the SMARTMODEL library and instantiate each SmartModel component. In the VHDL design file that uses SmartModel components, enter the following library and use clauses:

   ```
   library SMARTMODEL;
   use SMARTMODEL.components.all
   ```

   The library logical name SMARTMODEL must be mapped to appropriate directories in your .synopsys_vss.setup file.

7. Add the following line to your .synopsys_vss.setup file:

   ```
   TIMEBASE :   PS
   ```

8. Instantiate SmartModels in your VHDL design. For information on required configuration parameters and instantiation examples, refer to "Using SmartModels with SWIFT Simulators" on page 18.

9. Compile your testbench as shown in the following example:

   ```
   % vhdlan testbench
   ```

10. Invoke the VSS simulator as shown in the following example:

   ```
   % vhdlsim design
   ```

   For information about vhdlsim and the VHDL debugger, refer to the *VSS User's Guide*.

create_smartmodel_lib Command Reference

The command reference for create_smartmodel_lib is as follows.

Syntax

create_smartmodel_lib [--] [-nc] [-create] [-srcdir *dirpath*] [-analyze] [-nowarn] [-modelfile *file*] {-model *model_name*}

Arguments

--	Displays the usage and all the command line options of the utility.
-nc	Suppresses the Synopsys copyright message.
-create	Creates the VHDL source files (.vhd files) for the SMARTMODEL library and saves the source files in the $LMC_HOME/synopsys directory.
-src_dir *dir*	Lets you specify the location of the VHDL source files that you create. The default location is $LMC_HOME/synopsys.
-analyze	Analyzes the SMARTMODEL library source files (.vhd files) by invoking vhdlan. The analyzed files (.sim and .mra files) are saved in the $LMC_HOME/synopsys/smartmodel directory. This directory is specified by the SMARTMODEL logical name mapping in the setup file.
-nowarn	Suppresses the generation of warning messages that notify you of any port name mappings. See "VHDL Reserved Port and Window Names" in the *VSS Expert Interface Manual* for more information about port name mappings.

-modelfile *file*	A list of SMARTMODEL component names is read from *file*. Names are separated by spaces. Only those component names specified are included in the SMARTMODEL component library.
-model *modelname*	Each specified *modelname* is included in the resulting SMARTMODEL component library. Repeat this option to specify multiple models. Only those component names specified are included in the SMARTMODEL component library.

Description

The create_smartmodel_lib command, if issued without options, uses as input all of the files in the $LMC_HOME/models directory, creates and analyzes the template files, and saves them in the $LMC_HOME/synopsys/smartmodel directory. If you do not have write permission for $LMC_HOME/synopsys/smartmodel, the command terminates with an error message. In that case, you must use the -src_dir option to specify a writable directory in which to place the VHDL templates. You must also specify that directory through the SMARTMODEL library mapping in the .synopsys_vss.setup file in your current working directory.

Using FlexModels with VSS

To use FlexModels with VSS in UNIX, follow this procedure. There is no custom integration for VSS on NT, but you can use Direct C Control. For information on using Direct C Control, refer to "Instantiating FlexModels with Direct C Control" on page 26.

1. If you want the improved performance that comes with bused wrappers, you can generate VHDL model wrapper files by invoking create_smartmodel_lib with any optional arguments. For more information on the syntax for this command, refer to "create_smartmodel_lib Command Reference" on page 144.

   % **$SYNOPSYS_SIM/sim/bin/create_smartmodel_lib** *arguments*

☞ **Note**
The bused wrappers enable improved performance but do not work with the examples testbench shipped with the model. To exercise the examples testbench, use the wrappers shipped with the model (see Table 19), as explained in the rest of this procedure. If you are using the bused wrappers, adjust accordingly.

2. Create a working directory and run flexm_setup to make copies of the model's interface and example files there, as shown in the following example:

```
% $LMC_HOME/bin/flexm_setup -dir workdir  model_fx
```

You must run flexm_setup every time you update your FlexModel installation with a new model version. Table 19 describes the FlexModel VSS interface and example files that the flexm_setup tool copies.

Table 19: FlexModel VSS VHDL Files

File Name	Description	Location
*model*_pkg.vhd	Model command procedure calls for HDL Command Mode.	*workdir*/src/vhdl/
*model*_user_pkg.vhd	Clock frequency setup and user customizations.	*workdir*/src/vhdl/
*model*_fx_vss.vhd	A SWIFT wrapper for the model.	*workdir*/examples/vhdl/
*model*_fx_comp.vhd	Component definition for use with the *model* entity defined in the SWIFT wrapper file. This is put in a package named "COMPONENTS" when compiled.	*workdir*/examples/vhdl/
model.vhd	A bus-level wrapper around the SWIFT model. This allows you to use vectored ports for the model in your testbench. This file assumes that the "COMPONENTS" package has been installed in the logical library "slm_lib".	*workdir*/examples/vhdl/
*model*_tst.vhd	A testbench that instantiates the model and shows how to use basic model commands.	*workdir*/examples/vhdl/

3. Update the clock frequency supplied in the *model*_user_pkg.vhd file in your working directory to correspond to the desired clock period for the model. After running flexm_setup this file will be located in:

```
workdir/src/vhdl/model_user_pkg.vhd
```

where *workdir* is your working directory.

4. Compile a dummy module to force linking of CLI library functions, as shown in the following example:

```
% cp $LMC_HOME/sim/vss/src/vss_dummy_calls.c ./vss_dummy_calls.c
% cli -ansi -s -add -cf vss_dummy_calls.c vss_dummy_calls
```

5. Link the FlexModel binary into the vhdlsim simulation executable:

```
% cli -ansi -s -build -libs $LMC_HOME/lib/platform.lib/slm_vss.o
```

where *platform* is hp700 or sun4Solaris.

The new version of vhdlsim you just created must be used when you simulate a design that includes FlexModels. This vhdlsim must be defined as the first vhdlsim in your UNIX search path.

6. Add the following line to your .synopsys_vss.setup file:

```
SLM_LIB   :    SLM_LIB_PATH
TIMEBASE :    PS
```

7. Compile the FlexModel VHDL files into logical library slm_lib as follows:

```
% vhdlan -c -w slm_lib $LMC_HOME/sim/vss/src/slm_hdlc.vhd
% vhdlan -c -w slm_lib $LMC_HOME/sim/vss/src/flexmodel_pkg.vhd
% vhdlan -c -w slm_lib workdir/src/vhdl/model_user_pkg.vhd
% vhdlan -c -w slm_lib workdir/src/vhdl/model_pkg.vhd
% vhdlan -c -w slm_lib workdir/src/vhdl/model_fx_comp.vhd
% vhdlan -c -w slm_lib workdir/src/vhdl/model_fx_vss.vhd
% vhdlan -c -w slm_lib workdir/src/vhdl/model.vhd
```

8. Add LIBRARY and USE statements to your testbench:

```
library slm_lib;
use slm_lib.flexmodel_pkg.all;
use slm_lib.model_pkg.all;
use slm_lib.model_user_pkg.all;
```

For example, you would use the following statement for the tms320c6201_fx model:

```
use slm_lib.tms320c6201_pkg.all;
use slm_lib.tms320c6201_user_pkg.all;
```

9. Instantiate FlexModels in your design, defining the ports and generics as required (refer to the example testbench supplied with the model). You use the supplied bus-level wrapper (*model*.vhd) in the top-level of your design to instantiate the supplied bit-blasted wrapper (*model*_fx_vss.vhd).

Example using bus-level wrapper (*model*.vhd) without timing:

```
U1: model
    generic map (FlexModelID => "TMS_INST1")
    port map ( model ports );
```

Example using bus-level wrapper (*model*.vhd) with timing:

```
U1: model
    generic map (FlexModelID    => "TMS_INST1",
    FlexTimingMode => FLEX_TIMING_MODE_ON,
    TimingVersion  => "timingversion",
    DelayRange     => "range")
    port map ( model ports );
```

10. Compile your testbench as shown in the following example:

```
% vhdlan testbench
```

11. Invoke the VSS simulator as shown in the following example:

```
% vhdlsim design
```

Using MemPro Models with VSS

To use MemPro models with VSS, follow this procedure. Note that on Solaris, VSS requires the Sunsoft compiler and Solaris 2.5 or later.

1. Compile a dummy module to force linking of CLI library functions:

```
% cli -ansi -s -add -cf \
    $LMC_HOME/sim/vss/src/vss_dummy_calls.c vss_dummy_calls
```

2. Link the MemPro binary into the vhdlsim simulation executable:

```
% cli -ansi -s -build -libs $LMC_HOME/lib/platform.lib/slm_vss.o
```

where *platform* is hp700 or sun4Solaris.

The new version of vhdlsim you just created must be used when you simulate a design that includes MemPro memory models. In order to use vhdldbx on a design that includes MemPro models, the vhdlsim you just created must be defined as the first vhdlsim in your UNIX search path.

3. For Solaris, set the LD_LIBRARY_DATA environment variable as follows:

```
% setenv LD_LIBRARY_DATA $SYNOPSYS/sparcOS5/sim/lib
```

4. Create a logical library named slm_lib. The default physical library mapping for this is $LMC_HOME/sim/vss/lib; however, you can create slm_lib any where you want.

Add the following line to your .synopsys_vss.setup file:

```
SLM_LIB   :    $LMC_HOME/sim/vss/lib
TIMEBASE :    PS
```

5. Compile the MemPro VHDL files into logical library slm_lib. Note that rdramd_pkg.vhd is not required unless you are simulating RDRAM MemPro models.

```
% vhdlan -c -w slm_lib $LMC_HOME/sim/vss/src/slm_hdlc.vhd
% vhdlan -c -w slm_lib $LMC_HOME/sim/vss/src/mempro_pkg.vhd
% vhdlan -c -w slm_lib $LMC_HOME/sim/vss/src/rdramd_pkg.vhd
```

☞ **Note**

The vhdlan program returns an "Error compiling file" warning message for rdramd_pkg.vhd and reverts to interpreted code for the file. Your designs containing MemPro RDRAMs will simulate properly, however.

6. After generating a model using MemPro, compile the VHDL code for the model, as shown in the following example:

```
% vhdlan -c mymem.vhd
```

7. If you compiled the model to a library (SLM_MODELS in these examples), add a LIBRARY statement to your testbench:

```
LIBRARY SLM_MODELS
```

This makes models in SLM_MODELS accessible from your design. You can refer to the model using a standard VHDL convention, such as a CONFIGURATION statement:

```
CONFIGURATION mymem_a OF mymem_tst IS
    FOR atest
            FOR ALL : mymem use CONFIGURATION SLM_MODELS.mymem_behav;
            END FOR;
    END FOR;
END mymem_a;
```

8. Add LIBRARY and USE statements to your testbench:

```
LIBRARY SLM_LIB
USE SLM_LIB.mempro_pkg.all
```

This also provides access to MemPro testbench commands.

For more information on using the MemPro testbench interfaces, refer to the "HDL Testbench Interface" and "C Testbench Interface" chapters in the *MemPro User's Manual*.

9. Instantiate MemPro models in your design. Define ports and generics as required. For information on generics used with MemPro models, refer to "Instantiating MemPro Models" on page 32. For information on message levels and message level constants, refer to "Controlling MemPro Model Messages" on page 33.

10. Compile your design as shown in the following example:

    ```
    % vhdlan design_name
    ```

11. Invoke VSS on your design as shown in the following example:

    ```
    % vhdlsim design_name
    ```

Using Hardware Models with VSS

To use hardware models with VSS, follow this procedure:

1. Make sure VSS is set up properly and all required environment variables are set, as explained in "Setting Environment Variables" on page 141. Also, make sure you have the VSS-LMSI key in your license file for the interface licensing.

2. Add the hardware model install tree to your path variable, as shown in the following example:

   ```
   % set path=(/install/sms/bin/your_platform/ $path)
   ```

3. Create the *model*.vhd wrapper file for your hardware model. You can use the nawk script provided in "VSS Template Generator Script for Hardware Models" on page 153 to generate this file. Copy the script and paste it into an executable file called hwm2vhdl.nawk.

4. If you generate the wrapper by hand, you must provide:

   ❍ an entity-architecture pair declaration so VSS can reference it in a later component instantiation statement.

   ❍ a package for defining constants, declaring components, and instantiating components.

VSS Example with TILS299 Hardware Model

The following example uses the TILS299 hardware model to show how to set up hardware models for use with VSS:

1. After creating the wrapper .vhd file, analyze the TILS299.vhd using vhdlan, as shown in the following example:

   ```
   % vhldan TILS299.vhd
   ```

2. Place the hardware model in the testbench file and invoke the simulator. For this TILS299 example, we used the Synopsys VHDL Debugger, as follows:

   ```
   % vhdldbx -t ns TB_TILS299
   ```

   The ns argument invokes the simulator with nanosecond timesteps.

VSS Utilities

The following hardware modeler simulator commands are supported in VSS.

lmsi list devices | ids

You can use the lmsi list devices command to list all hardware model instances by device name, and the lmsi list ids command to list all hardware model instances by id name. For example:

```
# lmsi list devices
device name            id#    instance name          logging
TILS299                0      /TB_TILS299/U0         Off
# lmsi list ids
id#    device name            instance name          logging
0      TILS299                /TB_TILS299/U0         Off
```

You can also log test vectors for the hardware model. To log by ID number, specify an *id#* and a *filename*. The extension .TST is appended to the vector file name. If no file name is specified, VSS writes to a file named *device_name.id#*.TST. For example:

#lmsi logon *id# filename*

To log vectors by instance name, specify an *instance_name* and *filename*. The extension .TST is appended to the output file name. For example:

#lmsi logon *instance_name filename*

To log vectors for all hardware model device instances, specify **all**. A log file is created for each instance. The output files are named *device_name.id#*.TST. For example:

#lmsi logon all

To turn off vector logging, replace logon with logoff and omit the *filename* in the above examples.

VHDL Model Generics with VSS

You can also control hardware model behavior using VHDL generics in your hardware model instantiations. The nawk script on page 153 creates VHDL wrappers for hardware models with these VHDL generics set to values that are reasonable for most simulations. However, you can modify the values of the VHDL generics in your *model*.vhd files to suit your verification needs. For more information on supported VHDL generics, refer to the Synopsys *VHDL Simulation Interfaces Manual*. Following are descriptions of some of the most useful generics:

LMSI_TIMING_MEASUREMENT

You can use the LMSI_TIMING_MEASUREMENT generic to direct where timing values for your simulation session come from. There are two legal values:

ENABLED   The hardware modeler measures and records actual pin-to-pin timing values and passes them to the simulator.

DISABLED   The hardware modeler passes to the simulator the pin-to-pin timing values from the .TMG file. This is the default value.

LMSI_DELAY_TYPE

You can use the LMSI_DELAY_TYPE generic to specify whether the hardware modeler returns pin values to the simulator with minimum, typical, or maximum delays, as you can see in the following legal values:

MINIMUM   Return minimum delays for pin values to the simulator.

TYPICAL   Return typical delays for pin values to the simulator. This is the default.

MAXIMUM   Return maximum delays for pin values to the simulator.

LMSI_LOG

You can use the LMSI_LOG generic to specify whether the hardware modeler logs test vector or not. There are two legal values:

ENABLED   The hardware modeler logs test vectors.

DISABLED   The hardware modelers does not log test vectors. This is the default value.

VSS Template Generator Script for Hardware Models

Here is the nawk script that you can use to generate VHDL wrappers for the hardware models. Because of the length this script, you will have to cut-and-paste one page at a time from this PDF file to get the whole thing copied to your environment.

```
*********************************************
#  In your design directory type:
#
#  nawk -f hwm2vhdl.nawk $HWM/<model>.NAM > <outfile>.vhd
#
#  (where "$HWM" is the full path to your hardware modeling directory)
#  Instantiate .vhd into your design.
#
# THE SCRIPT:
#
# Script to generate a VSS/Scirocco VHDL shell for a hardware model
# using the <model>.NAM file

BEGIN {
   pin_type = 0
   is_it_a_vector = "No"
   data_type = ""
   prev_signal = ""
   prev_test = ""
   prev_number = ""
   prev_dir = ""
   ending = ";"

   printf "library SYNOPSYS;\n"
   printf "   use SYNOPSYS.ATTRIBUTES.all;\n"
   printf "library IEEE;\n"
   printf "   use IEEE.std_logic_1164.all;\n\n"

}

$2 ~ /generic_device_name/ {
   device = $3
   printf "entity " device " is\n"
   printf "   generic\n"
   printf "      (\n"
   printf "       timing : LMSI_TIMING_MEASUREMENT           := DISABLED;\n"
   printf "       delay_type : LMSI_DELAY_TYPE               := TYPICAL;\n"
   printf "       delay : LMSI_DELAY                         := ENABLED;\n"
   printf "       log : LMSI_LOG                             := DISABLED;\n"
   printf "       timing_violations : LMSI_TIMING_VIOLATIONS := DISABLED;\n"
   printf "       xprop : LMSI_XPROP                         := DISABLED;\n"
   printf "       xprop_method : LMSI_XPROP_METHOD           := HIGH\n"
   printf "      );\n\n"
   printf "   port\n"
   printf "      (\n"
}

$4 ~ /\(in_pin\)/ || $4 ~ /\(out_pin\)/ || $4 ~ /\(io_pin\)/ \
            || $4 ~ /\(power_pin\)/ {
   pin_type++
}

$2 ~ /\=/ || ($0 ~ /^$/ && pin_type ~ /3/) {
   if (pin_type == 1) {
     direction = "in   "
   }
   else if (pin_type == 2) {
     direction = "out  "
   }
   else if (pin_type == 3) {
     direction = "inout"
   }
   else {
     next
   }
   current_signal = $1 " "
   gsub(/\{/, "", current_signal)
   gsub(/\'/, "", current_signal)

   current_test = current_signal
```

```
          gsub(/[0-9]+ /, " ", current_test)

          n = split(current_signal, array_a, "[a-zA-Z]")
          current_number = array_a[n]
          gsub(/ /, "", current_number)

          if (prev_signal ~ /[0-9]+ /) {
             if (current_test == prev_test)  {
                if (is_it_a_vector == "No") {
                   data_start = prev_number
                }
                if ((current_number == prev_number - 1) || (current_number == prev_number + 1))
                {
                   is_it_a_vector = "Yes"
                }
                prev_signal = current_signal
                prev_test = current_test
                prev_number = current_number
                next
             }
             else {
                if (is_it_a_vector == "Yes") {
                   total = prev_number + data_start
                   if (prev_number > data_start) {
                      data_end = data_start
                      data_start = prev_number
                   }
                   else {
                      data_end = prev_number
                   }
                   data_type = "_vector (" data_start " " "downto " data_end ")"
                   prev_signal = prev_test
                }
             }
          }
          if (prev_signal != "") {
             gsub(/ /, "", prev_signal)
             n = split(prev_signal, array_c, "[a-zA-z0-9_]")
             y = 20 - n
             if (y > 0) {
                for (i = 1; i <= 20-n; i++) {
                   prev_signal = prev_signal " "
                }
             }
             if (($0 ~ /^$/) && (pin_type == 3)) {
                ending = ""
             }
             printf "      " prev_signal " : " prev_dir " std_logic" data_type ending "\n"
          }
          data_type = ""
          is_it_a_vector = "No"
          updown = ""
          prev_signal = current_signal
          prev_test = current_test
          prev_dir = direction
          prev_number = current_number
       }

       END {
          printf "       );\n"
          printf "end " device ";\n\n"
          printf "architecture LMSI of " device " is\n"
          printf "   attribute FOREIGN of LMSI : architecture is \"Synopsys:LMSI\";\n"
          printf "   begin\n"
          printf "end LMSI;\n\n"
       }
```

8

Using MTI VHDL with Synopsys Models

Overview

This chapter explains how to use SmartModels, FlexModels, MemPro models, and hardware models with MTI VHDL. The procedures are organized into the following major sections:

Setting Environment Variables

First, set the basic environment variables. If you are not using one of the model types, skip that step. In some cases the procedures that follow in this chapter include steps for setting additional environment variables.

1. Set the LMC_HOME variable to the location of your SmartModel. FlexModel, and MemPro model installation tree, as follows:

   ```
   % setenv LMC_HOME path_to_models_installation
   ```

2. Make sure that MTI VHDL is set up properly in your environment.

3. Set the LM_LICENSE_FILE or SNPSLMD_LICENSE_FILE environment variable to point to the product authorization file, as shown in the following example:

   ```
   % setenv LM_LICENSE_FILE path_to_product_authorization_file
   % setenv SNPSLMD_LICENSE_FILE path_to_product_authorization_file
   ```

You can put license keys for multiple products (for example, SmartModels and hardware models) into the same authorization file. If you need to keep separate authorization files for different products, use a colon-separated list (UNIX) or semicolon-separated list (NT) to specify the search path in your variable setting.

Caution

Do not include la_dmon-based authorizations in the same file with snpslmd-based authorizations. If you have authorizations that use la_dmon, keep them in a separate license file that uses a different license server (lmgrd) process than the one you use for snpslmd-based authorizations.

4. If you are using the hardware modeler, set the LM_DIR and LM_LIB environment variables, as shown in the following examples:

   ```
   % setenv LM_DIR hardware_model_install_path/sms/lm_dir
   % setenv LM_LIB hardware_model_install_path/sms/models: \
    hardware_model_install_path/sms/maps
   ```

   If you put your models in a directory other than the default of /sms/models, modify the above variable setting accordingly.

5. Depending on your platform, set your load library variable to point to the platform-specific directory in $LMC_HOME, as shown in the following examples:

   Solaris:

   ```
   % setenv LD_LIBRARY_PATH $LMC_HOME/lib/sun4Solaris.lib:$LD_LIBRARY_PATH
   ```

   Linux:

   ```
   % setenv LD_LIBRARY_PATH $LMC_HOME/lib/x86_linux.lib:$LD_LIBRARY_PATH
   ```

   AIX:

   ```
   % setenv LIBPATH $LMC_HOME/lib/ibmrs.lib:$LIBPATH
   ```

   HP-UX:

   ```
   % setenv SHLIB_PATH $LMC_HOME/lib/hp700.lib:$SHLIB_PATH
   ```

   NT:

   Make sure that %LMC_HOME%\lib\pcnt.lib is in the Path user variable.

Using SmartModels with MTI VHDL

To use SmartModels with MTI VHDL, follow this procedure:

1. Open the modelsim.ini file in a text editor and uncomment the lines corresponding to the platform you are using:

```
; ModelSim's interface to Logic Modeling's SmartModel SWIFT software
;libsm = $MODEL_TECH/libsm.sl
; ModelSim's interface to Logic Modeling's SmartModel SWIFT software
(Windows NT)
; libsm = $MODEL_TECH/libsm.dll
; Logic Modeling's SmartModel SWIFT software (HP 9000 Series 700)
; libswift = $LMC_HOME/lib/hp700.lib/libswift.sl
; Logic Modeling's SmartModel SWIFT software (IBM RISC System/6000)
; libswift = $LMC_HOME/lib/ibmrs.lib/swift.o
; Logic Modeling's SmartModel SWIFT software (Sun4 Solaris)
; libswift = $LMC_HOME/lib/sun4Solaris.lib/libswift.so
; Logic Modeling's SmartModel SWIFT software (Sun4 SunOS)
; do setenv LD_LIBRARY_PATH $LMC_HOME/lib/sun4SunOS.lib
; and run "vsim.swift".
; Logic Modeling's SmartModel SWIFT software (Windows NT)
; libswift = $LMC_HOME/lib/pcnt.lib/libswift.dll
```

2. To create the SmartModel Library VHDL wrappers or templates, run the MTI sm_entity script with any optional arguments. The sm_entity script takes SmartModel names as input and writes the VHDL output to STDOUT. You can redirect the output to a file. Run sm_entity as follows. For more information, refer to .

```
% sm_entity -c model > model.vhd
```

For example:

```
% sm_entity -c ttl373 > ttl373.vhd
```

generates the following VHDL file, which has both entity and component declarations for the model. Edit the resulting VHDL file to add the portions of text that are highlighted in the following example:

```
library IEEE;
use IEEE.std_logic_1164.all;
entity ttl373 is
generic ( TimingVersion : STRING := "SN74LS373";
DelayRange : STRING := "MAX";
ModelMapVersion : STRING := "01008" );
port ( C : in std_logic;
D1 : in std_logic;
D2 : in std_logic;
D3 : in std_logic;
D4 : in std_logic;
```

```
D5 : in std_logic;
D6 : in std_logic;
D7 : in std_logic;
D8 : in std_logic;
OC : in std_logic;
Q1 : out std_logic;
Q2 : out std_logic;
Q3 : out std_logic;
Q4 : out std_logic;
Q5 : out std_logic;
Q6 : out std_logic;
Q7 : out std_logic;
Q8 : out std_logic );
end;

architecture SmartModel of ttl373 is
attribute FOREIGN : STRING;
attribute FOREIGN of SmartModel : architecture is "sm_init
$MODEL_TECH/libsm.sl ; ttl373";
begin
end SmartModel;
library ieee; use ieee.std_logic_1164.all; package comp is
component ttl373
generic ( TimingVersion : STRING := "SN74LS373";
DelayRange : STRING := "MAX";
ModelMapVersion : STRING := "01008" );
port ( C : in std_logic;
D1 : in std_logic;
D2 : in std_logic;
D3 : in std_logic;
D4 : in std_logic;
D5 : in std_logic;
D6 : in std_logic;
D7 : in std_logic;
D8 : in std_logic;
OC : in std_logic;
Q1 : out std_logic;
Q2 : out std_logic;
Q3 : out std_logic;
Q4 : out std_logic;
Q5 : out std_logic;
Q6 : out std_logic;
Q7 : out std_logic;
Q8 : out std_logic );
end component;
end comp;
```

3. Compile the *model*.vhd into a library called slm_lib, as follows:

```
% vlib slm_lib
% vmap slm_lib slm_lib
% vcom -work slm_lib model.vhd
```

4. Instantiate the SmartModel component in your testbench by specifying the required SWIFT parameters in the generic map. Here is an example instantiation for the TTL373 model, with the library and use statements, the instance (U1), and the TimingVersion and DelayRange options specified in the generic map for the TTL373 SmartModel Library component.

   Use the SmartModel Library (slm_lib) just as you would use any other VHDL resource library. Here is an example:

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
library SLM_LIB;
use SLM_LIB.COMPONENTS.ALL;

entity TestBench is
end TestBench;

architecture ArchTestBench of TestBench is

signal A, B, C: STD_LOGIC;

U1 : TTL373 generic map (TimingVersion => "SN74LS373",
DelayRange => "Typ")
port map (A => D1, B => D2, C => Q1);

P1 : process
begin
```

   For more information on SmartModel configuration parameters, refer to "Using SmartModels with SWIFT Simulators" on page 18.

5. Compile the top-level testbench to a work library (*MYWORK*) as shown in the following example:

```
% vlib MYWORK
% vcom -work MYWORK top.vhd
```

6. Invoke the simulator by running vsim, as shown in the following example:

```
% vsim -lib MYWORK CFGTEST
```

   For information on how to use MTI VHDL, refer to the "*ModelSim User's Manual*."

sm_entity Command Reference

The command reference for sm_entity is as follows.

Syntax

sm_entity [options] [SmartModels]

Arguments

-read   Read SmartModel names from standard input.

-xe   Do not generate entity declarations.

-xa   Do not generate architecture bodies.

-c   Generate component declarations.

-all   Select all models in the SmartModel Library.

-v   Display progress messages.

By default, sm_entity generates an entity and architecture. Optionally, you can include the component declaration (-c), exclude the entity (-xe), or exclude the architecture (-xa).

Using FlexModels with MTI VHDL

To use FlexModels with MTI VHDL, follow this procedure. This procedure covers users on UNIX and NT. If you are on NT, substitute the appropriate NT syntax for any UNIX command line examples (percent signs around variables and backslashes in paths).

1. Create a working directory and run flexm_setup to make copies of the model's interface and example files there, as shown in the following example:

    % **$LMC_HOME/bin/flexm_setup -dir** *workdir model_***fx**

You must run flexm_setup every time you update your FlexModel installation with a new model version. Table 20 describes the FlexModel interface and example files that the flexm_setup tool copies.

Table 20: FlexModel MTI VHDL Files

File Name	Description	Location
*model*_pkg.vhd	Model command procedure calls for HDL Command Mode.	*workdir*/src/vhdl/
*model*_user_pkg.vhd	Clock frequency setup and user customizations.	*workdir*/src/vhdl/
*model*_fx_mti.vhd	A SWIFT wrapper for the UNIX model.	*workdir*/examples/vhdl/
*model*_fx_mti_nt.vhd	A SWIFT wrapper for the NT model.	*workdir*/examples/vhdl/
*model*_fx_comp.vhd	Component definition for use with the *model* entity defined in the above SWIFT wrapper file. This is put in a package named "COMPONENTS" when compiled.	*workdir*/examples/vhdl/
model.vhd	A bus-level wrapper around the SWIFT model. This allows you to use vectored ports for the model in your testbench. This file assumes that the "COMPONENTS" package has been installed in the logical library "slm_lib".	*workdir*/examples/vhdl/
*model*_tst.vhd	A testbench that instantiates the model and shows how to use basic model commands.	*workdir*/examples/vhdl/

2. On NT, add the following to your modelsim.ini file:

```
libsm = $MODEL_TECH/libsm.dll
```

and add the following to your PATH:

```
%LMC_HOME%\lib\pcnt.lib
```

This is so MTI can find the slm_mti.dll file.

3. Update the clock frequency supplied in the *model*_user_pkg.vhd file to correspond to the desired clock period for the model. After running flexm_setup, this file is located in:

```
workdir/src/vhdl/model_user_pkg.vhd
```

where *workdir* is your working directory.

4. Add the following to your vsystem.ini or modelsim.ini file.

```
slm_lib=$LMC_HOME/sim/mti/lib
VHDL93 = 1
```

5. Compile the FlexModel VHDL files into logical library slm_lib as follows:

```
% mti_path/bin/vlib $LMC_HOME/sim/mti/lib
% mti_path/bin/vcom -work slm_lib $LMC_HOME/sim/mti/src/slm_hdlc.vhd (UNIX)
% mti_path/bin/vcom -work slm_lib $LMC_HOME/sim/mti/src/slm_hdlc_nt.vhd (NT)
% mti_path/bin/vcom -work slm_lib $LMC_HOME/sim/mti/src/flexmodel_pkg.vhd
% mti_path/bin/vcom -work slm_lib workdir/src/vhdl/model_user_pkg.vhd
% mti_path/bin/vcom -work slm_lib workdir/src/vhdl/model_pkg.vhd
% mti_path/bin/vcom -work slm_lib workdir/examples/vhdl/model_fx_comp.vhd
% mti_path/bin/vcom -work slm_lib workdir/examples/vhdl/model_fx_mti.vhd (UNIX)
% mti_path/bin/vcom -work slm_lib workdir/examples/vhdl/model_fx_mti_nt.vhd (NT)
% mti_path/bin/vcom -work slm_lib workdir/examples/vhdl/model.vhd
```

6. Add LIBRARY and USE statements to your testbench:

```
library slm_lib;
use slm_lib.flexmodel_pkg.all;
use slm_lib.model_pkg.all;
use slm_lib.model_user_pkg.all;
```

For example, you would use the following statement for the tms320c6201_fx model:

```
use slm_lib.tms320c6201_pkg.all;
use slm_lib.tms320c6201_user_pkg.all;
```

7. Instantiate FlexModels in your design, defining the ports and generics as required (refer to the example testbench supplied with the model). You use the supplied bus-level wrapper (*model*.vhd) in the top-level of your design to instantiate the supplied bit-blasted wrapper (*model*_fx_mti.vhd for UNIX or *model*_fx_mti_nt.vhd for NT).

Example using bus-level wrapper (*model*.vhd) without timing:

```
U1: model
    generic map (FlexModelID => "TMS_INST1")
    port map ( model ports );
```

Example using bus-level wrapper (*model*.vhd) with timing:

```
U1: model
    generic map (FlexModelID    => "TMS_INST1",
    FlexTimingMode => FLEX_TIMING_MODE_ON,
    TimingVersion  => "timingversion",
    DelayRange     => "range")
    port map ( model ports );
```

8. Compile the testbench as shown in the following example:

```
% vcom testbench
```

9. Invoke the MTI VHDL simulator as shown in the following example:

```
% vsim design
```

Using MemPro Models with MTI VHDL

To use MemPro models with MTI VHDL, follow this procedure:

1. Perform *one* of these platform-dependent steps.

   a. On NT platforms, verify that the shared library is visible from the current working directory. The path to the shared library (**%LMC_HOME%\lib\pcnt.lib**) was set at MemPro installation.

   b. On UNIX and Linux platforms, append the MemPro shared library location to the library search path environment variable setting.

   On Solaris or Linux workstations:

   ```
   % setenv LD_LIBRARY_PATH \
   $LMC_HOME/lib/plat.lib:$LD_LIBRARY_PATH
   ```

   where *plat* is sun4Solaris or x86_linux, respectively.

   On HP-UX workstations:

   ```
   % setenv SHLIB_PATH \
   $LMC_HOME/lib/hp700.lib:$LD_LIBRARY_PATH
   ```

2. Create a local copy of modelsim.ini in the current working directory and a local logical library named slm_lib:

   ```
   % vlib slm_lib
   % vmap slm_lib ./slm_lib
   ```

3. Open the modelsim.ini file in a text editor, uncomment the line that enables VHDL-1993, and uncomment the lines that enable SWIFT; uncomment only the lines that correspond to the platform you are using:

```
; Turn on VHDL-1993 as the default. Normally is off.
; VHDL93 = 1
...
...

; ModelSim's interface to Logic Modeling's SmartModel SWIFT software
;libsm = $MODEL_TECH/libsm.sl
; ModelSim's interface to Logic Modeling's SmartModel SWIFT software
(Windows NT)
; libsm = $MODEL_TECH/libsm.dll
;  Logic Modeling's SmartModel SWIFT software (HP 9000 Series 700)
; libswift = $LMC_HOME/lib/hp700.lib/libswift.sl
;  Logic Modeling's SmartModel SWIFT software (IBM RISC System/6000)
```

```
; libswift = $LMC_HOME/lib/ibmrs.lib/swift.o
;  Logic Modeling's SmartModel SWIFT software (Sun4 Solaris)
; libswift = $LMC_HOME/lib/sun4Solaris.lib/libswift.so
;  Logic Modeling's SmartModel SWIFT software (Sun4 SunOS)
;[7Cdo setenv LD_LIBRARY_PATH $LMC_HOME/lib/sun4SunOS.lib
;[7Cand run "vsim.swift".
;  Logic Modeling's SmartModel SWIFT software (Windows NT)
; libswift = $LMC_HOME/lib/pcnt.lib/libswift.dll
```

☞ **Note**

This "UNIX-like" syntax is valid on NT workstations as well as all UNIX platforms.

4. Compile the MemPro VHDL files into your local slm_lib.

   a. If you *are* simulating RDRAM MemPro models:

   ```
   % vcom -work slm_lib $LMC_HOME/sim/mti/src/slm_hdlc.vhd
   % vcom -work slm_lib $LMC_HOME/sim/mti/src/mempro_pkg.vhd
   % vcom -work slm_lib $LMC_HOME/sim/mti/src/rdramd_pkg.vhd
   ```

   b. If you *are not* simulating RDRAM MemPro models:

   ```
   % vcom -work slm_lib $LMC_HOME/sim/mti/src/slm_hdlc.vhd
   % vcom -work slm_lib $LMC_HOME/sim/mti/src/mempro_pkg.vhd
   ```

5. After generating a model using MemPro, compile the VHDL code for the model.

   a. If you *are* simulating SDRAM Module MemPro models, you must compile these three files in the following order:

   ```
   % vcom -work slm_models model_eeprom.vhd
   % vcom -work slm_models model_sdram.vhd
   % vcom -work slm_models model.vhd
   ```

   b. If you *are not* simulating SDRAM Module MemPro models:

   ```
   % vcom -work slm_models model.vhd
   ```

6. If you compiled the model to a library (SLM_MODELS in these examples), add a LIBRARY statement to your testbench:

   ```
   LIBRARY SLM_MODELS
   ```

This makes models in SLM_MODELS accessible from your design. You can refer to the model using a standard VHDL convention, such as a CONFIGURATION statement:

```
CONFIGURATION mymem_a OF mymem_tst IS
    FOR atest
            FOR ALL : mymem use CONFIGURATION SLM_MODELS.mymem_behav;
            END FOR;
    END FOR;
END mymem_a;
```

7. Add LIBRARY and USE statements to your testbench:

```
LIBRARY SLM_LIB
USE SLM_LIB.mempro_pkg.all
```

This also provides access to MemPro testbench commands.

For more information on using the MemPro testbench interfaces, refer to the "HDL Testbench Interface" and "C Testbench Interface" chapters in the *MemPro User's Manual*.

8. Instantiate MemPro models in your design. Define ports and generics as required. For information on generics used with MemPro models, refer to "Instantiating MemPro Models" on page 32. For information on message levels and message level constants, refer to "Controlling MemPro Model Messages" on page 33.

9. Compile your design as shown in the following example:

```
% vcom design_name
```

10. Invoke the simulator on your design as shown in the following example:

```
% vsim design_name
```

Using Hardware Models with MTI VHDL

To use hardware models with MTI VHDL, follow this procedure:

1. Make sure MTI VHDL is set up properly and all required environment variables are set, as explained in "Setting Environment Variables" on page 155.

2. Add the hardware model install tree to your path variable, as shown in the following example:

```
% set path=(/install/sms/bin/your_platform/ $path)
```

3. Modify the modelsim.ini or project_name.mpf file to include the hardware modeling information. Locate the line:

```
        [lmc]
```

Remove the semicolons from the libhm line and the libsfi line you will be changing for your platform. Provide the correct path for the SFI. For example:

```
; ModelSim's interface to Logic Modeling's hardware modeler SFI software
libhm = $MODEL_TECH/libhm.sl
; Logic Modeling's hardware modeler SFI software (HP 9000 Series 700)
libsfi = hardware_model_install_path/lib/platform/libsfi.ext
; Logic Modeling's hardware modeler SFI software (IBM RISC System/6000)
; libsfi = <sfi_dir>/lib/rs6000/libsfi.a
; Logic Modeling's hardware modeler SFI software (Sun4 Solaris)
; libsfi = <sfi_dir>/lib/sun4.solaris/libsfi.so
; Logic Modeling's hardware modeler SFI software (Sun4 SunOS)
; libsfi = <sfi_dir>/lib/sun4.sunos/libsfi.so
; Logic Modeling's hardware modeler SFI software (Window NT)
; libsfi = <sfi_dir>/lib/pcnt/lm_sfi.dll
```

where *ext* is so for Solaris. a for AIX, or sl for HP-UX.

4. Run the hm_entity script to generate a .vhd file for the hardware model as shown in the following example. For details on hm_entity, refer to "hm_entity Command Reference" on page 167.

5. You are now ready to use the model in your simulation.

MTI VHDL Example Using TILS299 Hardware Model

Here is an example that uses the TILS299 hardware model. Follow these steps:

1. Put the TILS299 hardware model in the testbench.

2. Create a working library directory by invoking vsim -gui and selecting Library/Create. This creates a working directory called work.

3. Compile the .vhd files, as shown in the following example:

```
% vcom -work work TILS299.vhd TB_TILS299.vhd
```

This step compiles the two VHDL files and puts them in the specified work library. Note that the TILS299.vhd file must be specified first or you get an error because the TB_TILS299.vhd utilizes the TILS299 entity.

4. Invoke the simulator as shown in the following example:

```
% vsim
```

5. When the window comes up, select the testbench to load.

6. Use the View/Wave pull-down menu to get the wave window. In the wave window, use File/Load Format wave.do to get the waveforms. After the waveform viewer comes up and the vsim prompt appears, enter "run 10000".

7. You can also use some of the hardware model utilities listed below, but the commands must be entered at the simulator command prompt because they are not VHDL statements. For the TILS299 example, you can also put these commands into the .do file. Here is an example wave.do file:

```
lm_vectors on /tb_tils299/U0 TEST.VEC
add wave  -logic {/clk}
add wave  -logic {/clr}
add wave  -logic {/s1}
add wave  -logic {/s0}
add wave  -logic {/g1}
add wave  -logic {/g2}
add wave  -logic {/sr}
add wave  -logic {/sl}
add wave  -logic {/qa}
add wave  -logic {/qh}
add wave -literal {/t}
```

hm_entity Command Reference

The hm_entity script creates .vhd files for hardware models.

Syntax

hm_entity [options] *shell_software_filename*

Arguments

-xe	Do not generate entity declaration.
-xa	Do not generate architecture body.
-c	Generate component declaration
-93	Use extended identifiers where needed

Example

For example, the following hm_entity invocation:

```
% hm_entity TILS299.MDL > TILS299.vhd
```

generates a .vhd file that looks like the following:

```
                    library ieee;
    use ieee.std_logic_1164.all;
    entity TILS299 is
        generic( DelayRange : STRING := "Max" );
        port ( G2 : in std_logic;
               CLR : in std_logic;
               SR : in std_logic;
               CLK : in std_logic;
               S0 : in std_logic;
               G1 : in std_logic;
               SL : in std_logic;
               S1 : in std_logic;
               QA : out std_logic;
               QH : out std_logic;
               H : inout std_logic;
               E : inout std_logic;
               G : inout std_logic;
               A : inout std_logic;
               C : inout std_logic;
               B : inout std_logic;
               F : inout std_logic;
               D : inout std_logic );
    end;

    architecture Hardware of TILS299 is
        attribute FOREIGN : STRING;
        attribute FOREIGN of Hardware : architecture is "hm_init
    $MODEL_TECH/libhm.sl; TILS299.MDL";
    begin
    end Hardware;
```

MTI VHDL Utilities

The following hardware modeler simulator commands are supported in MTI VHDL:

lm_vectors on | off *instance_name filename*

The lm_vectors utility turns on vector logging for the hardware model instance. The vectors record stimulus to the input and I/O pins and responses from the output and I/O pins during simulation.

lm_measure_timing on | off *instance_name filename*

The lm_measure_timing utility causes the modeler to measure timing between an input transition and resulting output transition on the hardware model. Note that this is only supported on LM-family systems.

lm_timing_checks on | off *instance_name*

The lm_timing_checks utility allows you to enable or disable timing checks such as setups and holds.

lm_loop_patterns on | off *instance_name*

The lm_loop_patterns utility causes the hardware modeler to continually replay the pattern history of a specified device instance.

lm_unknowns on | off *instance_name*

The lm_unknowns utility turns off unknown propagation. This "on_unknown" feature is also in the .OPT file for hardware models. It modifies the system's default handling of device input and I/O pins that are set to unknown by the simulator. This utility does not turn on unknown propagation unless it is also turned on in the .OPT file, but it can override the setting in the .OPT file to turn this feature off when it is set to on in the .OPT file.

9

Using Cyclone with Synopsys Models

Overview

This chapter explains how to use MemPro models and hardware models with Cyclone. The procedures are organized into the following major sections:

Setting Environment Variables

First, set the basic environment variables. If you are not using one of the model types, skip that step. In some cases the procedures that follow in this chapter include steps for setting additional environment variables.

1. Set the LMC_HOME variable to the location of your MemPro installation tree, as shown in the following example:

   ```
   % setenv LMC_HOME path_to_models_installation
   ```

2. Set the LM_LICENSE_FILE or SNPSLMD_LICENSE_FILE environment variable to point to the product authorization file, as shown in the following example:

   ```
   % setenv LM_LICENSE_FILE path_to_product_authorization_file
   % setenv SNPSLMD_LICENSE_FILE path_to_product_authorization_file
   ```

You can put license keys for multiple products (for example, MemPro models and hardware models) into the same authorization file. If you need to keep separate authorization files for different products, use a colon-separated list (UNIX) or semicolon-separated list (NT) to specify the search path in your variable setting.

Caution

Do not include la_dmon-based authorizations in the same file with snpslmd-based authorizations. If you have authorizations that use la_dmon, keep them in a separate license file that uses a different license server (lmgrd) process than the one you use for snpslmd-based authorizations.

3. Set the SYNOPSYS_CY environment variable to point to the Cyclone installation tree, as shown in the following example:

   ```
   % setenv SYNOPSYS_CY Cyclone_install_path
   ```

4. Set the MA_CY environment variable to point to the ma_cyclone directory, as shown in the following example:

   ```
   % setenv MA_CY ModelAccess_install_path
   ```

5. If you are using the hardware modeler, set the LM_DIR and LM_LIB environment variables, as shown in the following examples:

   ```
   % setenv LM_DIR hardware_model_install_path/sms/lm_dir
   % setenv LM_LIB hardware_model_install_path/sms/models: \
    hardware_model_install_path/sms/maps
   ```

   If you put your models in a directory other than the default of /sms/models, modify the above variable setting accordingly.

6. Depending on your platform, set your load library variable to point to the platform-specific directory in $LMC_HOME, as shown in the following examples:

   Solaris:

   ```
   % setenv LD_LIBRARY_PATH $LMC_HOME/lib/sun4Solaris.lib:$LD_LIBRARY_PATH
   ```

   Linux:

   ```
   % setenv LD_LIBRARY_PATH $LMC_HOME/lib/x86_linux.lib:$LD_LIBRARY_PATH
   ```

   AIX:

   ```
   % setenv LIBPATH $LMC_HOME/lib/ibmrs.lib:$LIBPATH
   ```

   HP-UX:

   ```
   % setenv SHLIB_PATH $LMC_HOME/lib/hp700.lib:$SHLIB_PATH
   ```

   NT:

   Make sure that %LMC_HOME%\lib\pcnt.lib is in the Path user variable.

Using SmartModels with Cyclone

For information on using SmartModels with Cyclone, refer to "Using SmartModels with SWIFT Simulators" on page 18.

Using FlexModels with Cyclone

To use FlexModels with Cyclone, you use Direct C Control. For information on the required SWIFT parameters for FlexModels (which differ from regular SmartModels) and how to use Direct C Control, refer to "Using FlexModels with SWIFT Simulators" on page 24.

Using MemPro Models with Cyclone

To use MemPro models with Cyclone, follow this procedure. Note that RDRAM models are not supported on Cyclone.

1. For HP-UX, Cyclone incorrectly uses "hpux10.lib" in paths to platform-specific directories. The correct path leaf should be "hp700.lib." Correct the paths by creating symbolic links as follows:

   ```
   % ln -s $LMC_HOME/lib/hp700.lib $LMC_HOME/lib/hpux10.lib
   % ln -s $LMC_HOME/mempro/lib/hp700.lib $LMC_HOME/mempro/lib/hpux10.lib
   ```

2. Create a logical library named slm_lib. The default physical library mapping for this is $LMC_HOME/sim/cyclone/lib, but you can put the slm_lib anywhere you want. Add the following line to your .synopsys_vss.setup file, and adjust accordantly if you put slm_lib anywhere but the default location:

   ```
   SLM_LIB   :   $LMC_HOME/sim/cyclone/lib
   ```

3. Compile the MemPro VHDL files into logical library slm_lib.

   ```
   % cyan -nc -synthoff -lang vhdl -w slm_lib \
       $LMC_HOME/sim/cyclone/src/slm_hdlc.vhd

   % cyan -nc -synthoff -lang vhdl -w slm_lib \
       $LMC_HOME/sim/cyclone/src/mempro_pkg.vhd
   ```

4. After generating a model using MemPro, compile the VHDL code for the model, as shown in the following example:

   ```
   % cyan -nc -synthoff -lang vhdl mymem.vhd
   ```

5. If you compiled the model to a library (SLM_MODELS in these examples), add a LIBRARY statement to your testbench:

   ```
   LIBRARY SLM_MODELS
   ```

This makes models in SLM_MODELS accessible from your design. You can refer to the model using a standard VHDL convention, such as a CONFIGURATION statement:

```
CONFIGURATION mymem_a OF mymem_tst IS
    FOR atest
            FOR ALL : mymem use CONFIGURATION SLM_MODELS.mymem_behav;
            END FOR;
    END FOR;
END mymem_a;
```

6. Add LIBRARY and USE statements to your testbench:

```
LIBRARY SLM_LIB
USE SLM_LIB.mempro_pkg.all
```

This also provides access to MemPro testbench commands.

For more information on using the MemPro testbench interfaces, refer to the "HDL Testbench Interface" and "C Testbench Interface" chapters in the *MemPro User's Manual*.

7. Instantiate MemPro models in your design. Define ports and generics as required. For information on generics used with MemPro models, refer to "Instantiating MemPro Models" on page 32. For information on message levels and message level constants, refer to "Controlling MemPro Model Messages" on page 33.

8. Compile your design as shown in the following example:

```
% cyan design_name.vhd
```

9. Invoke Cyclone on your design as shown in the following example:

```
% cysim (-4state | -2state) -i design_name
```

Using Hardware Models with Cyclone

This section describes how to set up and configure Release 3.5a of ModelAccess for Cyclone. After completing the setup tasks, for usage information refer to "Using Hardware Models with Cycle-Based Simulators" on page 182.

The hardware modeling configuration you choose affects the performance you get when running hardware models in Cyclone simulations. This section reviews the fundamentals of the ModelSource and LM-family hardware modeling systems, and then provides guidelines for a number of possible configurations.

ModelSource System Hardware and Software

If you are using a ModelSource system, your hardware modeling system configuration consists of one or more MS-3400 or MS-3200 units, plus a ModelSource Processor. (For a description of a ModelSource Processor, refer to the *ModelSource Hardware Installation Guide*. For information about the software, refer to the *ModelSource User's Manual*.) The ModelSource Processor is connected to the rest of your network via Ethernet, and to the MS-3400/MS-3200 units via fiber-optic cable.

The ModelSource Processor provides the CPU for the ModelSource units, and at a minimum, it executes the runtime modeler software (RMS) for the modeling system. However, you might decide to run your simulation from the ModelSource Processor workstation as well, unless you are using an LM-1400 as the ModelSource Processor.

The R3.3a and later ModelSource RMS has been enhanced to deliver higher performance in all configurations, and has been optimized to generate the maximum performance gain over previous releases of the RMS when used by a single user running the simulation from the ModelSource Processor workstation. This enhanced release of the RMS is available for Sun Solaris and HP 700 ModelSource Processor workstations.

LM-1400/LM-family System Hardware and Software

If you are using one of the LM-family hardware model servers (LM-1200 or LM-1400), your hardware modeling system configuration consists of the LM-family unit. This family of modelers includes a dedicated CPU within the modeling system chassis. The LM-family system connects to the rest of your network via Ethernet. The LM-family CPU runs the standard RMS. You run your simulations from other workstations on the Ethernet network.

Configuration Options

Figure 7 on page 177 illustrates some of the supported Cyclone configurations, labeled from A (the highest performance choice) to D (lower performance options).

Option A

The recommended configuration for highest performance in cycle-based simulation is an MS-3400 or MS-3200 hardware modeling system with the simulation executing on the ModelSource Processor workstation (which has the SBus or EISA card connection to the modeling systems). This configuration eliminates network overhead in the communication between the modeling system processor and the simulation.

Option B

In this configuration, the Cyclone simulation is executing on a different workstation from the ModelSource Processor workstation. In this case, the simulation workstation and the ModelSource Processor workstation must be on the same Ethernet subnet.

Option C

Because the LM-1400 has its own dedicated CPU within the LM-1400 chassis, the simulation must be run on a separate workstation. For best performance with an LM-1400 (or any of the LM-family hardware model servers), keep the simulation workstation and the LM-1400 on the same Ethernet subnet.

Option D

In this configuration, the hardware modeling system (which can be either a ModelSource system or an LM-family hardware model server) exists on a different Ethernet subnet from the workstation on which the Cyclone simulation is running. Because of the extra overhead of the router, this is a lower performance configuration.
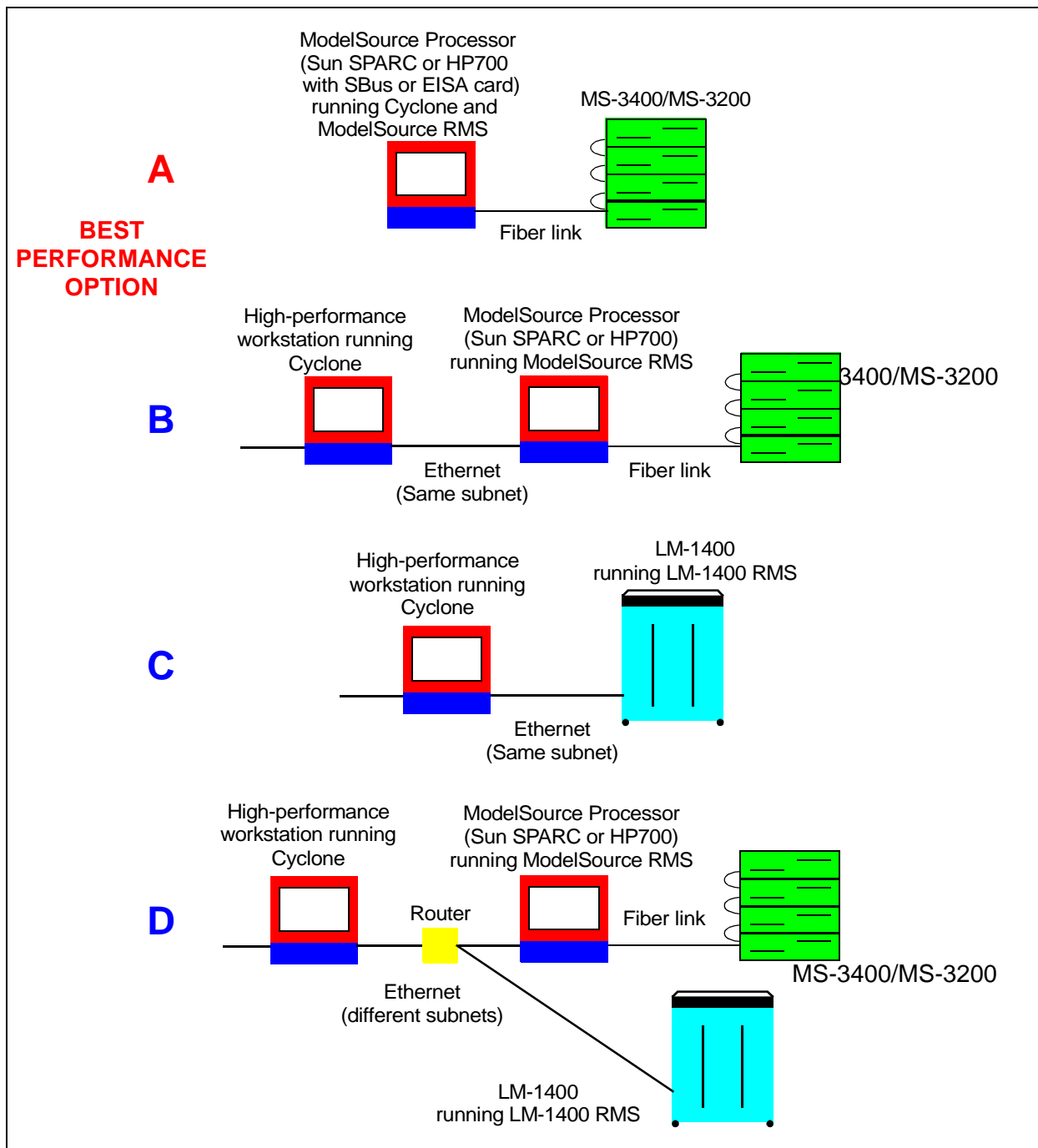
Figure 7: Cyclone Configuration Guidelines

Cyclone User Setup

Before proceeding with the setup instructions that follow, perform these tasks:

- Install the Cyclone simulator package as described in the *Cyclone Installation Guide*.

- Install and configure the hardware modeling system, including hardware and software (R3.5a or later), as outlined in the Quick Reference in Chapter 1 of either the *ModelSource Hardware Installation Guide* or the *LM-family Hardware Installation Guide.*

- If necessary, boot the modeler.

- Make sure all required environment variables are set, as explained in "Setting Environment Variables" on page 171

The ma_cyclone Software Tree

The ModelAccess for Cyclone (ma_cyclone) directory structure is illustrated in Figure 8.

ma_cyclone/

bin/	**C/**	**setup**/
sun5.5.1/	lm_hw_slang.c	setup.csh
geninterface	lm_hw_slang.h	setup.sh
verifySetup		synopsys_lm_hw.setup
pa_hp102/		synopsys_lm_hw.setup.hp700
genInterface		synopsys_lm_hw.setup.solaris
verifySetup		
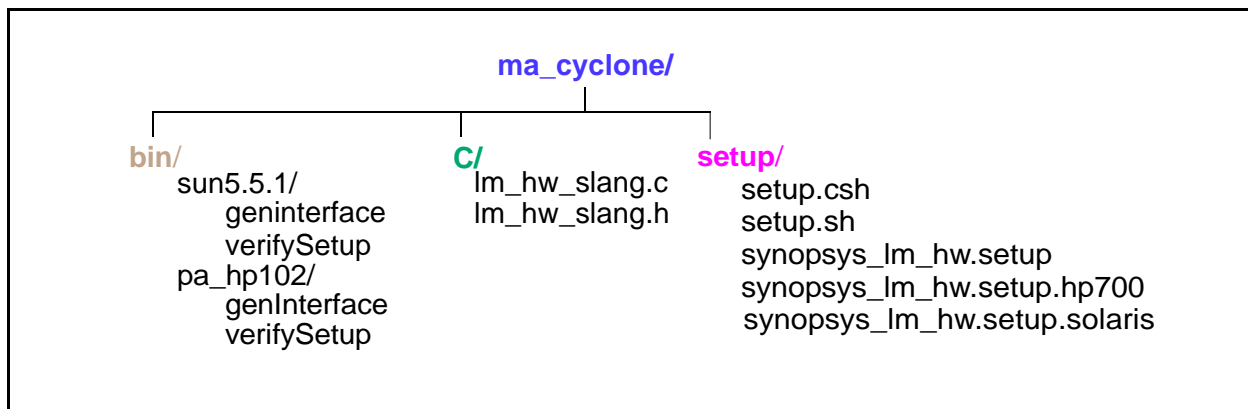
Figure 8: ModelAccess for Cyclone Installation Tree

The setup process consists of the following tasks:

- "Setting Up Your Environment" on page 179

- "Running verifySetup" on page 179

- "Running geninterface" on page 179.

- "Confirming License File Settings (ModelSource Only)" on page 181.

Setting Up Your Environment

Make sure all required environment variables are set properly, as explained in "Setting Environment Variables" on page 171. If any of the required environment variables are not defined, the source command will fail, with an error message indicating the cause of the error.

Running verifySetup

Run the provided verifySetup program. This verifies that your environment is set up correctly so that genInterface can run.

1. To run verifySetup, change directory to /tmp, then execute verifySetup, as follows:

```
% cd /tmp
% verifySetup
```

The verifySetup program returns messages confirming the setup information that will be used (both the environment setup information, and the genInterface setup options taken from the synopsys_lm_hw.setup file). For example, if the hardware modeling system is not booted and available on the network, verifySetup reports the error.

```
% verifySetup
Copyright 1988-1996 Synopsys, Incorporated.; 05 Sep 1996; R1.0

**** Environment Setup ****
User home: /home/klt
MA_CY: /tools/lmc/sms/ma_cyclone
LM include directory: /tools/lmc/sms/include
LM library directory: /tools/lmc/sms/lib/sun4.solaris
CY include directory: /tools/cyclone/sparcOS5/cyclone/include

**** Setup Files ****
Modeler: engineering1
SFI ERROR: modeler not responding (Message Number: 972)
```

Running geninterface

The genInterface program takes hardware model Shell Software files as input, and creates the following files:

- A VHDL shell for each hardware model you specified

- A dynamically-linkable C library, which is used in communicating simulator requirements to/from the hardware modeling system (via the hardware modeling Simulator Function Interface software)

With the output of genInterface, you proceed as with any other VHDL input by compiling the hardware model VHDL files (elaborate and analyze) along with your other VHDL design files and then simulating the design. Figure 9 gives an overview of the entire process, and the following sections describe each step in detail.

Invoke genInterface from the directory in which you want the interface files to be created. On the command line, specify the hardware models you want to use with Cyclone. For details on genInterface syntax, refer to "genInterface Command Reference" on page 186

☞ **Note**
The genInterface program relies on the software described in "Cyclone genInterface Setup Files" on page 190.   The verifySetup program helps you verify that these prerequisites have been set up correctly.
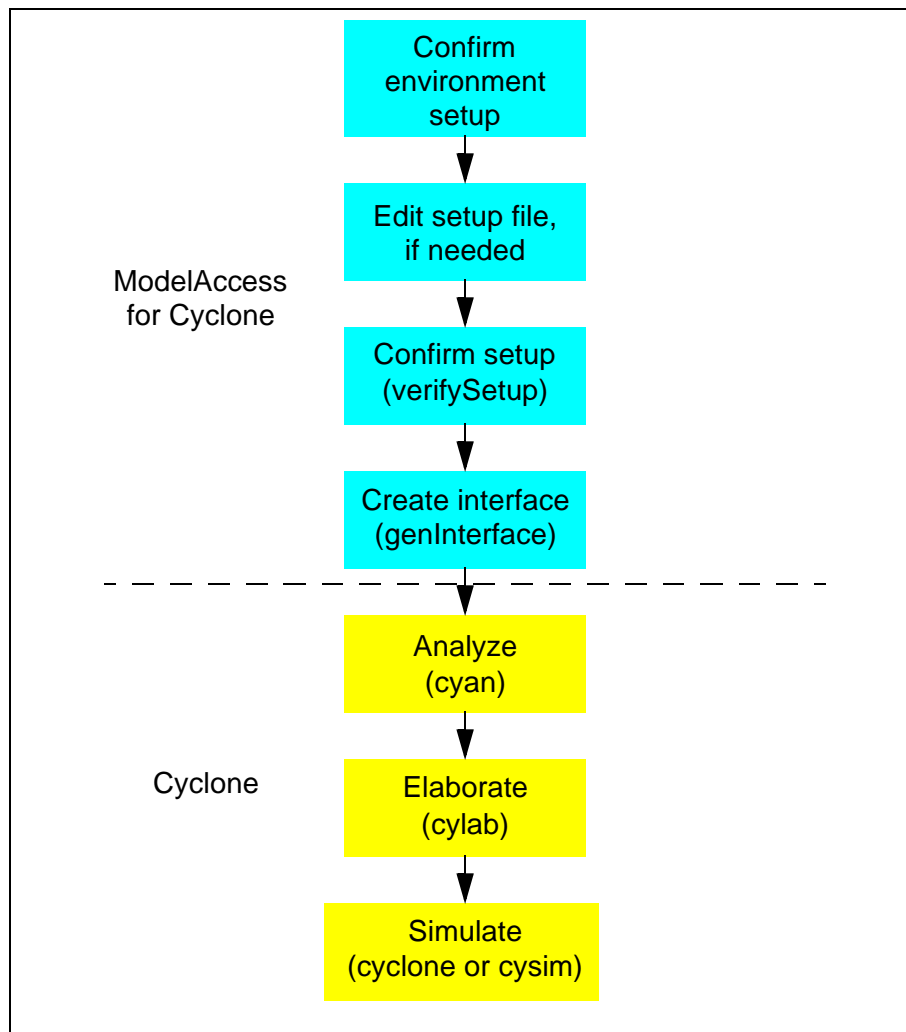
Figure 9: Process Flow Chart

Confirming License File Settings (ModelSource Only)

The genInterface program is not license-protected. However, in order to use the output of genInterface to run Cyclone simulations with ModelSource hardware models, several licenses are required:

- MSCBS licenses the use of hardware models with cycle-based simulators.

- MS3400 or MS3200 specifies the number of MS-3400 or MS-3200 units licensed.

In addition, you need the appropriate licenses to run Cyclone.

Note

> The LM-family hardware model servers (LM-1400 and LM-1200) are not license-protected and do not have a license file. This step is required only for ModelSource systems (MS-3400, MS-3200).

For information about installing hardware model licenses or updating an existing license file, refer to the *Hardware Modeling Release Notes*. To confirm that your licenses are working correctly, follow these steps:

1. Invoke the lm utilities:

```
% lm

Copyright 1988-1996 Synopsys, Incorporated.; 17 Aug 1998; R3.4a
Default Modeler is "venkat"
   LM Utilities Menu
   1)  Create Logic Models
   2)  Verify Logic Models
   3)  Perform Maintenance
   4)  Run Diagnostics
   5)  Show Modeler Configuration
   h)  Help
   q)  Quit
selection:
```

2. Select item 5, "Show Modeler Configuration".

```
selection: 5
   Modeler Configuration
   1)   Show Modelers
   2)   Show Logic Models
   3)   Show Users
   4)   Show Versions
   5)   Show Model Users
   6)   Show Licenses
   h)   Help
   q)   Quit
selection:
```

3. Select item 6, "Show Licenses".

```
selection: 6
Modeler Name (* = ALL) [venkat]:
 License Server set to: 5300@hal
"venkat" Licenses Used
 No licenses being used on the modeler
"venkat" Total Licenses Present in the License File
 Feature                   # licenses     Version       Exp. Date
MSFAULTYes3.400 31-Dec-1999
MSCBSYes  3.400 31-Dec-1999
MS3400100 3.400 31-Dec-1999
MS3200100 3.400 31-Dec-1999
 ++++++++++++++++++++++++++++++++++++++++++++++++
```

Using Hardware Models with Cycle-Based Simulators

ModelAccess for Cyclone allows you to prepare your hardware models for use in a Cyclone cycle-based simulation. This section describes how to use hardware models in a Cyclone simulation. We begin with an overview of hardware modeling in the Cyclone environment and then provide instructions for using genInterface.

Timing Delays

Synopsys hardware models typically include pin-to-pin delay information and can optionally include timing checks. However, in cycle-based simulation, the simulator ignores delay information and timing checks in the hardware models.

Cycle-Based Simulation Constraints

Before using a hardware model in cycle-based simulation, review the design, coding, and testbench guidelines provided in the Cyclone documentation set. Although there are no inherent limitations because of hardware modeling technology (when compared to a VHDL model or C-language model of the same device), you must follow the same usage guidelines for a circuit using hardware models as you would follow for a circuit using any other types of models, when creating a cycle-based simulation testbench.

How Hardware Models Interface with Cyclone

Cyclone provides the Slang C-language interface to enable you to integrate external C and C++ models into the Cyclone runtime environment. Hardware models are also integrated into the Cyclone environment using a special-purpose implementation of the Slang interface.

A Slang C (or C++) software model consists of a collection of C language entry points, compiled into a shared object library, plus a VHDL shell that determines which entry points are called at runtime. A Slang hardware model requires a shared object library, one VHDL shell per model, the hardware model's Shell Software, and the model itself, installed in the hardware modeler. A conceptual diagram of a Slang hardware model is shown in Figure 10.
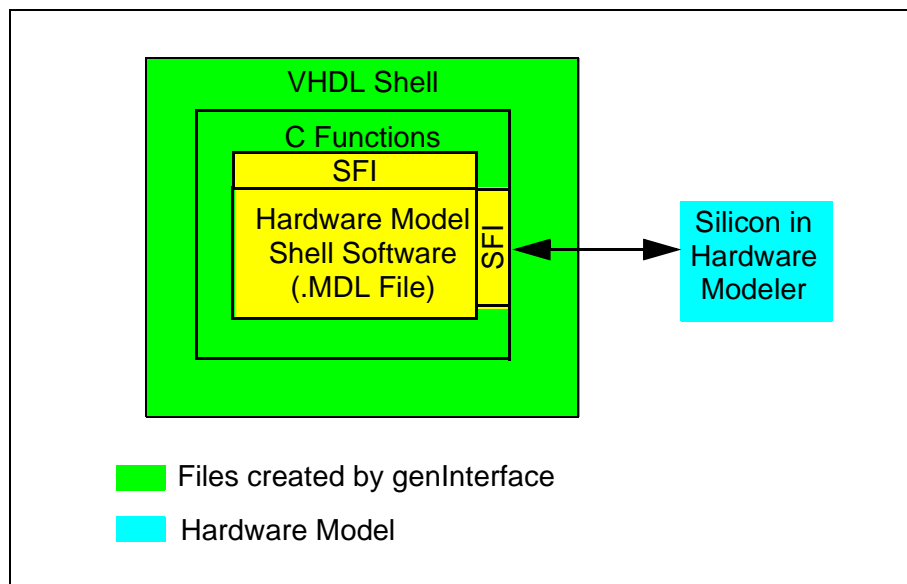
Figure 10: Slang Hardware Model Conceptual Diagram

👉 **Note**

> The genInterface program creates the C library and VHDL shell files needed by Cyclone to evaluate hardware models.

Editing the Setup File

The genInterface program has its own setup file (synopsys_lm_hw.setup) that you use to specify various options to be applied to the entire genInterface session, or to particular models within the session, including:

- Deleting intermediate files

- Overwriting existing files

- Overwriting pin names (per model)

Default values are provided for all required items, so you only need to edit this file if you want to alter the default values. If you decide to edit the file, copy it from $MA_CY/setup/synopsys_lm_hw.setup to your own local working directory. The copy must be renamed to .synopsys_lm_hw.setup. Now you can edit and customize the local .synopsys_lm_hw.setup file appropriately for your session.

If you want to change the global settings on a Solaris system, you must edit the following file:

   $MA_CY/setup/synopsys_lm_hw.setup.solaris

(The other file extension is .hp700.)

To change the file, copy lines from the default synopsys_lm_hw_setup file shown in Figure 11 and uncomment the lines.

```
# delete_files yes              # default yes

# overwrite_files no            # default no

# for PPC403GA use
#    delete_files yes
#    overwrite_files no
#    pin_name_ovr "-DSR/-CTS" "NSRSCTS"
#    pin_name_ovr "-HALT" "NHALT"
```

Figure 11: Default synopsys_lm_hw.setup File

Deleting Intermediate Files

```
# delete_files {yes|no}              # default yes
```

By default, genInterface deletes the intermediate files it creates. If you want to retain the intermediate files, specify "delete_files no" in the setup file and delete the leading '#' character. (Typically, you need to save these files only for debugging purposes; the files are not used by Cyclone.)

Overwriting Existing Files

```
# overwrite_files {yes|no}              # default no
```

By default, genInterface does not overwrites files in the target directory. This is to protect you from accidentally overwriting earlier versions of .vhd files that you might have customized. If genInterface detects a file with the same name in the target directory, it generates the following warning:

```
genInterface warning: retaining older version of ./model_name.vhd file
```

If you receive this warning, you must choose one of the following:

- If you want to save the old .vhd files, rename them, and then run genInterface again. You can add your custom code to the newly-updated .vhd files.

- If you don't want to save the old .vhd files, delete them from the target directory or change the overwrite_files setting to yes before you run genInterface/

Attention

Whenever you receive this warning, you must correct the situation and re-run genInterface so that a complete, integrated set of .vhd files and the corresponding C library are created. The genInterface program keys the results of each session, so if you attempt to mix files from different genInterface sessions in your Cyclone simulation, you receive a fatal simulation error (LM_HW integration error: Keys do not match).

Selecting Options Per Model

The synopsys_lm_hw.setup file allows you to set specific options per model, including the following:

- delete_files

- overwrite_files

- cflags -DLM_HW_DEBUG

- pin_name_ovr

The cflags debugging options are intended for system administrators, and are explained in "cflags" on page 191.

The pin_name_ovr statement, which enables the overwriting of automatically-generated pin names, is only available on a per-model basis, as explained below.

Overwriting Pin Names Per Model

The Shell Software syntax for hardware model pin names uses special characters and VHDL keywords that are not allowed in legal VHDL signal names. Therefore, when genInterface creates VHDL shells for each model, it converts illegal VHDL signal names to legal equivalents. This process is explained in "Rules for Signal Renaming" on page 192. If you prefer to use your own VHDL signal names, you can use the pin_name_ovr statement to specify the mapping from the original name to the new name.

The syntax for this statement is:

```
for model_name use
    pin_name_ovr "shell_sw_name1" "VHDLname1"
    pin_name_ovr "shell_sw_name2" "VHDLname2"
end
```

For example, the pin name -ALE is allowed in the Shell Software, but not in VHDL. By default, genInterface removes the leading hyphen (-) and replaces it with the string NE_, creating the new pin name NE_ALE. If you prefer the alternate legal name NALE, add the following lines to your setup file:

```
for I80960M use
    pin_name_ovr "-ALE" "NALE"
end
```

genInterface Command Reference

After successfully running verifySetup, you can run genInterface, specifying the hardware models you want to include in Cyclone simulation.

Syntax

genInterface {-m *modeler_name*} [*mdlfile1 mdlfile2 ...* | -f *model_list* | -a]

Arguments

-m *modeler_name*   This optional switch specifies the hardware modeling system for genInterface to use. The modeler must be installed on the network and be booted and running. If a modeler_ name is not specified, genInterface searches the modelers.lis file for the name of an available modeler.

The modeler does not need to have the hardware model installed; it must only be booted and running Runtime Modeler Software v3.3 or later.

mdlfile1	You can list individual models by their Model (.MDL) file name, such as IPENTIUM.MDL. \Separate multiple file names with a blank (space) character.
-f *model_list*	You can create a file listing the Model (.MDL) files to be included. Create the file with one .MDL file name per line.
-a	The **-**a option allows you to generate an interface that includes all available model files found in directories specified by the LM_LIB environment variable.

☀ Hint

The -a option is convenient when you want to create one interface incorporating all hardware models in your environment. However, depending on how your LM_LIB environment variable is set, this could be a large file.

Examples

The following example creates interface files for the hardware models listed in the "my.models" file:

```
% genInterface -f my.models
```

The following example creates interface files for all hardware models in the LM_LIB search path, using the hardware modeling system named "engineering1":

```
% genInterface -m engineering1 -a
```

The following example creates interface files for the hardware models IPENTIUMPRO, I82451GX, I82452GX, I82453GX, and I82454GX:

```
% genInterface IPENTIUMPRO.MDL I82451GX.MDL I82452GX.MDL \
I82453GX.MDL I82454GX.MDL
```

Assuming that all setup files have been left at their default values, the genInterface command creates the following files in the current directory:

- liblm_hw.so (Solaris only)
- IPENTIUMPRO.vhd
- I82451GX.vhd
- I82452GX.vhd
- I82453GX.vhd
- I82454GX.vhd

The following example shows genInterface executed on the ID3052EA hardware model (Model file ID3052EA.MDL):

```
% genInterface -m engineering1 ID3052EA.MDL
Copyright 1988-1996 Synopsys, Incorporated.; 05 Sep 1996; R1.0

Processing Common files.....Done
Processing ID3052EA.MDL file......Done
Running make....Done
Running clean....Done
```

This command creates one dynamic library and a ".vhd" file for each model specified in the command line. For the Solaris example shown above, the liblm_hw.so library and ID3052E.vhd file are created.

The genInterface program requires an ANSI C compiler. If you receive compiler errors while attempting to run genInterface, for information about updating setup files, refer to "Cyclone genInterface Setup Files" on page 190.

Cyclone Simulation

After successfully running verifySetup and genInterface, you can simulate using Cyclone. For detailed information on using Cyclone, refer to the *Cyclone Reference Manual*. Following are some Cyclone usage notes for hardware models.

Analyzing the Design

You analyze the generated VHDL files (created by genInterface), just as you would any other files in your design.

Elaborating and Simulating the Design

Performance Monitoring

You can monitor the performance of the hardware modeler and append the results to the simulator log file after simulation. To enable performance monitoring, in the window where you are running the simulator, enter the following:

```
% setenv LM_OPTION "monitor_performance"
```

For more information, refer to "Performance Monitoring" in the *ModelSource User's Manual.*

2-state and 4-state Simulation

If you are using Release 1.1 or later of ModelAccess for Cyclone with Cyclone Release 1998.02 or later, you can specify either 2-state (0, 1) or 4-state (0, 1, X, Z) simulation. However, for earlier releases, 2-state simulation is not supported; when running cylab, you must specify -4state to create 4-state (0, 1, X, Z) code. Similarly, for earlier releases you must specify -4state for 4-state simulation with cysim.

Defining LD_LIBRARY_PATH

When using the output of genInterface with Cyclone, the LD_LIBRARY_PATH environment variable must include "." (the current directory). This is also required by Cyclone, so if you have set LD_LIBRARY_PATH as documented in the Cyclone user documentation, LD_LIBRARY_PATH will be correct for genInterface.

Cyclone Elaboration Warnings

Cyclone issues two elaborations warnings for each hardware model in your design. This is because Cyclone divides the circuit into two types of blocks, sequential and combinatorial. At every clock edge the sequential blocks get executed first, and then the combinatorial blocks get executed. The hardware model is neither fully sequential nor fully combinatorial, so Cyclone declares it as a special block. Cyclone discourages you from using special blocks by issuing warnings; however, special blocks are fine for hardware models, so you can ignore the warnings for hardware models.

"Keys Do Not Match" Error Message

If you receive the following error message during Cyclone simulation:

```
LM_HW integration error: Keys do not match
```

this indicates that you do not have a consistent set of genInterface output; for example, the liblm_hw.so file was not generated in the same genInterface session as one or more of the .vhd files, so the information is not valid for simulation. This can occur if you run genInterface more than once on the same hardware model files with the overwrite_files option left at its default setting of "no."

To correct this error, refer to "Overwriting Existing Files" on page 185; then rerun genInterface on the complete set of hardware models you want to use in the Cyclone simulation. Analyze and elaborate the new genInterface output before proceeding with your simulation.

Cyclone genInterface Setup Files

This section describes the ModelAccess for Cyclone setup file syntax and usage.

Setup File Definition

Two sets of setup files are provided for genInterface:

1. Model-dependent setup information is stored in the following file:

   ```
   $MA_CY/setup/synopsys_lm_hw.setup
   ```

   This file is typically copied by each user from this central location into the their own directory, where it can be edited for a particular session. Use of this setup file is described in "Editing the Setup File" on page 184.

2. System-dependent setup information is stored in these three platform-specific files:

   ❍ $MA_CY/setup/synopsys_lm_hw.setup.hp700

   ❍ $MA_CY/setup/.synopsys_lm_hw.setup.solaris

   These files are provided to allow a system administrator to update compiler and linker information, if necessary. If the ANSI C compiler (acc) is used, then no editing of these files should be required.

Search Path

Upon invocation, the genInterface program searches for the synopsys_lm_hw.setup file and the.synopsys_lm_hw.setup.*platform* files in the following locations, in the order listed:

1. Current working directory (files preceded by ".")

2. User's home directory (files preceded by ".")

3. $MA_CY/setup (files without a "." prefix).

System-Dependent Setup Options

The system-dependent setup files allow you to change default settings for genInterface. A sample of the HP-UX version of the file is shown in Figure 12.

```
        # compiler acc                  # default acc

        # cflags values are cumulative
        cflags +z +DA1.1 +DS2.0
        # cflags -fPIC                   # default -fPIC
        # cflags -DLM_HW_DEBUG
        # cflags -DLM_HW_PIN_DEBUG

        # linker ld                      # default ld

        # lflag values are cumulative
        # lflags -b                      # default -b
```

**Figure 12: Sample System-Dependent Setup File
(.synopsys_lm_hw.setup.hp700)**

compiler

The genInterface program requires access to a C compiler. The ANSI C compiler (acc), which is required for use with Cyclone, is also recommended for genInterface.

By default, genInterface searches for the acc compiler. If this is not correct for your environment, update the information following the compiler keyword, as follows:

```
compiler gcc
```

cflags

The -DLM_HW_DEBUG and -DLM_HW_PIN_DEBUG flags create a special, debug version of the Cyclone interface. By default, these options are always commented out (preceded by a #). There is no need to enable these options unless you are requested to do so by the Synopsys Technical Support Center.

linker and lflags

By default, genInterface uses the ld linker with the flags specified in each platform-dependent setup file. If you choose to use another linker, contact the Synopsys Technical Support Center. For instructions, refer to "Getting Help" on page 14.

Cyclone genInterface Processing

This section describes how genInterface processes input Shell Software to create the VHDL shell needed by Cyclone. Note that genInterface ignores .NAM files when processing pin names.

Rules for Signal Renaming

Because certain characters and keywords are permitted in Shell Software pin names but are illegal as VHDL signal names, genInterface must convert these signal names in order to generate correct VHDL. The rules that genInterface uses to map the signal names to legal values are explained in the following sections.

The following rules are applied by default. You can explicitly specify the mapping for any signal name by using the pin_name_ovr statement in the genInterface setup file, as described in "Overwriting Pin Names Per Model" on page 186.

Renaming Buses

The genInterface program sorts all pins in ascending order. Groups of pins having the same basename are combined into buses. If part of the bus is in a different mode (for example, inout and out), then the bus is split by mode, and the basenames are made unique.

For example, consider the following bus, described in the Shell Software:

```
out_pin
  A[21:12]          = 110,109,108,107 106,105,104,103,99,98
....
io_pin
  A[11:6]           =  97, 96, 95, 94, 93, 92
  A[29:22]          = 119,118,117,116,115,114,113,112
```

This is converted as follows in the generated VHDL file:

```
A  : INOUT std_logic_vector(11 downto 6);
AA : OUT std_logic_vector(21 downto 12);
AB : INOUT std_logic_vector(29 downto 22);
```

Replacing Special Characters

Cyclone allows only alphanumeric characters and underscores (_) in the generation of valid signal names. Hardware model Shell Software allows special characters such as slash (/), asterisk (*), minus (-), and underscore (_).

The genInterface program follows the conversion rules specified in Table 21.

Table 21: Rules for Special Character Mapping

Character in Shell Software	Conversion when at beginning of name	Conversion when appearing within name	Conversion when at end of name
Slash (/)	SL_	_SL_	_SL
Star (*)	ST_	_ST_	_ST
Minus (-)	NE_	_NE_	_NE

Table 21: Rules for Special Character Mapping (Continued)

Character in Shell Software	Conversion when at beginning of name	Conversion when appearing within name	Conversion when at end of name
Underscore (_)	UN_	_ (no conversion)	_UN
Any other special character	Random alphanumeric	Random alphanumeric	Random alphanumeric

For every generated name, genInterface then compares the name with the present list of names. If there is a match, a random string is added at the end of the name until it is unique.

The following examples illustrate how genInterface converts existing Shell Software names in the generated VHDL file.

```
'-CS'        is converted to:       NE_CS

'-BM0/BYTE'  is converted to:       NE_BM0_SL_BYTE

'DT/-R'      is converted to:       DT_SL_NE_R

'-TOUT2/-IRQ3'is converted to:      NE_TOUT2_SL_NE_IRQ3

'IO94_-RCLK_-BUSY/RDY'is converted to:IO94_NE_RCLK_NE_BUSY_SL_RDY
```

Keyword Replacement

Certain VHDL keywords cannot be used as signal names (for example, IN, OUT, PROCESS). The genInterface program scans the list of signal names replaces disallowed *keyword* is found, that name is replaced by S_*keyword*. If another signal already exists by this name, a random string is appended to the end of the present signal name.

For example, the Shell Software notation:

```
IN[6:1]
```

would be converted as follows:

```
S_IN : IN std_logic_vector(6 downto 1);
```

10

Using Leapfrog with Synopsys Models

Overview

This chapter explains how to use SmartModels, FlexModels, MemPro models, and hardware models with Leapfrog. The procedures are organized into the following major sections:

- "Setting Environment Variables" on page 195
- "Using SmartModels with Leapfrog" on page 197
- "Using FlexModels with Leapfrog" on page 198
- "Using MemPro Models with Leapfrog" on page 198
- "Using Hardware Models with Leapfrog" on page 201

Setting Environment Variables

First, set the basic environment variables. If you are not using one of the model types, skip that step. In some cases the procedures that follow in this chapter include steps for setting additional environment variables.

1. Set the LMC_HOME variable to the location of your SmartModel. FlexModel, and MemPro model installation tree, as follows:

   ```
   % setenv LMC_HOME path_to_models_installation
   ```

2. Set the LM_LICENSE_FILE or SNPSLMD_LICENSE_FILE environment variable to point to the product authorization file, as shown in the following example:

   % **setenv LM_LICENSE_FILE** *path_to_product_authorization_file*

   % **setenv SNPSLMD_LICENSE_FILE** *path_to_product_authorization_file*

   You can put license keys for multiple products (for example, SmartModels and hardware models) into the same authorization file. If you need to keep separate authorization files for different products, use a colon-separated list (UNIX) or semicolon-separated list (NT) to specify the search path in your variable setting.

   Caution

   Do not include la_dmon-based authorizations in the same file with snpslmd-based authorizations. If you have authorizations that use la_dmon, keep them in a separate license file that uses a different license server (lmgrd) process than the one you use for snpslmd-based authorizations.

3. If you are using the hardware modeler, set the LM_DIR and LM_LIB environment variables, as shown in the following examples:

   % **setenv LM_DIR** *hardware_model_install_path***/sms/lm_dir**
   % **setenv LM_LIB** *hardware_model_install_path*/**sms/models: **
   *hardware_model_install_path***/sms/maps**

   If you put your models in a directory other than the default of /sms/models, modify the above variable setting accordingly.

4. Set the CDS_VHDL variable to the location of your Leapfrog installation and make sure that Leapfrog is set up properly in your environment.

5. Depending on your platform, set your load library variable to point to the platform-specific directory in $LMC_HOME, as shown in the following examples:

   Solaris:

   % **setenv LD_LIBRARY_PATH $LMC_HOME/lib/sun4Solaris.lib:$LD_LIBRARY_PATH**

   Linux:

   % **setenv LD_LIBRARY_PATH $LMC_HOME/lib/x86_linux.lib:$LD_LIBRARY_PATH**

   AIX:

   % **setenv LIBPATH $LMC_HOME/lib/ibmrs.lib:$LIBPATH**

   HP-UX:

   % **setenv SHLIB_PATH $LMC_HOME/lib/hp700.lib:$SHLIB_PATH**

   NT:

   Make sure that %LMC_HOME%\lib\pcnt.lib is in the Path user variable.

Using SmartModels with Leapfrog

To use SmartModels with Leapfrog, follow this procedure:

1. To build the SmartModel interface, first cd to the lib directory in the Cadence tree and execute the lfsmGen command:

   ```
   % cd $CDS_VHDL/lib
   % lfsmGen
   ```

   This step produces a liblfsm.so.1.1 file on SunOS, liblfsm.so on Solaris, liblfsm.sl on HP-UX, and liblfsm.a on AIX.

2. To build the VHDL libraries needed to simulate with SmartModels, cd to the $CDS_VHDL/bin directory and execute the lfsmLibPckGen command:

   ```
   % cd $CDS_VHDL/bin
   % lfsmLibPckGen
   ```

   This step produces a lfsmLibPck file.

3. Determine where you want the SmartModel VHDL libraries to go and cd to that location. Then execute the lfsmLibPck you built in the previous step.

   ```
   % lfsmLibPck
   ```

   This step can take 30 minutes or more. When the process completes you get the following two VHDL files that you need to analyze in LeapFrog:

   - SMILibrary.vhd

   - SMpackage.vhd

   The SMILibrary.vhd file contains entity-architecture pairs for all SmartModels in your $LMC_HOME tree. These include the generics used to configure SmartModel SWIFT parameters.

   Note that SmartModels are identified as follows:

   ```
   attribute FOREIGN of SmartModel : architecture is
   "LFSM:LFSmartModels";
   ```

   The SMpackage.vhd file contains component declarations for the SmartModels. You must specify required SWIFT parameters for every generic in a component that you want to simulate within Leapfrog. For more information on required SWIFT parameters, refer to .

⚡ **Caution**
On the HP platform all users must use the same $LMC_HOME in order to prevent erroneous simulation results or fatal simulation errors. This precaution is necessary because the lfsmGen command modifies the liblfsm.sl file to require $LMC_HOME, and on the HP platform the liblfsm.sl file references an absolute path name to libswift.sl, the SWIFT library. When the absolute path name is not the same as the user's $LMC_HOME, the result is the loading of two different versions of libswift.sl during the simulation.

Using FlexModels with Leapfrog

To use Leapfrog with FlexModels, follow the same steps laid out for SmartModels in "Using SmartModels with Leapfrog" on page 197. On Leapfrog, you use FlexModels with Direct C Control. For information on the required SWIFT parameters for FlexModels (which differ from regular SmartModels) and how to use Direct C Control, refer to "Using FlexModels with SWIFT Simulators" on page 24.

Using MemPro Models with Leapfrog

To use MemPro models with Leapfrog, follow this procedure:

1. If you have built your own Foreign Model Interface (FMI) shared library, you need to perform this step in order to combine your shared library with the MemPro FMI shared library.

⏱ **Attention**
If you do not build your own FMI library, skip to Step 3.

Leapfrog binds in only one shared FMI library at runtime. If your design uses FMI, you need to build a new FMI shared library that contains your library and the MemPro library. A MemPro archive library can be found at:

HP-UX

        $LMC_HOME/lib/hp700.lib/libfmi_ar.a

Solaris

        $LMC_HOME/lib/sun4Solaris.lib/libfmi_ar.a

You must create a new archive that includes the MemPro archive, library table file declaration object file, and your archive. Detailed instructions for this process can be found in "Foreign Model Integration" in the *Cadence Leapfrog C Interface User Guide*.

As shown in the following example, you must combine the contents of the MemPro library table file with your own FMI application library table:

```
#include <fmilib.h>

extern fmiModelTableT CpipeModelTable;
extern fmiModelTableT myFMITable;

fmiLibraryTableT fmiLibraryTable = {
  {"Cpipe", CpipeModelTable},
  {"myFMIlib", myFMITable},
  {0, 0}
};
```

Link the MemPro archive library with the new library table object file and any other FMI application object files you wish to include, following the instructions in the *Cadence C Interface User Guide*.

The following examples show compiling the library table object files and linking MemPro libfm_ar.a with the library table object file and the FMI application you developed, shown in the examples as new_FMI_table.o and *your_archive*.a:

HP-UX

```
% /bin/cc -D_NO_PROTO -c +Z -I$CDS_VHDL/include new_FMI_table.c
% /bin/ld -b -o libfmi.sl new_FMI_table.o your_archive.a \
    $LMC_HOME/lib/hp700.lib/libfmi_ar.a
```

Solaris

```
% /opt/SUNWspro/bin/cc -c -KPIC -I$CDS_VHDL/include new_FMI_table.c
% /opt/SUNWspro/bin/cc -G -o libfmi.so new_FMI_table.o \
    your_archive.a $LMC_HOME/lib/sun4Solaris.lib/libfmi_ar.a
```

2. Set up the library search path to locate the MemPro shared library.

Attention

You must add the MemPro shared library to the beginning of your SHLIB_PATH or LD_LIBRARY_PATH contents. If the MemPro shared library is added to the tail of the path list, the library search order will be incorrect and Leapfrog will not simulate properly.

HP-UX

```
% setenv SHLIB_PATH $LMC_HOME/lib/hp700.lib:$SHLIB_PATH
```

Solaris

```
% setenv LD_LIBRARY_PATH \
    $LMC_HOME/lib/sun4Solaris.lib:$LD_LIBRARY_PATH
```

3. Create a logical library named slm_lib. The default physical library mapping for this is $LMC_HOME/sim/leapfrog/lib. If you do not want to install the library in the LMC_HOME tree, you can create it in any other location.

   Add the following line to your cds.lib file:

   ```
   define slm_lib $LMC_HOME/sim/leapfrog/lib
   ```

4. Compile the MemPro VHDL files into logical library slm_lib. Note that rdramd_pkg.vhd is not required unless you are simulating RDRAM MemPro models.

   ```
   % cv -w slm_lib $LMC_HOME/sim/leapfrog/src/slm_hdlc.vhd
   % cv -w slm_lib $LMC_HOME/sim/leapfrog/src/mempro_pkg.vhd
   % cv -w slm_lib $LMC_HOME/sim/leapfrog/src/rdramd_pkg.vhd
   ```

5. After generating a model using MemPro, compile the VHDL code for the model, as shown in the following example:

   ```
   % cv mymem.vhd
   ```

6. If you compiled the model to a library (SLM_MODELS in these examples), add a LIBRARY statement to your testbench:

   ```
   LIBRARY SLM_MODELS
   ```

   This makes models in SLM_MODELS accessible from your design. You can refer to the model using a standard VHDL convention, such as a CONFIGURATION statement:

   ```
   CONFIGURATION mymem_a OF mymem_tst IS
       FOR atest
               FOR ALL : mymem use CONFIGURATION SLM_MODELS.mymem_behav;
               END FOR;
       END FOR;
   END mymem_a;
   ```

7. Add LIBRARY and USE statements to your testbench:

   ```
   LIBRARY SLM_LIB
   USE SLM_LIB.mempro_pkg.all
   ```

   This also provides access to MemPro testbench commands.

   For more information on using the MemPro testbench interfaces, refer to the "HDL Testbench Interface" and "C Testbench Interface" chapters in the *MemPro User's Manual*.

8. Instantiate MemPro models in your design. Define ports and generics as required. For information on generics used with MemPro models, refer to "Instantiating MemPro Models" on page 32. For information on message levels and message level constants, refer to "Controlling MemPro Model Messages" on page 33.

9. Compile your design as shown in the following example:

```
% cv -w work design_name
```

10. Invoke Leapfrog on your design as shown in the following example:

```
% sv design_name
```

Using Hardware Models with Leapfrog

To use hardware models with Leapfrog, follow this procedure. For the latest information on supported features, refer to the Cadence documentation.

1. Make sure Leapfrog is set up properly and all required environment variables are set, as explained in "Setting Environment Variables" on page 195.

2. Add the hardware model install tree to your path variable, as shown in the following example:

```
% set path=(/install/sms/bin/platform/ $path)
```

3. Run the install.sh script so that the hardware models option is turned on and the LMlibrary.vhd, LMpackage.vhd, and LMproc.vhd are created. Also make sure the cds.lib file points to the correct libraries, including LMSFI, which is located in the install area.

4. Create your own library directory for files that will be generated for the hardware model. In the Leapfrog Notebook, you can set up the directory so that the library files get placed in there by using the Library menu.

5. Generate a custom Leapfrog simulator executable (sv) to work with the hardware model and imported Verilog model. This is done in the install.sh. The install generates a new svvlog.exe. When this completes, you are ready to run using the custom sv executable.

Leapfrog Example with TILS299 Hardware Model

The following example uses the TILS299 hardware model to show how to set up hardware models for use with Leapfrog:

1. Create a testbench to instantiate the hardware model (for example, TB_TILS299.vhd).

2. Invoke Leapfrog. This brings up the Notebook window, where you can compile, elaborate, and simulate your VHDL testbench. Type "leapfrog&".

3. In the Notebook window, select your .vhd testbench file and click on the compile button.

4. Once compiled, use the Unit menu and select elaborate. Fill in the Design Unit with your compiled information and fill the snapshot with SIM. For example, mywork is the directory specified to place compiled work, so we use mywork.TB.TILS299.

5. To simulate, select simulate from the Unit menu and fill in the information for simulating in the snapshot line. For example: mywork.tb_tils299:test/sim. This syntax can also be found at the end of the elaborate.

6. To see waveforms, use the simulator window to select Tools > Navigator. When you select the hardware model instance in the subscopes, the signal pins come up in the Object window. Select all the signals to be traced in the waveforms and right-click to select Set trace simple.

7. Go back to the simulator window and select Tools > Waveview. When the cwaves window comes up with all your signal pins, click on run on the simulator window to simulate.

Leapfrog Utilities

The following hardware modeler simulator commands are supported in Leapfrog:

lm_log_test_vectors (*"instance_name"***, 1/0,** *"filename")***;**

enables (1) or disables (0) vector logging for hardware models.

lm_timing_measurements (*"instance_name"***, 1/0,** *"filename"***);**

Enables (1) or disables (0) timing measurements for hardware models.

lm_enable_timing_checks ([*device_name(s).***...**]**)**

Enables timing checks for hardware models.

lm_disable_timing_checks ([*device_name(s)***....**]**)**

Disables timing checks for hardware models.

lm_unknowns (*"option=value"***, ...device_or_pin);**

Determines how unknown values are handled by hardware models. Supported options include:

- propagate=yes/no
- value=previous/high/low

- sequence_count=0-20

- random_seed=0-65535

- device_or_pin

lm_loop_instance (*[instance_name]*)**;**

Makes the hardware modeler enter loop mode, where it continually replays the pattern history of the specified instance.

**lm_pam_shortage("actions=save/sleep/finish/free/suspend/drop_faults",
     "sleep_minutes=n, "sleep_count=n", "save_file=filename");**

Lets you specify the action the hardware modeler is to take when it has used up most of the available pattern memory.

lm_pattern_history (*[device_name(s)....]*)

Saves the pattern memory for a private device instance.

Examples

You can use any of these utilities by calling them from VHDL code or invoking them from the debugger prompt with an lm_*procedure* call.

Example call from VHDL code:

```
Variable ret : Natural;
...
ret := lm_log_test_vectors(":U1",1,"U1.VEC");
wait for 800 ns;
ret := lm_log_test_vectors(":U1",0,"U1.VEC");
```

Example invocation from the debugger prompt with lm_*procedure* call:

```
> call lm_log_test_vectors(U1,1,U1.VEC)
```

11

Using NC-VHDL with Synopsys Models

Overview

This chapter explains how to use SmartModels, FlexModels, and MemPro models with NC-VHDL. The procedures are organized into the following major sections:

Setting Environment Variables

First, set the basic environment variables. If you are not using one of the model types, skip that step. In some cases the procedures that follow in this chapter include steps for setting additional environment variables.

1. Set the LMC_HOME variable to the location of your SmartModel, FlexModel, and MemPro model installation tree, as follows:

   ```
   % setenv LMC_HOME path_to_models_installation
   ```

2. Set the LM_LICENSE_FILE or SNPSLMD_LICENSE_FILE environment variable to point to the product authorization file, as shown in the following example:

   ```
   % setenv LM_LICENSE_FILE path_to_product_authorization_file
   % setenv SNPSLMD_LICENSE_FILE path_to_product_authorization_file
   ```

You can put license keys for multiple products (for example, FlexModels and MemPro models) into the same authorization file. If you need to keep separate authorization files for different products, use a colon-separated list (UNIX) or semicolon-separated list (NT) to specify the search path in your variable setting.

⚡ **Caution**——————————————————————————————————————

Do not include la_dmon-based authorizations in the same file with snpslmd-based authorizations. If you have authorizations that use la_dmon, keep them in a separate license file that uses a different license server (lmgrd) process than the one you use for snpslmd-based authorizations.

3. Make sure that NC-VHDL is set up properly in your environment.

4. Depending on your platform, set your load library variable to point to the platform-specific directory in $LMC_HOME, as shown in the following examples:

Solaris:

```
% setenv LD_LIBRARY_PATH $LMC_HOME/lib/sun4Solaris.lib:$LD_LIBRARY_PATH
```

Linux:

```
% setenv LD_LIBRARY_PATH $LMC_HOME/lib/x86_linux.lib:$LD_LIBRARY_PATH
```

AIX:

```
% setenv LIBPATH $LMC_HOME/lib/ibmrs.lib:$LIBPATH
```

HP-UX:

```
% setenv SHLIB_PATH $LMC_HOME/lib/hp700.lib:$SHLIB_PATH
```

NT:

Make sure that %LMC_HOME%\lib\pcnt.lib is in the Path user variable.

Using SmartModels with NC-VHDL

To use SmartModels with NC-VHDL, follow this procedure:

1. Add the following line to your cds.lib file to specify the logical library sm_library for SmartModels, as shown in the following example:

```
DEFINE sm_library ./sm_library
```

2. Run the ncshell utility to generate a wrapper for the model that you want to use, as shown in the following example:

```
% ncshell -import swift into vhdl model -work sm_library
```

This step produces a wrapper file (*model*.vhd) and a component declaration (*model*_comp.vhd) for the specified model in the sm_library work directory.

If you want to generate wrappers for all SmartModels in your $LMC_HOME tree, add the -all switch to the ncshell invocation. In this case, ncshell creates one file (shell.vhd) that contains all the model wrappers and another file (component.vhd) that contains the component declarations.

Hint

NC-VHDL also works with wrappers created for Leapfrog. If you want to reuse SmartModel wrappers created for Leapfrog, use ncvhdl to recompile the SMLibrary.vhd and SMpackage.vhd files. For more information on using SmartModels with Leapfrog, refer to "Using SmartModels with Leapfrog" on page 197.

---

3. Add LIBRARY and USE statements to your testbench:

```
library sm_library;
use sm_library.component.all;
```

4. Instantiate SmartModels in your design using the wrapper files that you generated in Step 2. For information on required configuration parameters and instantiation examples, refer to "Using SmartModels with SWIFT Simulators" on page 18.

5. Compile the other VHDL files into the work library, as shown in the following example:

```
% ncvhdl -w work testbench.vhd
```

6. Elaborate your design as shown in the following example:

```
% ncelab cfgtest
```

7. If you are using any SmartCircuit models in your design, set the LMC_TIMEUNIT environment variable to -12 for 1 ps resolution, as shown in the following example:

```
% setenv LMC_TIMEUNIT -12
```

This sets a global timing resolution for all SmartModels in your simulation. If this variable is not set, the default timing resolution is 100 ps, which is the resolution used by most SmartModels. To see if a model is a SmartCircuit model, refer to the model datasheet. For more information on the LMC_TIMEUNIT environment variable, refer to the Cadence documentation for NC-VHDL.

8. Invoke the NC-VHDL simulator on your design as shown in the following example:

```
% ncsim design
```

Using FlexModels with NC-VHDL

To use FlexModels with NC-VHDL, follow this procedures:

1. Add the following lines to your cds.lib file:

   define slm_lib *slm_lib_path*
   define work *work_lib_path*

2. Generate a VHDL wrapper file for the model by invoking ncshell, as shown in the following example:

   % **ncshell -import swift -into vhdl** *model_**fx** -**nocompile -work slm_lib**

3. Create a working directory and run flexm_setup to make copies of the model's interface and example files there, as shown in the following example:

   % **$LMC_HOME/bin/flexm_setup -dir** *workdir model_***fx**

You must run flexm_setup every time you update your FlexModel installation with a new model version. Table 22 describes the FlexModel NC-VHDL interface and example files that the flexm_setup tool copies.

Table 22: FlexModel NC-VHDL Files

File Name	Description	Location
*model*_pkg.vhd	Model command procedure calls for HDL Command Mode.	*workdir*/src/vhdl/
*model*_user_pkg.vhd	Clock frequency setup and user customizations.	*workdir*/src/vhdl/
*model*_fx_comp.vhd	Component definition for use with the *model* entity defined in the SWIFT wrapper file. This is put in a package named "COMPONENTS" when compiled.	*workdir*/examples/vhdl/
model.vhd	A bus-level wrapper around the SWIFT model. This allows you to use vectored ports for the model in your testbench. This file assumes that the "COMPONENTS" package has been installed in the logical library "slm_lib".	*workdir*/examples/vhdl/
*model*_tst.vhd	A testbench that instantiates the model and shows how to use basic model commands.	*workdir*/examples/vhdl/

4. Update the clock frequency supplied in the *model*_user_pkg.vhd file in your working directory to correspond to the desired clock period for the model. After you run flexm_setup this file is located in:

```
workdir/src/vhdl/model_user_pkg.vhd
```

where *workdir* is your working directory.

5. Create a logical library named slm_lib. The default physical library mapping for this is $LMC_HOME/sim/*simulator*/lib; however, you can put the physical library anywhere you want.

6. Add LIBRARY and USE statements to your testbench:

```
library slm_lib;
use slm_lib.flexmodel_pkg.all;
use slm_lib.model_pkg.all;
use slm_lib.model_user_pkg.all;
```

For example, you would use the following statement for the tms320c6201_fx model:

```
use slm_lib.tms320c6201_pkg.all;
use slm_lib.tms320c6201_user_pkg.all;
```

7. Instantiate FlexModels in your design, defining the ports and generics as required (refer to the example testbench supplied with the model). You use the supplied bus-level wrapper (*model*.vhd) in the top-level of your design to instantiate the bit-blasted wrapper generated in Step 2 (*model*_fx.vhd) using ncshell.

Example using bus-level wrapper (*model*.vhd) without timing:

```
U1: model
    generic map (FlexModelID => "TMS_INST1")
    port map ( model ports );
```

Example using bus-level wrapper (*model*.vhd) with timing:

```
U1: model
        generic map (FlexModelID => "TMS_INST1",
        FlexTimingMode => FLEX_TIMING_MODE_ON,
        TimingVersion  => "timingversion",
        DelayRange     => "range")
    port map ( model ports );
```

8. Compile the FlexModel VHDL files into logical library slm_lib as follows:

```
% ncvhdl -w slm_lib $LMC_HOME/sim/ncvhdl/src/slm_hdlc.vhd
% ncvhdl -w slm_lib $LMC_HOME/sim/ncvhdl/src/flexmodel_pkg.vhd
% ncvhdl -w slm_lib workdir/src/vhdl/model_user_pkg.vhd
% ncvhdl -w slm_lib workdir/src/vhdl/model_pkg.vhd
% ncvhdl -w slm_lib workdir/examples/vhdl/model_fx_comp.vhd
% ncvhdl -w slm_lib model_fx.vhd
% ncvhdl -w slm_lib workdir/examples/vhdl/model.vhd
% ncvhdl-w work testbench
```

9. Elaborate your design as shown in the following example:

```
% ncelab cfgtest
```

10. Invoke the NC-VHDL simulator as shown in the following example:

```
% ncsim design
```

Using MemPro Models with NC-VHDL

To use MemPro models with NC-VHDL, follow this procedure:

1. If you have built your own Foreign Model Interface (FMI) shared library, perform this step.

⏰ **Attention**

If you do not build your own FMI library, skip to Step 2.

NC-VHDL binds in only one shared FMI library at runtime. If your design uses FMI, you need to build a new FMI shared library that contains your library and the MemPro library. A MemPro archive library can be found at:

HP-UX

```
$LMC_HOME/lib/hp700.lib/libfmi_ar.a
```

Solaris

```
$LMC_HOME/lib/sun4Solaris.lib/libfmi_ar.a
```

You must create a new archive that includes the MemPro archive, library table file declaration object file, and your archive. Detailed instructions for this process can be found in the "Foreign Model Integration" chapter of the *Affirma NC VHDL Simulator C Interface User Guide*.

As shown in the following example, you must combine the contents of the MemPro library table file with your own FMI application library table:

```
#include <fmilib.h>

extern fmiModelTableT CpipeModelTable;
extern fmiModelTableT myFMITable;

fmiLibraryTableT fmiLibraryTable = {
  {"Cpipe", CpipeModelTable},
  {"myFMIlib", myFMITable},
  {0, 0}
};
```

Link the MemPro archive library with the new library table object file and any other FMI application object files you wish to include, following the instructions in the *Cadence C Interface User Guide*.

The following examples show compiling the library table object files and linking MemPro libfm_ar.a with the library table object file and the FMI application you developed, shown in the examples as new_FMI_table.o and your_archive.a:

HP-UX

```
% /bin/cc -D_NO_PROTO -c +Z -I$CDS_VHDL/include new_FMI_table.c
% /bin/ld -b -o libfmi.sl new_FMI_table.o your_archive.a \
    $LMC_HOME/lib/hp700.lib/libfmi_ar.a
```

Solaris

```
% /opt/SUNWspro/bin/cc -c -KPIC -I$CDS_VHDL/include new_FMI_table.c
% /opt/SUNWspro/bin/cc -G -o libfmi.so new_FMI_table.o \
    your_archive.a $LMC_HOME/lib/sun4Solaris.lib/libfmi_ar.a
```

2. Create a logical library named slm_lib. The default physical library mapping for this is $LMC_HOME/sim/ncvhdl/lib. If you do not want to install the library in the LMC_HOME tree, you can create it in any other location.

   Add the following line to your cds.lib file:

   ```
   define slm_lib $LMC_HOME/sim/ncvhdl/lib
   ```

3. Compile the MemPro VHDL files into logical library slm_lib. Note that rdramd_pkg.vhd is not required unless you are simulating RDRAM MemPro models.

   ```
   % ncvhdl -w slm_lib $LMC_HOME/sim/ncvhdl/src/slm_hdlc.vhd
   % ncvhdl -w slm_lib $LMC_HOME/sim/ncvhdl/src/mempro_pkg.vhd
   % ncvhdl -w slm_lib $LMC_HOME/sim/ncvhdl/src/rdramd_pkg.vhd
   ```

4. After generating a model using MemPro, compile the VHDL code for the model, as shown in the following example:

   ```
   % ncvhdl mymem.vhd
   ```

5. If you compiled the model to a library (SLM_MODELS in these examples), add a LIBRARY statement to your testbench:

   ```
   LIBRARY SLM_MODELS
   ```

   This makes models in SLM_MODELS accessible from your design. You can refer to the model using a standard VHDL convention, such as a CONFIGURATION statement:

   ```
   CONFIGURATION mymem_a OF mymem_tst IS
      FOR atest
            FOR ALL : mymem use CONFIGURATION SLM_MODELS.mymem_behav;
            END FOR;
      END FOR;
   END mymem_a;
   ```

6. Add LIBRARY and USE statements to your testbench:

   ```
   LIBRARY SLM_LIB
   USE SLM_LIB.mempro_pkg.all
   ```

   This also provides access to MemPro testbench commands.

   For more information on using the MemPro testbench interfaces, refer to the "HDL Testbench Interface" and "C Testbench Interface" chapters in the *MemPro User's Manual*.

7. Instantiate MemPro models in your design. Define ports and generics as required. For information on generics used with MemPro models, refer to "Instantiating MemPro Models" on page 32. For information on message levels and message level constants, refer to "Controlling MemPro Model Messages" on page 33.

8. Compile your design as shown in the following example:

   ```
   % ncelab -w work design_name
   ```

9. Invoke NC-VHDL on your design as shown in the following example:

   ```
   % ncsim design_name
   ```

Using Hardware Models with NC-VHDL

To use hardware models with NC-VHDL, follow this procedure. For the latest information on supported features, refer to the Cadence documentation.

1. Make sure NC-VHDL is set up properly and all required environment variables are set, as explained in "Setting Environment Variables" on page 205.

2. Add the hardware model install tree to your path variable, as shown in the following example:

   ```
   % set path=(/install/sms/bin/platform/ $path)
   ```

3. Create your own library directory for files that will be generated for the hardware model.

4. Run the ncshell command to generate .vhd wrapper files, as shown in the following example:

   ```
   % ncshell -import lmsfi -into vhdl models/TILS299/TILS299.MDL
   ```

   You can also use the -all switch to create .vhd files for multiple hardware models.

NC-VHDL Example with TILS299 Hardware Model

The following example uses the TILS299 hardware model to show how to set up hardware models for use with NC-VHDL:

1. Create a testbench to instantiate the hardware model (for example TB_TILS299.vhd).

2. Run ncvhdl to compile your .vhd files, as shown in the following example:

   ```
   % ncvhdl TILS299.vhd TILS299_comp.vhd TB_TILS299.vhd
   ```

3. Elaborate the design, as shown in the following example:

   ```
   % ncelab -messages work.tb_tils299:test
   ```

4. Invoke the NC-VHDL simulator, as shown in the following example:

   ```
   % ncsim -gui work.tb_tils299:test
   ```

NC-VHDL Utilities

The following hardware modeler simulator commands are supported in NC-VHDL:

lm_log_test_vectors (*"instance_name"***, 1/0,** *"filename")***;**

Enables (1) or disables (0) vector logging for hardware models.

lm_timing_measurements (*"instance_name"***, 1/0,** *"filename"***);**

Enables (1) or disables (0) timing measurements for hardware models.

lm_enable_timing_checks ([*device_name(s)***....])**

Enables timing checks for hardware models.

lm_disable_timing_checks ([*device_name(s)***....])**

Disables timing checks for hardware models.

lm_unknowns (*"option=value"***, ...device_or_pin);**

Determines how unknown values are handled by hardware models. Supported options include:

- propagate=yes/no
- value=previous/high/low
- sequence_count=0-20
- random_seed=0-65535
- device_or_pin

lm_loop_instance (*[instance_name]***);**

Makes the hardware modeler enter loop mode, where it continually replays the pattern history of the specified instance.

lm_pam_shortage("actions=save/sleep/finish/free/suspend/drop_faults",
     "sleep_minutes=n, "sleep_count=n", "save_file=filename");

Lets you specify the action the hardware modeler is to take when it has used up most of the available pattern memory.

lm_pattern_history (*[device_name(s)....]***)**

Saves the pattern memory for a private device instance.

Examples

You can use any of these utilities by calling them from VHDL code or invoking them from the debugger prompt with an lm_*procedure* call.

Example call from VHDL code:

```
Variable ret : Natural;
...
ret := lm_log_test_vectors(":U1",1,"U1.VEC");
wait for 800 ns;
ret := lm_log_test_vectors(":U1",0,"U1.VEC");
```

Example invocation from the debugger prompt with lm_*procedure* call:

```
% ncsim> call lm_log_test_vectors :U0 1 299.VEC
```

12

Using QuickSim II with Synopsys Models

Overview

This chapter explains how to use SmartModels, FlexModels, and hardware models with QuickSim II. The procedures are organized into the following major sections:

- "Setting Environment Variables" on page 217

- "Using SmartModels and FlexModels with QuickSim II" on page 219

- "Using Hardware Models with QuickSim II" on page 244

☞ **Note**
MemPro models are not supported on QuickSim II.

Setting Environment Variables

First, set the basic environment variables. If you are not using one of the model types, skip that step. In some cases the procedures that follow in this chapter include steps for setting additional environment variables.

1. Set the LMC_HOME variable to the location of your SmartModel. FlexModel, and MemPro model installation tree, as shown in the following example:

   % **setenv LMC_HOME** *path_to_models_installation*

2. Set the LM_LICENSE_FILE or SNPSLMD_LICENSE_FILE environment variable to point to the product authorization file, as shown in the following example:

   % **setenv LM_LICENSE_FILE** *path_to_product_authorization_file*

   % **setenv SNPSLMD_LICENSE_FILE** *path_to_product_authorization_file*

You can put license keys for multiple products (for example, SmartModels and hardware models) into the same authorization file. If you need to keep separate authorization files for different products, use a colon-separated list (UNIX) or semicolon-separated list (NT) to specify the search path in your variable setting.

Caution

Do not include la_dmon-based authorizations in the same file with snpslmd-based authorizations. If you have authorizations that use la_dmon, keep them in a separate license file that uses a different license server (lmgrd) process than the one you use for snpslmd-based authorizations.

3. Set MGC_HOME to the location of your Mentor installation and make sure QuickSim II is set up properly in your environment.

       % **setenv MGC_HOME** *path_to_Mentor_installation*

4. If you are using the hardware modeler, set the LM_DIR and LM_LIB environment variables, as shown in the following examples:

       % **setenv LM_DIR** *hardware_model_install_path*/**sms/lm_dir**
       % **setenv LM_LIB** *hardware_model_install_path*/**sms/models: **
        hardware_model_install_path/**sms/maps**

   If you put your models in a directory other than the default of /sms/models, modify the above variable setting accordingly.

5. Depending on your platform, set your load library variable to point to the platform-specific directory in $LMC_HOME, as shown in the following examples:

   Solaris:

    % **setenv LD_LIBRARY_PATH $LMC_HOME/lib/sun4Solaris.lib:$LD_LIBRARY_PATH**

   Linux:

    % **setenv LD_LIBRARY_PATH $LMC_HOME/lib/x86_linux.lib:$LD_LIBRARY_PATH**

   AIX:

    % **setenv LIBPATH $LMC_HOME/lib/ibmrs.lib:$LIBPATH**

   HP-UX:

    % **setenv SHLIB_PATH $LMC_HOME/lib/hp700.lib:$SHLIB_PATH**

   NT:

   Make sure that %LMC_HOME%\lib\pcnt.lib is in the Path user variable.

Using SmartModels and FlexModels with QuickSim II

This section explains how to use SmartModels and FlexModels with QuickSim II. To use FlexModels with QuickSim II, you use Direct C Control. For information on Direct C Control, refer to "Using FlexModels with SWIFT Simulators" on page 24. The rest of this section explains required installation steps and how to use model symbols in the schematic capture front-end to the simulator. This information is organized in the following major subsections:

Installing the QuickSim II SWIFT Interface

Synopsys ships the part of QuickSim II that communicates with the SWIFT interface for versions of QuickSim II prior to the D.1 release. If you are using a version of QuickSim II prior to D.1, you must install the Mentor Graphics application software for each Mentor Graphics user tree.

> ⏰ **Attention**
>
> Beginning with version D.1 of QuickSim II, Mentor Graphics assumed responsibility for their integration of the SWIFT interface. If you are using version D.1 or higher, refer to the Mentor Graphics documentation for information about using SWIFT.

Every time you install or update Mentor Graphics application software, you must create a user tree for the SWIFT SmartModel Library. Use the MGC install tool to create duplicate Mentor Graphics user trees. User trees typically require between 10–20 MB of disk space. For questions about creating Mentor Graphics user trees, refer to the Mentor Graphics documentation. Follow these steps:

1. If you are using a version of QuickSim II prior to D1, for each Mentor Graphics home directory (user or master tree) that requires access to the SWIFT interface, execute the following command:

   UNIX

   ```
   % $LMC_HOME/bin/mgc_ins -m $MGC_HOME -l $LMC_HOME
   ```

   NT

   You will be running a mkns shell in the MGC environment; for more information on the mkns shell, refer to Mentor Graphics QuickSim II documentation.

In the Control Panel, set the following drives to the appropriate system environment variables:

```
DRIVE:/port LMC_HOME
DRIVE:/port MGC_HOME
```

2. Add "$LMC_HOME" followed by a blank line to your location map file.

3. Add SmartModel Library Menus to Design Architect. Normally, as part of installation, the Admin tool automatically adds SmartModel menus to Design Architect for the models you installed. Use the instructions in this section to add the SmartModel menus only if, after installation, you do not find SmartModel menu entries under the Design Architect "Libraries" pull-down menu. To include SmartModel menu selections in the Design Architect (DA) menus, follow these steps:

   a. Set the AMPLE_PATH environment variable. If this variable already exists, use one of these commands as appropriate:

      UNIX

      ```
      % setenv AMPLE_PATH ${AMPLE_PATH}:$LMC_HOME/special/qsim/menus
      ```

      If you have a custom userware directory, you can create links that point to "$LMC_HOME/special/qsim/menus/des_arch."

      NT

      DRIVE:/*path_to_menus*/

   b. If the AMPLE_PATH environment variable does not exist, use one of these commands as appropriate:

      UNIX

      ```
      % setenv AMPLE_PATH $LMC_HOME/special/qsim/menus
      ```

      If you have a custom userware directory, you can create links that point to "$LMC_HOME/special/qsim/menus/des_arch."

      NT

      DRIVE:/*path_to_menus*/

4. Generate the menus as shown in the appropriate example:

   UNIX

   ```
   % $LMC_HOME/bin/mgc_menu.pl
   ```

   NT

   ```
   > LMC_HOME%\bin\mgc_menu.cmd
   ```

Menu entries are created for the models that you have installed. If menu entries are missing for models you believe you have in your library, use the Admin tool to verify your installed models. If you change your model installation, rerun mgc_menu to update the menu to reflect the new model list.

After successful menu activation, the "Libraries" pull-down menu of the Schematic Editor contains an entry for "Logic Modeling SmartModel Library." If you have SimuBus models installed, the pull-down menu also contains an entry for "Logic Modeling SimuBus Products."

Using SmartModels/FlexModels with QuickSim II

This chapter provides information about using SmartModels (including FlexModels) with Mentor Graphics (MGC) tools. This information is organized in the following major sections:

- Schematic Capture
- Logic Simulation
- Custom Symbols

Schematic Capture

Adding a SmartModel Library model to a design schematic involves identifying the desired symbol, instantiating it, and then editing its properties as necessary. Synopsys supplies a complete Schematic Editor menu system in the QuickSim II environment that you can use to identify and instantiate a component. You can also instantiate symbols from the command line and edit property values interactively using Design Architect. This chapter provides information about both approaches to building a design, following an introductory discussion of the symbols and their properties.

For more information about Design Architect and the Schematic Editor, refer to the Mentor Graphics documentation.

Symbols

Synopsys provides symbols representing default package pinouts for each SmartModel. Some models have more than one symbol associated with them. This is true for:

- Models of simple logic gates, which are supplied with DeMorgan equivalent negative logic symbols in addition to standard symbols
- Models that offer both pin and bus symbols

For information about symbol compatibility with different versions of the Design Architect software, refer to the *SmartModel Library Release Notes.*

Pin and Bus Symbols

For many high pin-count parts, you get pin and bus symbols. Bus symbols may be more convenient to use than their pin equivalents, because they take up less real estate on the schematic and are easier to connect and to read. Figure 13 illustrates the differences between pin and bus symbols.
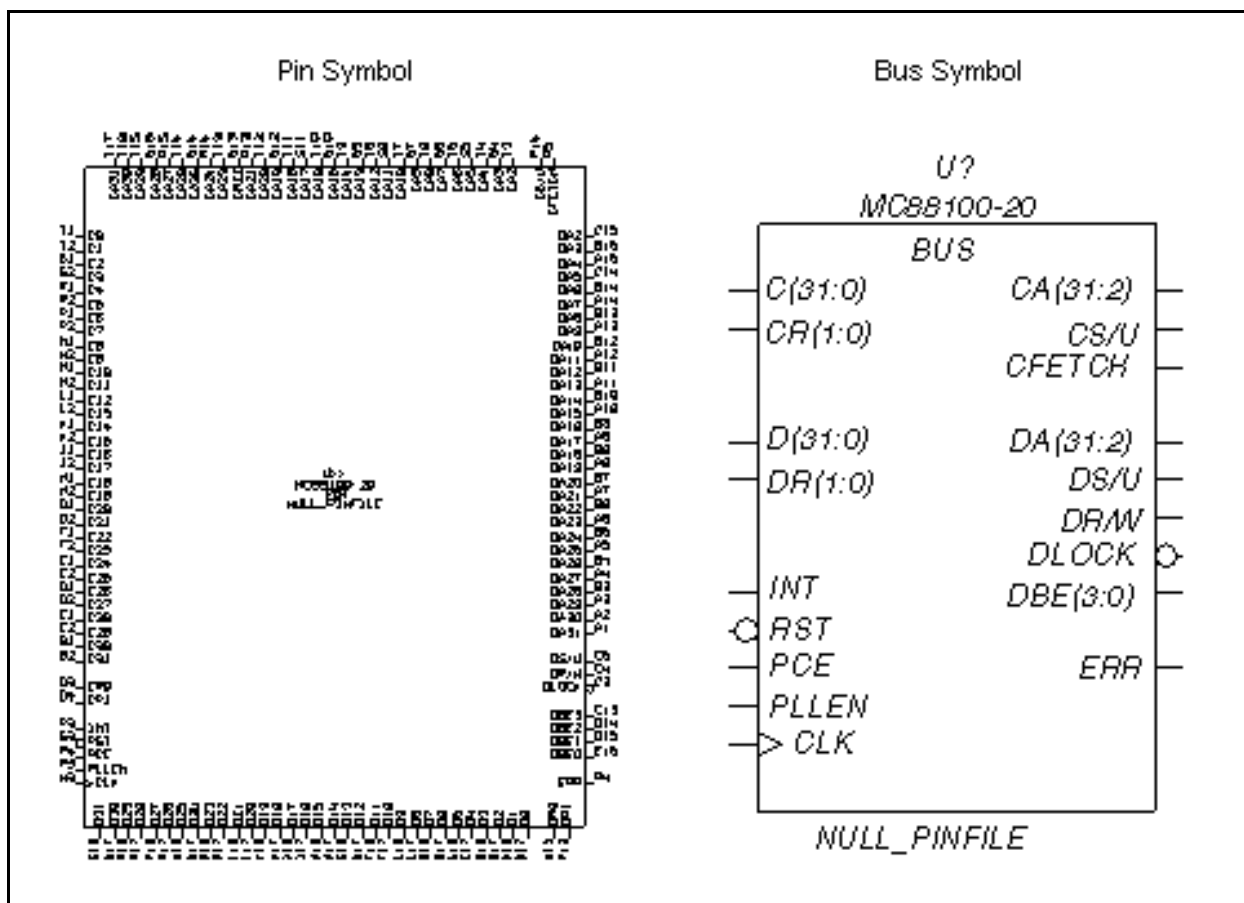
Figure 13: Sample Pin and Bus Symbols

Symbol Properties

Assigning specific values to symbol properties completes the definition of a SmartModel. The properties used on the symbols provided for the Mentor Graphics Design Architect environment include:

- Symbol properties used by SWIFT interface models
- Symbol properties required for simulation
- Optional symbol properties

You can edit symbol property values either with the Schematic Editor pop-up menu or by using the QuickSim II CHANGE TEXT VALUE command. These properties are hidden or visible depending on the visibility attributes selected by your library manager. Figure 14 illustrates the positions of the visible properties on a symbol supplied by Synopsys.
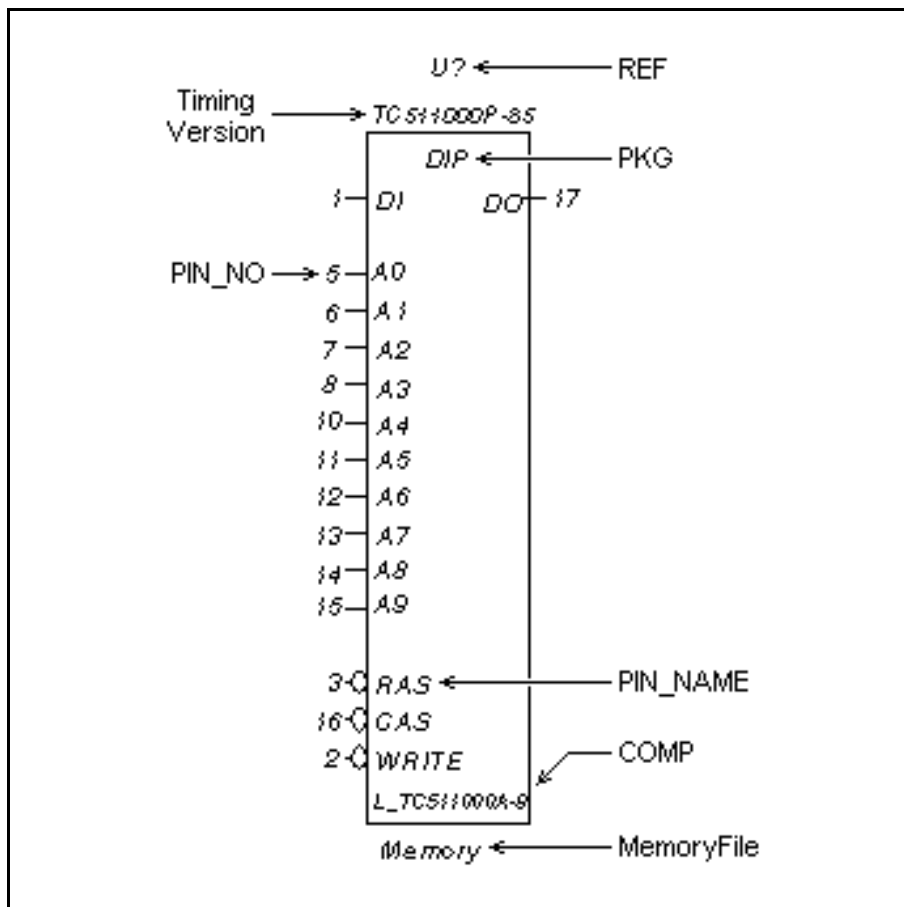
Figure 14: Visible Symbol Properties

Symbol Properties used by SWIFT Models

Table 23 lists symbol properties that are used by SWIFT interface models.

Table 23: Symbol Properties used by SWIFT Models

Property	Description
TimingVersion	Timing version to use with a model. Any value assigned to the TimingVersion property must be a valid timing version for that model.
MemoryFile	Name and path of a memory image file.

Table 23: Symbol Properties used by SWIFT Models (Continued)

Property	Description
PCLFile	Name and path of a compiled PCL file.
JEDECFile	Name and path of a JEDEC fuse map file.
SCFFile	Name and path of an MCF file for SmartCircuit models.

☞ **Note**

FlexModels use a slightly different set of symbol properties. For information on the required configuration properties for FlexModels, refer to "Using FlexModels with SWIFT Simulators" on page 24.

You can use either an absolute or relative path name to point to a file. If you use a relative path name it is resolved relative to the value of $MGC_WD.

Symbol Properties Required for Simulation

Table 24 lists symbol properties that are required for simulation.

Table 24: Symbol Properties Required for Simulation

Property	Description
MODEL	The MODEL property contains a label registered as type "SWIFT".
PIN	Logic simulation requires that each pin on a symbol have a property. A PIN property has two values associated with it: ● User pin name ● Compiled pin name You can change the user pin name to adhere to drafting standards, but you must not change the compiled pin name.
PINTYPE	Each model pin has an associated PINTYPE property, which describes the pin entry point type (i.e., IN, OUT, IXO, or IO). QuickSim II requires this property.
SWIFT_TEMPLATE	The SWIFT_TEMPLATE property always has a value that specifies the model name. This property cannot be changed.

Table 24: Symbol Properties Required for Simulation (Continued)

Property	Description
PKG	Each model has a PKG property equal to the physical package type (for example, DIP, LCC) that the symbol's pin numbers match. When using the bus symbol for a component, the PKG property is set to the value "BUS".

Optional Symbol Properties

Table 25 lists optional symbol properties.

Table 25: Optional Symbol Properties

Property	Description
COMP	The COMP property provides an interface to layout or other applications. Synopsys does not use this property.

The COMP property is assigned the default value "TimingVersion" with the property attribute "expression". This causes the COMP property to track the value of the TimingVersion property for Synopsys symbols. |
PIN_NAME	The PIN_NAME property is the visible text on a symbol representing a pin's name. Changing this text has no effect on the model's operation.
PIN_NO	The PIN_NO property value matches the physical pin number of the component for the default package. Changing the value of this property has no effect on the model's operation.
REF	The REF (reference) property provides an identifier for use in Advanced Verification messages. Changing the value of this property has no effect on the model's operation.

Building a Design Using the Menus

The Synopsys entries in the Design Architect menu system can be useful when building a design for the first time because all of the alternatives at each menu level are apparent.

To add SmartModels to a design using the menus, follow these steps:

1. Identify the desired model using the Schematic Editor menu system to traverse the menus.

2. Instantiate the model's symbol on the schematic sheet.

3. Edit property values as necessary using Design Architect or the Schematic Editor.

The Menu System

The menu system consists of several levels, starting with the pull-down menu that is accessed with the Libraries choice from the Design Architect menu bar. At that point, the following menu choices are available:

● Logic Modeling SmartModel Library

● Logic Modeling SimuBus Products (this entry is present only if you have installed SimuBus models)

👉 Note

Normally, the Admin tool installs the Logic Modeling entries in the Design Architect Libraries menu automatically, as part of model installation. If, after installing your models, you do not find at least the Logic Modeling SmartModel Library entry, you can perform the menu installation yourself. For more information, refer to "Installing the QuickSim II SWIFT Interface" on page 219.

Following are descriptions of the relevant menu levels:

Top-level

The top-level menu offers a number of choices, including component libraries and the first SmartModel product menu.

Function

The first SmartModel product menu offers a choice of functions, as follows:

❍ General purpose logic menu

❍ Logic block menu

❍ Memories menu

❍ Processor menu

❍ Programmable logic menu

❍ Support peripheral menu

Subfunction

Each item on the function menu has its own subfunction menu, which is used to further specify the symbol for the model being instantiated.

Vendor

> Each subfunction menu selection has an associated vendor menu, which displays a list of part manufacturers for that subfunctional group of models.

Part

> Each vendor menu selection has an associated part menu, which displays a list of all SmartModels for the selected subfunction class and vendor.

Component

> Each part menu selection has an associated component menu, which displays a list of all the timing and/or symbol versions available for a particular model.

Example

The following sequence of menu selections activates a Motorola MC88100:

> **SmartModel Library > Processor > Microprocessor > Motorola >**
>
> **88100 > MC88100-20 (BUS)**

Choosing the function, subfunction, and vendor brings up the part menu, which shows all the Motorola microprocessor models. Choosing the MC88100 brings up the component menu, which shows both the component and the symbol.

Building a Design Without the Menus

Users who are familiar with the SmartModel Library may prefer to use Schematic Editor commands to build designs. This approach can be faster than using the menu system. The basic steps are the same:

1. Identify the model you need.

2. Instantiate the model's symbol on the schematic sheet.

3. Use Design Architect or the Schematic Editor to edit property values, as necessary.

Creating An Instance

Use the $add_instance command to instantiate a part in the Schematic Editor, as shown in the following minimal command:

```
% $add_instance ("$LMC_HOME/special/qsim/symbols/")
```

The symbol name and TimingVersion property value can also be included on the same line, as follows. (All punctuation marks are required.)

```
% $add_instance \
("$LMC_HOME/special/qsim/symbols/","",,,["TimingVersion",""])
```

If the TimingVersion property value is not specified, the default timing version is activated. If a symbol is not specified when applicable, the default is used. The defaults are "positive" for logic version, and "pin" for symbol type.

Selecting Alternate Symbols

When there are DeMorgan equivalent symbols, the positive version is the default. Specify "NEG" as the symbol name to get the negative logic symbol (if desired), as follows:

```
% $add_instance ("$LMC_HOME/special/qsim/symbols/ttl00","NEG",,, \
["TimingVersion","SN74AS00"])
```

When activating parts manually, remember that the pin symbol is the default when both pin and bus symbols are available. Specify the bus symbol (if desired), as follows:

```
% $add_instance ("$LMC_HOME/special/qsim/symbols/mc68030_hv","BUS",,, \
["TimingVersion","MC68030-33"])
```

Use the Mentor Graphics Design Viewpoint Editor (DVE) to set the primitive type for SWIFT in DVE. To ensure that SWIFT instances are evaluated as primitives, you can add to the primitive rule using the add primitive command within DVE. In the following example, the add primitive command causes all instances of the MODEL property value "SWIFT" to be evaluated as primitives by QuickSim II.

```
% add primitive "model" -noexcept -string "SWIFT"
```

The string "SWIFT" can be substituted with any other labels that you have registered with the model type of "swift".

Logic Simulation

The following sections in this chapter provide information about using SmartModels for logic simulation in the Mentor Graphics's QuickSim II environment. For related information, refer to the following Mentor Graphics manuals:

- *Common Simulation User's and Reference Manual*
- *Getting Started with QuickSim II Training Workbook*
- *QuickSim II User's Manual*

Current Support Levels

Please note the following important items before starting a simulation. SmartModel Library models currently:

- **Do** support the implementation of location maps. You can use location maps to install a library anywhere on the system. Set an environment variable and a location map variable before using location maps.

- **Do** support extended-time (64-bit) simulations.

- **Do not** support the unit delay timing mode.

Default Timing Mode

The default timing mode for SWIFT SmartModels is "typ". You can use the timing-mode switch at QuickSim invocation time to force the timing mode to be "min", "typ", or "max".

Signal Levels and Drive Strengths

SmartModels recognize the nine signal levels and drive strengths listed in Table 26. QuickSim II maps indeterminate strengths to unknowns for "12-state" simulations.

The models generate strong and resistive states. The high-impedance unknown state (XZ) is used when a model places an output in the high-impedance state.

Table 26: Signal State Values

	Signal Level		
Drive Strength	Low (0)	High (1)	Unknown (x)
Strong (S)	**0S**	**1S**	**XS**
Resistive (R)	**0R**	**1R**	**XR**
High Impedance (Z)	0Z	1Z	**XZ**

The state values shown in bold type are generated by the models. All values are recognized.

QuickSim II Command Line Switches

When QuickSim II is invoked from the command line, the following switches are the only ones recognized by SmartModels.

Timing Mode Switch

Use the -timing_mode switch to set the global timing mode to minimum, maximum, or typical. Unit delay timing mode is not currently supported. Use the following syntax when setting this switch:

```
-timing_mode { min | max | typ }
```

☞ Note

SmartCircuit models override settings made with the timing mode switch by means of the model command file (MCF) when the MCF is configured with a particular timing mode.   For more information about configuring SmartCircuit models of FPGA and CPLD devices, refer to the *SmartModel Library User's Manual*.

Time Scale Switch

Use the -time_scale switch to adjust time values (delays and checks) to the desired resolution by specifying the time scale in nanoseconds (ns). The default is 0.1 ns. Use the following syntax when setting this switch:

```
-time_scale timescale
```

Constraint Mode Switch

Use the -constraint_mode switch to enable or disable timing constraint checking. The default is "off". Any value other than "off" causes a model to perform constraint checking. Use the following syntax when setting this switch:

```
-constraint_mode { off | state_only | messages }
```

QuickSim II Command Interaction

There are many QuickSim II simulator commands that interact with SmartModels in a design. Table 27 lists some of the QuickSim II commands that affect SmartModels.

Table 27: QuickSim II Command Interaction

QuickSim II Command	Effect on SmartModels
INITIALIZE	Causes a model to reevaluate until the simulation reaches DC stability. It will not reset the model or set internal values to the "state_value" specified.
SIGNAL INSTANCE	Reports the status of selected instances with the "swift_dump" parameter.
REPORT OBJECT	Not supported by SmartModels. Use the SIGNAL INSTANCE command to query a model and report its status.
REREAD MODELFILE	Reloads any of the configuration files used by a model, including memory image, JEDEC, MCF, SCF, and PCL command files. Configuration files are only re-read if the simulation has changed the configuration.
RESET STATE	Causes all models to reinitialize their states to the original time zero (power-up) conditions.
RESTORE STATE	Restores all of the model's internal states as part of the operation.
SAVE STATE	Records all of the model's internal states as part of the operation.
WRITE MODELFILE	Causes a model to "dump" its memory image to a file.

SWIFT Command Channel

You can use the SWIFT command channel to pass commands directly through to SmartModels. Use the QuickSim II SIGNAL INSTANCE command to access the command channel.

To issue a command for selected instances, use the following:

```
signal instance swift_model -p "command_name [ arguments ]"
```

To issue a command for a session, use the following:

signal instance *swift_session* **-p** "*command_name* [*arguments*]"

For more information on using the SWIFT command channel, refer to "The SWIFT Command Channel" on page 21.

Checking the Model's Status

Use the QuickSim II SIGNAL INSTANCE command to query a model directly. Select one or more instances in the design and then issue the command to display the internal element status of all selected instances, as shown in the example below.

```
signal instance swift_dump
'/I$1': ' '
'/I$1': 'Note: <<Status Report>>'
'/I$1': ' Model template: pal20r4i'
'/I$1': ' Version: not available'
'/I$1': ' InstanceName: /I$2742'
'/I$1': ' TimingVersion: MMI_20R4A-COM'
'/I$1': ' DelayRange: TYP'
'/I$1': ' JEDECFile: /user/bobb/design/schematic/selack.jedec'
'/I$1': ' Timing Constraints: Off'
'/I$1': ' SmartModel Instance /I$2742(U103:MMI_20R4A-COM), sheet1
            of schematic at time 0.00 nsec'
'/I$1': ' '
'/I$1': 'Note: SmartModel Windows Description:'
'/I$1': ' Q20 "PAL Internal Register connected to pin 20"'
'/I$1': ' Q19 "PAL Internal Register connected to pin 19"'
'/I$1': ' Q18 "PAL Internal Register connected to pin 18"'
'/I$1': ' Q17 "PAL Internal Register connected to pin 17"'
'/I$1': ' SmartModel Windows not enabled for this model.'
'/I$1': ' SmartModel Instance /I$2742(U103:MMI_20R4A-COM), sheet1
            of testbed/schematic at time 0.00 nsec'
'/I$1': ' ''
```

Reconfiguring Models for Simulation

You can use QuickSim II to reconfigure models for additional simulations by:

- Editing properties

- Changing timing modes of model instances

- Enabling or disabling constraint checking

Editing Properties

Adding or changing the value of a JEDECFile, SCFFile, PCLFile, or MemoryFile property causes the simulator to read the file and initialize the model to the power-up state. Select the following menu choices to change a property:

Edit > Properties > Add, Edit > Properties > Change, and Edit > Properties

Changing Timing Modes

SmartModels support minimum, typical, and maximum timing modes. Unit-delay mode is not supported. Select the following menu choices to display the form for changing the timing mode of specific model instances:

Setup > Kernel > Change > Timing Mode

You can also use instance names to specify which instance to change.

Constraint Checking

Select the following menu choices to enable/disable the various timing constraint checks (for example, setup, hold):

Setup > Kernel > Constraint Mode > Change

To enable constraint checking, select either "State Only" or "Messages" on the Change Constraint Mode form. To disable constraint checking, select "Off" on the Change Constraint Mode form.

SmartModel Library Message Formats

SmartModels issue four different kinds of messages to provide relevant information to users during simulations. These include:

- Error messages
- Warning messages
- Trace messages
- Notes

Error messages can be generated by timing or usage checks. Warning messages, error messages, and notes can all be generated by usage checks, depending on the situation. Hardware verification models also issue trace messages, if enabled.

Error messages itemize selected information. For example, a setup time violation causes an error message that documents:

- Pin name
- Part (by instance), reference designator, and component name

- Sheet name

- Design name

- Simulation time

- Signals and edges, as appropriate

- Setup times (as they occurred and as required by vendor data sheet)

Here are some sample messages:

```
'/I$2751':' '
'/I$2751':'? Error: Unknown signal level on CLK pin.'
'/I$2751':'? This will probably cause problems later in the
             simulation.'
'/I$2751':  '? SmartModel Instance /I$2751(U102:MC68030-20), sheet1 of
             schematic at time 0.0'
'/I$2790': ' '
'/I$2790': 'Note: Loading the memory image file "/user/bobb/design/
           schematic/rom_image.0_7"'
'/I$2790': 'SmartModel Instance /I$2790(U201:I27512), sheet2 of
           schematic at time 0.0'
'/I$2790': ' --- 14 values have been initialized.'
'/I$2751': ' '
'/I$2751': '! Warning: Unknown signal level on DSACK0_PIN.
             Assuming DSACK0_PIN is 1.'
'/I$2751': '! SmartModel Instance /I$2751(U102:MC68030-20), sheet1 of
             schematic at time 200.0'
'/I$2751': 'Trace: Returning read data to PCL program:'
'/I$2751': ' [0]=00000BFE, [1]=00000000, [2]=00000000, [3]=00000000'
'/I$2751': ' [4]=00000000, [5]=00000000, [6]=00000000, [7]=00000000,
        [8]=00000000'
'/I$2751': ' SmartModel Instance /I$2751(U102:MC68030-20), sheet1 of
             schematic at time 1750.0'
'/I$2751': ' '
'/I$2751': 'Trace: PCL Bus Cmd: Read. Control=06, Addr=00000004,
           Bytes=4.'
'/I$2751': ' SmartModel Instance /I$2751(U102:MC68030-20), sheet1 of
             schematic at time 1750.0'
```

Using SmartModel Windows with QuickSim II

SmartModel Windows is a SmartModel Library feature that can be used for more efficient system-level verification and debugging by allowing you to view window elements during simulation runs for microprocessor, PLD, memory, and peripheral models.

Window elements that can be viewed include registers, pointers, states, or latches (for example), depending on the part being modeled. This section provides information about how to interact with SmartModel Windows using QuickSim II. SmartModel Windows can be used to:

- Review window element values and set breakpoints

- Single-step through simulations

- Change window element values before proceeding with a simulation

- Trace instruction execution

- Rename instances

- Combine register elements

Most SmartModels contain predefined window elements that correspond to the manufacturer's specifications. In addition, SmartCircuit models allow users to define their own window elements so that the actual structure of the device can be examined. To determine if a specific model is equipped with SmartModel Windows, check the model's online datasheet.

How SmartModel Windows Works

SmartModel Windows couples the models and the simulator so that model elements can be used almost as if they were nets in the design. Normal QuickSim II commands are used with SmartModel Windows, except that an instance designator must be added to a QuickSim II command to address a model's window elements (even at the top level). The general format for using QuickSim II commands with SmartModel Windows is:

```
command model_instance/element_name
```

Use any of the following commands to enable window elements with the simulator:

```
ADD LISTS model_instance/element_name

ADD MONITORS model_instance/element_name

ADD TRACES model_instance/element_name
```

Window elements must be activated with one of the preceding commands in order for SmartModel Windows to begin displaying data. Each model's online datasheet lists its predefined window elements, which are available during simulation. Using windows you can read or display elements, force new values onto them, and stop the simulation based on their values.

Tracing Instruction Execution

SmartModel Windows provides the ability to trace peripheral component activity in a design. Most larger peripherals and microprocessors equipped for SmartModel Windows have a 1-bit element named TRACE_ENABLE.   Setting TRACE_ENABLE to one (1) causes trace messages to display in the transcript window.

Use the following command to enable instruction tracing for a model:

```
FORCE model_instance/TRACE_ENABLE 1
```

Some trace message examples follow:

```
'I$2752': 'Trace: Logical Master writing to PMMU Operand Address CIR.
'I$2752': 'SmartModel Instance /I$2752(U103:MC68851-12), sheet1 of
          my_design at time 819500.0
'I$2752': 'Trace: MC68851 is starting a table search using CRP.
'I$2752': 'SmartModel Instance /I$2752(U103:MC68851-12), sheet1 of
          my_design at time 822150.0
```

Setting Breakpoints and Word Triggering

Use the ADD REAKPOINT command to stop the simulation at critical points and examine internal window elements. You can set breakpoints based on the contents of specific elements inside components within the design. For example, the following command causes the simulation to stop at the breakpoint when the specified condition is met.

```
ADD BREAKPOINT (model_instance/TC==0B)
```

You can use Boolean expressions with the Add Breakpoint command to set up complex word triggers that provide a logic analyzer during simulation. For example, the following command causes the simulation to stop at the breakpoint when both of the two specified conditions are met.

```
ADD BREAKPOINT ((model_instance/SCC!=0)&&(model_instance/TC==B))
```

Trigger terms do not have to refer to the same instance or model. In addition, net values and window element contents can be combined to make trigger terms.

Single-Step Simulation

The ADD BREAKPOINT command defaults to stopping the simulator when a signal or expression in a window element changes state. As a result, you can use the ADD BREAKPOINT command to single-step through a simulation.

Renaming Instances

Use the ADD SYNONYM command to rename instances with easier-to-remember substitute names. For example, to rename a model for an MC68851 with an instance name of "I$289" to an easier-to-remember name such as "PMMU," use the following command:

```
ADD SYNONYM 'PMMU' I$289
```

Once you add a synonym you can use it in place of the original instance name in commands. For example, the following command lists the MC68851 translation control register.

```
ADD LISTS HEX PMMU/TC
```

Combining Register Elements

You can use the ADD BUS command to combine meaningful 1-bit elements of a PLD into a single bus that can be viewed or changed after the PLD has been programmed. This saves effort compared to dealing with each 1-bit element one at a time. Table 28 shows the elements of a sample device, the Texas Instruments TIBPAL22V10. All window elements for this example are 1-bit wide with read and write access.

Table 28: Elements in a TIBPAL22V10 Device

Element	Description
Q23	PAL Internal Register connected to pin 23
Q22	PAL Internal Register connected to pin 22
Q21	PAL Internal Register connected to pin 21
Q20	PAL Internal Register connected to pin 20
Q19	PAL Internal Register connected to pin 19
Q18	PAL Internal Register connected to pin 18
Q17	PAL Internal Register connected to pin 17
Q16	PAL Internal Register connected to pin 16
Q15	PAL Internal Register connected to pin 15
Q14	PAL Internal Register connected to pin 14

☞ **Note** ───

> Because the block diagram of this part does not denote specific names for the elements, their names reflect the output pin numbers on the DIP symbol (for example, pin 23 maps to Q23).

For example, to program the first six elements listed in Table 28 as counters and register the last four elements as data pins from an I/O port, you would use the following commands:

```
ADD BUS CNTR I$230/Q23, I$230/Q22, I$230/Q21, I$230/Q20, I$230/Q19,
             I$230/Q18
ADD BUS DATA I$230/Q17, I$230/Q16, I$230/Q15, I$230/Q14
ADD LISTS CNTR DATA -C -HEX
```

Changing Program Flow by Setting Values

You can use the SmartModel Windows feature to shorten large repetitive loops. For example, if a DMA controller has initiated a DMA transfer of 1,024 words to main memory, you can view the transfer of the first couple of words before stopping the simulation. By artificially setting the value of the DMA's transfer control register, you can control which part of the transfer to view. You can then view the last few words as they are transferred without having to wait for the entire process.

Be careful when inserting values into window elements, especially when forcing data into program counters and instruction registers. This SmartModel Windows feature is recommended only for users who completely understand the implications of what is being inserted into an element.

When forcing a value onto an element, the FORCE command is always interpreted as if the -CHARGED switch were present. This means that the forced value vanishes when another event attempts to update the window element. It is not possible to FIX or WIRE a forced value on a window element.

SmartModel Window Elements

SmartModel Window elements for SmartCircuit models can be defined only at simulator startup. This affects the way several QuickSim II commands interact with SmartCircuit models:

● The SAVE STATE and RESTORE STATE commands produce unpredictable effects if any SmartCircuit window elements are defined after saving the state.

● For SmartModel Windows to work with the SAVE STATE and RESTORE STATE commands, the window elements defined at SAVE STATE must exactly match those defined at the start of the current simulation session.

- The REREAD MODELFILE command does not redefine window elements for SmartCircuit models. Using this command to redefine window elements after simulation startup disables the window elements.

Custom Symbols

Synopsys provides symbols representing default package pinouts for SmartModels. However, you may need to create custom symbols for some of the following reasons:

- To conform to internal drafting requirements

- To make a symbol match a component's pinout

- To match external drafting specifications (for example, military specifications)

Users who choose to create custom symbols as an alternative to using the symbols provided can:

- **Modify a SmartModel Symbol.** Start with the SmartModel symbol and modify it to match your drafting requirements. The value of the user PIN property can now be changed without corrupting the value of the compiled PIN property.

- **Create a New Symbol.** Create the symbol with your pin values, figure out the corresponding pin names used by the model, and change the user pin values to those names.

To create custom symbols, follow these steps:

1. Provide required SWIFT properties on the symbol.

2. Register the component.

3. Map pin names to standard SWIFT pin names.

SWIFT Properties

The following symbol properties are required to interface with a SWIFT model:

- model

- TimingVersion

- pin

- pintype

- swift_template

Refer to Table 23 and Table 24 for information about other required symbol properties.

Component Registration

When creating new symbols, you must add the SWIFT model to the component interface by "registering" the model with a type of "SWIFT".

For example, suppose you have created a new component called "my_ttl00" and you want to use the SmartModel Library "ttl00" binary as the simulation model. You would register the ttl00 model as follows:

```
reg_model $MY_DIR/my_comp_lib/my_ttl00 -type SWIFT -label 'my_label'
```

PIN_NAME Mapping

The two methods for creating custom symbols described in "Custom Symbols" on page 239 cannot be used to map bus symbol pins to model pins. You must use a pin_map file to accomplish this sort of custom symbol creation, as explained in the following sections.

PIN Property

A PIN property can have two distinct values in Design Architect, as follows:

- Compiled pin value
- User pin value

The compiled pin value must be the same value that is used in the model. When initially adding a pin to a symbol, both these values are set to the specified value. For example, naming a pin "A" causes both its user pin value and the compiled pin value to be "A".

Changing a PIN property value causes the compiled pin value to track the user pin value. Specifically changing the compiled PIN property value disables this tracking mechanism. To re-enable tracking, set the value of the compiled PIN property to null ("").

PIN_NAME Property

SmartModel Library symbols include a property called PIN_NAME that is used purely for graphical purposes. The PIN_NAME property is provided because SmartModel Library symbols do not completely match the Mentor Graphics requirements for pin names. Deleting a PIN_NAME property does not affect model functionality in any way.

🕐 **Attention**───
   Do not confuse the PIN property with the PIN_NAME property on
   SmartModel symbols.
──

Purpose of the pin_map File

Use the pin_map file to map custom symbol pins to model pins. If your symbol does not have buses, then you can use the "user pin value" and "compiled pin value" combinations previously described to do this mapping without using the pin_map file. If you have bus pins on your symbol, then you need to use a pin_map file and ensure that the PKG symbol property is set to the value "BUS". Following is a general description of the pin_map file which describes both cases.

> ☞ **Note**
> _____
>
> Error messages cite the pin names used by the model, not those on the symbol or in the pin_map file.
> _____

How the pin_map File Works

At startup, a PKG symbol value of "BUS" triggers the simulator to look for a pin_map file for that model. The pin_map cross-reference file is a free-format ASCII file. It contains statements that use the following syntax:

```
pin_type symbol_pin [ = ] model_pin_names ; [ # comment_text ]
```

Following are descriptions of the fields and options.

pin_type	Must be the same value as the PINTYPE property of the model. Valid values are IN, OUT, IXO, and IO. Do not change this value.
symbol_pin	The new pin name you want to use on your symbol. This is the symbol's user PIN property, not its PIN_NAME property.
=	Optional.
model_pin_names	A statement can have from 0 to 767 model_pin_names, separated by spaces, tabs, or new lines. The model_pin_names are ordered from most significant to least significant and refer to the PIN property, not the PIN_NAME property.
;	Ends a statement.
#	Starts a comment, which runs to the end of the line.

Example of a pin_map File

The following example pin_map file customizes the standard symbol supplied with the model of the National Semiconductor DP8429 DRAM controller shown in Figure 15.
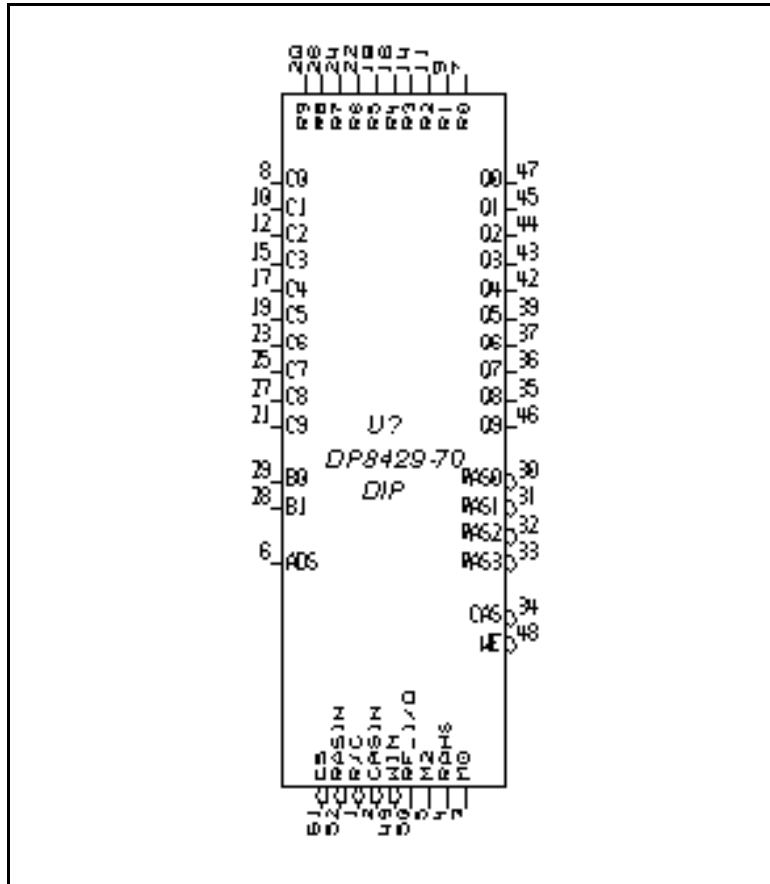
Figure 15: National Semiconductor DP8429 DRAM Controller

In the example pin_map file shown below, the names that are changed on the first symbol are labeled CS, RASIN, R/C, CASIN, WIN, RA, RB, RC, RD, and M2. They will have an "L" added to the name to denote that they are asserted low. Notice that a bus has been defined for each of these sets of pins: Q0 through Q9, R0 through R9, C0 through C9, and B0 through B1.

```
#
#  DP8429 PIN NAME CHANGES
#
IN CSL =  CS  ;
IN  RASINL =  RASIN  ;
IN  CASINL =  CASIN  ;
IN RCL =  R/C  ;
IN WINL  =  WIN  ;
IN RFSH  =  M2  ;
IN R    =  R9 R8 R7 R6 R5 R4 R3 R2 R1 R0;
```

```
IN C    =  C9 C8 C7 C6 C5 C4 C3 C2 C1 C0;
IN B    =  B1 B0;
OUT  WEL =   WE  ;
OUT  RAL =   RAS3  ;
OUT  RBL =   RAS2  ;
OUT  RCL =   RAS1  ;
OUT  RDL =   RAS0  ;
OUT  Q   =   Q9 Q8 Q7 Q6 Q5 Q4 Q3 Q2 Q1 Q0;
```

Conditional Pin Mapping

Use a conditional clause in the pin_map file to cause the model to use different parts of a pin_map file based on the value of a certain property. The syntax is:

```
% property_name  property_value
```

This method is used by the models to map the pins from the BUS symbols to the model.

The pin map parser searches for the property_name in the design database and then compares the property_value. If the property is not present, or if the actual value of the property does not match the property_value exactly, everything in the file until the next percent sign (%) is ignored.

The following example shows the pin_map file that provides mapping from the pin to the bus symbols for the Logic Devices LSH32 32-bit barrel shifter.

```
#
# Bus Package for the LSH32
#
% PKG BUS
IN  I = I31 I30 I29 I28 I27 I26 I25 I24 I23 I22 I21 I20 I19 I18
        I17 I16 I15 I14 I13 I12 I11 I10 I9 I8 I7 I6 I5 I4 I3 I2 I1 ;
OUT  Y = Y15 Y14 Y13 Y12 Y11 Y10 Y9 Y8 Y7 Y6 Y5 Y4 Y3 Y2 Y1 Y0 ;
IN  SI = SI4 SI3 SI2 SI1 SI0 ;
OUT SO = SO4 SO3 SO2 SO1 SO0
```

Figure 16 illustrates both symbol types.
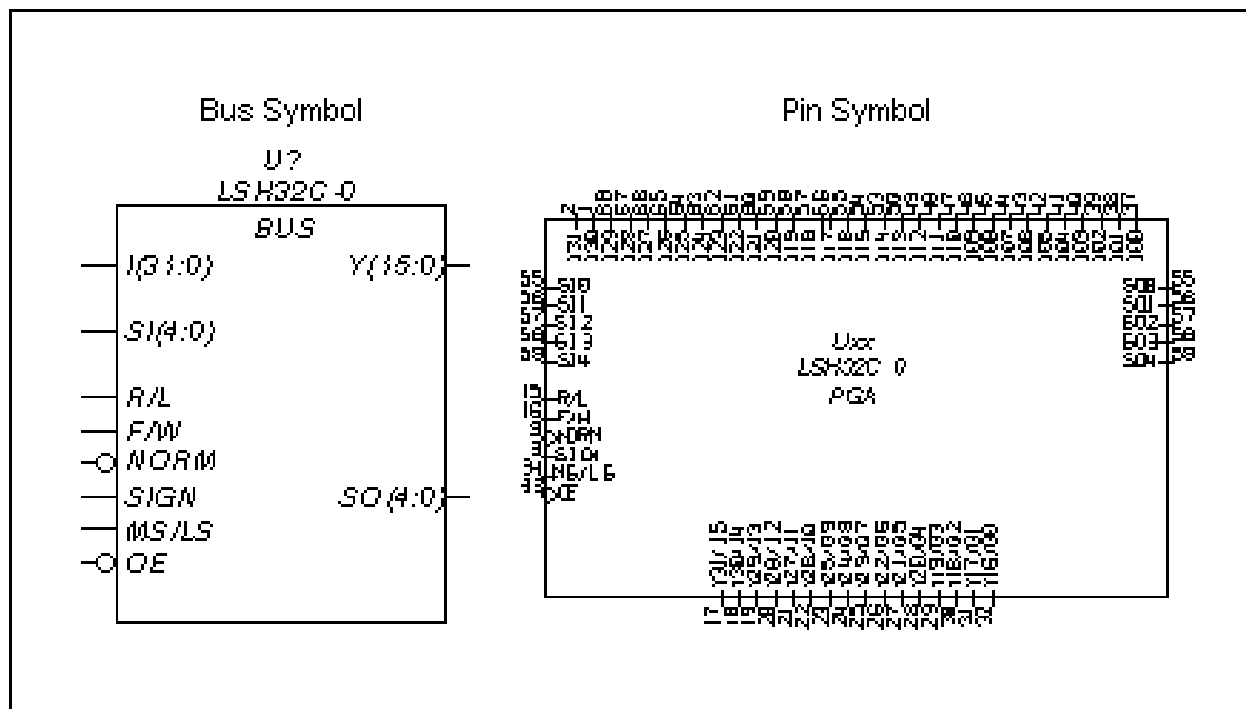
Figure 16: Bus and Pin Symbols

👉 **Note**

The properties on custom symbols must be the same as those on standard SmartModel symbols.

Using Hardware Models with QuickSim II

This section describes how to configure Release 3.5a of ModelAccess for QuickSim II. ModelAccess is the software you use to interface hardware models with the simulator. Before you begin, review the release notes for ModelAccess for QuickSim in the *Hardware Modeling Release Notes*. If you are using the C-series releases of QuickSim II, you must use R3.0 or better of the ModelAccess for QuickSim II interface software.

These instructions assume that you have already installed the following software:

- Mentor Graphics software, including QuickSim II V8.6 or later; and the Design Data Port package, as described by Mentor Graphics Corporation.

- R3.1a or later of ModelSource or LM-family hardware modeling software.

Setting up Hardware Models in QuickSim II

To set up the LM-family ModelAccess interface software for QuickSim II, complete the following steps:

1. "Running lmc_hm_install" on page 245
2. "Rebuilding the Mentor Graphics Tree" on page 246

Running lmc_hm_install

To run the ModelAccess installation script, enter the following commands. If you are on NT, execute these commands in a MGC "mkns" shell.

```
% cd install_dir/sms/maqs_30/lmc_hm.$vco/bin
% lmc_hm_install -m mgc_home -l lm_home -p ma_home
```

where:

- *$vco* is the vendor CPU operating system suffix that corresponds to your platform, as shown in Table 29.

- *mgc_home* is the directory path that contains the Mentor Graphics software tree. You can use $MGC_HOME if you have set it, or a pathname such as /home/mentor.

- *lm_home* is the directory path that contains the LM-family and ModelSource system software; for example, /home/lm.

- *ma_home* is the directory path that contains the ModelAccess interface software; for example, /home/lmc_hm.sss.

Table 29: Mentor Graphics Vendor CPU Operating System Suffixes

Host	Vendor CPU Operating System Suffix (*$vco*)
Sun SPARC (Solaris)	ss5
HP 9000 Series 700	hpu
Intel Pentium (Windows NT)	ixn

When the script completes, the following message appears on the screen:

```
1. Invoke the Mentor Installation tool
   > cd /an_idea_tree/install8
   > install
```

```
2. From the "Mentor Graphics Install" tool window
   > Admin
     > Rebuild MGC Tree

3. In the "Prompt" window enter the proper path to the
   MGC_HOME to be rebuilt. Click on
   > OK

4. "Rebuild MGC Tree Results" window appears. AFTER Rebuild
   completes. Click on
   > OK

5. From the "Mentor Graphics Install" tool window
   > File
     > Exit

6. "Install Warning" window appears. Click on
   > OK
```

☞ **Note**

This process rebuilds the Mentor tree with the newly installed hardware modeler package.

Rebuilding the Mentor Graphics Tree

The final step is to rebuild the Mentor Graphics tree using the Mentor Graphics installation script.

Using install, version C.1

1. To invoke this program, enter the following:

    ```
    % cd mgc_home/install8 ./install
    ```
2. When the install tool appears, use the mouse to select the Admin > Rebuild MGC Tree pull-down menu item. The program prompts you to enter the MGC tree pathname.

3. If you have defined $MGC_HOME, that path will appear; otherwise, enter the full pathname of the Mentor Graphics tree that you want to rebuild, such as the *mgc_home* pathname described in "Rebuilding the Mentor Graphics Tree" on page 246.

4. Click on OK or press the Return key to accept this pathname. The install program takes several minutes to rebuild the Mentor Graphics tree. The program prints a number of messages to the Results screen. You should ensure that no errors or warnings are printed, especially warnings generated by the lmc_hm package indicating that you are missing certain Mentor Graphics software packages.

5. When the rebuild is complete, click on OK to delete the Results screen.

6. Use the mouse to select the File > Exit pull-down menu item.

This completes the ModelAccess installation procedure. You are now ready to begin model registration and simulation.

Using Hardware Models in QuickSim II

This section describes how to prepare and use hardware models in QuickSim II. We begin with an overview of the Mentor Graphics design environment that describes why models must be registered in order to function in this environment. The section also describes how to use the lm_model utility to register hardware models.

The operation of the hardware modeling system during simulation is transparent to the user in most respects. However, a number of signal instance commands are available to enable or disable QuickSim II or hardware modeling features for selected instances during simulation. This section provides descriptions and examples of those commands.

The Mentor Graphics Design Environment

In the post-V8.0 QuickSim-family environment, an instance of a component placed on a schematic references a component interface. A component interface contains a set of descriptors that define aspects of a component, such as its functionality, graphical representation (symbol), and timing constraints.

There can be several variations of each descriptor for a component, such as:

- Several functional descriptions of the component using different modeling methods such as BLM, VHDL, or hardware models.

- Several graphical descriptions (symbols) of the component such as ANSI, MG_STD, or your company standard.

- Several technology descriptions (timing constraints) of the component with different timing grades.

The Mentor Graphics analysis tools use the value of the MODEL property as a label to identify the descriptors that define the model. For example, Figure 17 shows an instance with a MODEL property of $lm. The $lm label is the default label for the functional description of a hardware model. By examining the model table of the component interface, a match can be found between the MODEL property and the files that comprise the functional description of the hardware model.
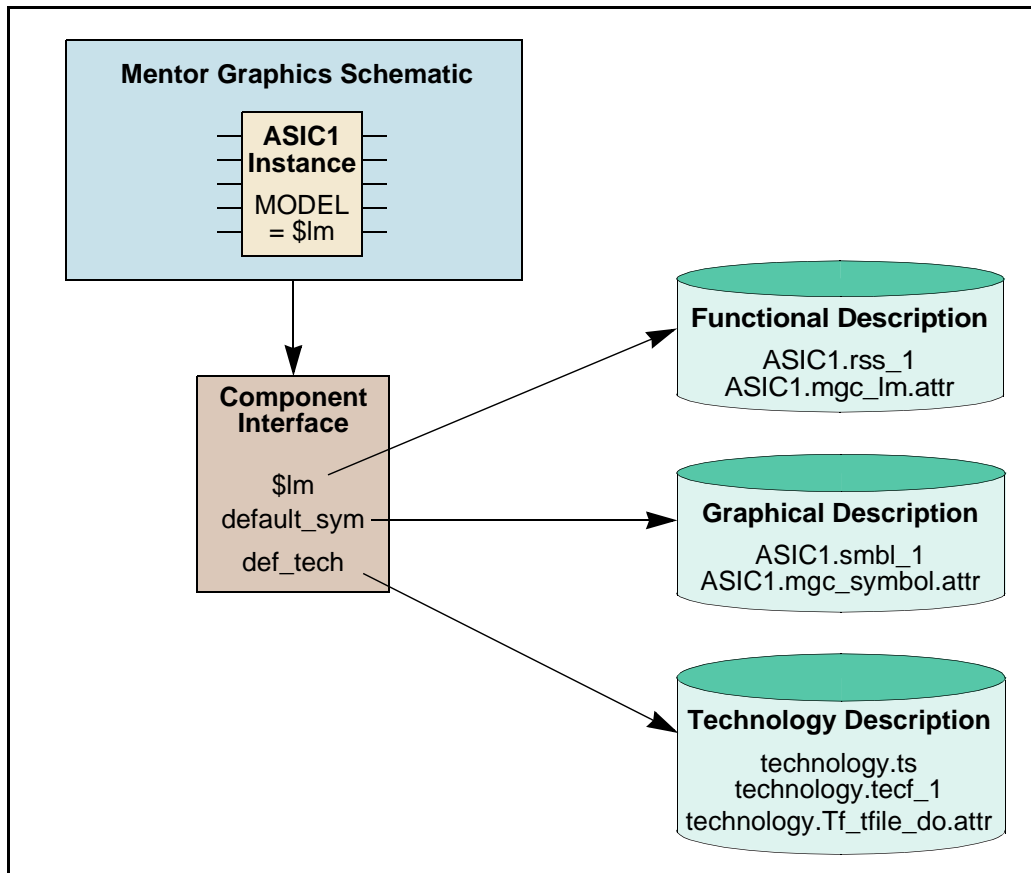
Figure 17: Sample Component Interface for a Hardware Model

Mentor Graphics analysis tools use the following rules to determine the appropriate descriptors:

1. If a label match is found, the analysis tool uses the descriptor identified by the label.

2. If a label match is not found, the analysis tool uses the default label for the descriptor.

3. If a label match is not found and there is no default label, the descriptor is optional and is not used.

Model Registration

Because multiple descriptions can exist for the same component, you must register each model to specify the model's component interface and descriptors. The lm_model utility is a tool for registering hardware models. If a QuickSim II component does not already exist, lm_model creates one, along with a component interface that specifies the functional, graphical, and technology descriptions for the model.

The lm_model utility registers a hardware model in three steps, using the model's Shell Software as source files:

1. Creates a symbol for the model and registers it with the component interface.

2. Creates, compiles, and registers a technology File, which contains the timing description of the model in a Mentor Graphics proprietary format. The user can choose to use either this technology file or the Shell Software timing files during simulation. For more information, refer to "Timing Shell Selection" on page 258.

3. Registers the functional description with the component interface.

The lm_model command, as illustrated in Figure 18, calls a number of other utilities. The reg_model utility and Technology Compiler (tc) are Mentor Graphics utilities; for more information about these utilities, refer to your Mentor Graphics documentation. For more information on the tmg_to_ts converter, refer to "lm_model Command Reference" on page 264. For more information on the lm_model utility, refer to "tmg_to_ts Command Reference" on page 267.
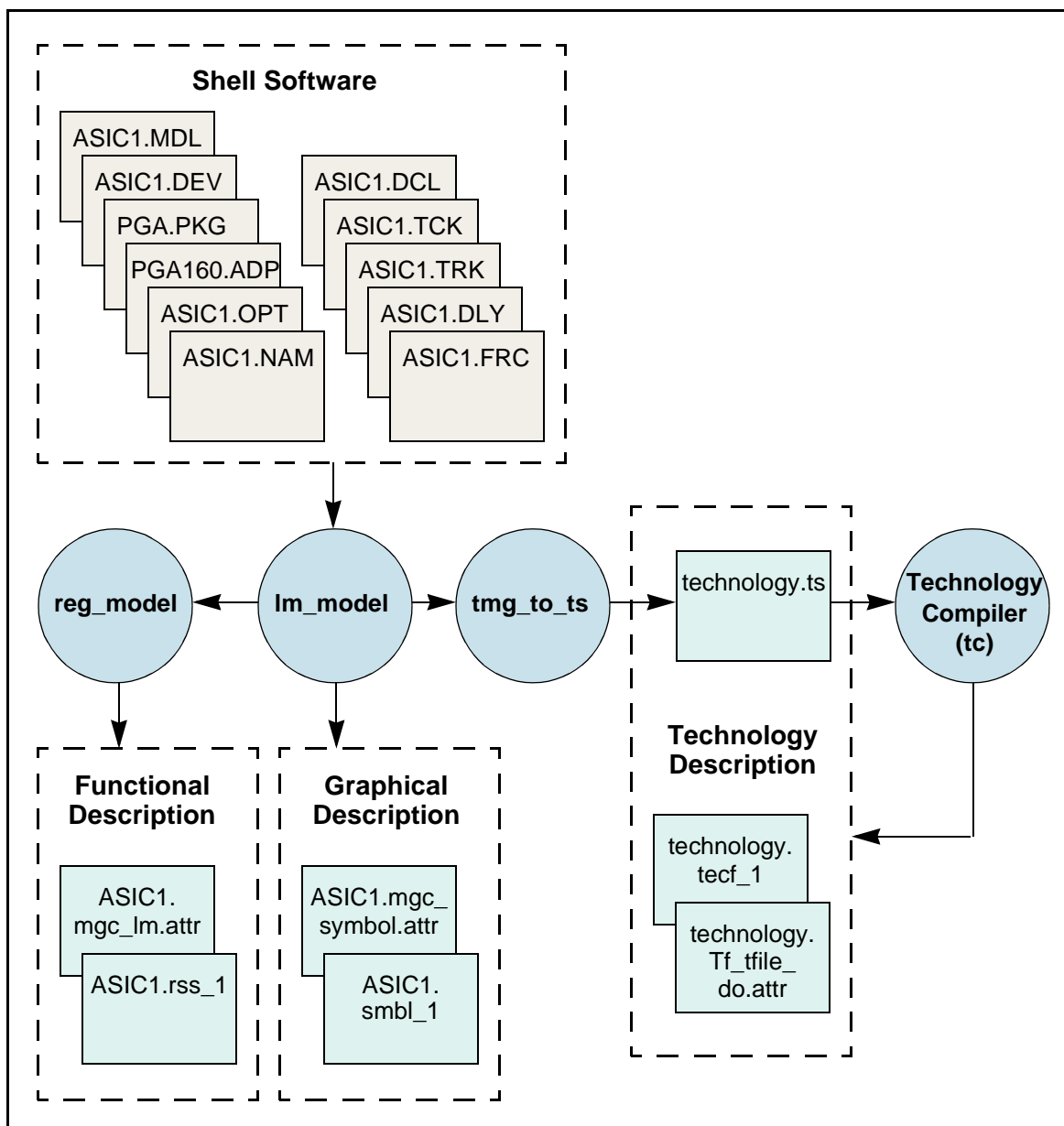
Figure 18: Hardware Model Registration

Registering a Model with lm_model

All hardware models, whether user-created or purchased from Synopsys, must be registered with the lm_model utility before you can use them in the QuickSim II simulation environment. The following list shows the basic steps involved in preparing a hardware model for simulation:

1. Running the lm_model Utility, discussed next.

2. Checking the Transcript for any errors or warnings.

3. Editing the Symbol to meet any additional symbol conventions (optional).

4. Verifying the Technology File(not required if Shell Software timing files are used in place of this file; for more information, refer to "Timing Shell Selection" on page 258).

Running the lm_model Utility

You can use the lm_model shell command to register a hardware model. To run lm_model, use the following syntax.

Syntax

lm_model *input_dir* [*output_path*] [**-Dir** *name*] [**-LAbel** *label*]
    [**-Mdl** *mdl_filename*] [**-Step** Register|Symbol|Timing|Update] [**-Replace**]

For a complete description of lm_model syntax and options, refer to "lm_model Command Reference" on page 264.

Example

The following example shows how you might register a 74LS74 model which has a model file named 74LS74A.MDL. This example assumes that $MGC_WD is set to /user/models, which contains a Shell Software directory called 74ls74.

```
lm_model 74ls74 -m 74LS74A
```

This command creates a /user/models/74LS74 component directory—if one did not already exist—containing the files shown in Table 30.

Table 30: Sample Component Directory

File	Description
74LS74A.rss_1	Registered Shell Software
74LS74A.mgc_lm.attr	Compiled and registered Shell Software
part.part_1	EDDM part
part.Eddm_part.attr	EDDM part
74LS74.smbl_1	Symbol graphics
74LS74.mgc_symbol.attr	Symbol graphics
74LS74A_tech.ts	Source technology file
74LS74A_tech.tecf_1	Compiled technology file
74LS74A_tech.Tf_tfile_do.attr	Versioned technology file

> ☞ **Note**
>
> Previous versions of lm_model copied Shell Software source files into the component directory and registered these files. However, the current version registers only a reference pathname to the Shell Software files and does not copy the files.

Checking the Transcript

The transcript displays information about the progress of the registration, in addition to notes, warnings, and errors.

The lm_model utility checks that the Shell Software is syntactically and semantically correct; this is equivalent to running the lm Check Shell Software utility. If lm_model encounters an error condition, it stops execution and prints a message describing the source of the error. Warning and information messages print to the screen, but do not halt the execution of lm_model.

If you get an error, you should fix the problem in the Shell Software and then run lm_model again to complete the registration.

The following is an lm_model transcript for the 74LS74 model:

```
//  ModelAccess for QuickSim II v2.0, (a.k.a. lmc_hm v2.0)
//  lm_model  v8.5_2.1 Fri Oct 18 18:32:32 PDT 1996
//  Note:  Input directory "74ls74"
//         resolves to "/user/johnd/lmc/qa/lmc_hm/work.sss/74ls74".
//  Note:  Output directory "74LS74"
//         resolves to "/user/johnd/lmc/qa/lmc_hm/work.sss/74LS74".
//
//  Note:  Using "74LS74A.MDL" file for conversion.
//  Note:  Compiling symbol generator program.
//  Note:  Linking symbol generator program.
//  Note:  Creating symbol.
//  tmg_to_ts  v8.5_2.1  Sat Oct 19 20:18:24 PDT 1996
//  Falcon Framework  v8.5_2.5 Thu May 30 17:31:43 PDT 1996
//
//  Copyright (c) Mentor Graphics Corporation, 1982-1995, All Rights Reserved.
//                    UNPUBLISHED, LICENSED SOFTWARE.
//          CONFIDENTIAL AND PROPRIETARY INFORMATION WHICH IS THE
//         PROPERTY OF MENTOR GRAPHICS CORPORATION OR ITS LICENSORS.
//
//  Mentor Graphics software executing under Sun SPARC SunOS.
//


//   TC - The Technology Compiler   v8.5_2.2  Sat Jun 22 10:56:50 PDT 1996
//  Falcon Framework  v8.5_2.5 Thu May 30 17:31:43 PDT 1996
//
//  Copyright (c) Mentor Graphics Corporation, 1982-1995, All Rights Reserved.
//                    UNPUBLISHED, LICENSED SOFTWARE.
```

```
//              CONFIDENTIAL AND PROPRIETARY INFORMATION WHICH IS THE
//            PROPERTY OF MENTOR GRAPHICS CORPORATION OR ITS LICENSORS.
//
//  Mentor Graphics software executing under Sun SPARC SunOS.
//


//
//  Note:  lm_model completed successfully
```

Editing the Symbol

During registration, the lm_model utility reads the Shell Software to determine the device's input, output, and I/O pin names. The utility then flattens all buses to individual bits and generates a Mentor Graphics Design Architect script that creates the symbol.

The process uses the following rules to create the symbol:

- All input pins are placed starting in the lower left corner and proceeding upwards.

- All output pins are placed starting in the lower right corner and proceeding upwards.

- I/O pins are placed for minimizing the symbol's height.

- All buses are grouped with the least significant bit placed lower on the symbol than the most significant bit.

- A single grid spacing is left between buses, grouped scalar pins, input and I/O pins, and output and I/O pins.

Since symbol standards vary, you may need to use the Symbol Editor in Design Architect to modify the appearance of the automatically-generated symbol. For more information, refer to your Design Architect documentation.

Verifying the Technology File

During registration, lm_model calls the tmg_to_ts converter. This converter extracts timing information from the following Shell Software files to create a Technology File:

- Variable declarations (.DCL) file

- Timing checks (.TCK) file

- State tracking (.TRK) file

- Delays (.DLY) file

- Force values (.FRC) file

The technology file specifies propagation delays and some timing checks, as well as technology-dependent data for the simulation model. Table 31 shows how Shell Software timing statements are converted into technology file statements.

☞ **Note** ──

Many Shell Software statements have no technology file equivalents. The tmg_to_ts converter includes each "untranslatable" statement in the technology file as a comment and/or generates a warning message. For this reason, we recommend that you use the Shell Software timing files instead of the technology file during simulation. For instructions, refer to "Timing Shell Selection" on page 258.

───

Table 31: Shell Software to Technology File Conversion

Shell Software Statements	Technology File Statements
cycle_time *input_state* (*storage_pin*) = *timing_spec*	fMIN = *min_freq* on *storage_pin* (*input_trans*) fMAX = *max_freq* on *storage_pin* (*input_trans*)
decrement *name*	—
default_delay *timing_spec*	tP = *timing_spec* on *eval_storage_pin* (*input_trans*) to *output_pin* (*output_trans*)
delay from *input_state* (*eval_storage_pin*) to *output_state* (*output_pin*) = *timing_spec*	tP = *timing_spec* on *eval_storage_pin* (*input_trans*) to *output_pin* (*output_trans*)
force_value *output_pin* = *pin_value*	—
hold after *input_state*[1] (*storage_pin*) of *input_state*[2] (*input_pin*) = *timing_spec*	tH = *timing_spec* on *input_pin* (*input_state*[2]) to *storage_pin* (*input_trans*[1])
if (*condition*) {*statements*} else_if (*condition*) {*statements*} else {*statements*} end_if	—
increment *name*	—
print (*severity*, *arguments*)	—
pulse_width *input_state* (*storage_pin*) = *timing_spec*	tW = *timing_spec* on *storage_pin* (*input_state*) No equivalent to maximum pulse width time.

Table 31: Shell Software to Technology File Conversion (Continued)

Shell Software Statements	Technology File Statements

recovery after (*condition*)
  during (*condition*)
    before *input_state*(*storage_pin*) =
        timing_spec
  else_during (*condition*)
    before *input_state*(*storage_pin*) =
        timing_spec
  else
    before *input_state*(*storage_pin*) =
        timing_spec
  end_during
	—

set *name* = *value*	—
setup before *input_state*[1] (*storage_pin*) of *input_state*[2] (*input_pin*) = *timing_spec*	tS = *timing_spec* on *input_pin* (*input_state*[2]) to *storage_pin* (*input_trans*[1])
stable valid (*input_pin*) while (*store_pin* = *input_state*)	tSTAB = $0 : 0$ on *input_pin* (V) to *store_pin* (*trans1*, *trans2*)
var *enumerated_list name* = *identifier*	—
var counter *name* = *number*	—
when (*condition*) {*statements*} else_when (*condition*) {*statements*} else {*statements*} end_when	with condition No equivalents to else_when and else clauses

Modifying a Hardware Model

Whenever you change a hardware model's Shell Software, you need to rerun lm_model.
However, you may be able to use the -Step option to perform just the steps you need.
The following list provides some guidelines about how to take advantage of the -Step
option:

● If you change, add, or delete a pin name in the Shell Software, then you must rerun
  all three steps of lm_model (the default). Because you are recreating the symbol,
  you must also use the -Replace option. For example:

> ```
> lm_model 74ls74 -r
> ```

● If you change, add, or delete a timing specification in the Shell Software timing files
  and you are using the Technology File in QuickSim II, you should use lm_model
  with the -Step Timing switch. For example:

> ```
> lm_model 74ls74 -s t
> ```

This step is equivalent to running tmg_to_ts to create the technology file and then running tc to compile and register it with the component interface.

● If you make any changes to the Shell Software other than changing pin names and timing information, you should use lm_model with the -Step Register switch; for example:

```
lm_model 74ls74 -s r
```

This step is equivalent to running reg_model.

● If you have not changed any pin names and want to run both the registration and timing steps, you can use lm_model with the -Step Update switch to update the component interface without recreating the symbol. For example:

```
lm_model 74ls74 -s u
```

This switch is particularly useful, because symbol generation is the most time-consuming step of registration and you lose all manual edits you have made to a symbol when you regenerate it.

● If you already have a working symbol, you can use -Step Update to register the hardware model functionality with the existing component. For example, you would use -Step Update if you have a different type of model for the same component. You can then change the MODEL property in the schematic in order to specify whether you want to use the hardware model or another type of model for an instance.

Simulating with Hardware Models in QuickSim II

Once you have registered each hardware model in your design and set the MODEL property to the appropriate label for instances that reference those models, you are ready to simulate. You can use the SIGnal INSTance command to turn on and off a number of QuickSim II or hardware modeling features for selected instances during simulation.

Signal Instance Command Summary

Table 32 provides a summary of these features and the specific commands used to implement them; the subsections that follow describe the features in more depth. Some features can also be implemented through Shell Software statements or the lm utilities; for details, refer to the *Shell Software Reference Manual*. For instructions on how to select one or more instances, refer to your QuickSim II documentation.

☞ **Note**
If the simulator is reset with the $reset_state function, any prior Signal Instance commands are lost because the simulator is reset to the same state it was at invocation.

Table 32: Signal Instance Command Summary

Feature	Command	Description
Model evaluation	enable	Enables evaluation of the instance by QuickSim II (default)
	disable	Disables evaluation of the instance by QuickSim II
Timing shell selection	lst [-p all]	Selects hardware model Shell Software files to describe the instance's timing
	nolst [-p all]	Selects the Technology File to describe the instance's timing (Default)
Unknown handling and propagation	xp [-p *pin_name*]	Maps an unknown input state to the previous state (Default)
	x0 [-p *pin_name*]	Maps an unknown input state to a logic zero state
	x1 [-p *pin_name*]	Maps an unknown input state to a logic one state
	xz [-p *pin_name*]	Maps an unknown input state to a float state
	propagate	Propagates unknowns through the hardware model
	nopropagate	Turns off unknown propagation (default)
	default_ propagation -p *number*	Sets the number of additional sequences to be played to the instance when unknown propagation is enabled (default = 0)
	random_seed -p *seed*	Sets the value of the seed for the random sequence generator when unknown propagation is enabled (default = 0)
Indeterminate strength mapping	is	Maps an indeterminate strength (i) to a strong strength (s) (default)
	iz	Maps an indeterminate strength (i) to a high-impedance strength (z)
Test vector logging	logvectors -p *filename*	Turns on test vector logging
	nologvectors	Turns off test vector logging (default)

Table 32: Signal Instance Command Summary (Continued)

Feature	Command	Description
Timing measurement	tm [-p *filename*]	Turns on timing measurement: returns the actual measured delays to QuickSim II
	notm	Turns off timing measurement: uses the delay values specified in the Shell Software or in the technology file (default)
Loop mode	loop	Turns on loop mode: the modeling system repeatedly plays a pattern history to the physical device
	noloop	Turns off loop mode (default)
Information	dump	Reports all available information about the selected instance of a hardware modeled device
	lmc [-p shell|allshell]	Reports the type of timing shell (Shell Software or technology file) for the selected instance
	vector	Reports the runtime vector count of the selected instance

Model Evaluation

By default, all component instances are evaluated in QuickSim II. If you want to disable evaluation of models for selected instances, you can use the SIGnal INSTance disable command. This command isolates sections of the design and shortens the simulation time for debugging purposes. To turn model evaluation on again for selected instances, you can use SIGnal INSTance enable.

Timing Shell Selection

The SIGnal INSTance lst command lets you use the hardware model's Shell Software timing files instead of the default technology file during evaluation of the selected instances. You can use the SIGnal INSTance nolst command to switch back to the technology file for selected instances.

The optional -p all argument enables you to choose the type of timing shell for all hardware models in the design, if you have at least one instance selected. For example, you could use the following command before simulating:

```
sig inst lst -p all
```

QuickSim II ignores the technology files for all hardware models in the design and take the timing (delays and timing checks) directly from the Shell Software. If you decided you wanted to use the technology files instead for all hardware models, you could use the following command to switch back to the default timing shell without having to select every instance:

```
sig inst nolst -p all
```

To select Shell Software timing every time you invoke QuickSim II with a particular design, you can create or edit a quicksim.startup file under the design viewpoint. Add the following line to the file to directly call the function that implements this Signal Instance command:

```
$signal_instances("lst", "all", "/I$1");
```

Substitute the instance name of any hardware modeled device for /I$1.

You can also use the actual measured delays from the device as an alternative timing option with hardware models. For more information about this feature, refer to "Timing Measurement" on page 261.

Performance Monitoring

You can monitor the performance of the hardware modeler and append the results to the simulator log file after simulation. To enable performance monitoring, in the window where you are running the simulator, enter the following:

```
% setenv LM_OPTION "monitor_performance"
```

For more information, refer to "Performance Monitoring" in the *ModelSource User's Manual.*

Unknown Handling and Propagation

The unknown handling and propagation commands enable you to modify the hardware modeling system's default handling of device input and I/O pins that the simulator sets to unknown.

Unknown Mapping

Since the hardware modeling system cannot present an unknown logic level to a physical device, unknown values presented to inputs of hardware models must be mapped to known values. The SIGnal INSTance xp, x0, x1, and xz commands map unknowns for all instances of the selected components to the previous state, logic zero, logic one, or high-impedance (float), respectively. By default, unknowns are mapped to the previous state. Unknowns mapped to high-impedance are also mapped to the previous state.

You can customize unknown handling per pin by using the -p *pin_name* argument. For example, you can issue the following:

```
sig inst x0
sig inst x1 -p clk1
```

These commands map all unknowns for the selected components—except for unknowns received on the clk1 pin—to logic zero (0). Any unknowns received on clk1 are mapped to logic one (1).

These Signal Instance commands perform the same function as the Shell Software on_unknown statement, and the set_previous, set_low, set_high, and set_float attributes of the in_pin and io_pin statements. For more information, refer to the *Shell Software Reference Manual*. Note that explicit Shell Software settings override any Signal Instance commands.

Unknown Propagation

The SIGnal INSTance propagate command turns on unknown propagation for all instances of the selected components. The modeling system propagates the unknowns through the model using multi-sequence pattern play. The SIGnal INSTance nopropagate command turns off unknown propagation for all instances of the currently selected component, which is the default behavior.

When unknown propagation is on, two pattern sequences are used by default. However, you can specify up to twenty additional sequences with the default_propagation -p *number* command, for a total of 22 sequences. You can also specify the value of the seed for the random sequence generator with the random_seed -p *seed* command. The value of the seed is 0 by default, but any number from 0 to 65,535 can be used.

For example, you can issue the following:

```
sig inst propagate
sig inst default_propagation -p 8
sig inst random_seed -p 7896
```

These commands turn unknown propagation on for all instances of the selected components. The modeling system plays a total of ten sequences (the primary, secondary, and eight additional sequences) per instance to the device, and uses the random sequence seed 7,896.

These Signal Instance commands perform the same function as the Shell Software on_unknown statement. Note that explicit Shell Software settings override any Signal Instance commands, except for when the SIGnal INSTance nopropagate command is used. This exception allows the simulator to turn off unknown propagation if the modeling system is running out of pattern memory. For more information about unknown propagation, refer to the *Shell Software Reference Manual* and the *LM-family Modeler Manual* or the *ModelSource User's Manual*.

Indeterminate Strength Mapping

The SIGnal INSTance is and SIGnal INSTance iz commands enable you to map indeterminate strength pin values received on inputs of hardware models to either strong (hard) or high-impedance (float) strengths. The modeling system treats high-impedance strength pin values as unknowns and maps or propagates them accordingly. By default, the system maps indeterminate strengths to strong strengths.

Test Vector Logging

The SIGnal INSTance logvectors -p *filename* command turns on modeling system test vector logging for the selected instance. With test vector logging enabled, the inputs to the device and sensed outputs from the device are stored to *filename*. By convention, the *filename* used for the test vector output is *device_name*.VEC. The SIGnal INSTance nologvectors command turns off test vector logging for the selected instance, which is the default behavior.

For example, consider the following commands:

```
sig inst logvectors -p '$ASIC2/vectors/vector11.VEC'
dofile '$ASIC2/dofiles/run11.do'
sig inst nologvectors
```

In this example, the modeling system creates a test vector file called vector11.VEC. This file contains the vectors played to and sensed from the selected instance during the simulation run by the dofile. The SIGnal INSTance nologvectors command turns off the modeling system test vector logging capability.

After logging vectors, you can replay them directly to the device and note any discrepancies using the lm Play Vectors utility. This utility is particularly useful for ASIC verification. For more information about ASIC verification and test vector (.VEC) file format, refer to the *LM-family Modeler Manual* or the *ModelSource User's Manual*.

Timing Measurement

The SIGnal INSTance tm [-p *filename*] command turns on the modeling system timing measurement for the selected instance. The system then returns to the simulator the actual measured delay values for that instance. If you provide an optional *filename*, the system also saves the measured delays to a timing measurement (.TIM) file. By convention, *device_name*.TIM is the *filename* used for the timing measurement output.

The SIGnal INSTance notm command turns off timing measurement for the selected instance, which is the default behavior. If timing measurement is disabled, the Technology File delays (or the Shell Software delays if SIGnal INSTance lst is specified) are returned to the simulator.

For example, consider the following commands:

```
sig inst tm -p '$ASIC2/timing/timing11.TIM'
dofile '$ASIC2/dofiles/run11.do'
sig inst notm
```

In this example, the modeling system creates a timing measurement file called timing11.TIM. This file contains the delays of the selected instance measured during the simulation run created by the dofile. The SIGnal INSTance notm command turns off the modeling system timing measurement capability.

This Signal Instance command performs a similar function to that of the lm Measure Timing utility. Timing measurement is particularly useful for ASIC verification. For more information about ASIC verification and timing measurement (.TIM) file format, refer to the *LM-family Modeler Manual* or the *ModelSource User's Manual*.

👉 **Note** ──

The timing measurement (.TIM) file can be converted to a Shell Software delays (.DLY) file by using the lm Create Timing File utility. (For more information, refer to the *LM-family Modeler Manual* or the *ModelSource User's Manual*.) The delays file can then be converted into a technology file, if desired, by using the lm_model utility with the -Step Timing option.

──

Loop Mode

The SIGnal INSTance loop command turns on the modeling system pattern looping capability (loop mode) for the currently selected instance. In loop mode, the modeling system continually replays the complete pattern history of the selected instance to the device. The SIGnal INSTance noloop command turns off pattern looping.

Pattern looping is a model development feature useful for analyzing the device behavior and pattern history with an oscilloscope or logic analyzer connected to the device. However, while loop mode is enabled, no other user can access the modeling system; patterns are replayed to the selected device exclusively until loop mode is disabled. For this reason, QuickSim II returns an error if this command is specified when more than one user is accessing the modeling system.

Printing Model Information

A number of Signal Instance commands are available for printing information about the selected model instances. The SIGnal INSTance dump command prints all information available about the currently selected instances, including:

- Instance ID

- Name of the modeling system in which the hardware model is located

- Setting for test vector logging (On or Off)

- Setting for indeterminate strength mapping (S or Z)

- Setting for Shell Software timing (On or Off)

- Model name, as specified in the Shell Software device_name

- Setting for timing measurement (On or Off)

- Setting for loop mode (On or Off)

- Runtime vector count

- Evaluation status (Enabled or Disabled)

The SIGnal INSTance lmc and SIGnal INSTance vector commands print subsets of the information provided by SIGnal INSTance dump:

- The SIGnal INSTance lmc -p shell command prints the Shell Software timing setting (On if you have specified SIGnal INSTance lst; Off if you have not) for the selected instances. SIGnal INSTance lmc -p allshell prints the Shell Software timing setting for all hardware model instances, if you have at least one instance selected. If you do not specify one of the -p arguments, this command will print a list of the available subcommands.

- The SIGnal INSTance vector command prints the runtime vector count of the selected instances.

Performance Monitoring

You can monitor the performance of the hardware modeler and append the results to the simulator log file after simulation. To enable performance monitoring, in the window where you are running the simulator, enter the following:

```
% setenv LM_OPTION "monitor_performance"
```

For more information, refer to "Performance Monitoring" in the *ModelSource User's Manual.*

Ending the Simulation Session

Termination of a normal simulation session notifies the hardware modeling system that the simulation session has ended. All modeling system resources being used by that simulation are then made available for other users.

If the simulation exits abnormally, "orphaned" processes may exist on the modeling system, even though the simulation has terminated. If lmdaemon is running on your workstation, it automatically deletes orphaned processes. You can also use the lm Abort User utility to remove the unwanted processes manually. Both of these methods release modeling system resources. (For more information about lmdaemon and the lm utilities, refer to the *LM-family Modeler Manual* or the *ModelSource User's Manual*.)

LM-family and ModelSource modeling systems support simulation save and restore capabilities. When a save simulation state is performed, the state of all hardware models being used by the simulation session is automatically saved into a QuickSim II save directory. Similarly, restoring the simulation state automatically restores the state of the model as used by the saved simulation, including all stored pattern history.

Attention

If you are using Shell Software that contains enhanced features—such as model state tracking or "when" conditions—the translation to the resulting technology file may be incomplete and contain "compromise" statements. If you elect to use the technology file instead of the Shell Software during simulation, the device may exhibit incorrect timing and/or behavior. To eliminate this possibility, translate the technology file from pre-R2.0 Shell Software, which does not contain these statements, or use the Shell Software directly during simulation by issuing the SIGnal INSTance lst command on the hardware model instance. For more information about this procedure, refer to "Timing Shell Selection" on page 258.

lm_model Command Reference

The lm_model shell command registers a hardware model by invoking hardware model registration and conversion programs.

Syntax

lm_model *input_dir* [*output_path*] [**-Dir** *name*] [**-Ifc** *interface*] [**-LAbel** *label*]
    [**-Mdl** *mdl_filename*] [**-Step** Register|Symbol|Timing|Update] [**-Replace**]
    [**-VERBose**] [**-Help**] [**-Usage**] [**-VERSion**] [**-Old**] [**-LM**]

Required Argument

input_dir　　　　　　　Specifies the pathname to the directory containing the files to be registered. This argument is required and must appear first. All pathname specifications use the following location convention:

If the pathname is relative, the *input_dir* is assumed to be in the working directory, as specified by $MGC_WD. If $MGC_WD is not set, the *input_dir* is assumed to be in the current working directory.

If the pathname starts with a dollar sign (**$**), the *input_dir* is assumed to be in the location of the location map variable specified after the dollar sign.

If the pathname is an absolute pathname, the *input_dir* is assumed to be in the location of the absolute pathname.

Optional Arguments

output_path　　　　　　Specifies the pathname to the directory that contains the component information. If an *output_path* is not specified, then it defaults to the parent directory of the *input_dir*.

-Dir *name*　　　　　　　Specifies just the new name of the output component directory within the *output_path*; for example, MC68020. By default, the name is created from the base name of *input_dir* by removing any leading dollar ($) characters and converting all lowercase characters to uppercase. If the output component directory name is the same as the input directory name, lm_model will generate an error and fail rather than overwrite the input directory.

-Ifc *interface*　　　　Specifies the component interface(s) with which to register the model. Multiple component interfaces can be specified. By default, the model is registered with all component interfaces.

-LAbel *label*　　　　　Specifies the label(s) to register with the component interface. Multiple labels can be specified. By default, the model is registered with the $lm label, which corresponds to the functional description.

-Mdl *mdl_filename*　　　Specifies a particular model (.MDL) file within the *input_dir*. By default, the system uses the model file with the same base name as the output component directory, which is defined by the -Dir switch.

-Step Register|Symbol|Timing|Update

Selects particular registration step(s):

- Step Register registers the model's functional description.

- Step Symbol creates and registers the symbol.

- Step Timing creates, compiles, and registers the technology file.

- Step Update is equivalent to -Step Register and -Step Timing.

By default, lm_model performs all the registration steps.

-Replace	Deletes the existing component directory and then recreates it. If you try to overwrite an existing symbol without using this switch, lm_model fails and generates an error message.
-VERBose	Prints additional messages while lm_model is executing.
-Help	Prints help information on each of the available options, then immediately exits.
-Usage	Expands the command line and displays each argument and switch. After printing the usage message, lm_model immediately exits.
-VERSion	Prints the single-line version message, then immediately exits.
-Old	Registers the model using the pre-V8.3 method, for compatibility purposes.
-LM	Does not affect lm_model execution. The system accepts this argument for compatibility purposes.

Examples

The lm_model utility provides several ways of specifying input and output files and directories. The following examples list a given input directory (model file) and desired output component directory, and then show the lm_model command line you would use to get this result.

Example 1

Input directory: /user/models/74ls74 ($MGC_WD is set to /user/models)

Model file: 74LS74.MDL (same as the component name)

Component (output) directory: /user/models/74LS74 (default output path and directory)

Command: `lm_model 74ls74`

Example 2

Input directory: /user/models/74ls74 ($MGC_WD is set to a directory other than /user/models)

Model file: 74LS74A.MDL (different from the component name)

Component (output) directory: /user/models/74LS74 (default output path and directory)

Command: `lm_model /user/models/74ls74 -m 74LS74A`

Example 3

Input directory: /user/models/74ls74 ($MGC_WD is set to /user/models)

Model file: 74LS74.MDL (same as the component name)

Component (output) directory: /user/project_xyz/74LS74 (default directory; non-default path)

Command: `lm_model 74ls74 /user/project_xyz`

Example 4

Input directory: /user/models/74ls74 ($MGC_WD is set to /user/models)

Model file: 74LS74.MDL (same as the component name)

Component (output) directory: /user/project_xyz/latch_7474 (non-default path and directory)

Command: `lm_model 74ls74 /user/project_xyz -d latch_7474`

tmg_to_ts Command Reference

The tmg_to_ts utility reads the Shell Software timing files to create a technology file. Comments from the Shell Software timing statements are not copied to the technology file. You must use the technology compiler (tc) to compile the technology file that is created by the tmg_to_ts utility before using the technology file in QuickSim II.

In general, you should run the lm_model utility—which calls both tmg_to_ts and tc— rather than running the stand-alone tmg_to_ts utility. If you just want to update a model's Technology File, you can run lm_model with the -Step Timing option. For more information on Technology File creation, refer to "Verifying the Technology File" on page 253.

Syntax

tmg_to_ts *input_dir* [**-**Out *filename*] [**-**Replace] [**-**Help] [**-**Usage] [**-**Version]

Required Arguments

input_dir   Specifies the pathname to the Shell Software timing files that you want to convert. If the *input_dir* is not a full path, it is assumed to be relative to the current directory, specified by $MGC_WD.

Optional Arguments

-Out *filename*   Specifies an alternative filename for the output file. By default, the output file is called technology.ts. All pathname specifications use the following location convention:

                               If the pathname is a relative pathname, the output file is placed relative to the component directory.

                               If the pathname starts with period and slash (**./**), the output file is placed in the current working directory as specified by $MGC_WD, if it exists.

                               If the pathname starts with a dollar sign (**$**), the output file is placed in the location of the location map variable specified after the dollar sign.

                               If the pathname is an absolute pathname, the output file is placed in the location of the absolute pathname.

-Replace   Replaces the existing contents of the output directory with the new output.

-Help   Prints help information on each of the available options, then immediately exits.

-Usage   Expands the command line and displays each argument and switch. After printing the usage message, tmg_to_ts immediately exits.

-Version   Prints the single-line version message, then immediately exits.

A

Using VERA with FlexModels

Overview

VERA is a testbench automation tool that works as a front-end to Verilog or VHDL simulators. For general information on VERA, refer to:

http://www.synopsys.com/products/vera

This appendix explains how to use VERA with FlexModels, including a special section on how to use VERA with FlexModels with VCS. This information is presented in the following sections:

- "Using FlexModels with the VERA UDF Interface" on page 270

- "Creating a VERA Testbench" on page 271

- "VERA Testbench Example" on page 272

- "Incorporating FlexModels in a VERA Testbench" on page 274

- "Using VERA with VCS" on page 275

☞ **Note**

For information on using MemPro models with VERA, refer to the "VERA Testbench Interface" chapter in the *MemPro User's Manual*.

Using FlexModels with the VERA UDF Interface

FlexModels use the VERA UDF interface. To use FlexModels with Vera, you need to build the VERA dynamic library. Table 33 lists the files containing the Vera UDF information for FlexModels.

Table 33: FlexModel Files Containing VERA UDF Information

File Name	Description	Location
vera_user.c	Source file containing table of UDF functions used by FlexModels.	$LMC_HOME/sim/vera/src
vera_slm_pli.o	Object file for VCS and Verilog-XL. This file contains the compiled code for the UDF functions used by FlexModels.	$LMC_HOME/lib/*platform*.lib
vera_slm_mti.o	Object file for MTI Verilog and MTI VHDL. This file contains the compiled code for the UDF functions used by FlexModels.	$LMC_HOME/lib/*platform*.lib

When building the VERA dynamic library, you need to compile the vera_user.c file and link the object file (vera_slm_pli.o or vera_slm_mti.o) for the simulator you are using.

Attention

If you are building the VERA dynamic library for Verilog on Solaris, do not use the -B symbolic. Using this switch results in unresolved symbol warnings.

For more information on building the VERA dynamic library, refer to the UDF information in the *VERA System Verifier User Manual*.

Linking VERA with Verilog Simulators

For details on how to link Vera with Verilog simulators using the PLI, refer to the README file in the VERA installation directory:

```
$VERA_HOME/lib/vlog/README
```

In the instructions for "Including Your Own C-Code (UDF) with VERA" at the end of the README file:

1. Substitute the edited copy of vera_user.c in $LMC_HOME/sim/vera/src on the compile line.

2. Substitute vera_slm_pli.o for `<UDF_o_files>` on the link line if you are using VCS or Verilog-XL. For MTI Verilog, substitute vera_slm_mti.o.

Linking VERA with the MTI VHDL Simulator

For information on how to link VERA with MTI VHDL, refer to the README file in the VERA installation directory:

```
$VERA_HOME/lib/mti/README
```

In the instructions for "Linking in VERA UDFs Into The MTI-VHDL Simulator" at the beginning of the README file:

1. Substitute the edited copy of vera_user.c in $LMC_HOME/sim/vera/src on the compile line.

2. Substitute vera_slm_mti.o for `<o_files>` on the link line.

Creating a VERA Testbench

To create a VERA testbench to use with FlexModels, follow these steps:

1. Include the header files.

Table 34 lists the two required header files.

Table 34: VERA Header Files

File Name	Description	Location
flexmodel_pkg.vrh	Contains definitions for generic constants useful in FlexModel commands.	$LMC_HOME/sim/vera/src
*model*_pkg.vrh	Contains definitions for model class and model-specific constants useful in FlexModel commands.	$LMC_HOME/models/*model*_fx/*model*_fx*version*/src/vera

2. **Create an instance of the *Model*Fx class.**

    Before using FlexModel commands, you must create an instance of the *Model*Fx class in the VERA testbench.

3. **Send commands to a FlexModel through the model's methods.**

    In VERA Command Mode, you can use the same FlexModel features and commands that you use in HDL Command Mode. There are a few differences in command usage, however; refer to "Command Syntax Differences in VERA Command Model" in the *FlexModel User's Manual*. For details on specific commands, refer to "FlexModel Command Reference" in the *FlexModel User's Manual*.

VERA Testbench Example

The following example shows how to incorporate FlexModels in a VERA testbench.

```
#include <vera_defines.vrh>        // Vera Defines
#include "flexmodel_pkg.vrh"       // FlexModel generic constants defined
here
#include "model_pkg.vrh"           // Model class, and model-specific
constants defined here

program model_test
{

    /*
     * Create an instance of the model, argument 1 to the
     * constructor is the string name of the instance in
     * top level Verilog/VHDL testbench.
     * argument 2 is the path to the models clock pin
     * Here the assumption made is that the model is
     * instantiated in a Verilog testbench
     * Since the constructor has been called, this will
     * return at the next posedge of u1.CLK.
     */
    ModelFx    model_1 = new("modelInstName_1", "u1.CLK");

    // Create another instance, since time has already elapsed
    // above, this call will return immediately.
    ModelFx    model_2 = new("modelInstName_2", "u2.CLK");

    // NOTE : This example assumes that the aguments to the
    //        methods have been defined in the VERA testbench.

    // Check that no errors have occured
    if ( model_1.showStatus() == FLEX_VERA_FATAL ||
         model_2.showStatus() == FLEX_VERA_FATAL ) {
```

```
            // Errors exist, take suitable action
        }

        fork
        {
            // Send commands to the FlexModel Instance 1

            // Note that the id is encapsulated in the model
            // class and thus is not an argument to the commands.

            model_1.write(address1, data1, `FLEX_WAIT_F, status);
            model_1.write(address2, data2, `FLEX_WAIT_F, status);
            model_1.write(address3, data3, `FLEX_WAIT_F, status);
            model_1.write(address4, data4, `FLEX_WAIT_F, status);

            // Read Back
            model_1.read_req(address1, `FLEX_WAIT_F, status);
            model_1.read_req(address2, `FLEX_WAIT_F, status);
            model_1.read_req(address3, `FLEX_WAIT_F, status);
            model_1.read_req(address4, `FLEX_WAIT_F, status);

            // Check Results
            model_1.read_rslt(address1, tag, result1, status);
            model_1.read_rslt(address2, tag, result2, status);
            model_1.read_rslt(address3, tag, result3, status);
            model_1.read_rslt(address4, tag, result4, status);

            // Synchronize Instance 1 & 2

            // Note that the generic commands are also sent to
            // through the model's instance.

            model_1.synchronize(2, "synch_2", `timeout, status);
        }
        {
            // Send commands to the FlexModel Instance 2
            model_2.write(address1, data1, `FLEX_WAIT_F, status);
            model_2.write(address2, data2, `FLEX_WAIT_F, status);
            model_2.write(address3, data3, `FLEX_WAIT_F, status);

            // Synchronize Instance 1 & 2
            model_2.synchronize(2, "synch_2", `timeout, status);
        }
        join // End of fork

    } // End of program model_test
```

Incorporating FlexModels in a VERA Testbench

To incorporate FlexModels in your VERA testbench, use the following procedure. For more information on creating VERA interface files and using models in VERA, refer to the *VERA System Verifier User Manual.*

1. **Create a working directory and run flexm_setup to make copies of the model's interface and example files there, as shown in the following example:**

       % **$LMC_HOME/bin/flexm_setup -dir** *workdir model*_**fx**

   You must run flexm_setup every time you update your FlexModel installation with a new model version. Table 35 lists the files that flexm_setup copies to your working directory.

Table 35: FlexModel VERA Files

File Name	Description	Location
*model*_pkg.vr	Contains FlexModel VERA class and method definitions.	*workdir*/src/vera
*model*_pkg.vrh	Contains model definitions for use in VERA testbenches.	*workdir*/src/vera

2. **Compile the VERA source files in the LMC_HOME tree.**

   You need to compile three files: lstmodel.vr, swiftmodel.vr, and flexmodel_pkg.vr. The following is a sample compile script:

       % **vera -cmp lstmodel.vr -I$LMC_HOME/sim/vera/src**
       % **vera -cmp swiftmodel.vr -I$LMC_HOME/sim/vera/src**
       % **vera -cmp flexmodel_pkg.vr -I$LMC_HOME/sim/vera/src**

   If you are using VERA version 4.0 or earlier, you must compile the flexmodel_pkg.vr object with a "VERA_4" preprocessor flag. Your compile line would therefore look like the following example:

       % **vera -cmp flexmodel_pkg.vr -I$LMC_HOME/sim/vera/src -DVERA_4**

3. **Compile the model's VERA source file,** *model*_**pkg.vr**

   This file includes the flexmodel_pkg.vrh file, but the VERA compiler needs to find the other header files too; therefore, you must include the path to the other header files. The following is a sample compile script:

       % **vera -cmp** *model*_**pkg.vr -I$LMC_HOME/sim/vera/src** *workdir*/**src/vera**

4. **Create a VERA testbench.**

   For details, refer to "Creating a VERA Testbench" on page 271.

5. Compile the VERA testbench.

Although you need to include only the flexmodel_pkg.vrh and *model*_pkg.vrh files in your VERA testbench, the VERA compiler needs to find the other header files too; therefore, you need to include the path to the VERA header files included in LMC_HOME. The following is a sample compile script:

```
% vera -cmp vera_testbench -I$LMC_HOME/sim/vera/src workdir/src/vera
```

6. Run the VERA testbench in a Verilog or VHDL simulation environment.

When you run the Verilog or VHDL simulator, the VERA simulator needs to load your compiled VERA object files. You also need to load the following VERA object files:

- lstmodel.vro

- swiftmodel.vro

- flexmodel_pkg.vro

- *model*_pkg.vro

- *testbench*.vro

For more information on loading VERA object files, refer to the *VERA System Verifier User Manual.*

⏰ **Attention**

To prevent your simulation from ending prematurely in cases where the VERA testbench completes before the Verilog/VHDL testbench, use the +vera_finish_on_end switch on your simulator invocation line.

Using VERA with VCS

The following steps show how to use FlexModels with VERA and VCS. This is just one way of using the VERA simulator's UDF, multiple .vro files, and so on. For more information, refer to the *VERA System Verifier User Manual*. All steps shown here are also documented in that manual.

1. Compile the VERA source files:

```
% vera -cmp \
    -I$LMC_HOME/sim/vera/src \
    $LMC_HOME/sim/vera/src/lstmodel.vr

% vera -cmp \
    -I$LMC_HOME/sim/vera/src \
    $LMC_HOME/sim/vera/src/swiftmodel.vr
```

```
% vera -cmp \
    -I$LMC_HOME/sim/vera/src \
    $LMC_HOME/sim/vera/src/flexmodel_pkg.vr
```

If you are using VERA version 4.0 or earlier, you must compile the flexmodel_pkg.vr object with a "VERA_4" preprocessor flag. Your compile would therefore look like this:

```
% vera -cmp flexmodel_pkg.vr -I$LMC_HOME/sim/vera/src -DVERA_4
```

```
% vera -cmp \
    -I$LMC_HOME/sim/vera/src \
    -Iworkdir/src/vera \
    workdir/src/vera/model_pkg.vr
```

👉 **Note**

If you are building the VERA dynamic library on Solaris, do not use the -B symbolic switch. Using this switch results in unresolved symbol warnings.

2. Compile your VERA testbench:

```
% vera -cmp \
    -I$LMC_HOME/sim/vera/src \
    -Iworkdir/src/vera \
     testbench.vr
```

This step produces two files: testbench.vro and testbench.vshell.

3. Build vera_local.dl:

❍ Compile $LMC_HOME/sim/vera/src/vera_user.c

HP-UX

```
% /bin/c89 -c +z -I$VERA_HOME/lib \
$LMC_HOME/sim/vera/src/vera_user.c
```

Solaris

```
% cc -K pic -c -I$VERA_HOME/lib \
$LMC_HOME/sim/vera/src/vera_user.c
```

❍ Link in $LMC_HOME/lib/*platform*.lib/vera_slm_pli.o and vera_user.o during the link stage of building the vera_local.dl.

HP-UX

```
% ld -b +e syssci_prod_entry +e errno -o vera_local.dl \
     vera_user.o \
     $LMC_HOME/lib/hp700.lib/vera_slm_pli.o \
     $VERA_HOME/lib/vlog/libvlog_br.a \
     $VERA_HOME/lib/libVERA.a -lm -lc
```

Solaris

```
% ld -G -z text -o vera_local.dl \
     vera_user.o \
     $LMC_HOME/lib/sun4solaris.lib/vera_slm_pli.o \
     $VERA_HOME/lib/vlog/libvlog_br.a \
     $VERA_HOME/lib/libVERA.a
```

4. Set the VERA_HOME variable to point to the location of your VERA installation directory:

```
% setenv VERA_HOME path_to_VERA_installation
```

5. Set the SSI_LIB_FILES variable to point to the vera_local.dl that you built in Step 3:

```
% setenv SSI_LIB_FILES ./vera_local.dl
```

☞ **Note** ───

If you are using multiple dynamic libraries (.dl files), use a colon-separated list to specify the search path.

───

6. Modify the simv build.

Modify the simv build by adding the following:

❍ -P ${VERA_HOME}/lib/vera_pli_dyn.tab

❍ ${VERA_HOME}/lib/libSysSciTask.a

❍ the vshell file created in Step 2

☞ **Note** ───

For HP-UX, add -LDFLAGS -E.

───

For more information, refer to the installation and setup chapter in the *VERA System Verifier User Manual*.

7. Create a file for VERA to load at runtime.

This step assumes that the vro files are in the current working directory. You need to create a file that looks like the following example. The file name for this example file is files_to_load:

```
./lstmodel.vro
./swiftmodel.vro
./flexmodel_pkg.vro
./model_pkg.vro
./testbench.vro
```

For more information, refer to the documentation on vera_mload in the *VERA System Verifier User Manual*.

8. Run the simv executable

Add the +vera_mload switch as shown in the following example:

```
% simv +vera_mload = files_to_load +vera_finish_on_end
```

👉 **Note**

The +vera_finish_on_end switch prevents your simulation from ending prematurely in cases where the VERA testbench completes before the Verilog testbench.

B

LMTV Command Reference

Overview

LMTV is a PLI application that is used to interface SmartModels and FlexModels with Verilog-XL, NC-Verilog, and MTI Verilog. Note that VCS uses the SWIFT interface and not LMTV. You can control the features of the LMTV interface by using:

LMTV Command Line Switches

LMTV command line switches have a session-wide scope that impacts all SmartModel instances. Notice that the +laiobj switch, used by the LAI interface, is not used in either mode of the LMTV interface. Following are brief descriptions for each of the command line switches that you can use with the LMTV interface:

+notimingchecks   Disables timing checks (for example, setup and hold times) and their accompanying messages. The default is to perform the timing checks.

+[min | typ | max]delays   Specifies a single delay range for all SmartModel instances. The default is to use the delay range in the SmartModel's DelayRange or RANGE attributes.

+lmudtmsg or +laiudtmsg   Generates a list of the timing files loaded at simulation startup. This is equivalent to setting the command channel command TraceTimeFile to ON. The default is not to list the timing files. For more information about the command channel. refer to the *SmartModel Library User's Manual*.

+lmoldstr   Maps all SmartModel Library signal strengths to "strong" for all output events that have resistive strength. The default is to use resistive strength to reflect the true state of the SWIFT pin. Use this switch if you have a design that was created in the Verilog-XL-specific SmartModel Library environment and you want simulation conditions to match the Verilog-XL-specific SmartModel Library.

+lmoldtrans   Indicates that the historic style is to be used for transcribing messages. The historic style message contains references only to timing version names and does not specify any time units. The default is that messages contain references to both timing version names and model names. Timing values are in nanoseconds (ns). Use this switch if you want to match the Verilog-XL-specific SmartModel Library simulation conditions.

+lmresstr   Disables mapping of SmartModel Library signal strengths to "strong" strength, even if a historic model *model*.v file (vshell) is detected. Use this switch if you want your historic-mode design to use true resistive strengths. This switch only works with the SWIFT interface.

LMTV Commands

LMTV commands are predefined tasks that you place within your testbench or design. LMTV commands all begin with an $lm_ prefix. Some of them have historic counterparts, which begin with an $lai_ prefix. You can use any or all commands in either the SWIFT or the Historic SmartModel modes, except for the $lm_monitor_vec_map() command, which can be used only in SWIFT SmartModel mode.

☞ **Note**

The $lai_ commands are provided to support older designs. Therefore, you do not have to convert $lai_ commands to $lm_ commands. However, when starting a new design it is best to use the $lm_ commands and not the $lai_ commands.

Here is a list of the LMTV interface commands:

$lm_command() or $lai_command()

These commands provide access to the SWIFT command channel. You can use them to send a command to the session or to a model instance.

Syntax

$lm_command ("*session_cmmd_string*");

$lm_command (*inst_path*, "*model_cmmd_string*");

$lai_command ("*session_cmmd_string*");

$lai_command (*inst_path*, "*model_cmmd_string*");

Arguments

session_cmmd_string　　　　　The SWIFT interface command to be sent to the session.

inst_path　　　　　　　　　　The path name to the SmartModel instance to send the command to. Used only with model commands.

model_cmmd_string　　　　　　The SWIFT interface command to be sent to the model instance.

For more information about the SWIFT command channel, refer to *"The SWIFT Command Channel" on page 21.*

Examples

The following example sends the ReportStatus command to the instance "U1", causing it to generate a message reporting its configuration status.

```
% $lm_command ( "U1", "ReportStatus" );
```

The following example sends the TraceTimeFile off command to the session, causing it to stop issuing trace messages. Note that the absence of an instance name identifies the command as session-specific.

```
% $lm_command ( "TraceTimeFile  off" )
```

$lm_dump_file() or $lai_dump_file()

Use these commands to dump the memory contents of the instance *inst_path* into the file *filename*. This works only for memory models. If the specified file already exists, it is overwritten. Using this command eliminates the read cycles required to verify the success of a test.

You can reload the dumped file into a memory model using the $lm_load_file() command. The format of the dumped file is the same as the Synopsys memory image file format required by a memory model at initialization.

Syntax

$lm_dump_file (*inst_path*, *"filename"* [,*"file_type"*]);

$lai_dump_file (*inst_path*, *"filename"* [,*"file_type"*]);

Arguments

inst_path	The path name to the SmartModel instance whose memory information is to be dumped.
filename	The path name to the file that is to receive the dumped memory information from the model instance.
file_type	The type of configuration file to be dumped. The only allowed value is MEMORY, which is also the default. This argument is provided for compatibility with the historic environment.

$lm_help()

Use this command to display the syntax for all of the SWIFT interface commands.

Syntax

$lm_help();

Examples

The following example shows the results of issuing the command $lm_help.

```
C2 > $lm_help;

LMTV commands:
 lm_command( "session_command" ): execute a session command.
 lm_command( inst_path, "model_command" ):
   execute a model command.
 lm_dump_file( inst_path, "file_name", ["file_type"] ):
   dump memory into file.
 lm_load_file( inst_path, ["file_name", "file_type"] ):
   load file of programmable device or memory.
 lm_monitor_enable( inst_path [, "win_element" [,...]] ):
   enable window Monitor.
 lm_monitor_disable( inst_path [, "win_element" [,...]] ):
   disable window Monitor
  lm_monitor_vec_map( var_name, inst_path, "win_element" [, index]):
   map window to a variable for monitoring.
 lm_monitor_vec_unmap( [var_name,] inst_path ):
   unmap window to stop monitoring.
 lm_status( inst_path ):   dump instance status.

 Commands compatible with old release :
 lai_enable_monitor( "inst_path", [win_element],... ):
   enable window Monitor.
 lai_disable_monitor( "inst_path", [win_element],... ):
   disable window Monitor.
 lai_dump_file( "inst_path", "file_name", "file_type" ):
   dump memory into file.
 lai_load_file( "inst_path", "file_name", "file_type" ):
   load file of programmable device or memory.
 lai_status( "inst_path" ):  dump instance data.
```

$lm_load_file() or $lai_load_file()

Use these commands to load the memory contents of the file *filename* into the instance *inst_path*, which can be either a programmable device or a memory model. Using these commands eliminates the write cycles required to set up the contents of the model.

The load_file operation causes the selected model to reset its internal state to simulation startup conditions and then read the specified file.   After the file is read, the model is evaluated as a function of the new internal state and the current inputs and outputs are scheduled with zero delay. After this initial evaluation phase, the model behaves as it would normally.

You can load a model with any file type that would normally be accepted by the model at initialization. Additionally, the new configuration file you load is used for the specified model instance after any subsequent command to reset or reinitialize.

Syntax

$lm_load_file (*inst_path* [, "*filename*", "*file_type*"]);

$lai_load_file (*inst_path* [, "*filename*", "*file_type*"]);

Arguments

inst_path   The path name to the SmartModel instance into which the contents of *filename* is to be loaded.

filename   The path name to the configuration file that is to be loaded for the model instance specified by *inst_path*. The default is to use a path name defined with the defparam statement in the design.

file_type   The type of file to be loaded. Allowed values are MEMORY, JEDEC, PCL, and SCF. The default is to use the file type of the file defined with the defparam statement in the design.

$lm_monitor_enable() or $lai_enable_monitor()

$lm_monitor_disable() or $lai_disable_monitor()

Use these commands to enable or disable SmartModel Windows for one or more window elements of a model instance specified by *inst_path*. The SmartModel Windows feature allows you to view and change the contents of a model's internal registers through predefined windows, which usually reflect the model's internal state. After enabling SmartModel Windows, you can read from the register using an appropriate Verilog command or by adding the path name to the list of signals being traced. If you attempt to read from an internal register without enabling SmartModel Windows the window content is not read.

The $lm_monitor_enable() and $lm_monitor_disable() commands are provided for compatibility with the historic environment. You cannot access arrays of registers, as in memory window elements, using these commands. In addition, you cannot create dynamic windows needed for SmartCircuit models if you define a window in a configuration file. The $lm_monitor_vec_map() and $lm_monitor_vec_unmap() commands provide these capabilities.

☞ **Note**

Accessing internal states is memory-intensive, so you may notice some performance degradation when SmartModel Windows is enabled.

Syntax

$lm_monitor_enable (*inst_path* [, "*window_element*"   [,...]]);

$lm_monitor_disable (*inst_path* [, "*window_element*"   [,...]]);

$lai_enable_monitor (*inst_path* [, "*window_element*"   [,...]]);

$lai_disable_monitor (*inst_path* [, "*window_element*"   [,...]]);

Arguments

inst_path	The path name to the SmartModel instance for which SmartModel Windows is to be enabled.
window_element	The name of the internal register to read. This can be a single value or a list. The default is to read all internal registers of the instance.

Examples

The following example enables SmartModel Windows for all windows in instance "U1", then reads from the predefined window element IENA. Notice that you must enable SmartModel Windows before attempting to read from the window element.

```
// somewhere in the testbench ...
// enable access to all windows in instance U1
$lm_monitor_enable( U1 );
// display contents of window element IENA
$display("Value of register IENA is %h", $IENA);
```

The following example disables the window explicitly for the register IENA.

```
% $lm_monitor_disable( U1, "IENA" );
```

The following example disables all windows in instance U1.

```
% $lm_monitor_disable( U1 );
```

The following example does not read the register IENA, because SmartModel Windows was not enabled.

```
// somewhere in the testbench ...
$display("Value of register IENA is %h", $IENA);
```

$lm_monitor_vec_map() and $lm_monitor_vec_unmap()

Use these commands to enable or disable direct mapping between the user-defined variable *var_name* and a model instance's internal register *window_element*. This mapping allows you to read from, write to, or trace the internal register through your user-defined variable. You must define this variable with a width corresponding to that of the predefined window somewhere in the design hierarchy (typically in the testbench) before using these commands. Note that these commands only work in SWIFT SmartModel mode.

Using $lm_monitor_vec_map(), you can access arrays of registers, which is useful for addressing specific memory locations, as in the memory window elements feature. In addition, $lm_monitor_vec_map() allows dynamic window creation. Thus, if a SmartCircuit model changes its configuration file so that more windows are created, you can add those names to your testbench, and enable tracing directly.

Syntax

$lm_monitor_vec_map (*var_name*, *inst_path*, "*window_element*" [,*index*]);

$lm_monitor_vec_unmap ([*var_name*,] *inst_path*);

Arguments

var_name	The name of a user-defined variable to map to window_element. The variable must be already defined somewhere in the design hierarchy. The default for $lm_monitor_vec_unmap() is to unmap all mapped variables for that instance.
inst_path	The path name to the SmartModel instance whose internal register is to be mapped to the user-defined variable *var_name*.
window_element	The name of the internal register to be mapped to *var_name*. Can be part of an array.
index	The index of the array, if the window element is a memory window. The default is 0.

Examples

The following example defines three variables and maps them to specific memory locations in the memory array UMEM for memory model instance "U1". Note that these tasks cannot be performed using $lm_monitor_enable(). Although the example features an array of registers, the tasks are equally useful for scalar windows, where you can omit the index option or set it to 0.

```
// Assume a 4Kx8 memory model, on a controller board.
// Such a model would typically have one window called UMEM.
// This window is  a 4K deep array of 8 bit registers. In
// particular, the user is interested in these 3 locations:
//  Interrupt service routine, LOW ADDRESS: 100
//  Interrupt service routine, HIGH ADDRESS:  101
//  Control store :    200
// that are significant to the design.
reg [7:0] ISR_LOW;   // variable to map to location 100
reg [7:0] ISR_HIGH;  // variable to map to location 101
reg [7:0] CONTROL;   // variable to map to location 200
// enable monitoring of these variables
$lm_monitor_vec_map( ISR_LOW, U1, "UMEM", 100 );
$lm_monitor_vec_map( ISR_HIGH, U1, "UMEM", 101 );
$lm_monitor_vec_map( CONTROL, U1, "UMEM", 200 );

// ... at this time, you can read, write, or trace these
// variables.  For example, assign the address of the interrupt
// service routine to be 0x5000

ISR_LOW = 0x00 ;
ISR_HIGH = 0x50 ;

// or the same assignment can be done as follows:
define ISR {ISR_HIGH,ISR_LOW}
ISR = 16h5000 ;
// this one statement will access two different
// and independent memory locations at once.
// later in the simulation, you can disable monitoring
// for the 'CONTROL' register:
$lm_monitor_vec_unmap( CONTROL, U1 );
// or you can disable monitoring of all windows in that instance:
$lm_monitor_vec_unmap( U1 );
```

$lm_status() or $lai_status()

Use these commands to report the current status of the model instance *inst_path*. The report includes the names and values of internal windows.

Syntax

$lm_status (*inst_path*)

$lai_status (*inst_path*);

Arguments

inst_path   The path name to the SmartModel instance whose status is to be reported.

Examples

The following example shows the output of the $lm_status() command for model instance "U1".

```
C1 > $lm_status(U1);

Note: <>
Model template: mem
Version: not available
InstanceName: DESIGN.U1
TimingVersion: MEM-0
DelayRange: MAX
MemoryFile: memory.1
Timing Constraints: On
SmartModel Instance DESIGN.U1(mem:MEM-0), at time 1000.0 ns

Note: SmartModel Windows Description:
UMEM[2048] "2K x 8 Static RAM":
SmartModel Windows not enabled for this model.
SmartModel Instance DESIGN.U1(mem:MEM-0), at time 1000.0 ns
```

Index

Symbols

$add_instance command 227, 228
$display command 74
$lai_command command 282
$lai_disable_monitor command 70, 286
$lai_dump_file command 283
$lai_enable_monitor command 70, 72, 73, 286
$lai_load_file command 285
$lai_status command 70, 72, 290
$lm_command command 282
$lm_dump_file command 283
$lm_help command 284
$lm_load_file command 285
$lm_log_test_vectors 94
$lm_loop_instance 95
$lm_monitor_disable command 70, 286
$lm_monitor_enable command 70, 71, 72, 73, 286
$lm_monitor_vec_map command 70, 71, 74, 288
$lm_monitor_vec_unmap command 70, 288
$lm_status command 70, 72, 290
$lm_timing_information 96
$lm_timing_measurements 96
$lm_unknowns 97
+vera_finish_on_end switch 275

A

add breakpoint command 236
add bus command 237
add lists command 235, 237
add monitors command 235
add primitive command 228
add synonym command 237
add traces command 235
Admin tool 220
AIX

compiling C files 28
AMPLE_PATH environment variable 220
Analysis tools
   Mentor Graphics 247
   rules to determine descriptors 248
ANSI C compiler
   with Cyclone 191
Attributes
   SmartModel 18

B

Breakpoints
   setting with SmartModels 236
Bus symbols
   SmartModel 222
Buses
   renaming with hardware models 192

C

C compiler 191
ccn_report command 71
cds.lib file 200, 211
cds.lib path 208
CDS_INST_DIR environment variable 102
CDS_VHDL environment variable 196
cflags 191
Characters, mapping 192
Characters, replacing special 192
Check Shell Software utility 252
Command Channel
   SWIFT 231
Command interaction
   QuickSim II 231
Command line
   switches, LMTV 77
   switches, QuickSim II 230
Commands
   $add_instance 227, 228

N

W

Z