SYNOPSYS®

SmartModel Library User's Manual

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

February 2002

Contents

Figures

Tables

Preface

About This Manual

This manual contains user and reference information for SmartModel Library users. The focus is on how to use the SmartModel simulation models and tools. This manual does not contain information about installing the library—that information is presented in the *SmartModel Library Installation Guide*.

Related Documents

For general information about SmartModel Library documentation, or to navigate to a different online document, refer to the *Guide to SmartModel Documentation*. For the information on supported platforms and simulators, refer to *SmartModel Library Supported Simulators and Platforms*.

For detailed information about specific models in the SmartModel Library, use the Browser tool ($LMC_HOME/bin/sl_browser) to access the online model datasheets.

Manual Overview

This manual contains the following chapters and appendixes:

Preface	Describes the manual and lists the typographical conventions and symbols used in it; tells how to get technical assistance.
Chapter 1: SmartModel Library Features	Provides an overview of the library, including how model and tool versioning works and the different model types.
Chapter 2: About the Models	Overview of common model features and modeling assumptions.

Chapter 3: Browser Tool	How to use the Browser tool to select model versions and view product documentation.
Chapter 4: Memory Models	How to configure and use memory models.
Chapter 5: PLD Models	How to configure and use programmable logic device models.
Chapter 6: SmartCircuit FPGA Models	How to configure and use SmartCircuit models of FPGA and CPLD devices. Also describes how to use the debugging tools to enhance the usefulness of SmartCircuit models.
Chapter 7: Processor Models	How to configure and use full-functional and hardware verification models of microprocessors and microcontrollers.
Chapter 8: User-Defined Timing	How to use the user-defined timing feature to create your own custom timing versions for SmartModels.
Chapter 9: Back-Annotating Timing Files	How to use the Backanno tool to back-annotate timing values using Standard Delay Format (SDF) files.
Chapter 10: Library Tools	How to use the SmartModel Library command-line tools.
Appendix A: Reporting Problems	How to diagnose problems with SmartModels and request technical support when necessary.
Appendix B: Glossary	Definitions for terms that have special meaning in the context of this manual.

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

Comments?

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

doc@synopsys.com

To report an error that occurs 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 information to identify the source of the problem.

1
SmartModel Library Features

SmartModel Library Overview

The SmartModel Library is a collection of over 3,000 binary behavioral models of standard integrated circuits supporting more than 12,000 different devices. The library features models of devices from the world's leading semiconductor manufacturers, including microprocessors, controllers, peripherals, FPGAs, CPLDs, memories, and general-purpose logic. SmartModels connect to hardware simulators through the SWIFT interface, which is integrated with over 30 commercial simulators, including Synopsys VCS and Scirocco, Cadence Verilog-XL, and Mentor Graphics QuickSim II.

Instead of simulating devices at the gate level, SmartModels represent integrated circuits and system buses as "black boxes" that accept input stimulus and respond with appropriate output behavior. Such behavioral models are well suited for distribution in object code form because they provide improved performance over gate-level models, while at the same time protecting the proprietary designs created by semiconductor vendors.

All SmartModels are listed in the Model Directory, which you can find on the Web at:

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

This Web site provides the most up-to-date information about model availability and allows you to view model datasheets, which list all device components and manufacturers supported by each model in the library.

SmartModel Library Versioning

Models are the basic units in the library. You can install more than one version of any model in the same $LMC_HOME. Multiple model versions allow separate design teams to use different versions of the same model without interfering with each other. This means that design team #1 (for example) can get a bug fix that they need for a particular model without affecting design team #2 that may not care about that fix in the context of their work. You can also install new model shipments that you receive from Synopsys right into an existing SmartModel installation. The SmartModel environment makes it easy to make the latest simulation models available to design teams that need them without affecting the design teams that do not. Figure 1 illustrates the benefits of the flexible SmartModel Library versioning system. Different design teams can select new or revised models using custom configuration files. For more information, refer to "Configuration (LMC) Files" on page 48.

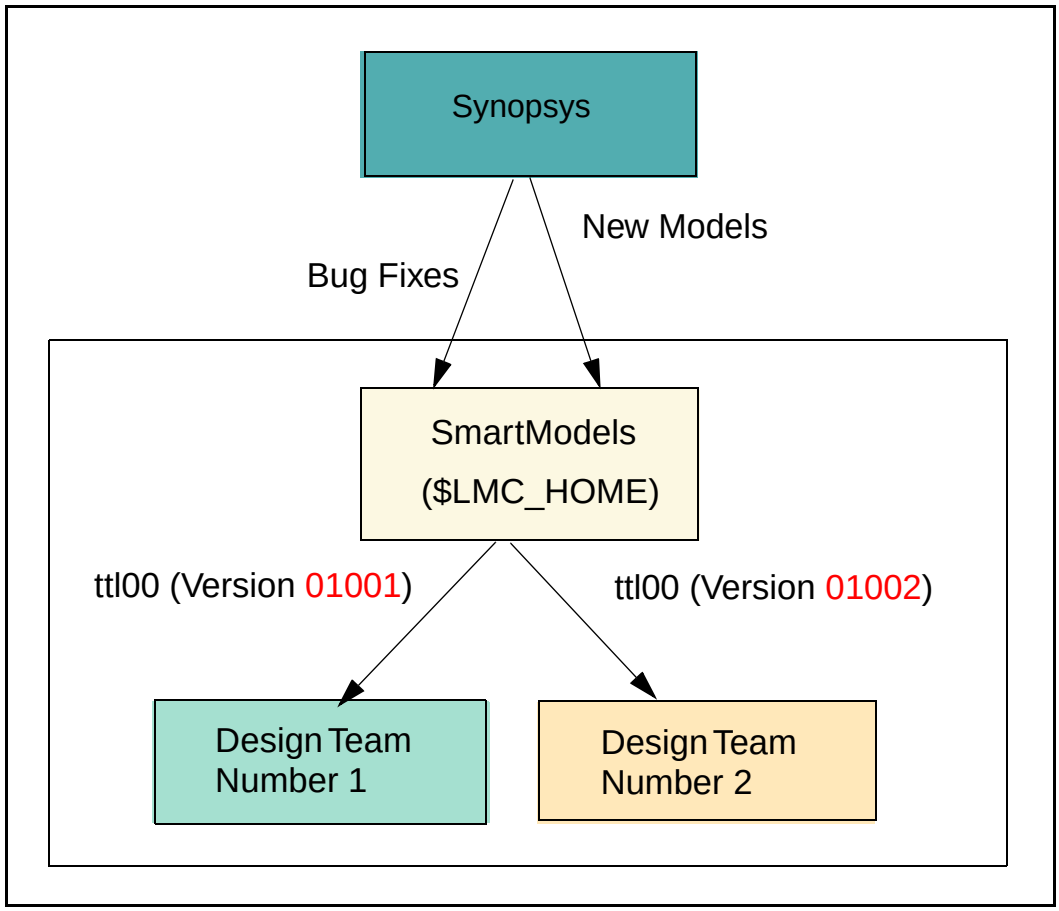


Figure 1: SmartModel Versioning Environment

Each versioned model supports multiple timing versions. Each timing version, in turn, can support multiple devices, or physical integrated circuits that you can order from a manufacturer. When you install a particular model version, you get all of the timing versions for that model. Similarly, when you purge a particular version of a model from the library, you purge all of its timing versions as well. Figure 2 illustrates how these concepts work together.

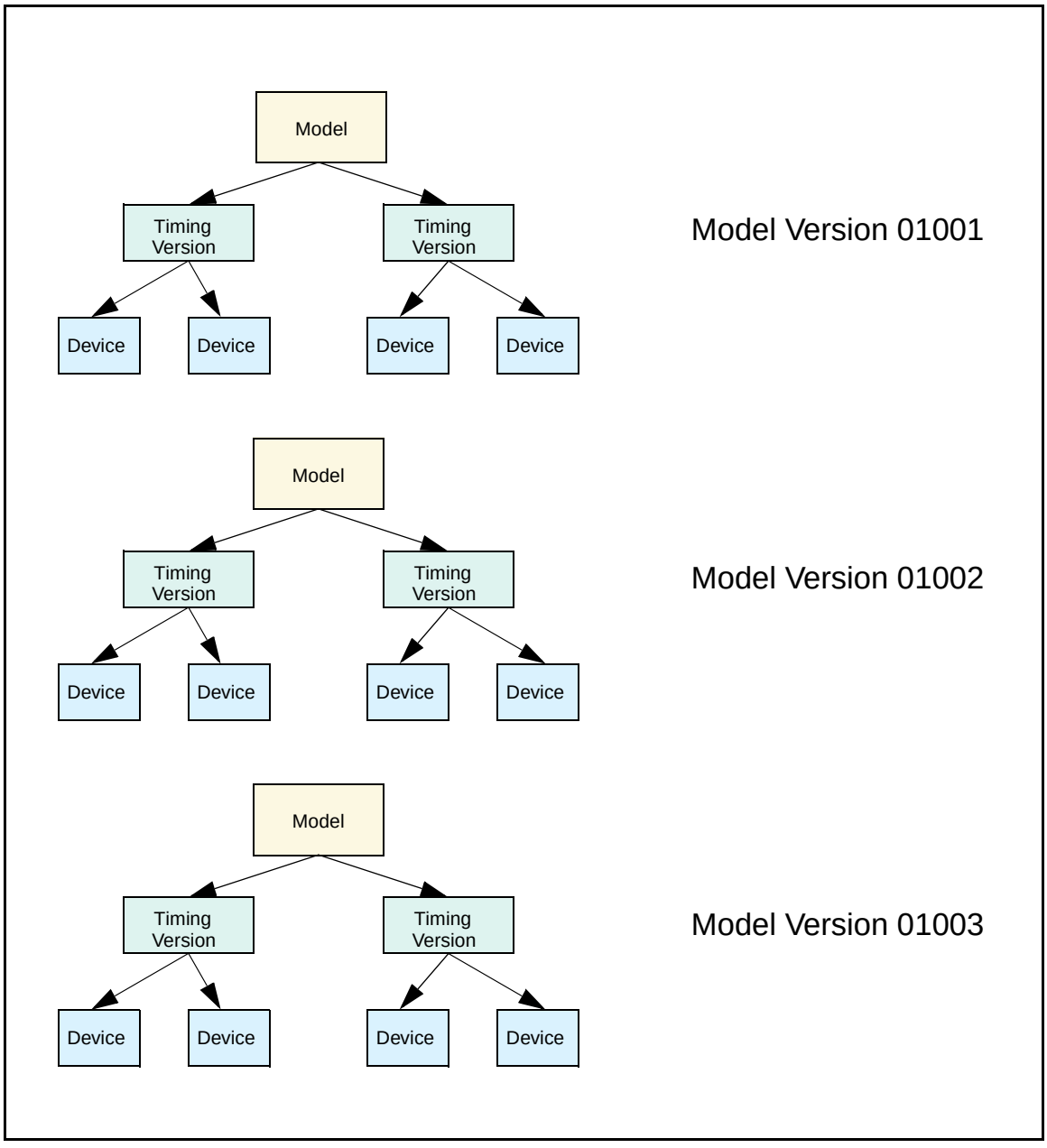


Figure 2: Model Versioning Overview

Note

To see all of the timing versions and components supported by any model in the library, review the model's datasheet.

Model Versions

Model versions have the five-digit format xxyyy (for example, 01001), where xx designates a major revision and yyy designates a minor revision. This five-digit version number appears on all model datasheets. Look in the banner section at the top of the datasheet for this key piece of information.

Tool Versions

SmartModel Library tools are also versioned and use the same numbering scheme as models. For some SmartModel Library tools, called model-versioned tools, the version of the tool to use is determined by the model. You cannot change the versions of model-versioned tools because a particular version of a model may depend on a specific tool version to function properly. Examples of model-versioned tools include compile_timing, ccn_report, smartbrowser, and smartccn. For other SmartModel Library tools, called user-versioned tools, you select the version of the tool to use via the default and custom configuration (LMC) files.

Note

For information on selecting specific model and tool versions, refer to "Selecting Models in $LMC_HOME" on page 44 and "Selecting Tool Versions" on page 45.

SmartModel Types

There are two basic types of SmartModels:

- Full-functional Models (FFMs) simulate the complete range of device behavior. Most SmartModels fall into this category.
- Bus-Functional Models (BFMs) simulate all device bus cycles. There are two types of BFMs in the SmartModel Library:
  - Hardware Verification (HV) models, which you control using Processor Control Language (PCL), a language that is similar to C.

- FlexModels, which you can control using Verilog, VHDL, or C.

For some devices, more than one type of model may be available, but these are exceptions, not the general rule. For detailed information about a specific SmartModel (including FlexModels), refer to the model's datasheet. For general information about FlexModels, refer to the *FlexModel User's Manual*.

Model Timing Versions

All SmartModels have at least one timing version. To see what timing versions are available for a particular model, use the Browser tool to display a list of timing versions for that model.

If you need a timing version that is not supplied with the library, or if you want to back-annotate customized delays into the model's simulation, you can create a custom timing version as described in "User-Defined Timing" on page 157.

Model Configuration

To configure a model means to define it completely, by doing the following:

- Setting environment variables and specifying the model version
- Creating technology-dependent setup files (JEDEC, MIF, or MCF). Different model types require different setup files. To find out the required setup file for a particular model, refer to the model's datasheet.
- Setting values for attributes that specify the instance name, timing version, propagation delay range, and location of setup files. The way you specify attributes depends on the simulator that you are using
  - Verilog simulators—use defparams in a .v file
  - VHDL simulators—use generics in a .vhd file
  - Schematic-capture based simulators—use model symbol properties

Note

For information on configuring SmartModels (including FlexModels) for use in your simulator, refer to the *Simulator Configuration Guide for Synopsys Models*.

SmartModel Windows

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. Using Windows, you can:

- Display the current value of an internal register
- Force new values into writable registers
- Set up a monitoring function to inform you when a register value changes

Note

FlexModels do not support SmartModel Windows.

You read and write to a model's internal registers through Windows using simulator-specific commands, one for each model instance. These commands can be issued on the command line or in a simulation script.

SmartModel Windows availability depends on the:

- **Model**—Only some models support Windows. Typically, these are processor, CPLD, FPGA, and memory models. Refer to the model's datasheet to find out if it supports Windows.
- **Simulator**—Currently, SmartModel Windows is supported by many simulators, including (but not limited to) Synopsys VCS, Cadence Verilog-XL and RapidSim, IBM AUSSIM, Mentor Graphics QuickSim II, ViewLogic ViewSim, MTI Verilog, and Lucent ATTSIM. Refer to your simulator documentation for information about Windows support.

Predefined Window Elements

For SmartModels that support Windows, Synopsys provides window elements with names, dimensions, and read/write features that correspond to registers specified by the manufacturer for the modeled device. These predefined window elements are documented in each model's datasheet.

Table 1 shows examples of predefined window elements taken from specific model datasheets.

Table 1: Examples of Predefined Windows from Model Datasheets

Model	Number of Elements	Element Name	Window Dimensions	R/W Access
pal32vx10	10	Q0-Q9	1 bit	Read/Write
i28f001bxb	2	COMMAND_REGISTER	8 bits	Read Only
		PROTECT_STATUS	1 bit	Read/Write
xc17128d	1	BIT_ADDR_REGISTER	18 bit	Read/Write
mt581c64k18b2	3	MEM	65K x 18 bit array	Read/Write
		MEM_addr	16	Read Only
		MEM_rw	2	Read Only

In addition to the information in Table 1, some model datasheets provide instructions on how to use a particular model's window elements.

Memory Windows

Notice that the predefined window elements in Table 1 represent either single-bit registers, one-dimensional arrays (multibit registers), or two-dimensional arrays. Two-dimensional arrays are used for memory array windows.

Within SmartModel Windows, memory array windows let you monitor events that take place in a memory array during simulation. The memory array can be part of a memory device or any other device that contains on-chip memory. Using this functionality, you can monitor read and write operations in the array without individually monitoring every array location. (For large memories, monitoring every array element is not practical.)

Each model's memory array has three associated windows: the Memory Array Window, the Memory Address Window, and the Memory Read/Write Window.

The Memory Array Window:

- Represents the memory array itself
- Has the same dimensions as the specific model's memory array (for example, a 16K x 32-bit memory)
- Supports read/write access

- Has a model-specific name that is specified in the model datasheet
- Has an initial value of "X" (unknown)

  For example, in Table 1 the model mt581c64k18b2 has a 64k x 18 bit array named MEM that has read/write access.

The Memory Address Window:

- Represents the array index
- Has as many bits as needed to contain the binary representation of the array size
- Is read-only
- Is named by appending "_addr" to the memory array window name.

  For example, in Table 1 the model mt581c64k18b2 has a 16-bit memory address window named MEM_addr that has read-only access.

  The memory address window is loaded with the appropriate index value each time there is a memory array transaction. Thus, the contents of this window always represent the index of the last location accessed by either a read or a write. In rare cases where more than one array location is accessed during a single model evaluation, the memory address window contains all Xs (unknowns); this is also the initial value of the memory address window.

The Memory Read/Write Window:

- Represents the read/write and access status
- Is always 2 bits wide
- Is read only
- Is named by appending "_rw" to the memory array window name
- Has an initial value of "1X"

  For example, in Table 1 the model mt581c64k18b2 has a 2-bit Memory Read/Write window named MEM_rw that has read-only access.

  The least-significant bit of the Memory Read/Write window is the Read/Write bit, which initially is X (unknown) and contains a:

  - 0 if the array is written to
  - 1 if the array is read from
  - X if an array read and write occurred in the same cycle

For ROM models, this bit always contains a 1.

The most-significant bit of the Memory Read/Write window is an access flag, which is toggled each time a model evaluation causes a read or a write of the memory array. Thus, you can detect a situation where there may have been two consecutive memory transactions of the same type at the same address.

Refer to the individual model datasheets for the names, dimensions, and read/write access characteristics of each model's predefined memory windows.

Using Window Elements

You read from and write to window elements using simulator-specific commands. For details, refer to the command set for your simulator.

For examples of using window elements with Cadence Verilog-XL or Mentor Graphics QuickSim II, refer to the *Simulator Configuration Guide for Synopsys Models.*

SmartModel Datasheets

SmartModel datasheets provide specific user information about each model in the library. The model datasheets supplement, but do not duplicate, the manufacturer's datasheets for the hardware parts. In general, the model datasheets describe:

- Supported hardware components and devices
- Bibliographic sources used to develop the model (i.e., specific vendor databooks or datasheets)
- How to configure and operate the model
- Any timing parameters that differ from the vendor specifications
- How to program the device (if applicable) or otherwise use it in simulation
- Differences between the model and the corresponding hardware device

Models are partitioned by function, including:

- Processors/VLSI
- Programmables
- Memories
- Standards/Buses
- General Purpose

SmartModel datasheets have standard sections that apply to all models and model-specific sections whose contents depend on the model type. The following sections provide general information about what to expect from the various sections in a SmartModel datasheet.

Note

FlexModel datasheets follow a different format than other SmartModel datasheets, but are similarly designed to provide you with all the information needed to successfully use the model.

Title Banner

The title banner provides information about the model name, title, function, subfunction, MDL version number, and date of the last change to the model.

MDL Version Numbers and Model History

With the SmartModel Library, model versions are called MDL version numbers. Not all MDL version number changes are significant to model users. For example, making an editorial change in a model's datasheet will cause the model's MDL version to increment, but model users would see no difference in the behavior of the model with the later version. For this reason, the model history section at the end of each SmartModel datasheet lists model history only for significant changes, where the model would behave differently in simulation.

Providing model history just for significant changes also means that there will often be gaps in the published model history. For example, the title banner on a model datasheet might reference MDL version 01024, but the model history section shows the last significant model change to be at MDL version 01021. This means that the intervening MDL version numbers (01022 and 01023) did not change model behavior in any way that would be visible to users. Note that all model bug fixes generate model history entries and cause the MDL version number to increment.

Supported Components and Devices

Each model datasheet includes a section entitled "Supported Components and Devices" which lists all of the hardware parts, by manufacturer, that the model can substitute for during simulation. In this section, each component represents one of the unique timing versions or speed grades supported by the model. Similarly, each device represents a hardware part that you can buy from the listed manufacturer.

Sources

This section lists all of the specific bibliographic references used for information about the behavioral and timing characteristics of the modeled device.

Model History

This section appears at the end of each datasheet. It only contains model history change information if there were significant changes to the model's behavior in the previous year. Read this section to get information on the latest model versions. Each change entry that appears in the "Model History" section notes the model version number associated with that particular change. For more information about model history, refer to "Model History and Fixed Bugs" on page 214.

Models versions are identified with the five-digit MDL version number that appears at the top of every SmartModel datasheet, in the Banner section.

Getting SmartModel Datasheets

You can get SmartModel datasheets several different ways:

- Through the Browser tool. To make a datasheet appear, just select the model you are interested in and click on the datasheet icon in the upper left portion of the vertical tool bar.
- Through the Model Directory on the Web:

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

Note

The Model Directory on the Web always provides datasheets for the latest SmartModel versions. The Browser tool shows you datasheets only for models installed at your site, which may or may not be the latest versions available.

SystemC/SWIFT Support

Synopsis provides a SystemC/SWIFT interface that supports Flex Models. SystemC is a C++ class library used for creating cycle-accurate models of software algorithms, hardware architecture, and interfaces for System-on-Chip (SoC) and system-level designs. As part of its class library, SystemC provides a cycle simulation environment, is designed to work with event-driven logic simulators, and provides extensive support for modeling device timing accurately. For more details see the *SmartModel Products Application Notes Manual*.

2
About the Models

Introduction

SmartModel Library models are behavioral simulation models of integrated circuits. This chapter provides information about standard and model-specific SmartModel features in the following major sections:

- Features Common to Most Models
- Implementation-Specific Model Features
- Modeling Assumptions
- Modeling Changing or Uncertain States

Features Common to Most Models

SmartModel Library models have many features in common. This bedrock technology helps give models from the library a similar look and feel that makes them easier to use. Common features include 64-bit time, supported logic values, status reporting, error checking, unknown handling, user-defined timing, and selectable propagation delays.

64-Bit Time

All SmartModels use 64-bits to compute elapsed simulation time. If simulation time exceeds this capacity, the models behave unpredictably.

Logic Values

Models in the SmartModel Library use a logic value system based on the IEEE 1164.1 VHDL nine-state, multivalue logic system, as shown in Table 2.

Table 2: SmartModel Logic Values

Symbol	Meaning
0	Strong 0
1	Strong 1
X	Strong X
L	Weak 0
H	Weak 1
W	Weak Unknown
Z	High-Impedance
U	Uninitialized (treated as unknown)
D	Don't care (treated as unknown)

Model Status Reports

You can generate a model status report by issuing the ReportStatus command through the command channel at any time during a simulation. Model status reports contain the following information:

- Model name and version
- Model attributes and their values
- Timing constraint setting
- Names, descriptions, and values of each window element
- Names and descriptions of each window array

The general format of the model status report message is as follows:

```
Note: <<Status Report>>
Model Logical Name: swiftnand
Model Physical Name: swiftnand
Model '.mdl' Version: 01000
Model Directory: /lmc_home/models/swiftnand
Model '.lmc' Name: /lmc_home/data/hp700.lmc
Model-reported Version of Main Shared Library: 01000
Model-reported Version of SmartLink Interface: v1.0
InstanceName: I$xx
TimingVersion: SWIFTNAND-1
DelayRange: MIN
Timing Constraints: On
```

Error Checking

SmartModel Library models provide usage and timing checks that display error, note, trace, or warning messages during simulation. The format and location of these messages depends on the design environment, but the content is essentially the same.

Usage Checks

Usage checks, which vary greatly with device type, help ensure a chip is used correctly. For example, a DMA controller model might include a check on whether or not all internal nodes and registers were initialized. An SRAM check might produce a message like: "Address, A0-A13, changed within Write cycle." These checks also document times, device names, instances, and error types.

For example:

```
WARNING: Ignoring unknown signal level on HALT pin, assume pin inactive.
(TEST68K MC68020RC12.1P) [MC68020-12], at 185.0 ns
```

Elements of this example have the following meanings:

- Error type:

  ```
  WARNING: Ignoring ...
  ```

- Design name:

  ```
  TEST68K
  ```

- Device name:

  ```
  MC68020RC12.1P
  ```

- Timing version:

  ```
  MC68020-12
  ```

Timing Checks

Timing checks include the component-specific set-up, hold, frequency, pulse width, and recovery times specified in the semiconductor vendor's specifications. Timing checks generate a single value—they do not have a range and thus are not affected by the propagation delay range.

For example:

```
ERROR: PULSE WIDTH on CLK (High) was 1.0 ns; 10.0 ns is the specified
minimum
(DEMO68K PAL16L8A-2MJ.167P) [MMI_16L8A-COM], at 12201.0 ns
```

Nominal and Worst-Case Specifications for Timing Checks

In cases where a manufacturer specifies both nominal and worst-case values for a timing parameter, the model always uses the worst-case specification.

Turning Off Timing Checks

You can turn off timing checks using any of the following methods:

- Through the SWIFT command channel with the SetConstraints ON | OFF command (not available for FlexModels).
- Using a simulator command implemented by the vendor. Note that this method may not be available; refer to your simulator documentation for information about turning timing checks on or off for SmartModels.
- With explicit settings in a user-defined timing file.

Handling of Unknowns

SmartModels make the most of each simulation by generating or propagating unknowns only when necessary. When appropriate, a model issues a warning message rather than propagating an unknown. This pessimistic unknown handling can preserve the usefullness of a simulation.

User-Defined Timing

Most SmartModels support user-defined timing. If you need a model timing version that Synopsys has not provided, you can create custom timing files to use in simulation. For more information, refer to "User-Defined Timing" on page 157.

Note

Netlist-driven SmartCircuit models do not support user-defined timing.

Selectable Propagation Delays

All SmartModels support a range of propagation delay values to represent minimum, typical, and maximum specifications.

Implementation-Specific Model Features

The availability of some SmartModel features depends on whether they are supported in your particular simulation environment. Consult your simulator documentation to determine which of these capabilities are supported and how to access them. Following are brief introductory descriptions of these implementation-specific model features.

Fault Simulation

Fault simulation allows concurrent evaluation of multiple faulty circuits in a design. The SmartModel Library supports this feature as an extension to the logic simulation capabilities of SmartModels.

Save and Restore Operations

If your simulation environment supports Save and Restore operations, you can save a simulation state, then later restore the saved state to the circuit using the SmartModel Library save and restore capability.

Timing Check Control

Timing check control allows you to control the timing constraint checks issued during simulation. By default, all SmartModels start up with setup, hold, and recovery timing checks enabled. The timing check control feature allows you to enable and disable these timing checks at any time during a simulation. This feature affects all timing checks of a model instance except those explicitly turned off in the external timing file.

Model Reset

SmartModels support the ability to reset a model to its initial state at any time during a simulation. The model reset operation resets the internal state variables, but not the input port values, attribute values, and mode settings, which retain their current values. For example, if timing checks are disabled before performing the reset operation, then timing checks remain disabled even though the default is for timing checks to be enabled.

Model Reconfiguration

SmartModels support the ability to force a model to reload any or all of its configuration files (memory image, JEDEC, MCF, PCL), or select a new timing version at any time during a simulation.

Modeling Certain Timing Relationships

In some cases a model's timing specifications do not map perfectly to a semiconductor vendor's specifications. One example of this type of variance is where a crystal clock input is used to update an internal state machine that in turn drives the outputs of the device. The vendor's datasheet describes only the timing relationships among external output signals without reference to the internal clock.

To accurately model the internal behavior of such a device, the model timing is specified relative to the internal clock. For example, in the model of the Intel 87C51FA 8-bit microcontroller, timing is specified relative to the internal clock (XTAL2). These timing values are derived from Intel's data—the values faithfully reproduce the relationships among external signals as they are described in the vendor's datasheet.

The internal clock-to-output timing relationships are modeled to approximate the vendor's output-to-output timing specifications. If you need to duplicate individual vendor specifications, you can use the user-defined timing feature of the SmartModel Library, but note that there is no way to model the complete set of vendor timings simultaneously.

License Unavailability

In order to prevent your simulation tool from continuing when no license is available for the DesignWare SmartModel or FlexModel, use the `NoLicenseFatal` feature described here.

DesignWare SmartModels (including FlexModels) based on the SWIFT standard take a command called `NoLicenseFatal` which can be set to either `ON` or `OFF`. The default is `OFF`.

```
setenv LMC_COMMAND "NoLicenseFatal ON"
setenv LMC_COMMAND "NoLicenseFatal OFF"
```

When set to `ON`, `NoLicenseFatal` causes a simulation session with models using the SWIFT interface to send a fatal error message to the simulator if any SmartModel in the simulation fails to authorize, causing the simulation to exit when the first Smartmodel licensing error (or denial) has occurred.

When set to `OFF`, the simulation will run, but any SmartModel or FlexModel without a license will have its outputs set to x.

Synopsys supports Scirocco, VCS, and MTI VHDL with this feature.

Because the memory modeling product, MemPro, does not use SWIFT, MemPro licensing errors alone will not terminate the simulation even with `NoLicenseFatal` set to `ON`. A testbench that contains MemPro models but no SmartModels will not terminate on a license failure in any condition.

Several simulators currently do not stop simulating when they encounter the fatal error caused by the license being unavailable when the `NoLicenseFatal` variable is set to `ON`. This includes simulators which rely on the LMTV integration (MTI Verilog, NC-Verilog, and Verilog-XL) as well as at least one other simulator, NC-VHDL, which relies on a vendor provided integration. However, note that all simulators, including those that use LMTV, show a fatal error in the lmc_trace file when the `NoLicenseFatal` variable has been set to `ON`.

Modeling Assumptions

In most cases SmartModel Library models represent all of a device's functional behavior, but there are exceptions. Sometimes a manufacturer does not quantify a component parameter or a model is designed in cooperation with a vendor before the actual device is available in silicon. In addition, there are some device capabilities that do not make sense in the context of logic simulation, because the chip's electronic environment is not the same as a simulation environment. The electrical programming of an EEPROM is a good example; programming voltage levels do not exist in logic simulation.

In some cases a model goes beyond the manufacturer's specifications in order to make simulation more useful. For example, the model of the Brooktree Bt458 RAMDAC (a color palette component) has a special test mode that does not exist in the component.

All model-specific exceptions between the behavior of an actual device and its model are documented in the model's datasheet. When appropriate, notes and warning messages generated by a model also inform users about exceptions.

The following sections provide details on modeling assumptions that are common to all models in the SmartModel Library. Refer to the individual model datasheets for information about model-specific assumptions.

Setup and Hold Timing Checks

When a range of values is specified by a semiconductor vendor, setup and hold timing errors are triggered by violations of the worst-case specifications.

Unprogrammed States in Memory Models

All memory locations in RAM models, if used without memory image files, are read as unknowns.

The uninitialized contents of ROM models vary according to the manufacturer's specifications and are noted in the model datasheets. ROM models do not need to be programmed as long as their data is not used during simulation.

Read Cycle Checks in SRAMs

Read cycle timing checks are not performed in SRAM models. This eliminates spurious read cycle timing violation messages during the simulation of most designs. The messages would be generated in designs where the SRAMs are continuously selected while new read addresses are supplied. Due to slight differences between high-to-low and low-to-high propagation delays, transitions from the previous address to the new one could pass through an undesired address for a very short period of time. The same message generation problem can occur with read cycle timing violations based on chip select pins.

Read cycle time is simply the minimum amount of time required to successfully access the SRAM; therefore, the model does not supply valid data until the read access time has been satisfied.

Simulating Processor Models in Partial Designs

To allow for simulations with designs that are not fully operational, processor models do not propagate unknowns on some inputs (the clock and various control pins, primarily). Instead, the models substitute a 1 or 0—whichever has the least effect on the simulation—and issues a warning message alerting you to the change.

Models with Boundary Scan Features

Models of boundary scan devices have a 2-byte state code assigned to each of 16 possible TAP controller states. When you access the TAP register using SmartModel Windows, the model returns a state code, indicating the current TAP controller state. The TAP register is a read-only element, so forcing the TAP register to a value has no effect. Figure 3 shows the TAP states and their associated state codes.

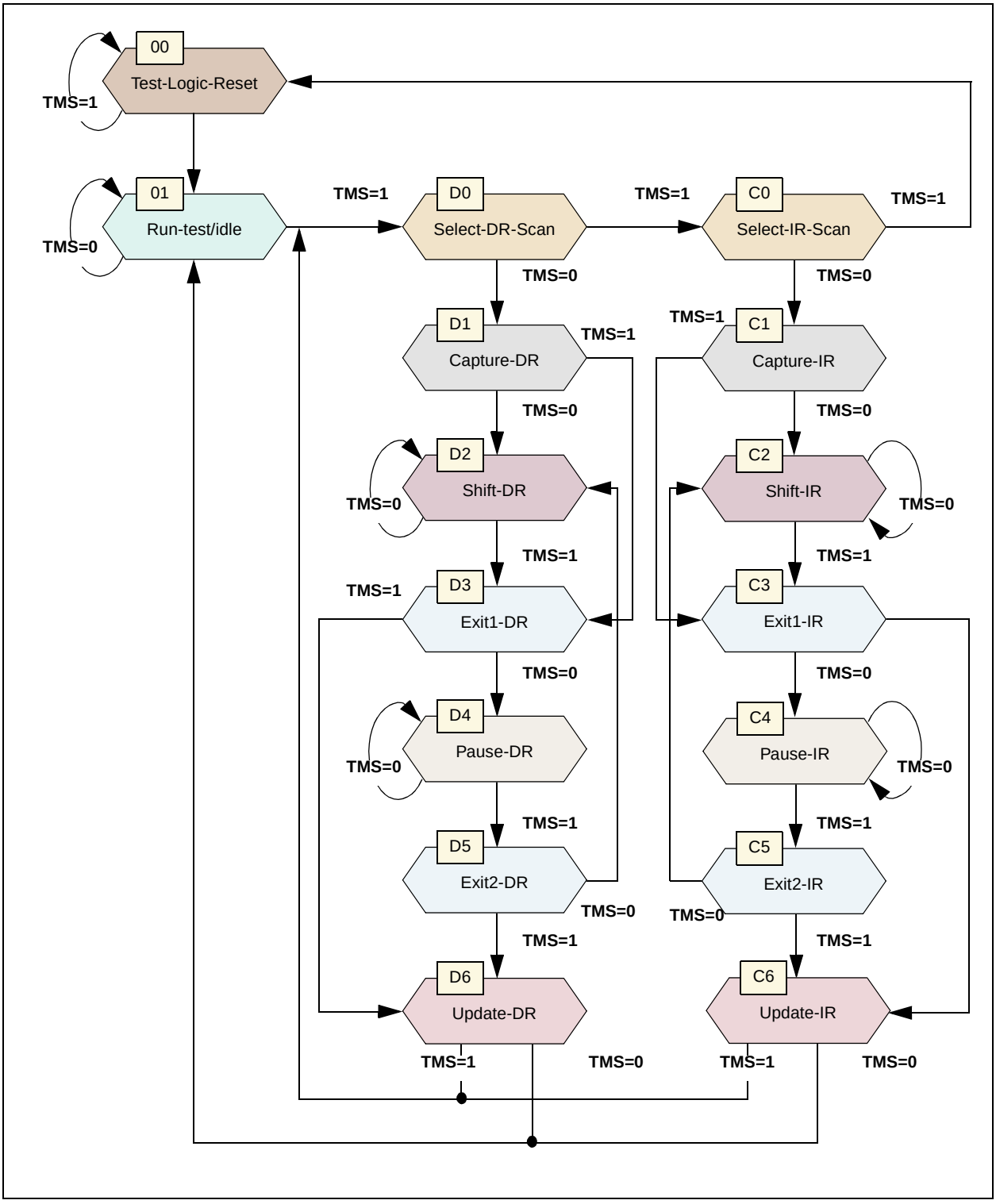


Figure 3: Diagram of TAP States

Approaches for Using Unknowns

Depending on where an unknown occurs in a circuit, it can propagate through your entire simulation. Later events can become buried in unknowns, making your simulation less useful than it could be. To gain more information, you would have to fix the first problem and then rerun the simulation.

SmartModel Library models are designed not to generate or propagate unknowns unnecessarily. A model uses error messages to inform you about its states and the assumptions made to substitute a good value for an unknown.

For example, Figure 4 shows a simple 9-bit register. When "enable" is asserted, on the next rising clock edge, the register puts the value of the D pins into the internal Q registers and also on the output pins. When "clear" is asserted, the internal registers are cleared to zero.

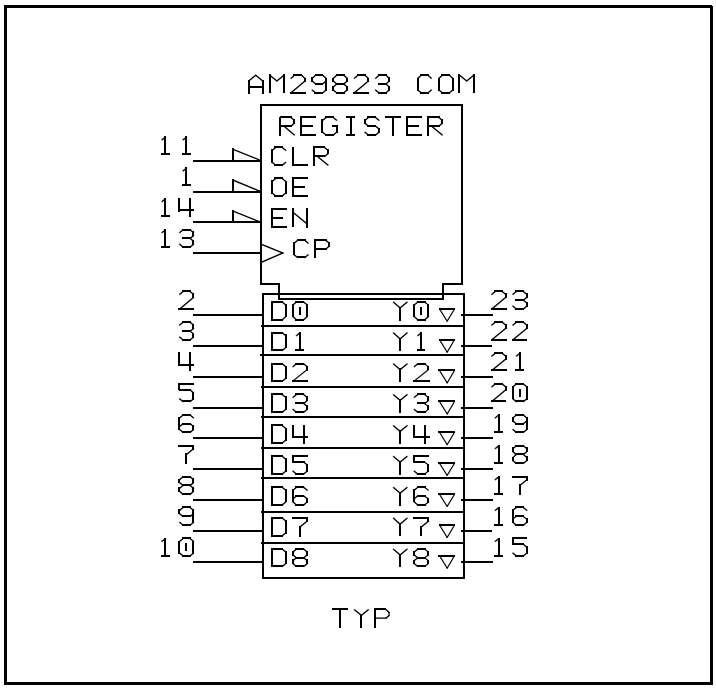


Figure 4: 9-Bit Register

Internally, this device is a series of positive edge-triggered, D-type flip-flops. Even in so simple a device, there are many opportunities for unknowns; the clock, clear, D, or any combination of these could be unknowns. In addition, the Q register could start as an unknown.

Let us look at how a "Smart Flip-Flop" handles unknowns. The operation of the Smart Flip-Flop is shown in Table 3. The following definitions apply to symbols used in the table:

- **CLR~.** Clear input, asserted when low.
- **Q.** Value of the Q output prior to evaluation.
- **"Smart" newQ.** Value of the Q output of a Smart Flip-Flop after the evaluation.
- **"Not Smart" newQ.** Value of the Q output of a flip-flop modeled without the techniques used to develop SmartModel Library models.
- **Caret (^).** A valid rising clock edge.
- **Hyphen (-).** The clock is known not to be rising; it is stable high or low, or could fall once from high to low.
- **X.** An unknown on either an input or an output, which refers to the logic values of X, W, U, or D described in "Features Common to Most Models" on page 29.

Table 3: Comparison of Generated Unknowns in the Example Flip-Flop

Case	Row	CLR~	Q	D	CLK	"smart newQ"	"not smart newQ
Case 1	1	1	X	1	^	1	1
Case 2	2	0	X	X	X	0	0
Case 3	3	X	X	0	^	0	X
Case 3	4	X	X	1	^	X	X
Case 4	5	X	0	X	-	0	X
Case 4	6	X	1	X	-	X	X

Case 1

Row 1 of the table shows that CLR~ is deasserted high, and there is a 1 on D. The original state of Q doesn't matter in this case, and there is a rising clock edge. The result is a 1 in Q.

Case 2

Row 2 shows that CLR~ is asserted low. The resulting Q is 0 even though there are unknowns on D and Q because the states of D, CLK, and Q do not matter with "clear" asserted.

Case 3

Rows 3 and 4 show what happens when both CLR~ and Q are unknown and there is a rising clock edge. In the "not smart" flip-flop the output is unknown; however, in the "Smart Flip-Flop" the output depends on the D input. For the "Smart Flip-Flop," if D is 0 as in Row 3, then newQ is known to be 0. This situation leads to either CLR~ being asserted, or the 0 at D being captured by the clock (CLK). Conversely, if D is 1 as in Row 4, then newQ is truly unknown. In this case, if CLR~ is asserted, then newQ is 0. However, if CLR~ is not asserted, then the 1 at D is captured by the clock, which makes newQ a 1.

Case 4

Rows 5 and 6 show what happens when both the CLR~ and D are unknown, and the clock has not changed while Q has been at a steady state. In the "not smart" flip-flop the output is unknown. However, in the "Smart Flip-Flop" the output depends on the previous state of the Q output.

For the "Smart Flip-Flop," if Q is 0 as in Row 5, then newQ remains at 0 either because CLR~ is asserted or no clock has occurred to change the output. Conversely, if Q is 1 as in Row 6, then newQ is truly unknown. In this case, if CLR~ is asserted then newQ is 0. However, if CLR~ is not asserted, then the output remains the same.

Modeling Changing or Uncertain States

SmartModel Library models simulate all the timing parameters and logic states specified by the device manufacturer, but in some situations the states of some pins are uncertain. In most memory parts, for example, the data input/output lines can be either high, low, high-impedance, or changing from one state to another. The models simulate each known state according to its specifications, and use unknowns to represent the uncertain or changing states. This concept is illustrated in Figure 5.

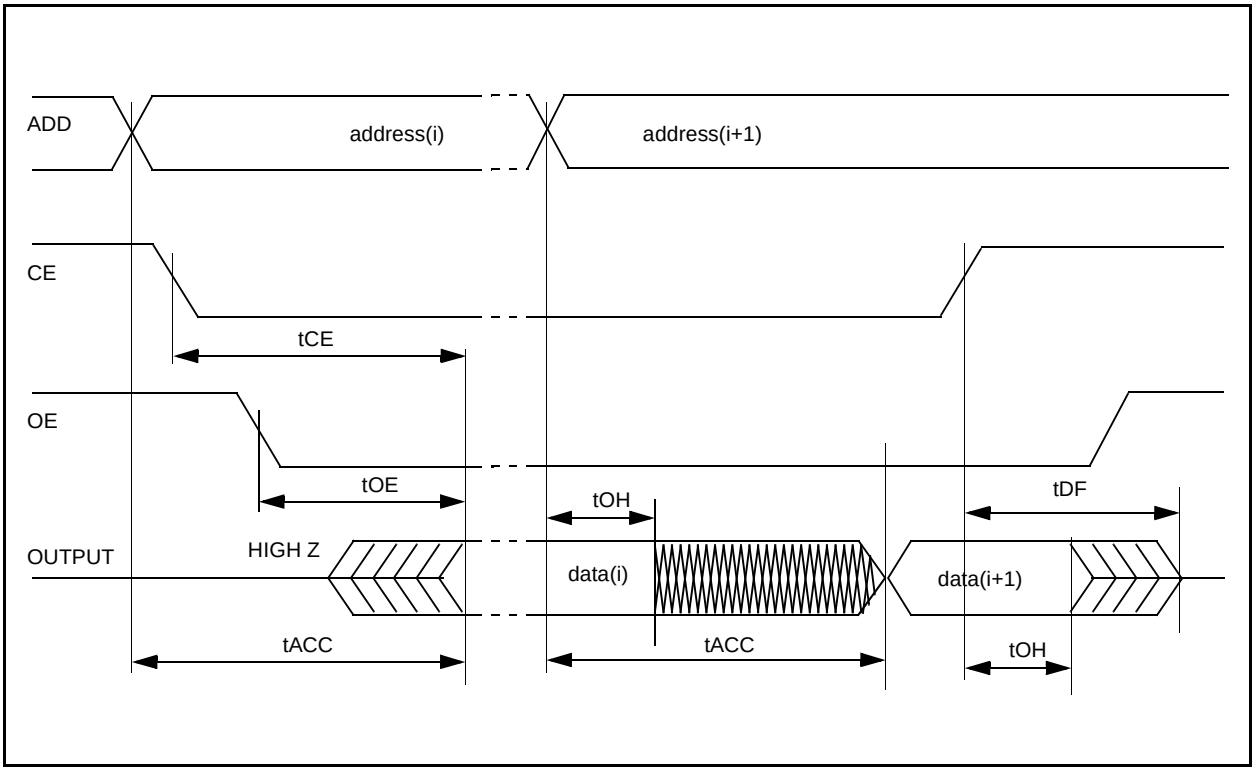


Figure 5: Changing States Timing Diagram

For example, a memory part manufacturer cannot be specific about the data line states between the end of the data hold time and the end of the data access time. During that time the I/O lines could be carrying the data from the last cycle or the current cycle. To prevent data from a previous cycle from being accepted as valid, the model generates unknowns during that time segment.

Though a memory part is used in the example, this modeling technique is useful in any situation where uncertainties exist—as in the transitions from and to high-impedance shown in Figure 5. If you push your designs to the limit, as you might when designing a memory cache, you may appreciate this logically pessimistic behavior.

The access delay feature of memory models can be turned off using the library's user-defined timing feature.

3
Browser Tool

Introduction

The Browser tool ($LMC_HOME/bin/sl_browser) provides a graphical user interface to the SmartModel Library and the online documentation. This chapter provides information about how to use the Browser to perform the day-to-day functions required to make optimal use of the SmartModel Library of behavioral simulation models. This information is organized in the following major sections:

- Selecting Models in $LMC_HOME
- Selecting Tool Versions
- Default Configuration (LMC) File
- Using the Browser Tool
- Configuration (LMC) Files
- Custom Configuration (LMC) Files
- Browser Tool GUI

Setting Environment Variables on NT Platforms

Many SmartModel Library installation and configuration steps require that you set environment variables. Most of the examples in this manual show how to set an environment variable in UNIX using a C shell. For NT, you set environment variables using the System Properties window. To access the System Properties window select **Start > Settings > Control Panel** and double-click the System icon. From the System Properties window, select the Environment tab, enter the variable name and value, and click Set. Then click on OK to dismiss the window.

By default, new variables that you enter become "User" environment variables. If you have administrator privileges you will also be allowed to create "System" environment variables. Note that any "User" variables that you create will override "System" variables set up by the system administrator at your site.

If the NT machine where you installed the SmartModel Library is to be used by multiple users it is probably best to set basic environment variables such as $LMC_HOME as "System" variables. To do this you need administrator privileges. First, highlight an existing entry in the System Variables portion of the System Properties window. Then enter the variable name and value and click Set.

Running Console Applications on NT Platforms

Many SmartModel Library user procedures assume that you have access to a UNIX shell such as the C shell. For example, instructions and examples for using SmartModel Library command-line tools assume that you have access to a C shell. If you are running on an NT platform, use the Console Application to run these tools. To access the Windows NT Console Application, select **Start** > **Programs** > **Command Prompt**.

References to environment variables on the NT command line must be delimited by the percent sign (%). This differs from the way environment variables are typed on the UNIX command line where the variable is simply introduced with the dollar sign ($). For example, $LMC_HOME/bin/mytool works on UNIX platforms, but must be typed as %LMC_HOME%\bin\mytool on NT.

Selecting Models in $LMC_HOME

You can install and maintain multiple versions of the same model can in the same $LMC_HOME. You select a specific version of a model to use in a design simulation using configuration (LMC) files.

The default model version is the most recently installed version. In cases where you do not want to use the latest installed version of a model, you can override the default model version by creating one or more custom configuration (LMC) files. The software locates the default and custom configuration files using the $LMC_HOME and $LMC_CONFIG environment variables. For more information about model versioning, refer to "SmartModel Library Versioning" on page 18.

When you invoke the Browser, the selection pane displays a list of timing versions for the models that will be used by the simulator based on your $LMC_HOME and $LMC_CONFIG variable settings. To find out what other versions of a model are installed in your library, use the Model Detail command.

Selecting Tool Versions

For some SmartModel Library tools, called model-versioned tools, the version of the tool that is used is determined by the model. You cannot change the versions of model-versioned tools because a particular version of a model may depend on a specific tool version to function properly. Examples of model-versioned tools include compile_timing, ccn_report, smartbrowser, and smartccn.

For other SmartModel Library tools, called user-versioned tools, you select the version of the tool to use via the default and custom configuration (LMC) files, in the same way that you select different model versions. You can select and use any version of a user-versionsed tool that you have installed. However, it is often best to use the latest version. Examples of user-versioned tools include ptm_make, mi_trans, and swiftcheck.

For more information about SmartModel Library tools refer to "Library Tools" on page 191.

Default Configuration (LMC) File

The default configuration (LMC) file that comes with the SmartModel Library contains a list of all installed SmartModel Library models, user-versioned tools, and their most recently-installed versions. The LMC file is platform-specific, and is typically named *platform*.lmc (for example, hp700.lmc). Normally, when a model version is installed in the library, the $LMC_HOME/data/*platform*.lmc file is updated with the most recently added model version. If, for example, Version 01002 has been more recently installed than Version 01004, then Version 01002 is the default version used even though Version 01004 has a higher version number.

Before using the Browser tool, you must specify the default configuration file for the Browser by setting your local $LMC_HOME environment variable to the install directory. The model versions specified in the default configuration file will be used by the simulator unless you define other model versions in one or more custom configuration (LMC) files.

Using the Browser Tool

Set your environment variables and search paths by following the setup instructions provided in the *SmartModel Library Installation Guide*. You are now ready to use the Browser.

Starting the Browser

At the command-line prompt, enter the following command to start the Browser:

```
% $LMC_HOME/bin/sl_browser
```

You can also customize some Browser features by creating an initialization file called an sl_browser.ini file. Features that you can customize include creating a custom user menu, displaying by model at startup, and disabling the display of user-defined timing (UDT) files. Following are procedures that you can follow to accomplish any of these tasks.

Creating a Custom User Menu

If you have your own custom programs that operate on a specific model, and you want to integrate these programs with the Browser, you can customize the Browser interface to create a User menu that invokes these programs.

In the current working directory or in $HOME (for access by you only); or in $LMC_HOME/data (for access by all who use the same $LMC_HOME), create a file named sl_browser.ini.

Enter the following lines in the file:

```
[USER TOOLS]
menu text 1=command1 to execute
menu text 2=command2 to execute
menu text 3=command3 to execute   ...
menu text n=commandn to execute
```

Note that you must enter the string [USER TOOLS] literally.

The following example sl_browser.ini file specifies the command /user/joeb/bin/foobar, and names the corresponding menu entry Foobar.

```
[USER TOOLS]
Foobar=/user/joeb/bin/foobar
```

In the Browser, before executing your program, you must select a model on which the program is to operate. For the model selected, the Browser automatically passes to your program three arguments:

```
arg1 arg2 arg3
```

where arg1 is the model name, arg2 is the version number, and arg3 is the timing-version (optional).

When you start the Browser, the User menu appears on the menu bar to the right of the Actions menu.

Displaying by Model Name at Startup

By default, the Browser window displays the library models by timing version name. If you want the Browser to display by model name instead, you can override this default using the sl_browser.ini file. Follow these steps:

1. If you already have an sl_browser.ini file, use it in the instructions that follow. Otherwise, create the file in the current working directory, in $HOME, or in $LMC_HOME/data.
2. Enter these lines in the file:

```
[OPTIONS]
show_models=true
```

Note that you must enter the string [OPTIONS] literally.

Disabling the Display of User-Defined Timing (UDT) Files

By default, the Browser window displays any compiled user-defined timing (UDT) files that are in $LMC_PATH. If you want to disable the display of UDT files, you can override this default using the sl_browser.ini file. Follow these steps:

1. If you already have an sl_browser.ini file, use it in the instructions that follow. Otherwise, create the file in the current working directory, in $HOME, or in $LMC_HOME/data.
2. Enter these lines in the file:

```
[OPTIONS]
show_udt=false
```

Note that you must enter the string [OPTIONS] literally.

The following example shows an sl_browser.ini file that creates the command /user/joeb/bin/repl and names the menu entry Replace. This example also configures the Browser to display by model name and disables the display of UDT files.

```
[USER TOOLS]
Replace=/user/joeb/bin/repl

[OPTIONS}
show_models=true
show_udt=false
```

Configuration (LMC) Files

A configuration file is also called an ".lmc" file or "LMC" file. LMC stands for "List of Model Configurations." There are two kinds of configuration files: default configuration files and custom configuration files. All configuration files must have the extension .lmc.

Configuration files contain a list of model names and user-versioned tool names, with a version number specified for each model and tool. Following is an example of a configuration file:

```
%PLT hp700

# Models added by Sl_Admin: Fri Feb 23 15:56:24 1996

%EXE swiftcheck 01009
%EXE mi_trans 04059
%EXE ptm_make 01006
%MOD atv2500 01000
%MOD bt458 01000
%MOD c5c_c8c_2 01000
%MOD dm74s188 01000
%MOD ecl01 01000
%MOD gal18v10 01000
%MOD hm658128 01000
%MOD ifc161 01000
...
%MOD ttl0 01000 7400 74LS00
 ...
%MOD windows 01000
%MOD z8536 01000
```

Configuration File Syntax

There are three commands that can appear in an LMC file, as follows:

%PLT *platform_name*

If this optional command is present, it indicates that a check is to be made to determine if the platform specified is the same as that on which the software is currently running. Examples of allowed values are decalpha, hp700, sunos, solaris, ibmrs, and pcnt. The first line in the above example,

```
%PLT hp700
```

indicates the hp700 platform. If PLT is absent, no checking is done.

Hint

If you want a single LMC file to be shared among several platforms, omit the PLT command.

%EXE *tool_name version*

This command specifies the versions of the most recently installed user-versioned SmartModel Library tools. Examples of user-versioned tools include the following:

- **swiftcheck**—performs integrity checks on the installed library
- **mi_trans**—translates memory image files for memory models
- **ptm_make**—creates simulator-specific portmap files, which list the port names of a model and map those port names to the physical pins of the device.

In the example, the lines:

```
%EXE swiftcheck 01009
%EXE mi_trans 04059
%EXE ptm_make 01006
```

indicate that, when the tools are called, the versions that will be used are 01009 for swiftcheck, 04059 for mi_trans, and 01006 for ptm_make.

%MOD *model_name model_version* **[***alias***[***alias***]**

This command specifies the model name, model version, and any aliases that might apply to the model. In the example, the line

```
%MOD ttl00 01000 7400 74LS00
```

indicates model ttl00, version 01000, with aliases of 7400 and 74LS00.

Custom Configuration (LMC) Files

A custom configuration file contains a list of installed SmartModel Library models and user-versioned tools to be used by the simulator. If you want to use tool or model versions that are different from those in the default configuration file, create one or more custom configuration files that specify the names and versions of those models and tools.

LMC files are platform-specific and must have the extension .lmc. You locate customer configuration files for the Browser and your simulator by setting the $LMC_CONFIG environment variable to the paths to the .lmc files. There are several reasons why you might want to create a custom configuration file:

- Freeze your design, so that it always references the same model versions.
- Access a specialized model version that no one else in your group should use.
- Check new models before releasing them to others.
- Archive your design, along with the model versions used.
- Access an updated model version to use a new or revised function, when other design teams do not want to disturb their designs by using the updated model.
- Revert to a previous version of one of the tools.

For NT, separate multiple entries for the $LMC_CONFIG environment variable using a semicolon-separated list, not a colon-separated list as in UNIX.

The model versions specified in custom configuration files are used by the simulator, overriding the versions of those same models that are specified in the default configuration file. However, the model versions you specify must be installed in the library at your location. To determine what versions of a specific model are installed in your library, use the model detail function.

Creating a Custom Configuration (LMC) File

To create a custom configuration file, follow these steps:

1. In your home directory, open a new file named (for example) *my_platform*.lmc. Although the file will be platform-specific, you do not have to use "*platform*.lmc".
2. In another window, open a read only copy of the default configuration file, $LMC_HOME/data/*platform*.lmc.
3. In the default configuration file, search for the models and tools whose version numbers you want to change.
4. As you find each item, copy its version record (the entire line on which the item appears) into the new file.
5. In the new file, change the version numbers to the ones you want.
6. Save the file.

As an alternative, instead of copying the version record, you can create new version records using the syntax rules described in "Custom Configuration (LMC) Files" on page 49. For example, to revert to the previous version of the swiftcheck tool and the model adsp1008a, you would follow these steps:

1. In the default configuration file, locate those two items, and copy the respective version records into your new file. The lines you copied might look like this:

   ```
   %EXE swiftcheck 04091
   %MOD adsp1008a  01003
   ```

   The version numbers represent the current versions; the previous versions are therefore 04090 and 01002.

2. Change each version number to the previous version and save the file.

   ```
   %EXE swiftcheck 04090
   %MOD adsp1008a  01002
   ```

Creating a Custom Model Filter

When you start up the Browser, it displays a large list of models and timing versions. Normally, each user's design does not use all of the models. To limit the display to only those models that are of interest to you, use the Model Filters dialog box. You can filter the display using any combination of names, vendors, functions, and license packages. For specific instructions, refer to "Model Filters Dialog Box" on page 66.

Creating a Custom Timing Version

If you need a timing version that is not already available with the SmartModel Library, you can create your own custom timing version by copying and modifying an existing timing file. Follow these steps to create a custom timing version:

1. Create a directory for the customized timing file.
2. In the Browser selection pane, select the model whose timing file you want to customize.
3. From the Actions menu, choose Copy Customizable Files, or click on the Copy Customizable Files button in the toolbar. The Copy Customizable Files dialog box opens.
4. If Timing Source File is not already selected, select it by clicking on its check box.
5. In the To Destination Directory text field, type the path name of the directory you just created.
6. Click on the Copy button. The dialog box closes. The file has been copied to the specified directory.

To edit the timing file, follow these steps:

1. Open the timing file.
2. Edit the timing file. If you are not familiar with the format and grammar of timing data, refer to "User-Defined Timing" on page 157.
3. Save the timing file as *model*.td in the directory you created.

To prepare the timing file for simulation, compile the timing file. For details, refer to "Running the Timing Compiler" on page 179.

Note

You must recompile the timing file each time you set your custom configuration file to a different model version or install a new model version in the library. Each compiled timing file (.tf file) is compatible only with the model version in the configuration files at the time the timing file is compiled.

To locate the timing file for the Browser and the simulator, set your $LMC_PATH environment variable to the directory that contains the compiled timing file, as shown in the example below:

```
% setenv LMC_PATH /user/me/
```

Your custom timing version is now ready for use in simulation. By default, the Browser displays this timing version and any other custom timing versions you may have in $LMC_PATH.

Determining the Most Recent Model Version

To determine whether you have the most recent model version available, follow these steps:

1. Select the model in the Browser's main window and click on the Model Detail toolbar button. This brings up a Model Detail window which lists all versions of that model installed in your $LMC_HOME.
2. Locate the latest datasheet for the model, using one of the following methods.
   a. Go to the Model Directory on the Web:

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

      Enter the timing version name and start the search. From the resulting list, select the required timing version. When the product information appears, click on the product code to show the datasheet.

   b. Contact your local Synopsys representative.

3. Look in the banner section at the top of the datasheet for the version number of the model. Compare the version number with that of the model in your library. If they match, you have the most recent version. If the datasheet shows a more recent version, the difference may or may not be significant.
4. Read the more recent datasheet's history section to find out how the model changed. (Model history sections appear at the end of each datasheet.) If the change was functional or due to a bug fix, then there could be a significant difference between the most recent version and the one you have. If the change was purely administrative and did not affect model functionality, you may not need the most recent version.

Displaying Model Datasheets

To display the datasheet for any model installed at your site, follow these steps:

1. From the Browser window selection pane, select a model.
2. Choose the Display Datasheet command from the Actions menu, or click on the Display Datasheet toolbar button to make the datasheet appear.

Displaying All Timing Versions of One Model

To display all timing versions of one model, follow these steps:

1. If the selection pane is displaying the models by timing versions, choose Display by Model Name from the View menu.
2. Locate the model whose timing versions you want to display.
3. Click on the folder icon next to the model name. In the selection pane, the timing versions, including any custom timing versions that are in $LMC_PATH, appear in hierarchical fashion, subordinate to the model.

Locating a Model in the Model List

To locate a model in the model list, follow these steps:

1. If the selection pane is displaying the models by timing versions, choose Display by Model Name from the View menu.
2. To locate the desired model, choose the Filter command from the Actions menu, or click on the Filter toolbar button. The Model Filters dialog box appears.
3. If the String Search option is not already selected, select it by clicking on its check box. In the String Search text field, type in the model name.

4. Deselect any other options that are selected by clicking on the check boxes. Only the String Search check box should be selected.
5. Click on Filter.
6. Click on Close. The Model Filters dialog box closes and the selection pane contains the model.

Note

You can also scroll to the desired model if it is currently in the selection pane.

If you do not find the model you are looking for, try expanding the search criteria by putting wild cards in the search string. Model names generally do not exactly match the vendor device names.

If you know the model is not in the library installed at your site, check with your system administrator to find out whether the model is available on the CD but was not installed. If the model is not available at your site, check the Model Directory on the Web at:

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

or call your local Synopsys representative to find out if the model has recently become available.

Displaying a Specific Vendor's Models

To display models of a specific vendor's devices, follow these steps:

1. Choose the Filter command from the Actions menu, or click on the Filter toolbar button. The Model Filters dialog box appears.
2. Select the Vendors option by clicking on its check box.
3. Deselect the String Search field and any other options that are selected by clicking on the appropriate check boxes. Only the Vendors check box should be selected.
4. In the Vendors list box, scroll to the desired vendor and select it.
5. Click on Filter.
6. Click on Close. The Model Filters dialog box closes and the selection pane lists all models of the specific vendor.

Displaying All Models That Have the Same Function

To display all models that have the same function, follow these steps:

1. If the selection pane is displaying the models by timing versions, choose Display by Model Name from the View menu.
2. Choose the Filter command from the Actions menu, or click on the Filter toolbar button. The Model Filters dialog box appears.
3. Select the Function/Subfunction option by clicking on its check box.
4. Deselect the String Search field and any other options that are selected by clicking on the appropriate check boxes. Only the Function/Subfunction check box should be selected.
5. In the Function/Subfunction list box, scroll to the desired function and select it.
6. Click on Filter.
7. Click on Close. The Model Filters dialog box closes and the selection pane lists all models that have the specified function.

Finding Out More Details About a Model

To find out more details about a model, follow these steps:

1. Locate the model you are interested in.
2. Double-click on the model (or use the Model Detail command on the Actions menu). The Model Detail dialog box appears. It contains information about the version, configuration file, timing versions, and other details.
3. Click on Close. The dialog box closes.

Finding Out What Model Version You Have

To find out what model version you have, follow these steps:

1. In the selection pane, select the model or timing-version you are interested in.
2. From the Actions menu or the toolbar, choose Model Detail. The Model Detail window opens. The model version is displayed in the Version box.

Loading a Custom Configuration File

To load a previously created custom configuration (LMC) file, follow these steps:

1. If you are currently running the Browser, exit it.
2. Set your $LMC_CONFIG environment variable to the path of your LMC file.
   - If you have no other configuration file already defined in $LMC_CONFIG, or if you do have an older file already defined but want the file you are now loading to replace the old file for this work session, enter the following on the command line:

     % **setenv LMC_CONFIG /user/johnq/***newfilename.lmc*
   - If you have a configuration file already defined in $LMC_CONFIG, and for this work session want to use this existing file in addition to the configuration file you are now loading, set $LMC_CONFIG to both file names, separated by a colon, in the order in which you want the files to be searched for models, as shown in the following example:

     % **setenv LMC_CONFIG/user/johnq/***newfilename.lmc* :
     /user/johnq/*oldfilename.lmc*

     For NT, you must separate multiple entries for the $LMC_CONFIG environment variable using a semicolon-separated list, not a colon-separated list as in UNIX.
3. Invoke the Browser. The selection pane displays a list of timing versions, including any model versions you specified in your new configuration file. The status pane still contains the path referenced by the $LMC_HOME environment variable.

Use Environment Settings (LMCs)

Selecting this menu entry returns you to the list of models or timing versions originally displayed in the Browser's selection pane. This list shows the models that the simulator will use at simulation time. To derive the list, the software uses the environment settings $LMC_CONFIG (which contains path names to any custom configuration files) and $LMC_HOME (which points to the default configuration file). Therefore, the list can be derived from more than one configuration (LMC) file, as indicated by the designation "(LMCs)" following the Use Environment Settings menu entry. If there are no custom configuration (LMC) files, then the list is derived entirely from the default configuration (LMC) file.

For each model called by the design, the software searches first for model versions specified in the custom configuration files listed in $LMC_CONFIG, in the order in which they are listed. If a model is not found, the software next searches for it in the default configuration (LMC) file, $LMC_HOME/data/*platform*.lmc.

For example, if:

- LMC_HOME=/d/lsl/latest

  and

- LMC_CONFIG=/user/me/my_platform.lmc:/user/joe/joes_platform.lmc

for each model, the model version is obtained as follows:

1. The software first searches for the model in /user/me/my_platform.lmc. If the model is found, that model version is used and the software stops searching for the model even though it might also be present in subsequent files.
2. If the software does not find the model in the first file, it searches /user/joe/joes_platform.lmc. As before, if the model is found, that model version is used and the software stops searching for that model.

Note

If the first model encountered is a version that is not in the library, the selection pane does not display that model at all, even though there might be other versions of it in the library. This behavior mirrors that of the simulator at simulation time. Models that are missing from the displayed list will not be found by the simulator. To detect and prevent potential simulation errors, check to see if any models called by your design are missing from the displayed list. If so, generate a Model Report and then repair any errors reported.

3. If the software does not find the model in the last custom configuration file, it searches the default platform.lmc file found in the directory /d/lsl/latest/data. If the model is found, that model version is used and the search ends.
4. If the model is not found in any file, the SWIFT interface indicates to the simulator that the model is invalid. Depending on the simulator, it may issue an error message. If you receive an error message about bad integration and a possible missing model, or if you are unable to create an instance of a model, first set your LMC_COMMAND environment variable to "verbose on" and rerun the simulation. If the messages produced are not sufficient to help you diagnose the problem, then run the swiftcheck utility program.

   Before running swiftcheck, make sure that one of the configuration files contains an EXE command that specifies a version of swiftcheck. Also, check your custom configuration files for a possible typing error in entering the model name, or for a possible reference to a model version not installed in the library at your site.

For more information about using LMC_COMMAND, refer to the *Simulator Configuration Guide for Synopsys Models*. For more information about swiftcheck, refer to "Checking SmartModel Installation Integrity" on page 207.

Repairing Errors Reported by a Model Report

To generate a model report, follow these steps:

1. Select one of the model report options by clicking on its radio button.
2. Click OK. The dialog box closes and the Model Report results window appears. This window contains the model report.

For more information about model reports, refer to "Model Report Dialog Box" on page 68.

Models found in configuration files but not in library

When you select this report, the Browser lists models found in the default configuration file (*platform*.lmc) or in your custom configuration files but not in the library installed at your site. The Browser reports these models as errors. To repair errors, follow these steps:

1. For each model, note whether or not it is used in your design.
2. For each model used in your design, use the Admin tool to install the model or model version in the library at your location, or ask your system administrator for help installing the models.

Models found in library but not in configuration files

When you select this report, the Browser lists models that were found in the library installed at your site but not in the default configuration file (*platform*.lmc) or in your custom configuration files. The Browser reports these conditions as errors. To repair errors, follow these steps:

1. For each model, note whether or not it is used in your design.
2. For each model used in your design, check all configuration files for an incorrectly entered model name.
3. Edit your custom configuration files to correct incorrect model names and to add missing models.

Browser Tool GUI

The Browser tool has standard graphical user interface features. Following are brief descriptions of the windows, menus, icons, and dialog boxes you can use to work with the models.

Browser Window

The Browser window is the command center from which you control the Browser as you work with models.

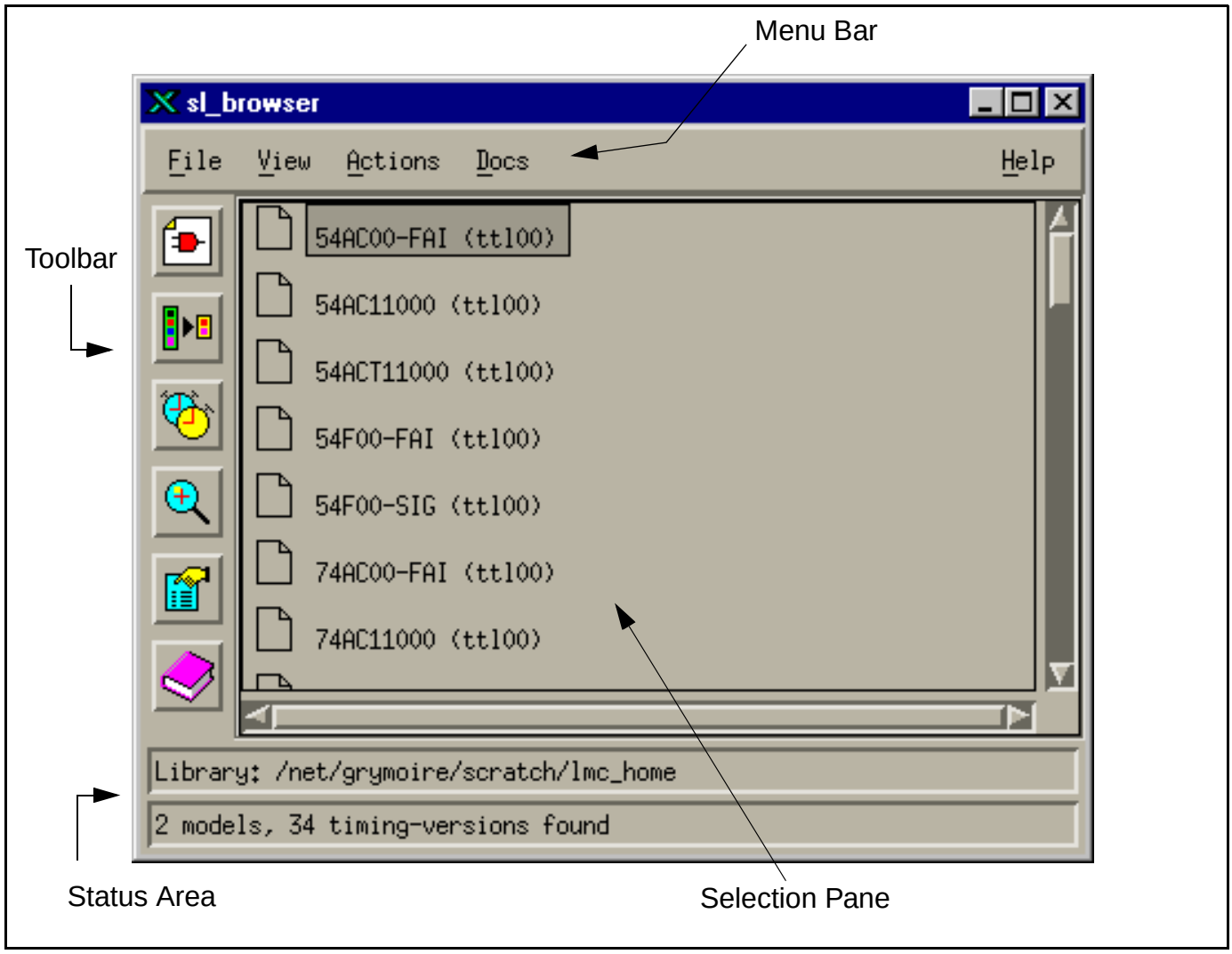


Figure 6: UNIX Browser Tool Window

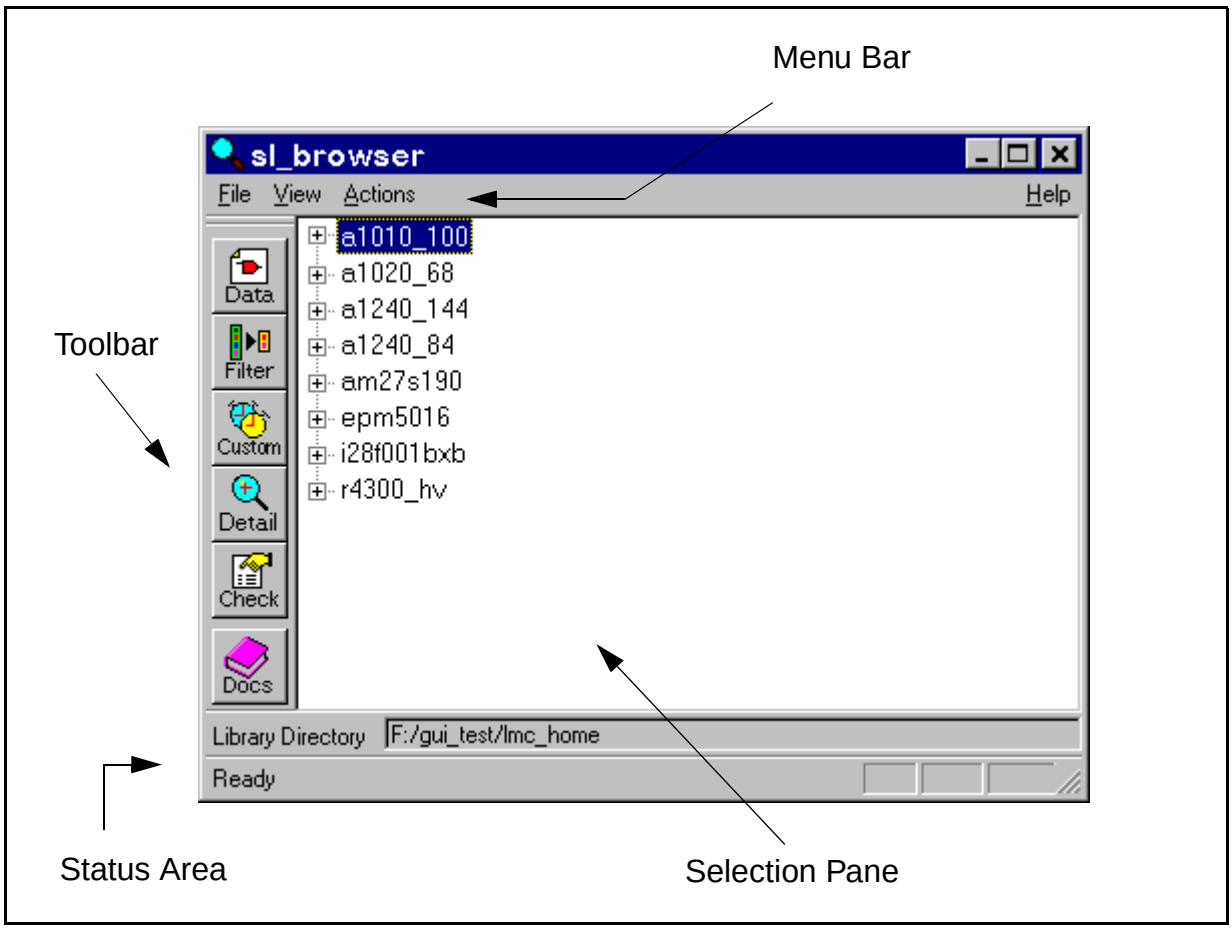


Figure 7: NT Browser Tool Window

The are four Browser window elements

:

- **Menu Bar**—At the top of the Browser window, the menu bar contains the File, View, Actions, Docs, and Help menus. You use these menus to initiate all Browser functions.
- **Toolbar—**Arranged vertically at the left side of the Browser window, the toolbar buttons provide easy access to functions also available through the Actions and Docs menus.
- **Selection Pane**—The selection pane takes up most of the window, and contains lists of models and timing versions.
- **Status Area**—The status area shows the path name to the SmartModel Library being operated on, and gives various status messages as appropriate.

You can customize the Browser window by adding an optional User menu or changing the default display. For more information, refer to "Creating a Custom User Menu" on page 46.

Menu Bar

The menu bar (see Figure 8) features several pull-down menus that you can use to perform the tasks described below.

Figure 8: Browser Tool Menu Bar

File Menu

The File menu has the following options:

Use Environment Settings (LMCs)

Displays an alphabetical list of models or timing versions. The list is based on configuration (LMC) files specified by the $LMC_CONFIG and $LMC_HOME environment variables.

Open Specific Configuration (LMC)

Opens the Open Configuration File dialog box to select a single LMC file from which to obtain a list of timing versions to display.

Print

Opens the Print dialog box (available only on NT platforms).

Print Preview

Opens the Print Preview (available only on NT platforms.)

Print Setup

Opens the Print Setup dialog box, where you can select a default or user-defined printer. (This menu item is only available on NT platforms.)

Exit

Ends the active Browser tool session.

View Menu

The View menu has the following options:

Display by Timing-Version Name

Displays an alphabetical list of timing version names from models in the LMC files.

Display by Model Name

Displays an alphabetical list of model names from the LMC files.

Show Model Names

Displays the model name associated with each timing version name. .

Expand Model

Expands the models by displaying hierarchical trees of the available timing versions for each selected model. You can do the same thing by clicking on the model icons.

Expand All

Expands all models by displaying hierarchical trees of the available timing versions for all models.

Collapse Model

Hides the timing versions for the selected models. You can do the same thing by clicking on the icons of the expanded models.

Collapse All

Hides the timing versions for all models.

Actions Menu

The Actions menu has the following options:

Display Datasheet

Displays the datasheet for the selected model or a timing version. (Same as the Display Datasheet toolbar button.)

Filter...

Opens the Model Filter dialog box, where you can select filtering criteria for displaying a subset of the model list. (Same as the Filter toolbar button.)

Copy Customizable Files (timing,...)

Opens the Copy Customizable Files dialog box, where you can copy timing files or other customizable files for the selected model or timing version to a specified directory for customization. (Same as the Copy Customizable Files toolbar button.)

Model Detail...

Opens the Model Detail dialog box, which provides version, platform, and other information about the selected model or timing version. (Same as the Model Detail toolbar button.)

Report...

Opens the Report dialog box, which displays environment information and allows you to perform consistency checks on the installed library. (Same as the Report toolbar button.)

User Menu

A custom User menu appears only if you create one using the optional sl_browser.ini file. Clicking on the dashed line lets you "tear off" the User menu and drag it to a convenient spot on your desktop for easy reference as you work. The following example menu entries invoke the external commands as specified in the sl_browser.ini file. You must select a model before choosing one of these entries.

Optional User Command 1

Optional User Command 2

...

Optional User Command n

Docs Menu

The Docs menu provides links to the SmartModel Library online documentation.

Help Menu

The Help menu displays the tool version number, copyright, and other information.

Toolbar

You can invoke the major Browser tool functions from the toolbar. The toolbar buttons provide another way to access functions available from the Actions and Docs menus. To display the function of each toolbar button, place the pointer on the button. Table 4 describes the different toolbar buttons and what you can do with them.

Table 4: Toolbar Button Descriptions

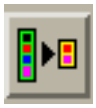
Button	Use To ...
	**Display Datasheet**— Displays the datasheet for the selected model or timing version. (Same as the Display datasheet command on the Actions menu.)
	**Filter—** Opens the Model Filters dialog box, which you can use to select the filtering criteria for displaying a subset of the model list. (Same as the Filter command on the Actions menu.)
	**Copy Customizable Files—** Opens the Copy Customizable Files dialog box, which you can use to copy timing files or other customizable files for the selected model or timing-version to a specified directory for customization. (Same as the Copy Customizable Files... command on the Actions menu.)
	**Model Detail—** Opens the Model Detail dialog box, which displays version, platform, and other information about the selected model or timing-version. (Same as the Model Detail command on the Actions menu.)
	**Report**— Opens the Report dialog box, which displays environment information and allows you to perform consistency checks on the installed library. (Same as the Report command on the Actions menu.)

Table 4: Toolbar Button Descriptions (Continued)

Button	Use To ...
	**Docs Button**— Opens the *SmartModel Library User's Manual* (this manual) in PDF format using the Acrobat Reader.

Selection Pane

The Selection Pane displays an ordered list of models or their timing versions. When you invoke the Browser, by default the selection pane displays a list of timing versions corresponding to the Use Environment Settings (LMCs) menu entry, and includes any user-defined timing (UDT) files that are contained in the $LMC_PATH variable. Each timing version name lists the corresponding model name in parentheses next to the timing version name.

In the View menu, the Expand Model, Collapse Model, Expand All, and Collapse All commands are grayed out.

To display a smaller subset of the list:

1. Choose the Filter command from the Actions menu, or click on the Filter toolbar button (second from top). The Model Filters dialog box is displayed.
2. Filter the list as described in "Model Filters Dialog Box" on page 66.

To display the list by model name, choose the Display by Model Name command from the View menu. The selection pane displays an ordered list of models by model name. In the View menu, the Expand Model, Collapse Model, Expand All, and Collapse All commands are no longer grayed out.

To toggle the hierarchical display of timing versions for a single model:

1. With the selection pane displaying an ordered list of models by model name, click on the folder icon of the desired model. The selection pane displays the model and its timing files as a hierarchical tree.
2. Click again on the folder icon of the same model. The timing files disappear.

Note

You can also toggle the hierarchical display using the Expand Model, Collapse Model, Expand All, and Collapse All commands.

To toggle the hierarchical display of timing versions for all models:

1. Select the Expand All command from the View menu. The selection pane displays all models and their timing files as hierarchical trees.
2. Choose the Collapse All command from the View menu. The timing files disappear.

To display a list of timing versions from a single configuration (LMC) file:

1. From the File menu, choose Open Specific Configuration (LMC). The Open Configuration File dialog box opens.
2. Select the desired configuration file and click OK. The Open Configuration File dialog box closes and the selection pane displays the corresponding set of timing versions.

Status Area

The status area displays the path to the model library that appears in the Selection Pane, the number of models and timing versions found, and various status messages.

The status area also displays tool tips for the toolbar buttons as you move the pointer over them.

Model Filters Dialog Box

The Filter function opens the Model Filters dialog box. Use it to specify filter options for displaying a subset of the model library. You can select one or more of the four filter options by checking the appropriate check box. The dialog box fields are as follows:

- **String Search field**—Contains a string that specifies a model name or timing version name to search for. The search field is initially set to display all models or timing versions. By default, this field is selected (checked) when the dialog box opens.
- **Vendors list box**—Contains a list of vendors to select as filter options. Use this field if you want to narrow the display to models of devices from specific vendors.
- **Function/Subfunction list box**—Contains a list of functions and subfunctions to select as filter options. Use this field if you want to confine the display to models that have specific functions or subfunctions.
- **Licensed Packages list box**—Contains a list of licensed packages to select as filter options. Use this field if you want to confine the display to models contained in one or more specific licensed packages, or if you want to know whether a specific model is contained in a particular licensed package.
- **Summary Filter Options field**—Displays the filter options currently selected.

Execute the filter function using the Filter button at the bottom of the dialog box.

To specify a model or timing version to search for, type the complete name of the model or timing version, or a partial name with the wild card character (*) in the String Search text field in each position where you have omitted characters.

To select one or more of the three remaining filter options, follow these steps:

1. Click on the check box for each desired option. Some combination of (No Vendor), (No Function), and (No Marketing Group) appears in the Summary of Filter Options list box. These appear ANDed with each other and with the String Search value.
2. For each option you checked, select one or more items from its list box. Use the scroll bar to traverse the list. To select more than one item, hold down the Ctrl key as you click on each item. As you select each item, its name appears in the Summary of Filter Options list box. If you select multiple items, they appear ORed together.

To filter the models, follow these steps:

1. When you have finished selecting filter options, click on the Filter button.
2. Move the Filter dialog box or click on the Close button to dismiss it. The list box in the Browser window displays a list of models and timing versions that meet the filtering criteria. The status pane displays the number of models and timing versions found.

Copy Customizable Files Dialog Box

The Copy Customizable Files function opens the Copy Customizable Files dialog box. Use it to copy a model's timing file (or other customizable file) to your directory for customization. For information about customizing timing files, refer to "User-Defined Timing" on page 157.

To copy a customizable file, follow these steps:

1. From the selection pane, select the model or timing version whose timing file or other customizable file you want to copy.
2. From the Actions menu, choose Copy Customizable Files. The Copy Customizable Files dialog box opens. All customizable files for that model appear in the dialog box.
3. If the file you want to customize is not already selected, click on its check box.

4. In the To Destination Directory text field, type the full path name of the directory where you want the file to be copied.
5. Click on the Copy button.

Model Detail Dialog Box

The Model Detail function opens the Model Detail dialog box. Use it to get information about a specific model, such as its configuration file, installed versions, timing versions, and installed platforms.

Initially, the list of installed versions in the Installed Versions field shows the top version selected. The Timing Versions and Installed Platforms list boxes show information specific to the installed version selected.

To display timing versions and platforms for another installed version, use the Installed Versions list box to select another version.

Model Report Dialog Box

The Report function opens the Model Report dialog box. Use it to view information about environment variables and select reports by clicking on their radio buttons.

The dialog box fields are as follows:

- **Environment Variable field**—At the top of the dialog box, this field contains paths referenced by the environment variables used by the Browser. Use this information to verify that the models are referenced as you intended.
- **Report Selection field**—Following the environment variable field, this field contains three options you can choose for generating reports:
  - Report Configuration (LMC) file errors

    Choose this option if you want to know about any models used by your design that cannot be found in the library installed at your site. This option is selected initially.
  - Report models found in Library but not in Configuration (LMC) files

    Choose this option if you want to know whether any models in the library installed at your site are not listed either in the default configuration file or in your custom configuration files.
  - List all models with their source Configuration (LMC) files

Choose this option if you want to see a mapping of each model to the configuration file in which the Browser found it. Using this report, along with the contents of the $LMC_CONFIG environment variable, you can verify that the file listed first in $LMC_CONFIG contains the version of the model you want to use in your design.

Save As... Dialog Box

You can use the Save As... button to save a model report for future viewing or printing.

If the existing file name you want to use is not in the currently selected directory, you can search for it in directories above and below the current directory.

To traverse the directory structure, follow these steps:

1. Select a directory from the Directories list box.
2. Click on the Filter button. The selected directory now appears as the top-level directory in the Directories list box. The Files list box contains the names of files in the selected directory that have the extension .rpt. Notice that the file names are not displayed until you click on the Filter button. Alternatively, if you know the file name directory you want to use, you can type it directly into the Filter or Selection fields.

   On NT platforms, use the standard Windows navigational tools to get to the directory where you want to save the model report.

To save a report under an existing file name, follow these steps:

1. Select the desired file from the Files list box. The file name is now appended to the directory path name in the Selection text field.
2. Click on OK.

To save a report under a new file name, follow these steps:

1. In the Filter text field, change the old path name to the new path name.
2. Click on OK.

Open Configuration File Dialog Box

The Open Specific Configuration (LMC) function opens the Open Configuration File dialog box. On NT, this dialog box is called Open. Use it to display the contents of a configuration (LMC) file (custom or default). If the file you want to open is not in the currently selected directory, you can search for it in directories above and below the current directory.

To traverse the directory structure, follow these steps:

1. Select a directory from the Directories list box.
2. Click on the Filter button. The selected directory now appears as the top-level directory in the Directories list box. The Files list box contains the names of files in the selected directory that have the extension .lmc. Notice that the file names are not displayed until you click on the Filter button.

   On NT, use the standard Windows navigational tools to find and open the configuration file that you want.

To open a file, follow these steps:

1. Select the desired file from the Files list box. The file name is now appended to the directory path name in the Selection text field.
2. Click on OK. The Browser window selection pane now displays the model names from the selected file. Note that the Browser does not display a model if the configuration file specifies a version that is not in the installed library.

By default, the Browser searches for files with the extension .lmc. However, you can search for files with a different extension, and you can search for a specific file. To search for files with a different name or extension, follow these steps:

1. In the Filter text field, edit the file name so that it contains the desired name or *.*ext*.
2. Click on the Filter button. The Files list box now contains names of files in the selected directory that have the current name or extension.

To search for files in a different directory, follow these steps:

1. Delete the text in the Filter text field.
2. Type in the desired path name, including the file name or extension.
3. Click on the Filter button. The Files list box now contains names of files in the specified directory that meet the filtering criteria.

   On NT, use the standard Windows navigational tools to navigate and find the configuration file that you want.

4
Memory Models

Configuring Memory Models

Memory models simulate internal memory locations. Most memory models are for memory devices, but there are some processor, interface, and oscillator models that also have on-chip memory. You configure memory models at simulation startup using a Synopsys memory image file (MIF). For on-chip memory in processor models you configure the memory from a PCL program using the model-specific PCL commands documented in the model's datasheet. Memory models check their initialization files when they are loaded for simulation, as shown in Figure 9.

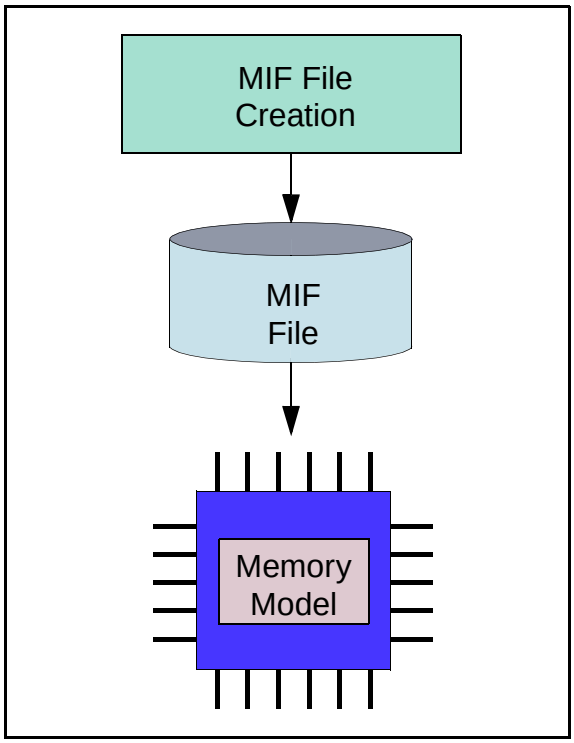


Figure 9: Process Flow for Memory Models

Using Memory Models

Memory models are flexible enough to support a wide range of potential uses in simulation. To make optimal use of this flexibility, keep the following points in mind:

- **Load minimum memory.** Because memory allocation is dynamic and due to the fact that you do not have to load all of the memory locations in a device, it is good practice to load only the minimum necessary to support the simulation. At simulation startup the model allocates only as much system memory as required by the data. The memory is allocated in groups of bytes based on the model's memory size in order to minimize the number of allocations. During simulation, if an address is used that is in an unallocated area, the model allocates new memory locations to support the data written to the model.
- **Key memory image file (MIF) names to your schematic.** It is best to name MIF files with the model instance name as part of the file name. Because model instance names are always unique, this is a handy way to pair memory devices with their correct MIF files.
- **Reuse MIF files.** You can load the same MIF file into as many models as you want by using the same file name with all instances on the schematic diagram.
- **Turn off access delays.** Memory models prevent invalid data from being accessed before the minimum read cycle time has been satisfied by outputting unknowns. If this feature is not useful for your particular simulation needs, you can disable it by creating a custom timing file. For information about custom timing files, refer to "User-Defined Timing" on page 157.
- **Monitor internal memory.** Change or monitor the values of internal memory locations using SmartModel Windows memory window elements. For processor models that have internal memory, you can use PCL commands to get the values of internal memory locations. .

The Memory Image File (MIF)

A memory image file (MIF) is an ASCII file containing memory data to be loaded into a model before simulation, thus saving the simulation time that would otherwise be required to load the memory. You can load a memory model with a file that has previously been dumped from the same or from a different memory model, as long as their word widths are compatible.

One common use of memory image files is for programming models of ROMs and PROMs. If these models are not configured with an associated MIF file, they issue a warning message at simulation startup.

Creating a Memory Image File (MIF)

To create a MIF file, follow these steps:

1. Using a text editor, create a file (for example, ram4kx4.img)
2. Enter the memory data, using the correct MIF file format.

Using a Memory Image File (MIF)

To load a model's simulated memory from a MIF file, follow these steps:

1. Configure the model to use the MIF file by assigning the file to the SWIFT MemoryFile parameter.
2. Start the simulation. MIF files are automatically loaded and checked for format errors.

If you do not load memory data at simulation startup, check the model datasheet to determine the default memory values at initialization. The contents of a memory device model's internal memory at initialization depend on the manufacturer's implementation. Typically, the default for RAMs is "unknown" and the default for ROMs is "1."

You can modify previously addressed locations multiple times using one MIF file. This way you can load the entire memory image with one line in your MIF file and then modify selected values on a case-by-case basis.

Memory Image File (MIF) Format

A MIF file contains one or more records. Each record specifies the data to be written to one or more memory locations.

MIF File Conventions

The following list shows the conventions and rules that apply to the syntax description for MIF file records:

- Braces ({ }) indicate a list of one or more entries.
- Brackets ([ ]) indicate optional entries.
- Italics indicate variables for which you specify actual values.
- Fields are not case-sensitive.
- More than one record can appear on a line.
- The character "X" or "x" indicates an unknown value, and is illegal except in a data word where the data is expressed in binary, octal, or hexadecimal (not decimal).

MIF File Record Syntax

Following is the syntax for MIF file records:

{*address1* [:*address2*] / *base_specifier data_value*;} [# *comment*]

address1
The memory location to which data is to be written, or the beginning address of a range.

:*address2*
The end address of a range. Either a colon (:) or a hyphen (-) can be used as a delimiter.

/*base_specifier*
A slash (/) separates the address specification from the data word. The *base_specifier* argument is one of the following:

```
'b.  Binary
'o.  Octal
'd.  Decimal
'h.  Hexadecimal (the default)
```

You can mix different base numbers within a record.

***data_value*;**
The value of the data word to be written to the specified memory locations. A semicolon (;) defines the end of each record.

 # comment
A comment can be included in a record by using the pound sign (#). All information from the pound sign to the end of the line is treated as a comment.

Example 1

The following example shows how various constructs can be used or combined in a MIF file. In this example, the width of the memory location is 8 bits.

```
0:3/0; #Colon separator for address range
4-6/0; #Hyphen separator for address range
'd7/'b10101110;    #Address and data can use a different
                   #numeric base
10/0; 11/'b10000000; #Two records on the same line
12:1e/'HxF; 'd31/'hX8; #Information is case-insensitive
20:7FF/4; #Load remaining addresses with 00000100
```

Example 2

When you specify the data value to load into memory, the safest practice is to specify values that match the width of the memory location; however, this is not required. If the data value has fewer bits than the memory location, the model pads the value with leading zeros. If the data value is larger than the memory location, the model rejects the data and issues a warning message.

The following example specifies that the hex value F (binary 1111) is to be loaded into memory location 0 (zero). If the memory location is 9 bits wide, the value entered is 000001111; if the location is 6 bits wide, the value is 001111; and so on.

```
0/F
```

Example 3

Unknown values are most easily specified in binary; often the unknown represents a single bit. In the following example, for an 8-bit memory location the binary value is loaded into 0F exactly as written; the hex value xF is loaded into FA as xxxx1111. For a 9-bit memory location, the binary value is loaded as 01010x0x1 and the hex value as 0xxxx1111.

```
0F/'b1010x0x1;
FA/xF;
```

Memory Image File (MIF) Address Mapping

Each record in a MIF file specifies an address followed by the data value to load at that address. To translate the address in a MIF file to a column and row address (or vice versa), follow the steps below. (Or, if you have an Intel Hex or a Motorola S-record memory image file, you can use the mi_trans tool to translate the file into the format used by memory models in the SmartModel Library. For more information, refer to "Translating Memory Image Files" on page 198.)

1. Use Table 5 to find the number of bits in the row and column addresses.

Table 5: Bits in Row and Column Addresses

Device Size	Row Bits	Column Bits
4 MB	11	11
1 MB	10	10
256 KB	9	9
64 KB	8	8

2. Write the address in the memory image file, in binary, padding with leading zeroes to get the correct number of bits. For example, with a 1 MB memory device, the address 4834 hexadecimal expressed in 20 bits is:

   ```
   0000 0100 1000 0011 0100
   ```

3. Divide the bits into two sets. The upper number of bits is the row address and the lower is the column address:

   ```
   00 0001 0010   00 0011 0100
   ```

   The row address (in hex) is 12; the column address is 34.

   As another example, we use a 256 KB memory and a MIF file address of 2405. Written as an 18-bit value, the address is:

   ```
   00 0010 0100 0000 0101
   ```

   Divide the 18-bit address into two 9-bit segments and translate back to hex:

   ```
   0 0001 0010   0 0000 0101
   ```

   In hex, the row address is 12, and the column address is 05.

Memory Image File (MIF) Format Checks

When a model configured with a MIF file is loaded for simulation, the model performs error checks to validate that the following conditions are true:

- Valid memory locations for the device were specified.
- Legal specifications were used.
- Data values do not exceed the memory width.

If a model's MIF file loads correctly, you will see a message similar to the following example:

```
Info: Loading the memory image file "U2.mem".
       (n=U2) (comp=lai_cy7c128-14-0-8) (loc=A1-48) (lai=CY7C128-25),
       at t=0 (0.0 ns).
--- 17 values have been initialized.
```

If a model's MIF file does not load correctly, the resulting message will be similar to the following example:

```
Info: Loading the memory image file "U2.mem".
       (n=U2) (comp=lai_cy7c194-14-0-8) (loc=A1-48) (lai=CY7C194-25),
       at t=0 (0.0 ns).
--- Invalid data value "DB" on line 2.
       It must be less than or equal to F. The line will
       be ignored.
--- Invalid data value "18" on line 3.
       It must be less than or equal to F. The line will be
       ignored.
--- Invalid data value "50" on line 7.
       It must be less than or equal to F. The line will be
       ignored.
--- 7 values have been initialized.
```

If you specify a MIF file and the model fails to locate it, a warning message identifying the model is generated when the model is loaded. As long as the memory image is not required during the simulation, neither the warning message nor the default parameter value on the memory device will adversely affect your simulation.

Dumping Memory Data

Models that simulate internal memory locations can write their contents to an external system file—referred to as a dump file. You can use the SWIFT command channel DumpMemory command to write the contents of a model's simulated memory locations to a dump file at any time during a simulation. If the specified file does not exist, it is created. If the file already exists, it is overwritten. The memory dump operation allows you to eliminate the read cycles required to verify the success of a test. If you issue the DumpMemory command on a model that does not have internal (simulated) memory locations, a warning message is issued.

The dump file format is the same as MIF file format—addresses and data are represented in hexadecimal, except that data is represented in binary if the data contains any unknown bits.

The size of the dump file is minimized by filtering data that remains in its initial or power-up state, and by writing out only one data line for contiguous addresses that contain the same data value. For example, consider the following memory contents—8 bits wide:

```
Addr 0 = 0
Addr 1 = 0
Addr 2 = 0
Addr 3 = 0
Addr 8 = 11111111
Addr 15 = 1100X1X0
```

All other addresses contain an initial value of X.

The dump file contents would be:

```
0:3/0; 8/FF; F/'b1100X1X0;
```

Note

If you subsequently load the dump file for the same instance of a memory model, you are guaranteed to put the memory back in exactly the same state it was in when it was dumped.

5
PLD Models

Configuring PLD Models

Like the actual devices, Programmable Logic Device (PLD) models in the SmartModel Library are programmable. To configure a PLD model, you use a JEDEC standard file. PLD models check their initialization files when loaded for simulation, as shown in Figure 10.

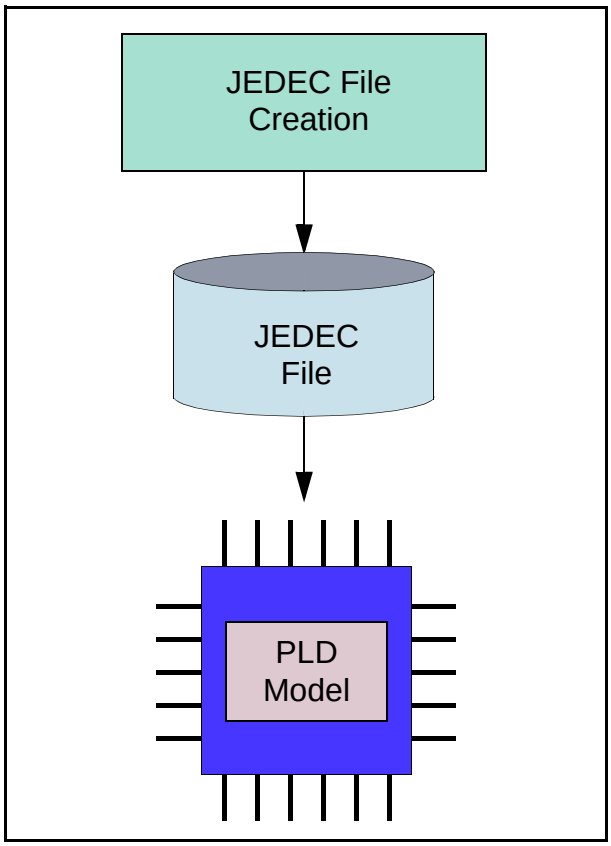


Figure 10: Process Flow for PLD Models

Like all models in the library, PLD and PAL models provide error checking during simulation.

Each PLD model datasheet contains a port name to pin number cross-reference table, which shows the mapping of a model's port names to a particular package type's pin numbering.

Programming PLD Models

PAL and PLD models are programmed with a file that conforms to JEDEC STANDARD No 3-A, Standard Data Transfer Format Between Data Preparation System and Programmable Logic Device Programmer, May 1986. To configure a model to use a particular JEDEC file, use the SWIFT JEDECFile parameter. Table 6 shows the programming and testing fields specified by the JEDEC standard.

Table 6: JEDEC Standard 3-A Fields and Their Uses in PLD Models

Identifier	Description	Use
not applicable	Design specification	required
N	Note	not used
QF	Number of fuses in the device	not used
QP	Number of pins in test vectors	not used
QV	Maximum number of test vectors	not used
F	Default fuse state	optional if Field L is used
L	Fuse list	optional if Field F is used
C	Fuse checksum	optional, and "0000" is valid
X	Default test condition	not used
V	Test vectors	not used
P	Pin sequence	not used
D	Device (obsolete)	not used
G	Security fuse	not used
R, S, T	Signature analysis	not used
A	Access time	not used

The F and L fields are complementary; if one is used, the other is optional. Fields marked "not used" still can be included in the model's programming file but they have no effect.

You can use a JEDEC standard file already created by programs like ABEL, CUPL, or PALASM, or you can create your own simplified version. If you create your own JEDEC file, the PLD models do not require all the fields specified by the standard; use the programming and testing fields described in the table.

Following is an example of a simple JEDEC programming file. The example specifies that the default fuse condition is a low-resistance link (0). The fuses from 0 to 39 are explicitly defined by the L field; and a checksum is used.

```
DUMMY HEADER*
F0*
L0000 01001110 00001000 11110000 11111111 01010001*
C021A*
```

Note

As defined in the JEDEC standard, if a fuse is specified more than once, the last state replaces all previous states for that fuse. If more than one checksum field is in the file, the last one is used, which allows a file to be easily modified or patched.

JEDEC File Format Checks

At simulation startup, each PLD model searches for its JEDEC programming file. If the programming file is missing or unspecified the model issues a warning message. If the programming file is properly specified, the model instead issues an informational message when loading is complete. The message format is similar to that shown in the example below.

```
Info: Loading the JEDEC file "U23.jed".
   (n=U23) (comp=EP1800) (loc=A9-12) (lai=EP1800), at t=0 (0.0 ns).
--- 12680 fuses have been blown.
```

When a PLD model with a JEDEC programming file is loaded for simulation, the model performs a series of error checks. The model checks the order of the fields in the file and then compares the character types and number of digits against the field type. The model also checks that all fuse links are specified, and that all the addresses are legal for the device. Finally, the model computes the checksum and checks it against the checksum in the JEDEC file.

The following example shows the kind of information message that the model generates after going through the error checking sequence.

```
Info: Loading the JEDEC file “U31.jed”.
   (n=U31) (comp=PAL16R4) (loc=B1-8) (lai=mmi_16r4), at t=0 (0.0 ns).
--- Invalid numeric character "L" on line 23.
A hexadecimal digit is expected.
--- The checksum value “3056” on line 63 does
not match the calculated value “3543”.
--- Not all of the fuses have been defined. The
first undefined fuse is at address “4”.
--- 1036 fuses have been blown.
```

Using PLD Models

To make PLD models easier and more efficient to use, follow these guidelines:

- Key JEDEC file names to your instance name.

  Use the instance name as part of your JEDEC file name. This is a handy way to pair PLD models with their programming files because each model has a unique instance name.

- Patch programming files rather than rewriting them.

  PLD models conform to the JEDEC standard, which contains provisions for easily patching a programming file. To patch a programming file, simply append the new fuse and/or checksum data to the end. The new data always replaces previous specifications.

- Reuse file names rather than changing the SWIFT JEDECFile parameter.

  In many simulation environments, changing the file name requires changing the schematic, recompiling, and then restarting the simulation. You can avoid this problem by reusing a model's file name for a file with new information in it rather than instantiating the part and changing the value of the property or parameter.

6
SmartCircuit FPGA Models

Introduction

SmartCircuit models simplify the integration and simulation of device models from the leading FPGA and CPLD device vendors. And the debugging tools designed specifically for use with SmartCircuit models enable you to monitor design state, trace cause and effect events, and analyze your design structure.

This chapter presents user and reference information for SmartCircuit models and their debugging tools organized as shown in the following table.

Type of Information	Is Located In ...
Overview	• "Using SmartCircuit Models" on page 84 • "SmartCircuit Technology Overview" on page 86 • "Debugging Tools Overview" on page 89
Procedure	• "Tracing Events In Your Design" on page 91 • "Viewing Internal Nodes During Simulation" on page 95 • "Browsing Your Design Using SmartBrowser" on page 106
Reference	• "SmartBrowser Command Reference" on page 111 • "Model Command File (MCF) Reference" on page 119 • "smartccn Command Reference" on page 122 • "ccn_report Command Reference" on page 125

For a detailed application note that discusses how to use the various SmartCircuit debugging tools to verify an FPGA design, refer to the *SmartModel Products Application Notes Manual*.

Using SmartCircuit Models

SmartCircuit models integrate smoothly with the software tools from leading CPLD and FPGA device vendors and third-party vendors that produce netlist or JEDEC files back-annotated with package-pin and/or timing information.

SmartCircuit models must be configured and initialized before they will operate. The model configuration phase consists of attaching SWIFT parameters to a model instance via the simulation environment. The software assigns default names to these parameters. With SmartCircuit models, you must set the SCFFile parameter to point to the location of the model command file (MCF) for the model.

If the model cannot find the specified MCF file at initialization, it issues an error message. When the model finds the specified MCF, it checks the file syntax and executes the commands that it contains, including the "load" command found in all MCF files. After loading the compiled configuration netlist (CCN), the model initializes. If the model cannot load the CCN file, it issues an error message.

The operation phase begins at time 0 after the model loads the MCF. The actual functions performed depend on the device being modeled; details are documented in the datasheets for the device and model.

Note

SmartCircuit models do not support JTAG functions or configuration through boundary scan pins.

Quick Start for SmartCircuit Models

Before using any SmartCircuit model in a design, refer to the model's datasheet for specific information about how to configure and use that particular model. Individual model datasheets provide information about technical issues or special usage considerations you may need to be aware of, such as specific options required in the FPGA vendor tools to target the netlist to SmartModels. For information on finding model datasheets using the Browser tool, refer to "SmartModel Datasheets" on page 25.

To use a SmartCircuit model in a design, follow these steps:

1. Instantiate the model in your design.

   For information about instantiating SmartModels and configuring them for use in your simulator, refer to the *Simulator Configuration Guide for Synopsys Models*.

2. Generate a design netlist or JEDEC file.

   Using the design tools provided by the device manufacturer or third-party vendor, generate a netlist or JEDEC file for your design. Refer to the model datasheet for information on the required netlist format or JEDEC file for your device.

3. Create a model command file (MCF).

   An MCF file is an ASCII file that contains instructions the model executes at startup. Your MCF can have any name you choose, but the convention is to give these files .mcf extensions for consistency.

   Using an ASCII editor, create and save a file that contains the following line:

   ```
   load -source netlist_name
   ```

   where *netlist_name* is the path to the netlist or JEDEC file you generated in Step 2.

   For example, if your netlist is named sample.edo, use the following command in your MCF file:

   ```
   load -source sample.edo
   ```

   The model automatically loads the specified netlist and translates it into a compiled configuration netlist (CCN), if necessary.

4. Verify or change the model's SCFFile parameter.

   To use a SmartCircuit model in a simulation, set the value of the SCFFile parameter to the path name of the model's MCF file. This can be done in one of two ways:

   - Edit the parameter value to use the name of the MCF you created in Step 3.

     -or-

   - Assign your MCF file the same name as the current value of the SCFFile parameter.

1. Run your simulation.

SmartCircuit Technology Overview

SmartCircuit technology uses a hierarchy of cell descriptions to model a family of FPGAs or CPLDs. Each cell provides logic building blocks that map to specific functions of the device. The cell descriptions for a device family are combined into a cell library for that family of devices.

Figure 11 illustrates the typical data flow for a SmartCircuit model; boldfaced items indicate tools or files supplied by Synopsys.

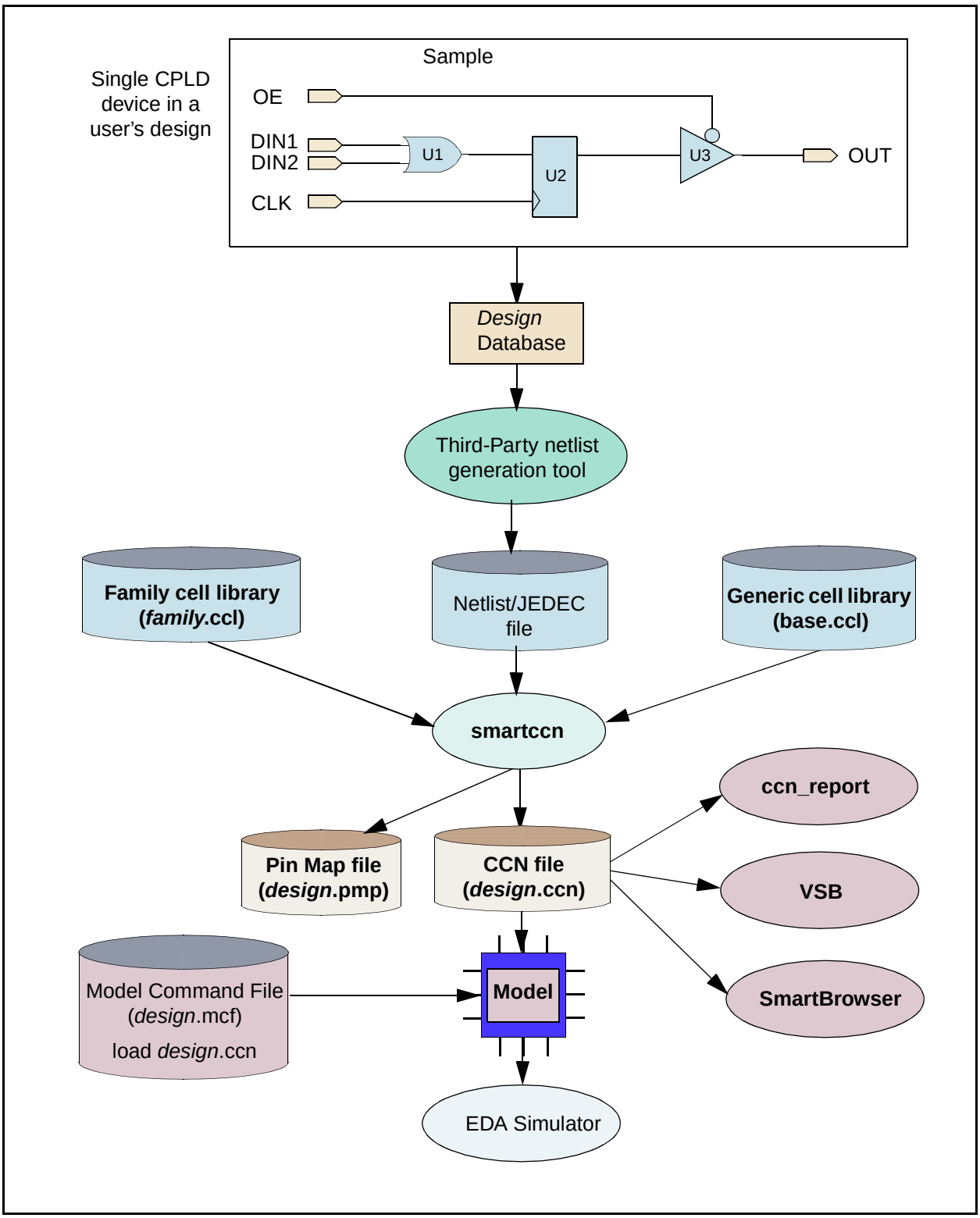


Figure 11: SmartCircuit Model Data Flow

At the top of Figure 11, starting with a single FPGA or CPLD device you have instantiated in your design, you use the device manufacturer's (third-party) compiler tool to generate an FPGA design netlist or a JEDEC file for the model along with any other required design-specific options. (Refer to the individual SmartCircuit model datasheets for information about the design netlist or JEDEC file format required for a specific model).

SmartCircuit models also obtain vendor-specific functions from a cell library (for example altera.ccl) that is based on device families from semiconductor vendors such as Actel, Altera, Vantis, Lucent, Cypress, Intel, Lattice, QuickLogic, and Xilinx.

The smartccn tool uses inputs from both of these sources to produce the complied configuration netlist (CCN) file needed to configure a SmartCircuit model for simulation. You can also use the smartccn tool to generate other files, such as pin-map files (.pmp). For more information, refer to "smartccn Command Reference" on page 122.

You can later extract information from the CCN file using the ccn_report tool, the SmartBrowser, or the Visual SmartBrowser (VSB). For example, you can generate a windows definition file, which allows you to monitor internal nets within your design.

As illustrated at the bottom of Figure 11, the simulator loads the model, which configures itself based on commands found in the MCF file.

User-Defined Timing for JEDEC-based Models

If you have created a timing file that contains timing specifications that are newer than those shipped with the model, you can compile the netlist with the more recent timing information. Note that the user-defined timing feature only works with JEDEC-based SmartCircuit models.

To compile the new timing file into a model, invoke smartccn with the -u switch and specify the name of the user-defined timing UDT file. The -u switch directs smartccn to use the timing contained in the specified UDT file.

The following example causes smartccn to compile a new CCN file for the mach110 model using a UDT file called *my_new_timing*.tf.

```
% smartccn sample.jed -u my_new_timing.tf -m mach110
```

Debugging Tools Overview

SmartCircuit debugging tools work in conjunction with the models to help you verify and validate your design at the back-end system level. After your design has been flattened, mapped, and fitted—to the point where the original structure is lost—SmartCircuit models and debugging tools help you visualize and fix design issues.

The SmartCircuit debugging tools have four major components:

- **Causal Tracing** (described on page 91)—enables you to identify the source of timing or functional problems for models of CPLD and FPGA devices. For example, you can use causal tracing to locate the source of setup, hold, or pulse width violations before running your simulation. Afterwards, you can trace a design error or a constraint violation to its source and quickly determine if the error is in the system or in the programmable logic design.
- **SmartCircuit Monitor** (described on page 96)—lets you view internal states and signal elements inside your programmable logic design. You select probe sites of interest and SmartCircuit Monitor reports back state and net information through the transcript window of your simulator.
- **SmartBrowser** (described on page 106)—allows you to dissect the design netlist to observe connectivity of elements and their properties.
- **Visual SmartBrowser (VSB)** (described in *Visual SmartBrowser User's Manual* – UNIX version or NT version)—provides all the capabilities of the command-line SmartBrowser, while adding improved visual traversal and display of your programmable logic design through an easy-to-use GUI interface.

SmartCircuit Monitor and Causal Tracing commands are placed in the model command file (MCF), along with the standard MCF commands. At simulation startup, the simulator reads the MCF for a model and interprets the commands it contains, including debugging commands. The output from these commands is piped to the simulator transcript window. Based on the information you gather with these tools, you can make changes to your design or MCF and then rerun your simulation.

Sample Circuit

To help you better understand how to use SmartCircuit models and the debugging tools, use the sample circuit in Figure 12 as a reference for the examples presented in this chapter.

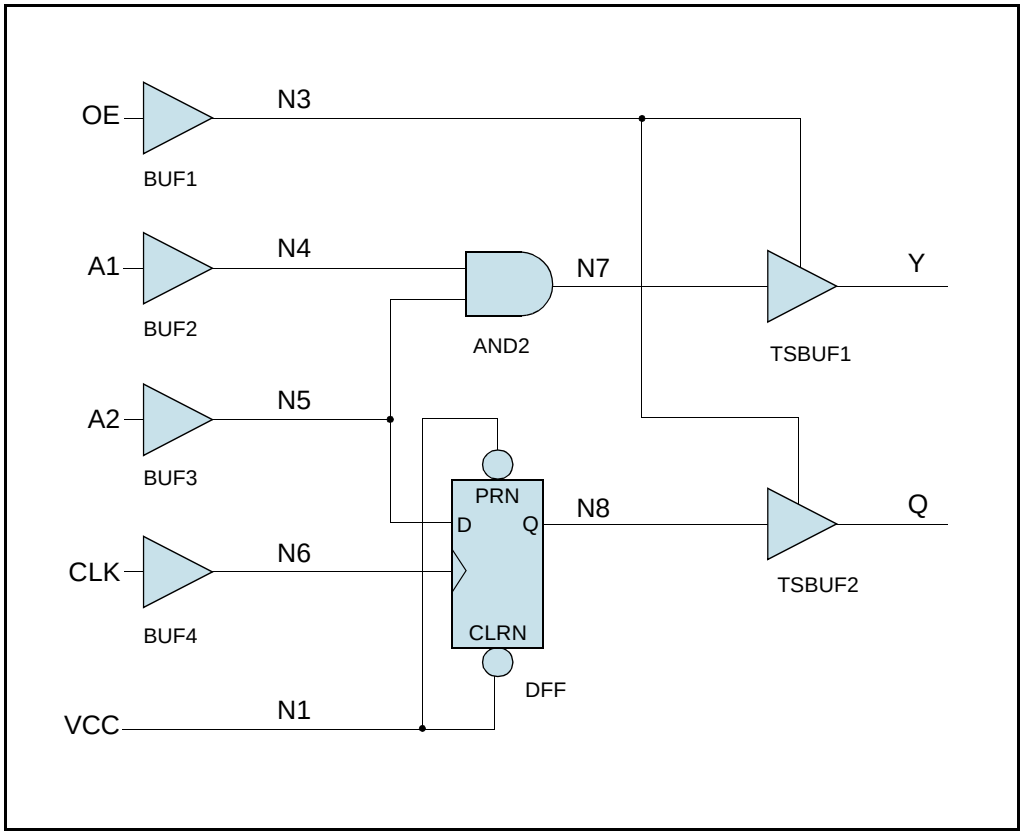


Figure 12: Sample SmartModel Circuit

SmartCircuit Model Pin Mapping

Place and route tools assign physical pin numbers to nodes in your design (schematic or HDL). When connecting a SmartCircuit model symbol in a schematic (or netlist), you must know which pins on the symbol correspond to the nodes in the design. Use the model's pin map file (.pmp) as a cross-reference between pin names and numbers.

The pin map file generated by using the -p switch with the ccn_report tool is a duplicate of the pin map file generated by the smartccn compiler. Pin map files contain cross-references between model port names, package pin numbers, and the design netlist. Figure 13 illustrates this relationship.

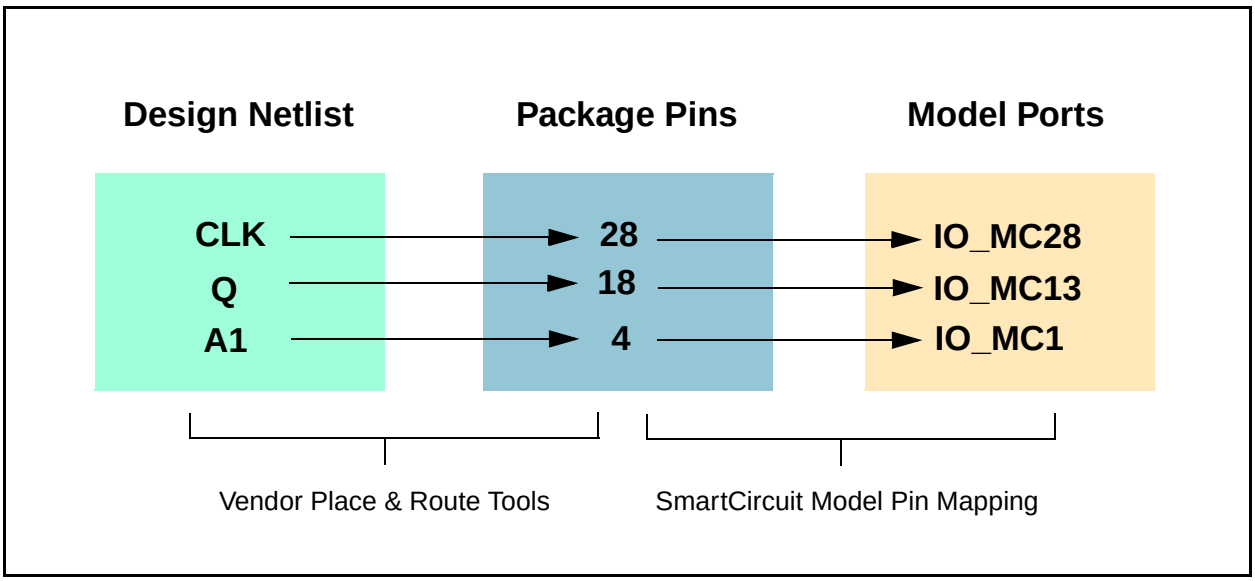


Figure 13: SmartCircuit Pin-to-Port Mapping

Note

Pin naming conventions are detailed in the datasheets for individual SmartCircuit models. For information on finding model datasheets using the Browser tool, refer to "SmartModel Datasheets" on page 25.

Tracing Events In Your Design

When you encounter problems with a design during a simulation run, it is often helpful to be able to trace events to see where the problem is occurring or to trace the causes of a problem to their roots. You can use the causal tracing capability to do just that.

Throughout this discussion of causal tracing, we often use the terms parent event and child event. Parent events and child events have a cause-and-effect relationship; some stimulus or circumstance on the parent event causes the child event to occur. An event that was caused by a preceding event, and in turn causes another event, is both a parent and a child at the same time.

Causal tracing consists of two commands: set cause and report. These commands have several forms you can use to define the scope of event tracing and produce reports detailing the cause or effect of an event. Report output appears in the model message transcript.

The signal values displayed within causal tracing are the same as those displayed by the monitor command. These signal values are described in Table 8 on page 97.

Causal Tracing Command Descriptions

Following are summaries of the causal tracing commands that you can put in the model's MCF file. For causal reporting purposes, when events occur simultaneously at a circuit element, only one event is determined to have caused any subsequent circuit changes.

- **report cause**—produces a list of parent events of the triggering event on a net or port that occur between start_time and stop_time.
- **report effect** (page 93)—produces a list of child events of the triggering event on a net or port that occur between start_time and stop_time.
- **set cause** (page 94)—determines what events are reported when a constraint violation occurs or determines the general scope of causal reporting.

You can use the report cause and report effect commands to produce reports showing the cause and effect of events that take place during your simulation. The following sections provide details about each command.

report cause Command

report cause *name start_time* [*stop_time*]

Produces a list of parent events of the triggering event on a net or port that occur between *start_time* and *stop_time*. If you don't specify a stop_time, the tool checks the event only at the time specified by *start_time*.

The report cause command traces the cause of a signal event on a node in a design. A node can be any net or port defined within a design.

A cause report is simply the chronological history of signal events that lead up to the trigger event. The report is in reverse chronological order—it starts at the trigger event and traces backward until it reaches the causal signal event that is responsible for the trigger event.

When used in conjunction with the set cause full command, the cause report includes the history of all signal events, starting at the trigger event and working backwards to the causal event.

When used in conjunction with the set cause nofull command, the cause report includes only the causal event.

Note

The report cause command can trace only one causal event at a time. When report cause encounters simultaneous signal events that have caused the trigger event, it traces only one event. Which event gets traced depends on the internal ordering of the model's causal history records.

report effect Command

The report effect command traces the effect of a signal event on a node within a design. A node can be any net or port defined within a design. The effect report is a history of all signal events, starting at the trigger event and working forward to all terminal events. A signal event on a design port is considered terminal, even though the port may be bidirectional.

For example:

report effect *name start_time* [*stop_time*]

produces a list of child events of the triggering event on a net or port that occur between *start_time* and *stop_time*. If you don't specify a *stop_time*, the tool checks the event only at the time specified by *start_time*.

When the report effect command encounters simultaneous signal events at an internal circuit element, one signal event is considered to have caused any subsequent signal events. The report ends at the point where the event being traced does not cause any subsequent events.

To produce an effect report for the sample circuit shown in Figure 12, your MCF would need to contain these lines:

```
report effect CLK 150
report effect A1  200
```

Unlike cause reports, effect reports are not bounded. In a cause report, the start and end times are known, and the report is bounded by these times. In effect reports, the end time is unknown when the report begins, so effect reports may be interrupted by other reports or by model messages.

Note

You cannot change the scope of effect reports. The scope of effect reporting is always full.

set cause Command

The set cause command has two variations: one that determines the general scope of cause reporting, and a second that determines the constraint violation scope of cause reporting.

For example,

set cause constraint | noconstraint

determines what events are reported when a constraint violation occurs. The default setting is noconstraint. The constraint argument causes the analysis report to contain all events involved in a constraint violation. For instance, if there is a setup or hold violation, then the DATA and CLK paths that caused the event to occur are reported. The noconstraint argument causes the model to produce a report listing only the errors, with no tracing information, when a constraint violation occurs.

Whereas,

set cause full | nofull

determines the general scope of causal reporting. The default setting is full, which produces an analysis report that contains all causal events, from the trigger through the earliest parent event. The report produced when you use the nofull argument is significantly shorter, and lists only the parent events.

Note

You must specify the full/nofull and constraint/noconstraint arguments using separate set cause commands.

To produce a cause report with the scope set to full for the sample circuit shown in Figure 12, your MCF file would need to contain these lines:

```
set cause full
report cause Y 150 250
report cause Q 280
```

Viewing Internal Nodes During Simulation

There are two different ways to view events on internal nodes during simulation using SmartCircuit models:

- "SmartModel Windows" on page 95
- "SmartCircuit Monitor" on page 96

Table 7 compares these different viewing methods so that you can choose which one is best for your needs.

Table 7: Windows and Monitors Tool Comparison

Viewing Tool	Read the State of Internal Nodes?	Write to the State of Internal Nodes?	Output Appears In ...
SmartModel Windows	Yes	Yes (1)	Simulator waveform viewer
SmartCircuit Monitor	Yes	No	Simulator transcript window

(1) Not all SmartModel Window elements are set up for write permission. For information on the read/write status of window elements, refer to the individual model datasheets.

Although the SmartBrowser tool (described on page 106) does not allow you to read or write to internal nodes, it does provide the complementary capability to view overall design topology and properties.

SmartModel Windows

SmartModel Windows is a feature that allows you to access internal net and state information during simulations. SmartModel Windows is especially useful with SmartCircuit models because it enables you to monitor and change design element values.

With SmartModel Windows, you can create windows for two types of design elements: states and nets. When you force a window element to some value, it remains at that value until a normal event occurs that changes the state of that element or until you apply a new forced value.

You define windows by placing statements in the MCF file. You can also place these statements in a separate file that is referred to by a "do" command in the MCF file.

Creating Buses and Windows

Sometimes it is useful to combine a number of design elements into a bus and then window the bus. You can also create a bus to rename a single design element. In some simulators, such as Cadence Verilog-XL, you need to use the bus command to alias a net or state name, because the net or state name contains illegal characters.

For example, you cannot window the net name /Block1/Net1 in the Verilog-XL environment because it contains the illegal character '/'. One solution to this limitation is to create a bus for each of the elements that contains illegal characters, and then create a window for the bus, as shown in the following example.

```
bus State1 /DFF/STATE
window State1
```

Use the SmartBrowser interactive utility or the Visual SmartBrowser (VSB) to identify which nets and states are available to be windowed. These tools both let you view a CCN file, browse the design, and list all nets, states, and instances along with their connections. For more information on the SmartBrowser tool, refer to "Browsing Your Design Using SmartBrowser" on page 106 of this manual. The VSB tool is documented in a separate manual entitled *Visual SmartBrowser User's Manual* (UNIX version or NT version).

You can use the AutoWindows feature of the ccn_report tool to automatically generate a windows definition file. Then, use the "do" command to include the windows definition file in your MCF file. For more information on creating AutoWindows, refer to "AutoWindows" on page 128.

SmartCircuit Monitor

The SmartCircuit Monitor enables you to observe any element in your design and receive messages in the simulator transcript window about any changes that occur on that element. The specified elements can be any nets or buses in your design.

Note

The number of monitors that can be active at one time is determined by the maximum length of a message string allowed by your simulator. Some simulators might allow as few as 256 characters in a message string, thus limiting the number of monitor statements you can use.

Let's take a look at an example based on the sample circuit shown in Figure 12. To monitor an element, you include a monitor command statement in the model's MCF file. For example, if you want to monitor input nets OE, A1, A2, and CLK; internal nets N3 - N8; and output nets Y and Q, the command in your MCF would look like this:

```
monitor OE A1 A2 CLK N3 N4 N5 N6 N7 N8 Y Q
```

When you start your simulation and load the MCF file, monitors are assigned to the elements specified on the command line.

To make the report easier to read, place a label at the beginning of each line of the report. Use the set label command, from within the MCF file, to specify the string. For example, to specify the label "SAMPLE>":

```
set label SAMPLE>
```

The complete MCF file for our example would look like this:

```
load -source sample.edo
set label SAMPLE>
monitor OE A1 A2 CLK N3 N4 N5 N6 N7 N8 Y Q
```

You can also assign monitors using the Visual SmartBrowser or the SmartBrowser. For information on using the Visual SmartBrowser, refer to the *Visual SmartBrowser User's Manual* (UNIX version or NT version). For information on using the SmartBrowser tool, refer to "Browsing Your Design Using SmartBrowser" on page 106.

When you assign a monitor to an external I/O port or any net connected directly to an external I/O port, the monitor is placed on the input side of the port. This enables the monitor to report the value being driven into the model. To monitor the value driven out of an external I/O port, you must access the external port value via the simulator interface.

Note

Some models have special input pin attributes, such as pull-up resistors. In such cases, the monitor command reports the resolved value of the simulator's input and the model's input pin attribute.

SmartCircuit Monitor Signal Values

The output produced by SmartCircuit Monitor commands includes a set of signal values (see Table 8). Each bit of a monitored signal is represented by a single character in the output.

Table 8: Monitor Signal Values

Output	Value	Description
0	Logic 0	Signal strength strong
1	Logic 1	Signal strength strong
X	Unknown	Signal strength strong
Z	High-impedance	High-impedance

Table 8: Monitor Signal Values (Continued)

Output	Value	Description
L	Logic 0	Signal strength resistive
H	Logic 1	Signal strength resistive
R	Unknown	Signal strength resistive
U	Uninitialized	Uninitialized

Using Unsupported Devices

If you need to use a device type that is not currently supported within a particular device family or model, you can use smartccn to generate an interface file. You might need an interface file in the following cases:

- When the device type specified by your netlist is not supported by any available models (for example, a new vendor device type).
- When the device type specified by your netlist should map to an available model, but is not recognized by that model (for example, the device designator is either obsolete or new).
- For a JEDEC-based model, when the component name specified on the command line is not recognized by that model (for example, when adding a new or a custom timing version).

The smartccn tool uses interface files to define the mapping between a device's pins and a model's ports. To use an unsupported device, follow these steps:

1. Select the correct model according to the following criteria:
    - If the device type specified within your netlist is not supported by any available models, then select a model from the same vendor family that has a pin count equal to or greater than the pin count of the device you are using.
    - If the device type specified within your netlist should map to an available model, but does not, then select the model that it should map to.
    - If you are using a JEDEC-based model with an unrecognized component name, then select the model that component designates.

Note

You can map an unsupported JEDEC-based device into an existing JEDEC-based model only if the devices have identical fuse maps.

2. Create an interface file by running smartccn with the -g switch as shown in the following example:

```
% smartccn -m model -g
```

   The output file is named *model*.inf.

3. Add your device to the interface file using an ASCII text editor as follows:

   a. Find the correct DEVICE specification section within the interface file. Note that although most models generate an interface file with a single DEVICE specification section, some models support multiple packages (for example, PGA and LCC). Such interface files contain a different DEVICE section for each unique package.

   b. Modify the correct DEVICE specification section by either adding a new DEVICE line containing your device name or replacing one of the existing DEVICE lines with you device name.

1. If you are mapping a new vendor device into an existing model, modify the device-to-model mapping using an ASCII text editor as follows. In the device specification section, modify the package pin number value within the pin records until you have one "corrected" PIN record for each of the user-configurable pins on the new device. You can rearrange the order of pin statements and freely map any user-configurable pin on the new device into any user-configurable pin in the existing model, but do not edit pin names and do not map a user-configurable pin to a non-configurable pin. Excess pins that are not mapped to the new device may have any package pin number, as long as they don't collide with the new pin numbers you are targeting. The excess pins are ignored during simulation.

   Some models have special non-configurable pins (for example, the PROGRAM pin in the Xilinx 4k family). You can identify these pins by examining the pin type designator within a pin record. To avoid simulation problems, these pins need to be mapped correctly. Do not map a user-configurable pin to a non-configurable pin (ideally, these pins should be mapped directly between devices).

Note

Do not change pin names. Also, do not add or delete PIN statements from a device declaration. Changes like these can make the model nonfunctional.

2. Generate a new CCN file by running smartccn with the -i switch, which forces the compiler to read its device-to-model mapping information from the specified interface file.

Example

The following example causes smartccn to generate a new CCN file for the mach110 model based on device-to-model mapping information in the specified interface file.

```
% smartccn -m mach110 -i mach110.inf
```

Interface File Format

Interface files contain one or more device declarations for one or more models. Each device declaration conforms to the format shown below. Any interface files you create must also conform to this format.

```
# Begin device declaration
  DEVICE : <device_name>[, ...]
  MODEL <model_name>
  LIBRARY <library_name>
  PIN <pin_name> <package_pin_number> [ { C | O | I | B } ]
  . . .
  END
# End device declaration
```

Keywords (DEVICE, MODEL, etc.) and pin names must appear in uppercase, while the model name and library name must be lowercase. Device names are case-insensitive.

Hint

To create a new device declaration block, you will probably find it easiest to copy and modify an existing block that you know is properly formatted

If you include multiple device declarations in an interface file, each declaration must use unique device names. Ordering of pin declarations is not significant. A pound sign (#) at the beginning of a line signifies a comment, which the compiler ignores.

You can determine the pin type by looking in the interface file for the declaration of a device with the same type.

Following are descriptions of the SmartCircuit interface (.inf) file keywords.

DEVICE : *device_name*
: Specifies the name of the device as it appears in the netlist file (for example, 3020PG84 or 2064pc68).

END
: Signifies the end of a device declaration (required).

MODEL *model_name*
: Specifies the name of the model.

LIBRARY *vendor_cell_library*
Specifies the name of the device model library.

PIN *pin_name package_pin_number* [C | O | I | B]

pin_name
The name that Synopsys assigns to each model pin (required).

package_pin_number
The physical pin number for the specified pin (required).

[C | O | I | B]
Pin type designator (optional); one of the following:

C Designates a user-configurable pin or a pin that is used by the configurable portion of the device.
O Designates a non-configurable output pin.
I Designates a non-configurable input pin.
B Designates a non-configurable input/output pin.

Using Unsupported Devices Example

Let's say you have a 240-pin FPGA model, but need to simulate with a 208-pin model from the same device vendor. If a 208-pin model is not available, you can create your own by modifying the interface file of the similar 240-pin model that you already have in the library. First, create an interface file (.inf) by running the smartccn tool on the existing 240-pin model as described in "Using Unsupported Devices" on page 98. The following interface file example was created by running smartccn on the xcs30_240 model from Xilinx.

Next, edit this interface file to make it work for the 208-pin device. For details on the .inf file syntax and editing rules, refer to "Interface File Format" on page 100. In this example, you need to add a DEVICE line to match the device you are targeting:

```
DEVICE XCS30-3PQ240C s30xlpq208-4
```

Then assign the extra pins that you don't need to pin numbers that don't exist. Don't comment out unneeded pins, because the smartccn netlist compiler requires the number of pins to match the model specification (240 in this case). Here, we assigned the unneeded pins to false pin numbers of 900 and above, as shown in the following edited version of the interface file. (Note that the ordering of pins is not significant.) The user edits to the original .inf file in the following example are highlighted.

```
# Edited inf file
DEVICE XCS30-3PQ240C s30xlpq208-4
MODEL xcs30_240
LIBRARY xilinx
LIBRARY simprims
# PIN <symbol_pinName> <package_pinNumber> <pinType>
```

```
# where pinType is either [I]nput, [O]utput, [B]idirectional,
[C]onfigurable
PIN CCLK 155 C
PIN DIN 153 C
PIN DONE 104 C
PIN ERR_INIT 77 C
PIN HDC 56 C
PIN LDC 60 C
PIN MODE 52 I
PIN PAD2 206 C
PIN PAD3 205 C
PIN PAD4 204 C
PIN PAD5 203 C
PIN PAD6 202 C
PIN PAD7 201 C
PIN PAD8 200 C
PIN PAD9 199 C
PIN PAD10 198 C
PIN PAD11 197 C
PIN PAD12 196 C
PIN PAD13 194 C
PIN PAD14 193 C
PIN PAD15 900 C
PIN PAD16 901 C
PIN PAD17 191 C
PIN PAD18 190 C
PIN PAD19 189 C
PIN PAD20 188 C
PIN PAD21 187 C
PIN PAD22 186 C
PIN PAD23 185 C
PIN PAD24 184 C
PIN PAD25 181 C
PIN PAD26 180 C
PIN PAD27 179 C
PIN PAD28 178 C
PIN PAD29 177 C
PIN PAD30 176 C
PIN PAD31 175 C
PIN PAD32 174 C
PIN PAD33 902 C
PIN PAD34 903 C
PIN PAD35 172 C
PIN PAD36 171 C
PIN PAD37 169 C
PIN PAD38 168 C
PIN PAD39 167 C
PIN PAD40 166 C
PIN PAD41 904 C
```

```
PIN PAD42 165 C
PIN PAD43 164 C
PIN PAD44 163 C
PIN PAD45 162 C
PIN PAD46 161 C
PIN PAD48 159 C
PIN PAD51 152 C
PIN PAD52 151 C
PIN PAD53 150 C
PIN PAD54 149 C
PIN PAD55 148 C
PIN PAD56 147 C
PIN PAD57 146 C
PIN PAD58 145 C
PIN PAD59 144 C
PIN PAD60 905 C
PIN PAD61 906 C
PIN PAD62 907 C
PIN PAD63 142 C
PIN PAD64 141 C
PIN PAD65 139 C
PIN PAD66 138 C
PIN PAD67 137 C
PIN PAD68 136 C
PIN PAD69 135 C
PIN PAD70 134 C
PIN PAD71 133 C
PIN PAD72 132 C
PIN PAD73 129 C
PIN PAD74 128 C
PIN PAD75 127 C
PIN PAD76 126 C
PIN PAD77 125 C
PIN PAD78 124 C
PIN PAD79 123 C
PIN PAD80 122 C
PIN PAD81 120 C
PIN PAD82 119 C
PIN PAD83 908 C
PIN PAD84 909 C
PIN PAD85 117 C
PIN PAD86 116 C
PIN PAD87 115 C
PIN PAD88 114 C
PIN PAD89 113 C
PIN PAD90 112 C
PIN PAD91 111 C
PIN PAD92 910 C
PIN PAD93 110 C
```

```
PIN PAD94 109 C
PIN PAD96 107 C
PIN PAD98 101 C
PIN PAD99 100 C
PIN PAD100 99 C
PIN PAD101 98 C
PIN PAD102 97 C
PIN PAD103 96 C
PIN PAD104 95 C
PIN PAD105 94 C
PIN PAD106 93 C
PIN PAD107 92 C
PIN PAD108 911 C
PIN PAD109 90 C
PIN PAD110 89 C
PIN PAD111 88 C
PIN PAD112 87 C
PIN PAD113 912 C
PIN PAD114 913 C
PIN PAD115 85 C
PIN PAD116 84 C
PIN PAD117 83 C
PIN PAD118 82 C
PIN PAD119 81 C
PIN PAD120 80 C
PIN PAD122 76 C
PIN PAD123 75 C
PIN PAD124 74 C
PIN PAD125 914 C
PIN PAD126 915 C
PIN PAD127 73 C
PIN PAD128 72 C
PIN PAD129 70 C
PIN PAD130 69 C
PIN PAD131 68 C
PIN PAD132 67 C
PIN PAD133 916 C
PIN PAD134 65 C
PIN PAD135 64 C
PIN PAD136 63 C
PIN PAD137 62 C
PIN PAD138 61 C
PIN PAD140 59 C
PIN PAD141 58 C
PIN PAD142 57 C
PIN PAD146 48 C
PIN PAD147 47 C
PIN PAD148 46 C
PIN PAD149 45 C
```

```
PIN PAD150 44 C
PIN PAD151 43 C
PIN PAD152 42 C
PIN PAD153 41 C
PIN PAD154 40 C
PIN PAD155 39 C
PIN PAD156 917 C
PIN PAD157 37 C
PIN PAD158 36 C
PIN PAD159 35 C
PIN PAD160 34 C
PIN PAD161 918 C
PIN PAD162 919 C
PIN PAD163 32 C
PIN PAD164 31 C
PIN PAD165 30 C
PIN PAD166 29 C
PIN PAD167 28 C
PIN PAD168 27 C
PIN PAD169 24 C
PIN PAD170 23 C
PIN PAD171 22 C
PIN PAD172 21 C
PIN PAD173 20 C
PIN PAD174 19 C
PIN PAD175 920 C
PIN PAD176 921 C
PIN PAD177 17 C
PIN PAD179 15 C
PIN PAD180 14 C
PIN PAD181 922 C
PIN PAD182 12 C
PIN PAD183 11 C
PIN PAD184 10 C
PIN PAD185 9 C
PIN PAD186 8 C
PIN PAD189 5 C
PIN PAD190 4 C
PIN PAD191 3 C
PIN PGCK1 2 C
PIN PGCK2 55 C
PIN PGCK3 108 C
PIN PGCK4 160 C
PIN PROGRAM 106 I
PIN SGCK1 207 C
PIN SGCK2 49 C
PIN SGCK3 102 C
PIN SGCK4_DOUT 54 C
PIN TCK 7 C
```

```
PIN TDI 6 C
PIN TDO 157 O
PIN TMS 16 C
END
```

With this edited version of the interface file, you can now use the smartccn tool to generate a new compiled configuration netlist file to use to simulate the 208-pin model, as explained in "Using Unsupported Devices Example" on page 101.

Browsing Your Design Using SmartBrowser

There are two different tools that you can use to browse your programmable logic design:

- Visual SmartBrowser (VSB)—a GUI tool that is documented in the *Visual SmartBrowser User's Manual* (UNIX version or NT version).
- SmartBrowser—a command-line tool that is documented in the following sections of this manual.

You can use either tool to read a compiled configuration netlist (.ccn) file and:

- Follow connectivity between all circuit objects
- List circuit objects
- Examine specific circuit objects
- Modify and save circuit properties
- Create window and monitor definitions
- Map illegal characters to valid strings
- Run command files and save log files

You might choose to use one tool or the other based on your preference for command-line tools like the SmartBrowser that can be used in batch mode with shell scripts or GUI tools like VSB that provide a more intuitive interface and superior visual display of design netlist information. You might also opt to issue SmartBrowser commands interactively in your simulator session, as described next, rather than using one of the browsing tools in a separate window.

Issuing SmartBrowser Commands Interactively

Most SmartBrowser commands can be issued through the SWIFT command channel. The command channel is a handy way to issue SmartBrowser commands interactively during your simulator session to any SmartCircuit model instance in your design. When you use the command channel you don't have to open a separate window and search for your design netlist before returning to your main task of probing and verifying a design in a simulator session. The ability to interact directly with any SmartCircuit model in your design is the primary reason to use the command channel rather than running SmartBrowser commands in a separate shell session. For general information on the command channel, refer to the *Simulator Configuration Guide for Synopsys Models*. The SmartBrowser commands, themselves, are documented in "SmartBrowser Command Reference" on page 111.

Note

Not all SmartBrowser commands can be issued through the command channel. For details on which commands are available for use with the command channel, refer to "SmartBrowser Command Reference" on page 111.

Using the SmartBrowser Tool in Standalone Mode

To invoke the SmartBrowser tool you need to specify the CCN file for the model that you want to examine. In addition, you can specify several other command switches that allow you to save a log file, run a command or log file in either interactive or batch mode, or view the SmartBrowser help file.

When you invoke the SmartBrowser tool, it looks in your home directory for an initialization file called .smartbrowse_rc and executes any commands in that file before doing anything else. The initialization file is an ASCII file that you create containing any SmartBrowser commands that you want to run. Initialization files are not required, but they are useful for tasks that you want performed each time you invoke the tool, such as defining command aliases.

For NT, invoke the SmartBrowser tool using the console command line. For more information, refer to "Running Console Applications on NT Platforms" on page 44.

Syntax

smartbrowse *ccn_filename* [-b] [-l *log_file*] [-m *model_name*] [-r *run_file*] [-re *log_file*] [-help]

Note

The *ccn_filename* must be the first argument on the command line, but switches can appear in any order.

Argument

ccn_filename	Use this required argument to specify the name of the compiled configuration netlist (.ccn) for the model.

Switches

-b	Starts the SmartBrowser tool in batch mode. Use this switch in conjunction with the -l, -r, and -re switches. When you invoke the SmartBrowser tool in batch mode, the tool runs silently and never enters interactive mode. Batch mode enables you to use the SmartBrowser tool in shell scripts.
-l *log_file*	Causes the SmartBrowser tool to write out a transcript of the SmartBrowser session to a file with the name specified by the *log_file* argument. You can also generate a log file by using the interactive log command.
-m *model_name*	Specifies the name of the model; required if the CCN file is in an old format, or if the SmartBrowser tool is invoked without the *ccn_filename* argument.
-r *run_file*	Causes the SmartBrowser tool to run the specified file before entering interactive mode. If you use this switch with the -b switch, the SmartBrowser tool will not enter interactive mode. All SmartBrowser interactive commands are legal in this file. Commands run in silent mode and are not echoed on the screen. To echo the commands, use the interactive run command.
-re *log_file*	Executes a log file before the tool enters interactive mode. As with the -r switch, if you also use the -b switch, the SmartBrowser tool will not enter interactive mode. All SmartBrowser interactive commands are

	legal in this file. Commands run in silent mode and are not echoed on the screen. To echo the commands, use the interactive rerun command. A log file differs from a run file, which you use with the -r switch, in that the log file contains output results in addition to commands and comments; a run file contains only commands and comments. In the log file, the executable commands are placed in square brackets (for example, [list nets]). When you run the log file using the -re switch only the commands in brackets are executed.
-help	Displays a help message for the SmartBrowser tool. You can also specify just -h, -he, or -hel and get the same output.

Using the SmartBrowser Tool to Create a Windows Definition File

One way to create a windows definition file is to use the interactive SmartBrowser commands "assign window" and "save mcf".

As when manually creating the windows definition file, you use the SmartBrowser list command to find the names of the elements you want windowed. Assign windows to the selected items, then use the "save mcf" command to save your definitions to a file. This file is subsequently referred to by a "do" command in the MCF file.

This section summarizes the SmartBrowser interactive commands that you can use for developing a windows definition file.

Regular assignments:

```
assign window   name(s) [= bus_name]
assign window   > bus_name
assign window instance
```

For AutoWindows:

```
set bus bitOrder big | little
set bus delimiter  [postfix_char]
assign window auto  # Find all window elements for current scope
```

Listing and saving all defined windows:

```
list mcf  # List all defined windows, monitors, and buses
set saveMcf noClobber | append | overwrite
save mcf  # Save all window, monitor, and bus definitions
```

The following SmartBrowser command creates three net windows definitions, one for each of the specified nets.

```
assign window net N4 N6 N8
```

Alternatively, you could use the following "assign window" command to bus the three nets together into a window element called foobus.

```
assign window net N4 N6 N8 = foobus
```

The next command illustrates how to create a bus and assign a window to a net that has a name containing illegal characters.

```
assign window state /DFF/STATE = fooState
```

Another way to create buses is to use the AutoWindows feature, as shown in the following example.

```
set bus bitOrder little # Set bus to little endian
set bus delimiter [ ]   # Bus numbering is between []s
assign window net data[0] > databus
```

This set of commands finds all nets of the form data[#] (with data[0] as the most significant) and buses them together to form a window element called databus.

Using SmartBrowser Commands

The SmartBrowser tool recognizes a large set of commands, all of which are described in "SmartBrowser Command Reference" on page 111. To make the command set easier to work with, the SmartBrowser tool supports abbreviation, aliasing, and automatic completion of interactive commands.

Scope of Commands

The output produced by many of the SmartBrowser commands depends on the amount of your design that is visible to the command; this view is referred to as the scope. Your current location is the current scope. As you traverse through a design, your current scope changes, as does the information that is displayed by commands.

At the top level of a design, the scope encompasses the entire design. As you travel deeper into the design, the scope becomes more and more focused. Some commands can traverse a design and produce reports that contain information outside the current scope, while other commands are limited to seeing only what is in the current scope.

Abbreviating Interactive Commands

You can abbreviate any interactive command, using the shortest string that uniquely identifies the command. In most cases that equals the first two characters of the command. For example, you can abbreviate the command "set scope" to "se sc".

Note

When using the SWIFT command channel to issue a SmartBrowser trace command, you must abbreviate the command to "tr".

Command Aliasing

The SmartBrowser tool supports command aliasing, which is similar to the command aliasing capability in UNIX. Aliasing enables you to provide shortcut names for SmartBrowser commands, combine multiple commands into a single alias name, define switch names for command keywords, and rename commands so that they are easier to remember.

Command Completion

Many of the SmartBrowser interactive commands require that you supply one or more keywords with the command, as well as an argument that identifies the element you want to work with. If you do not supply all the necessary keywords, or the argument, the SmartBrowser tool prompts you for the missing information. If you want to clear the command, press the Return key until you get the SmartBrowser command prompt back.

Command completion is particularly helpful if you cannot remember a command's syntax or if you type the command incorrectly. Note that command completion is disabled when using the SWIFT command channel.

SmartBrowser Command Reference

The following lists provide brief descriptions of the SmartBrowser commands. These commands enable you to:

- Obtain lists of design elements
- View, examine, and analyze designs, hierarchies, or cells
- Establish environment settings
- Save designs
- Perform other tasks related to your design

Although the SmartBrowser tool supports more than 60 commands, many of them are variations on a single command. For that reason, and to make the command set easier to understand, the command descriptions are divided into the following groups:

- "Analyze Commands" on page 112
- "Assign Commands" on page 112

Most of the SmartBrowser commands can also be issued through the SWIFT command channel, as noted on the following pages.

Analyze Commands

The following analyze commands can also be issued through the SWIFT command channel.

analyze cell *cell_class*
: Analyzes the specified cell for circuit errors.

analyze design
: Traverses a design hierarchy and performs an analyze cell function on every user-defined cell.

analyze hierarchy [*max_level*]
: Displays the instance hierarchy of the current instance scope and down. The *max_level* parameter specifies how many levels of hierarchy to trace.

Assign Commands

The following assign commands can also be used issued through the SWIFT command channel, except the assign timing command

assign monitor instance *name*
: Assigns a monitor to all nets attached to the specified instance.

assign monitor net *name1 . . . nameN* [= *bus_name*]
: Defines a monitor definition for the specified net. The optional = *bus_name* parameter lets you bus the specified elements together.

assign monitor net *name* > *bus_name*
: Defines a monitor for the specified net, looks for all elements related to the specified net, and buses the found elements together.

assign monitor state *name1 . . . nameN* [= *bus_name*]
: Defines a monitor definition for the specified state. The optional = *bus_name* parameter lets you bus the specified elements together.

assign monitor state *name > bus_name*
Defines a monitor for the specified state, looks for all elements related to the specified state, and buses the found elements together.

assign timing *timing_label* [*port_name*] *min_val* [*typ_val max_val*]
Modifies the min, typ, and max values of the specified timing label. If the timing label is port specific, you must specify the port name. If only the *min_val* is specified, then all values are set to *min_val*. Note that this command cannot be issued through the SWIFT command channel.

assign window auto
Automatically windows all nets in the current scope and all SCV states.

assign window instance *name*
Assigns a window to all nets attached to the specified instance.

assign window net *name1 . . . nameN* [= *bus_name*]
Defines a window definition for the specified net. The optional = *bus_name* parameter lets you bus the specified elements together.

assign window net *name > bus_name*
Defines a window for the specified net, looks for all related elements that could be buses to the specified net, and buses the found elements together.

assign window state *name1 . . . nameN* [= *bus_name*]
Defines a window definition for the specified net. The optional = *bus_name* parameter lets you bus the specified elements together.

assign window state *name > bus_name*
Defines a window for the specified net, looks for all related elements that could be buses to the specified net, and buses the found elements together.

Examine Commands

The following examine commands can also be issued through the SWIFT command channel.

examine instance *instance_name*
Displays detailed information about the specified instance and all instance-specific data.

examine net *net_name*
Displays details about the specified net.

examine port *port_name*
Displays details about the specified port attached to the current cell scope.

examine state *state_name*
Displays details about the specified state attached to the current cell scope.

examine timing *timing_label* [*port_name*]
Displays details about the specified timing label. If the timing label is port-specific, you must also include the port name.

List Commands

The following list commands can also be issued through the SWIFT command channel.

list all
Lists all instance, net, port, and state names and timing labels in the current cell scope.

list cells scvCells | userCells
Lists all vendor or user cell class names in a design.

list instances [*match_string*]
Lists all instance names defined in the current cell scope. The list can be constrained to only those instance names that match the *match_string*.

list mcf
Lists all monitors and windows defined during the current interactive session.

list nets [*match_string*]
Lists all net names defined in the current cell scope. The list can be constrained to only those net names that match the *match_string*.

list pinInterface
Lists all package pins and describes how those pins connect to the symbol pins and design ports.

list ports [*match_string*]
Lists all port names attached to the current cell scope. The list can be constrained to only those port names that match the *match_string*.

list states [*match_string*]
Lists all state names in an SCV cell of an instance hierarchy in the current cell scope and below. The list can be constrained to only those state names that match the *match_string*.

list timing
Lists all timing labels defined in the current cell scope.

Set and Show Commands

The following set and show commands can also be issued through the SWIFT command channel, except as noted.

set bus bitOrder big | little
Sets autoWindows bit order to big endian or little endian. The default is little endian.

show bus bitOrder
Displays the current bus bit ordering.

set bus delimiter *prefix* [*postfix*]
Defines the autowindows bus index delimiter. The default delimiter is square brackets ([and]).

show bus delimiter
Displays the current bus delimiter.

set help completion on | off
Toggles the interactive help completion capability. The default is on. (Cannot be issued through the SWIFT command channel.)

show help completion
Displays the current help completion setting. (Cannot be issued through the SWIFT command channel.)

set illegalchars *character*
Allows you to specify characters that may be illegal in their simulation environment. Replaces illegal characters with underscore characters (_).

show illegalchars
Displays the current illegal character settings.

show saveMcf
Displays the current setting for the "save mcf" file writing mode.

set listAll
Configures the default list command to list all elements.

set saveMcf noClobber | append | overwrite
Sets the writing mode for the "save mcf" command. The default mode is noClobber, which prevents an existing MCF file from being overwritten. Overwrite mode will create a new file, replacing an existing one by the same name. Append mode adds lines to the end of an existing MCF file.

set scope *instance_scope_name*
Enters the specified cell instance scope. The *instance_scope_name* argument may be either an absolute or a relative scope path. Use double dots (..) to enter the parent instance scope. The default scope is the top level of your design.

show scope
Displays the current instance scope level and path. Also displays information about the number of nets, ports, and instances.

set timing range min | typ | max
Establishes the timing range used for viewing instance-specific timing values. The default timing range is "min". (Cannot be issued through the SWIFT command channel.)

show timing range
Displays the current timing range setting. (Cannot be issued through the SWIFT command channel.)

set timing unit ps | ns | us | ms
Establishes the unit value used when viewing or modifying timing values. The default unit values is ps (picoseconds).

show timing unit
Displays the current value of the timing unit setting.

show doc
Displays all documentation for the current cell scope.

show version
Displays the version of the SmartBrowser tool.

Trace Commands

The following trace commands can also be issued through the SWIFT command channel. When used this way, SmartBrowser trace commands must be shortened to "tr" to prevent them from being misinterpreted as standard SmartModel command channel trace commands.

trace fin *instance_name* [*max_level*]

Traces and displays a connection tree of all inputs to the specified instance cell scope. The *max_level* parameter specifies how many levels of logic to trace. Here's an annotated example for a trace fin command issued for the TSBUF2 instance shown in Figure 12.

```
# In this example. the input port "IN" of TSBUF2 is connected to net "N8".

TSBUF2 (TRI) in: IN net: N8

# The previous net "N8" is connected to the output "Q" of the DFF instance,
which is driven by the "D" input.  The "D" input is connected to the net "N5".

# This notation (|<) indicates that the signal is connected to the net on the
previous level of hierarchy.

|<- DFF (DFF) out: Q in: D net: N5
|    |<- BUF3 (AND1) out: &1 in: &2 net: A2
|    | |<< <A2>
|<- DFF (DFF) out: Q in: CLK net: N6
|    |<- BUF4 (AND1) out: &1 in: &2 net: CLK
|    | |<< <CLK>
|<- DFF (DFF) out: Q in: CLRN net: VCC
|<- DFF (DFF) out: Q in: PRN net: VCC TSBUF2 (TRI) in: OE net: N3
|<- BUF1 (AND1) out: &1 in: &2 net: OE
|    |<< <OE>
```

trace fout *instance_name* [*max_level*]

Traces and displays a connection tree of all outputs from the specified instance cell scope. The *max_level* parameter specifies how many levels of logic to trace.

trace instances instance | net | port *name*

Reports all instances attached to the specified instance, net, or port.

trace nets instance | net | port *name*

Reports all nets attached to the specified instance, net, or port.

trace objs *name*

Traces all objects connected to the specified element, which may be an instance, net, or port.

trace pkgPin *package_pin_name*

Traces the *package_pin_name* to the related symbol in design port names.

trace ports instance | net | port *name*

Reports all ports attached to the specified instance, net, or port.

trace scvInstances instance | net | port *name*
Traces through the instance hierarchy until encountering an SCV cell, ultimately reporting all SCV cell instances connected to the specified element.

trace symbolPin *symbol_pin_name*
Traces the *symbol_pin_name* to the related package pin name and design port names.

trace topNet instance | net | port *name*
Finds the highest scope level net attached to the specified element.

General Commands

The following general commands can also be issued through the SWIFT command channel, except as noted.

alias *alias_name command1 . . . commandN*
Defines the specified *alias_name* for a command or list of commands.

unalias *alias_name*
Undefines the specified *alias_name*.

log *log_file*
Creates the specified log file. If another log file is open, the command closes that log file and then creates the new log file.

quit
Exits the SmartBrowser tool. (Cannot be issued through the SWIFT command channel.)

rerun *log_file*
Runs the specified *log_file* as though it is a command file, executing only statements within brackets (for example, [command]).

run *run_file*
Runs the specified SmartBrowser *run_file*.

save design *file_name*
Saves the current design, using the specified *file_name*. (Cannot be issued through the SWIFT command channel.)

save mcf *file_name*
Saves all monitors, windows, and buses defined during the current interactive session, using the specified *file_name*. You can use he "set saveMcf" command to configure the writing mode when saving to a file that already exists.

Model Command File (MCF) Reference

The primary function of an MCF file is to specify that the model load a compiled configuration netlist (CCN) or compile a source netlist. You can also use MCF files to include other command files, define bus names, and specify timing ranges. Files that you might want to include in the MCF are ones that perform basic tasks. For example, you can use an MCF file to include a standard setup file that creates monitors and windows for a specific family of devices. You can also use the MCF to maintain window element definitions either directly or by calling external files using the "do" command.

Additional MCF commands are available with the analysis tools; these commands report event causes and effects, monitor design elements, and define monitor report labels. A minimum MCF for a SmartCircuit model contains a load command, as shown below:

```
load -source filename
```

where *filename* is the name of the CCN file to be generated by smartccn.

The value of the SWIFT SCFFile parameter in your model instantiation determines the MCF file that a model reads at startup. To ensure that the model reads the correct MCF file, you can either edit the value of the SCFFile parameter to point to the appropriate MCF file for this model, or name the MCF file to match the current value of the SCFFile parameter.

Hint

When you use multiple configurable devices in a design, it is best to use MCF file names that are keyed to the model instance names. For example, the file name xc3030_u21.mcf uses both the model name and an instance specifier.

MCF Command Descriptions

Following are descriptions of the commands that you can use in an MCF file.

bus Command

bus *bus_name name1* [*name2 . . namen*]

Defines a bus alias for specified nets, states, external ports, or previously defined buses. Useful only with SmartModel Windows, SmartCircuit Monitors, and Causal Tracing reports.

bus_name
: The name of the bus alias.

name1
The name of a net, state, external port, or previously defined bus that is to be mapped to the bus_name alias. You must provide at least one name.

name2 . . namen
Optional additional names to be mapped to the bus_name alias.

The bus command must appear in the MCF file **after** the load command. In addition, you must define a bus name before you can use that bus name in another command.

do Command

do *filename*

Executes the file specified by *filename*. A do command can appear anywhere in an MCF file.

filename
The name of a script file to be executed by the model.

echo Command

echo *string*

Echoes the specified string to the simulation session transcript. An echo command can appear anywhere in an MCF file.

string
The string to be echoed to the simulation transcript window.

help Command

help

Displays a help message for the MCF file.

load Command

load [-source] *filename* [-nocheck] [-scale *factor*] [smartcnn switches]

Loads either a compiled configuration netlist (CCN) file (if the -source switch is not used) or a design source file (if the -source switch is used). The load command must appear in the MCF file after the set range command. You can specify the -source, the -nocheck, and the -scale switches on the same command line.

-source
Indicates that the model is to invoke smartccn to compile the design source file specified by *filename* and generate a CCN file, if necessary. The load command

supplies the smartccn compiler with the necessary model component and instance information. If a CCN file of the specified name already exists, the load -source command compares the date/time stamps of the source files against the date/time stamp of the CCN. The load -source command automatically recompiles the CCN for the model if one of the component files is newer than the existing CCN. No CCN compilation occurs if all files are up to date.

filename

Specifies the name of the CCN file to be loaded; or, if load is invoked with the -source switch, the name of the design source file to be compiled. The specified *filename* can include either an absolute or a relative path, though your simulator environment may restrict or limit the use of relative path names.

-nocheck

Disables reporting of timing constraint violations.

Hint

Compiling large JEDEC-based CPLD models can take a long time. If simulation initialization performance is important, compile the model separately, before simulating. Then use the load *design*.ccn command in your MCF file to picked up the compiled netlist.

-scale *factor*

Indicates that all timing values loaded from the CCN are to be multiplied by factor, which must be a positive, nonzero number. The -scale switch must follow -source and *filename*.

smartccn switches

Various smartccn compiler switches. The [smartccn switches] option allows you to specify command switches for the smartccn compiler.

set range Command

set range min | typ | max

Specifies the timing range (min, typ, or max) to be used from the CCN. The default value is set by the SWIFT DelayRange parameter from the model instantiation in the simulator environment. The set range command must appear in an MCF file before the load command. Using this command overrides the value set with DelayRange parameter.

window Command

window *name1* [*name2 . . . namen*]

Defines SmartModel Windows for specified nets, states, or bus aliases.

name1
The name of a net, state, or bus alias for which a window is to be created. You must provide at least one name.

name2 . . . namen
Optional additional nets, states, or bus aliases for which windows are to be created.

You can define windows for a design element from within the MCF file, either by placing the window command directly in the MCF (in which case it must be placed after the load command) or by creating a windows definition file and using the "do" command to include the file in your MCF.

Note
When you assign a window to an external I/O port, or to any net connected directly to an external I/O port, the window is placed on the input side of the port so that it can report the value being driven into the model. You cannot window the value driven out of an external I/O port.

On input or I/O ports that have special attributes (for example, pull-up resistors), the window command reports the resolved value of the simulator's input and the model's input pin attribute.

smartccn Command Reference

Before you can use a SmartCircuit model in a simulation, you must have a compiled configuration netlist (CCN). You can generate a CCN in two different ways:

- Use the load -source command in the model command file (MCF). This causes the model to automatically generate the CCN as needed.
- Run the SmartCircuit netlist compiler (smartccn) and then load the CCN using the load command in the MCF.

In most cases you should generate a CCN using the MCF rather than explicitly executing smartccn.

Note

Refer to the model datasheets for detailed information on configuring individual SmartCircuit models. For information on finding model datasheets using the sl_browser tool, refer to "SmartModel Datasheets" on page 25.

The smartccn tool translates a design netlist or JEDEC file, which describes the model's configuration, into a binary format that the SmartCircuit model understands.

During the compilation process, smartccn searches for the files it needs to create a CCN. The compiler also checks to see if a CCN already exists. If you invoke smartccn with the load -source command in the MCF, or if you invoke the tool in standalone mode with the -t switch, it compares the date and time stamps of the component files against the date and time stamp of the CCN. If one of the component files is newer than the existing CCN, smartccn automatically recompiles the CCN.

Like the models, some SmartModel tools can exist in the library in multiple versions. One such tool is smartccn. The correct version of smartccn to use is controlled by the model. You select only the model version. The version of smartccn is automatically determined based on the model version in effect when you invoke the tool. For information on selecting a model version, refer to "Selecting Models in $LMC_HOME" on page 44.

Syntax

Run the smartccn tool from the command line as shown in the following example. Be sure to enter the command, the required arguments, and any optional switches on a single line.

```
% smartccn -m model_name source_file [-switches]
```

Arguments

model_name — Specifies the model name for which you are compiling a netlist or JEDEC file.

source_file — Specifies the name of the netlist or JEDEC source file that smartccn is to read. Device manufacturers and some third-party vendors supply tools to produce netlists that are back-annotated for pin-package and timing information. Refer to the model datasheets to determine the *source_file* formats required by individual SmartCircuit models.

Switches

-m *model_name*	Specifies the name of the model to be used to simulate the netlist. The model must correspond to the targeted device in the netlist or, for JEDEC models, the device identified using the -c option.
-c *comp_name*	This switch is required for JEDEC models but has no effect for other model types. Use this switch to specify the name of the component to be used.
-g	Use this switch if you want the tool to generate an interface file named *model*.inf. You use an interface file to map new devices into an existing model.
-h	Specify this switch for help using the smartccn tool.
-i *interface_file*	Use this switch if you want smartccn to extract netlist-to-model mapping information from the specified interface file, rather than directly from the model.
-n *instance_name*	This switch can be used for JEDEC models only. It has no effect for other model types. You use this switch to specify that the .td timing data file for that model *instance_name* has been modified for use when compiling JEDEC files.
-o *output_file*	Causes smartccn to produce a CCN file named *output_file* rather than one named *source_file*.ccn.
-p	Causes smartccn to suppress generation of a CCN file and just create a pin map cross-reference file named *source_file*.pmp.
-q	Causes smartccn to suppress the generation of informational messages (but not warnings and errors).
-t	Causes smartccn to examine the time and date stamps of all component files and recompile the design if the existing CCN file is older.
-u *timing_file*	This switch can be used only for JEDEC models. It has no effect on other model types. You use this switch to specify use of an alternate .tf timing file.
-v	Returns the version of smartccn.

CCN Output Files

During the translation process, smartccn produces a compiled configuration netlist (CCN) file. By default, the CCN file has the same name as the netlist or JEDEC file that smartccn reads, but with the extension .ccn.

When you use the default file name extensions for your netlist or JEDEC file, smartccn replaces the extension with .ccn; but, when you use file name extensions other than the default, smartccn appends .ccn to the name you specified. The compiler places the output CCN file in the same directory as the netlist source file.

You can use the -o switch to specify an output CCN file name other than the default (*file*.ccn). This can be useful when you want to use a netlist or JEDEC file with a nondefault file name extension, but you want the CCN file to use the standard naming convention. The following example shows how to use the -o switch to produce a CCN file with a different name:

```
% smartccn sample.edo -m model_name -o sample2.ccn
```

ccn_report Command Reference

You can use the ccn_report tool to generate model reports based on information in the model's CCN file. To generate a report on a particular model, enter the model file name on the command line after the ccn_report invocation. You can optionally specify different switches depending on the information you need. Here is the syntax for using the ccn_report tool.

Syntax

```
ccn_report ccn_filename  { -A1 window_filename
                         | -A2 window_filename
                         | -A3 window_filename
                         | -A4 window_filename
                         | -h
                         | -i  illegal_chars
                         | -m  model_name
                         | -nr
                         | -p pinmap_filename
                         | -r replacement_char
                         | -v
                             [-n]
                             [-mn module_name]
                             [-o verilog_filename]
                             [-w window_filename]
                             [-y verilog_path]
                         | -vl
                             [-n]
                             [-o symbol_filename]  } output_filename
```

Argument

The ccn_report tool takes one required argument, as follows.

ccn_filename	Specifies the name of the CCN file to be used to generate the report.
output_filename	Specifies the name of the output file for the report.

Switches

There are many switches that you can optionally specify, as follows.

-A1 *window_file*	Uses the AutoWindows feature to generate a windows definition file with the specified name, using square brackets ([]) as the bus index delimiters.
-A2 *window_file*	Uses the AutoWindows feature to generate a windows definition file with the specified name, using angle brackets (<>) as the bus index delimiters.
-A3 *window_file*	Uses the AutoWindows feature to generate a windows definition file with the specified name, using parentheses (()) as the bus index delimiters.

-A4 *window_file*	Uses the AutoWindows feature to generate a windows definition file with the specified name, using trailing numbers as the bus index delimiters.
-h	Invokes the ccn_report help message.
-i *illegal_chars*	Defines the set of characters to be replaced during AutoWindows generation. The default illegal character is the slash (/).
-m *model_name*	Specifies the name of the model. This is required if the CCN file is in an old format, or if ccn_report is invoked without the *ccn_filename* argument.
-nr	Directs ccn_report to not replace any illegal characters. Overrides the -i switch.
-p *pinmap_file*	Directs ccn_report to produce a pin map file with the specified name.
-r *replacement_char*	Specifies a character to be used to replace illegal characters. The default is the underscore (_).
-v	Directs ccn_report to create a Verilog-XL module file.
-n	Renames the default port names to the design port names. This switch is used only with the -v and -vl switches.
-mn *module_name*	Renames the generated Verilog-XL module name to the specified name. This switch is used only with the -v switch.
-o *verilog_filename*	Renames the ccn_report output file to the specified name. This switch is used only with the -v switch.
-w *window_filename*	Directs ccn_report to generate a modified *model*.v file that contains the AutoWindows definitions. This switch is used only with the -v switch.
-y *verilog_path*	The path name to the *model*.v file. Allowed values are $LMC_HOME/special/cds/verilog/historic and $LMC_HOME/special/cds/verilog/swift (the default). This switch is used only with the -v switch.
-vl	Directs ccn_report to create a ViewLogic symbol file.
-n	Renames the default port names to the design port names. This switch is used only with the -v and -vl switches.

-o *symbol_filename*	Renames the ccn_report output file to the specified name. This switch is used only with the -v and -vl switches.

AutoWindows

Using a feature known as AutoWindows, ccn_report can automatically generate a SmartModel Windows definition file. You use AutoWindows by specifying one of the ccn_report options (-A1 through -A4). You can then include the resulting definition file in a model command file (MCF) using the "do" MCF command. An AutoWindows report lists all nets and states found in the design and then buses together any signals that follow the bus index delimiter rule selected by one of the -A1 through -A4 options.

When you recompile your design using your vendor tools, you must ensure that your windows definitions correspond to the elements in your new compiled configuration netlist (CCN file).

Note

Windowing all of your design elements using AutoWindows significantly degrades simulator performance. For information on more efficient ways to monitor individual design elements, refer to "Viewing Internal Nodes During Simulation" on page 95.

7
Processor Models

Configuring Processor Models

There are two basic types of processor models: full-functional and bus-functional. Full-functional models (FFMs) execute instructions loaded in the memory models within the design being simulated. Bus-functional models (BFMs) execute commands from an external source. A further distinction is that there are two different technologies used to develop BFMs. One is Hardware Verification (HV) models, which you control using an external program written in Processor Control Language (PCL). The other is FlexModels, which you can control using VHDL, Verilog, or C. BFMs represent device behavior by simulating bus cycles, rather than by executing assembly language instructions. Figure 14 illustrates the data flow for configuring FFM and HV processor models. For information on using FlexModels, refer to the *FlexModel User's Manual*.

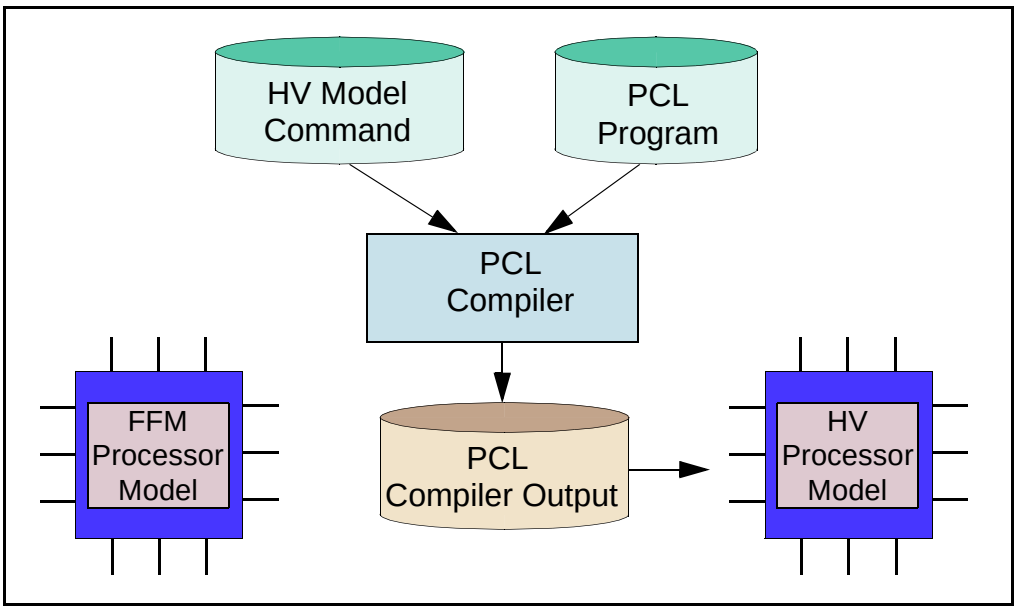


Figure 14: Data Flow for Processor Models

FFMs read programs stored in memory models, whereas HV models read compiled PCL files. Both types of processor models are useful for developing, debugging, and optimizing digital systems at different stages of the development cycle. During early development phases, when change is frequent and turnaround time critical, HV models are particularly useful because they are easy to use and run faster in simulation. An HV model's ability to verify proper handling of any combination of bus cycles is especially convenient. Towards the end of a design cycle, when a hardware design is more stable or the software must be verified, FFMs (or a hardware modeler) are essential. Table 9 compares the features of FFM and HV models.

Table 9: Comparison of HV and Full-Functional Processor Models

Model Feature	Hardware Verification	Full-Functional
Correct timing	Yes	Yes
Functionally correct pin behavior	Yes	Yes
Functionally correct bus behavior	Yes	Yes
Correct simulation of response to external interrupts	Yes	Yes
Full bus functionality (requests and grants)	Yes	Yes
High-level commands read from PCL file and executed	Yes	**No**
Machine code instructions fetched and executed	**No**	Yes

Simulating with HV Models

HV models are easy to use when debugging a hardware design. For example, if you wanted to run a simulation to verify a processor/memory interface with a Motorola MC68020 using an HV model, you would follow these steps:

1. Refer to the SmartModel datasheet for details about the PCL commands supported by the mc68020 model and write a PCL program to exercise the model functions that you want to verify in your design.

Verifying the basic interface requires only a couple of MC68020 bus cycles to indicate any obvious errors in the microprocessor/memory logic interface. To test this functionality, you could write a simple PCL program that looks like this:

```
/* check memory infc */

#include "mc68020.cmd"
main ()
{
      write (5,0x620,4,0xFFFF);
      read (5,0x624,4);
}
```

2. Use the compile_pcl command to compile the PCL program and create the PCL code the microprocessor model will load.
3. Run the simulation.

Hint

Of course, you can make your PCL program as extensive as you want, depending on your verification requirements. Check the individual model datasheet for a PCL program sample that exercises the model's basic functions, including an interrupt handler. You can cut-and-paste the program from the datasheet and then modify as needed.

PCL File Checks

When you initialize an HV model for logic simulation, the model looks for the compiled PCL program file specified by the SWIFT PCLFile parameter for that model instance. If the HV model cannot find the necessary PCL file, it issues a warning message. Similarly, if you use a memory model to store a program to drive a full-functional processor model, the memory model looks for the memory image file specified by the SWIFT MemoryFile parameter for that model instance. After successful initialization, the model generates a message that looks like the following example:

```
NOTE: Loading the PCL program from file "pclfile".
@i80386_hv/1:TESTMODEL(#189) [I80386-12], at 0 ns
```

Processor Control Language (PCL)

HV models represent system behavior by simulating external bus cycles directly rather than simulating the internal processing that leads to the assembly language instructions. You control the actions of an HV model by writing a PCL program that specifies how you want the model to respond to input stimulus, including interrupts. Each HV model supports its own set of model-specific PCL commands that implement the specific capabilities of the modeled device. HV model datasheets provide comprehensive information on the how to use the specific PCL commands supported by a particular model. In addition, the code definitions for all model-specific PCL commands and some handy defines are contained in the model's command header file (.cmd). This file must be included in your PCL source file using the #include preprocessor statement.

In form and structure, PCL looks very much like the C programming language, but despite the obvious similarities there are important differences. Do not assume that a PCL program will work just like a comparable C program. PCL includes the following features:

- A preprocessor with a limited set of directives that allows for the definition of constants and macros, as well as the inclusion of files
- Variables and constants (data types are limited to integers, arrays of integers, and pointers to integers)
- User-defined functions
- Program control statements, including loops and conditional logic
- Arithmetic and logical operators

Using PCL to Configure HV Models

The following procedure describes the basic steps required to configure an HV model for logic simulation:

1. Write a PCL program that directs the HV model to perform the desired set of operations.
2. Use the #include statement in the PCL program's Declarations section to specify the model's command header file (*model*.cmd).
3. Compile your PCL source code using the compile_pcl utility.
4. Configure your model instance to use the compiled PCL program via the SWIFT PCLfile parameter or symbol property.

PCL Program Structure

All PCL programs have a basic structure in common. In addition, some HV models require specific additions to this basic structure. In these cases, the HV model's datasheet provides detailed information about these additional structural requirements. This section provides information about the basic PCL program structure that is common to all HV models.

Every PCL program must contain a function named "main". Program execution begins with the first statement in the main function.

PCL Program Structure for Single-Process Models

Figure 15 shows the basic parts of a PCL program:

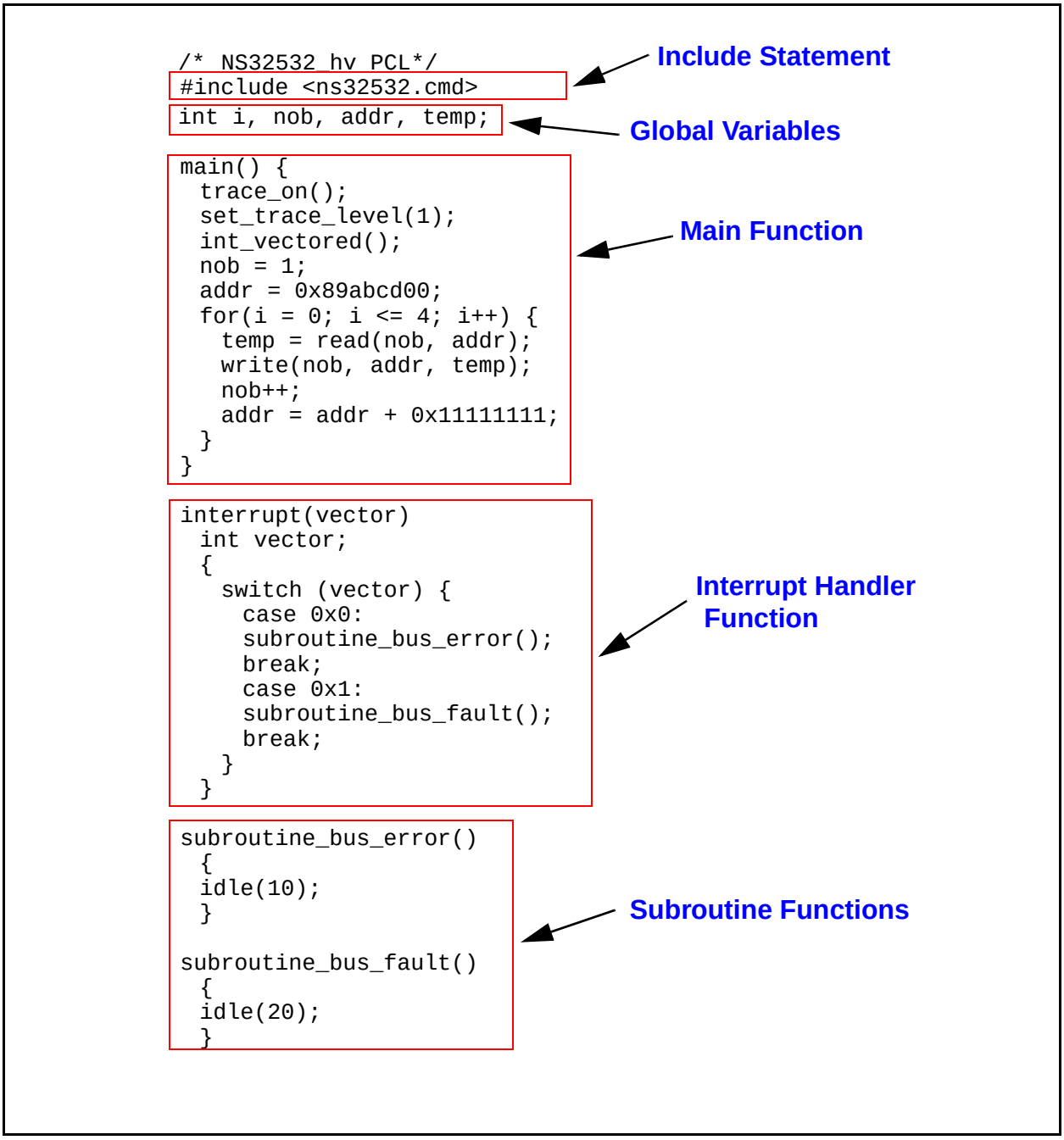


Figure 15: PCL Program Format Example

PCL Program Structure for Multiprocess Models

Some HV models use multiple process streams that execute concurrently. PCL programs for multiprocess models are very similar to those for single-process models. In addition to a "main" function, multiprocess models require two or more functions named processN (where N is the process number). Whereas single-process models have one interrupt handler, multiprocess models may have a separate handler for each process.

In single-process PCL programs the main function usually does the work of generating bus cycles, while in multiprocess PCL programs, the main function is reserved for initializing the model and handling variables. In multiprocess PCL programs the process functions generate bus cycles.

Interrupts and Exceptions

HV models do not have built-in interrupt or exception handlers. You must write an interrupt handler in your PCL program if you want your HV model to respond to interrupts in simulation.

The interrupt handler function must be named either interrupt or exception. In PCL these two function names are synonymous—one function will suffice to handle both types of events. When an HV model detects an interrupt, it generates a vector number. The PCL interpreter calls your interrupt handler function and passes the vector number as the only argument.

A typical interrupt handler function consists of a switch statement that evaluates the vector number and passes control to an appropriate case-labeled statement block, as shown in the following example:

```
exception(which)
int which;
{
  switch(which)
       {
            case 1 :    printf("Executing exception routine #1");
                                break;

            case 2 :    /* next exception handler routine */
            . . .
            case n :    /* last exception handler routine     */

            default:    break;
       }
}
```

At the end of the exception handler function, control resumes at the interrupted statement. Unless the model datasheet specifies otherwise, interrupt handlers may themselves be interrupted.

The Command Header File

Each HV model supplies its own set of PCL commands to simulate the operations or capabilities of a device. For example, most models have commands to generate various types of bus cycles and set the amount of trace information the model displays. These predefined commands are documented in the model's datasheet, and are also contained in the model's command header file.

Synopsys provides a command header file (also referred to as a ".cmd file"), *model*.cmd, which contains preprocessor definitions and predefined PCL commands for the model. You must include the command header file in your PCL program using the #include preprocessor directive; this must be the first statement in the PCL program. This allows the program to access the model's predefined commands and preprocessor definitions

The following syntax includes the command header file and causes the PCL program to obtain the specified file from its default location, the $LMC_HOME/models/*model* directory.

```
#include <i80386.cmd>
```

The following syntax includes the command header file and causes the PCL program to search for the file first in the local directory and then in the default location.

```
#include "i80386.cmd"
```

Note

If you want to define additional commands not in the command header file, you can create your own include file. The #include statement for this file can appear anywhere in the PCL program.

Returned Values

Some model-specific PCL commands (for example, "read") return one or more values to the PCL program. You can use commands that return a single value in the same way you use a function that returns an integer. You can also access a single returned value as the first element of the predefined retval array. However, when a command returns multiple values, you must access the values from the retval array.

The first value returned by a PCL command is placed in retval[0], the second in retval[1], and so on. To save returned values for later use, you must assign them to variables, because the retval[] array is overwritten by the next PCL command that returns values to the program.

Accessing the First Returned Value

When a command returns one or more values, PCL uses the first returned value as the command's exit value. Consequently, you can access the first returned value from a command in either of two ways:

```
read_d ( 0x03, 0x12341100, WORD );
read_value = retval[0];
```

Or, you can use the following:

```
read_value = read_d ( 0x03, 0x12341100, WORD );
```

Either method is appropriate to obtain the first value. All other returned values must be accessed through the retval[] array.

Unknown Values

In many simulation environments, an "X" or some similar character in place of a numerical value signifies that one or more bits of the digit are unknown. PCL cannot interpret unknown values, and so converts them to zeros without issuing any warnings or messages. If an HV model returns an unknown value to its PCL program, errors may occur.

The impact of this conversion of unknowns to zeros depends on the representation of the returned integer and on a given simulator's mechanisms for handling unknowns. In the following example, the ninth bit of a returned binary value is unknown. If this is translated into hexadecimal by the simulator, the hexadecimal digit containing the unknown becomes unknown, as follows:

```
0000 0000 0000 0000 0000 100X 0000 0001        Binary
 0    0    0    0    0    X    0    1          Hexadecimal
 0    0    0    0    0    0    0    1          PCL conversion
```

The PCL program converts the hexadecimal value of X to 0, so the number in hexadecimal becomes 0x001. Similarly, if the simulation environment represents this value in octal, the octal digit containing the unknown becomes unknown, as follows:

```
00 000 000 000 000 000 000 100 X00 000 001     Binary
 0  0   0   0   0   0   0   1   X   0   1      Octal
 0  0   0   0   0   0   0   1   0   0   1      PCl conversion
```

The PCL program converts the octal value of X to 0, so the number in octal becomes 04001.

PCL Constructs

PCL syntactical constructs include the following:

- Identifiers
- Data types
- Variables
- Constants
- Comments
- Operators
- Expressions
- Statements
- Function definitions

Following are details on how to use each of these constructs in PCL programs.

Identifiers

In PCL, the names that reference variables, named constants, macros, and functions are called identifiers. An identifier can be any sequence of alphanumeric characters and underscores, but must begin with an alphabetic character. The PCL compiler is case-sensitive; the identifiers "ABC" and "abc" are distinct.

Keywords are reserved by the PCL compiler for statements, data types, and other elements of the language. Using a keyword as an identifier causes a syntax error; however, note that the PCL keywords are defined to be in lower case. Table 10 lists the PCL keywords.

Table 10: PCL Keywords

`break`	`case`	`continue`
`default`	`do`	`else`
`for`	`if`	`int`
`return`	`switch`	`while`

Data Types

The only valid data types in PCL are integers, arrays of integers, and pointers to integers. All integer values are 32-bit signed numbers.

Variables

A variable is a named value that can be changed within a program. Variables must be declared before they can be used. The general form of a variable declaration is:

```
int variable_list;
```

The three different data types (integers, integer arrays, pointers to integers) are distinguished by the way the variable identifier is specified. A variable list is a list of identifiers, separated by commas (,). A variable declaration statement must end with a semicolon (;).

The following example shows how to declare three different integer variables.

```
int varname1, varname2, varname3;
```

PCL supports only one-dimensional arrays. You can declare an array by specifying a variable identifier, followed by the size of the array in brackets ([]). The array size must be an integer literal. All array indexes range from 0 to n-1, where n is the number of elements. You cannot initialize array variables in the declaration. The following example shows a variable declaration for array variables.

```
int array1[100], array2[10];
```

You can declare a pointer variable by putting an asterisk (*) before the identifier in the declaration, as shown in the following example:

```
int *ptr1;
```

Constants

A constant is a fixed value that cannot be changed by the program. Because of its limited data types, PCL supports only literal constants and not named constants. However, you can still create a named constant by using the #define directive.

You can specify a literal constant in decimal, octal, or hexadecimal format. Octal numbers begin with a "0" while hexadecimal numbers begin with "0x" or "0X."

The following example code shows various literal constants used in a fragment of PCL code:

```
if ( var1 == 10 )
  var2 = 0x64;     /* hex for 100 decimal */
else
  var3 = 0144;     /* octal for 100 decimal */
```

Comments

Comments in PCL begin with the string /* and end with */. Spaces are not allowed between the asterisk and slash in either of these strings. The compiler ignores any characters between the /* and */ strings.

Comments can be placed anywhere within a program, as long as they do not break a keyword or identifier. Nested comments (a comment containing another comment) are not allowed.

Operators

PCL supports the integer and logical operators shown in Table 11. Logical values in PCL are represented by integers, and so are treated as integer values by the compiler.

Table 11: PCL Operators

Operator	Description
!	Logical NOT
~	Bitwise complement
+	Addition
-	Subtraction, negation
*	Multiplication, indirection
/	Division
%	Remainder
<<	Left shift
>>	Right shift
<	Less than
<=	Less than or equal to
>	Greater than
>=	Greater than or equal to
==	Equal
!=	Not equal
&	Bitwise AND, address of
|	Bitwise inclusive OR

Table 11: PCL Operators (Continued)

Operator	Description
^	Bitwise exclusive OR
&&	Logical AND
||	Logical OR
'	Sequential evaluation
?:	Conditional
++	Increment
--	Decrement
=	Assignment
+=	Addition assignment
-=	Subtraction assignment
*=	Multiplication assignment
/=	Division assignment
%=	Remainder assignment
>>=	Right shift assignment
<<=	Left shift assignment
&=	Bitwise AND assignment
|=	Bitwise inclusive OR assignment
^=	Bitwise exclusive OR assignment

Operator Precedence and Associativity

Operators nearer the top of Table 12 have precedence over those placed lower in the table. Operators that share the same precedence are placed on the same row.

Table 12: PCL Operator Precedence and Associativity

PCL Operators	Associativity
() []	Left to right
++ -- - ~ ! * & +	**Right to left**
* / %	Left to right
+ -	Left to right
<< >>	Left to right
< > >= <=	Left to right
== !=	Left to right
&	Left to right
^	Left to right
|	Left to right
&&	Left to right
||	Left to right
?:	**Right to left**
= += -= *= /= %= &= |= ^= >= <=	**Right to left**
,	Left to right

Note

Operators are evaluated from left to right, except for those in the 2nd, 13th, and 14th rows.

Expressions

An expression in PCL is any syntactically correct combination of operators, constants, variables, and function calls. An expression must always evaluate to a value. Because a constant or a variable evaluates to a value, an expression can be as simple as a single constant.

The PCL compiler does not support the Boolean data type. In PCL logical expressions, a value of zero equates to FALSE and any nonzero value equates to TRUE.

Typically, an expression is used as one or more elements in a PCL statement. Expressions are most often used in assignment statements and in program control statements to determine entry and exit conditions.

Functions

Function definitions must conform to the following syntax:

[*return_type*] *function_name* ([*parameters*])
parameter declarations
{
local_variable_declarations
function_code
}

You can define functions before or after they are called. The only valid return type is int, which is also the default. Any other return type specified (for example, void, int *, char, char *) causes the PCL compiler to generate an error message.

Parameters are optional, but the pair of parentheses following the function name is required. Parameters are assumed to be of type int.

The printf() Function

The PCL compiler supports the built-in printf() function. The syntax is as follows:

printf (*format* [, *data_args*]);

The printf() function formats the data arguments according to the format specifications and writes the result to the simulator as a "note" message. The function supports a maximum of nine data arguments. PCL automatically handles declarations for this built-in function, so there is no printf() header file to include in your PCL program.

The format argument can contain plain characters, which printf() copies to the stream, and conversion specifications. Each conversion specification starts with a % character and ends with an alphabetic character determining the type of conversion. Between the % and terminating character, you can use the two modifiers shown in Table 13.

Table 13: Conversion Specification Modifiers

Modifier	Definition
-	Left-justify the argument to be converted
N	Pad the field to this width (N characters)

The printf() function recognizes the argument conversion types shown in Table 14.

Table 14: Argument Conversion Types

Modifier	Definition
d	Decimal integer
o	Unsigned octal number
x	Unsigned hexadecimal number
%	Print a % (no conversion)

Each conversion specification (excluding %%) must have a corresponding argument.

Because the following example has no conversion specification, the quoted string prints directly to the output.

```
printf ( "Print this string" );
```

In the next example, the value of the *exit_value* variable prints as a decimal integer at the specified point in the error message:

```
printf ( "Error %d has occurred", exit_value );
```

Preprocessor Directives

The PCL compiler includes a preprocessor, which recognizes these directives:

```
#define
#else
#endif
#if
#ifdef
#ifndef
#include
#undef
```

A preprocessor directive must be the first item on a line.

The #define and #undef Directives

The #define directive lets you define named constants and macros. Macro names can also include arguments to be passed into the macro body and operated on. The syntax is as follows:

#define *macro_name* (*argument* [, *argument*]) *macro_body*

Here is an example of the #define directive used for string substitution in the i80960sx_hv hardware verification model.

```
#define ONE_BYTE 1
```

The definition lets you specify the string "ONE_BYTE" for data transfer, rather than specifying its numerical value for the cycle type.

After a macro is defined, it retains its meaning until the end of the source file or until you explicitly undefine it with the #undef directive. You must undefine a macro before you can redefine it. Undefine the macro by placing an #undef directive between the two #define directives.

The syntax for the #undef directive is as follows:

#undef *macro_name*

The #include Directive

The #include directive lets you include the contents of other source files into the current source file. By using the #include directive you can place frequently used constants or functions into smaller, more easily maintained source files and include them in your main PCL program file only when necessary.

When used to include the command header (.cmd) file, the #include directive must be the first statement in a PCL program. Other #include statements can be anywhere in the file.

The #include directive syntax has two possible forms. To instruct the preprocessor to search for the specified file in the directory that is appropriate for your system, enclose the file name in angle brackets (<>), as follows:

```
#include <filename>
```

To instruct the preprocessor to search for the file in the directory where the PCL source file resides, enclose the file name in double quotes (" "), as follows:

```
#include "filename"
```

You must use either angle brackets or double quotes.

PCL Statement Types

In PCL, a statement is a discrete unit of programming code that conforms to the syntactical rules for the language. A PCL statement can be one of the following types:

- Null statement
- Assignment statement
- Compound statement
- Nested statement
- Program control statement

In addition, PCL uses a special type of statement called a predefined model command (also referred to as a model-specific command). Predefined model commands are specific to a particular model; that is, they implement the specific capabilities of the modeled device. For general information, see the command header file. For information about specific predefined model commands, refer to the model's online datasheet.

Null Statements

A null statement is an empty statement that contains no instruction or command to execute. A null statement has no effect other than to introduce an unknown amount of delay. You specify a null statement with a semicolon (;).

The following example shows the use of a null statement.

```
if ( a <= b)
       ;   /* null statement */
else
       c = a * 10;
```

Assignment Statements

An assignment statement assigns a value to a variable. The syntax for an assignment statement is as follows:

```
variable_identifier = expression;
```

Compound Statements

PCL allows the use of compound statements, which let you use a block of statements where a single statement is expected. You can declare local variables at the beginning of a compound statement. The syntax for a compound statement is as follows:

```
{
        [optional local variable declarations ]
        [statements ]
}
```

Nested Statements

PCL allows the use of nested statements. Nesting statements is useful for controlling program execution flow. Anywhere the syntax indicates the use of a statement, you can insert any valid PCL statement.

One common example is a nested if statement:

```
if (x == 0){
        if (y == 0)
                statement;}
```

Nested statements can get complex. The basic rule is that execution of a statement's syntax must complete before moving to the next statement. Therefore, if you write a nested if statement such as that shown below, the else clause belongs to the innermost if statement, not to the outermost one.

```
if (x == 0){
        if (y == 0)
                statement;
        else
                statement;}
```

Note that if you want to isolate the second if statement and have the else clause apply to the first if statement, you have to treat the second if statement as a compound statement, surrounding it with braces.

Although this discussion focuses on nesting if statements, nesting of other types of statements is just as valid and just as necessary to accomplish certain programming tasks. Basically, PCL follows the conventions of the C programming language for nesting statements.

PCL Program Control Statements

PCL program control statements specify or determine the next statement in the program to evaluate. The program control statements available in PCL are:

- break
- continue
- do
- for
- if
- return
- switch
- while

The break Statement

The break statement terminates execution of the most recent enclosing while, do, for, or switch statement. Control passes to the first statement following the terminated while, do, for, or switch statement.

```
break;
```

The continue Statement

The continue statement passes control to the next iteration of a do, for, or while loop.

```
continue;
```

The remaining statements in the current iteration of the loop body are not executed.

The do Statement

The do statement is a loop that executes the associated statements once, and then evaluates an expression to determine whether to continue or exit the loop. The syntax is:

```
do
   statements
while (expression);
```

If the expression evaluates to true (a nonzero value), the loop statements are executed again; otherwise, program flow continues to the next statement following the loop. Note that the statements inside the body of a do loop execute at least once.

The for Statement

A for loop executes its associated statement the number of times specified by the entry expressions. The syntax is:

```
for (init_expr; cond_expr; loop_expr)
        statements;
```

The *init_expr* is generally used to set an initial value for the loop control variable. The *cond_expr* is a relational expression that determines how many times the loop is executed. The *loop_expr* defines how the loop control variable changes each time the loop is repeated.

The following example shows a for loop used to initialize array element values:

```
for (i=0; i<=9; i++)
        sample_array[i] = 0;
```

The if Statement

The if statement allows you to execute a block of code based on a condition. The syntax is:

```
if (expression) statement1 [ else statement2 ];
```

If the expression evaluates to true (a nonzero value), then *statement1* executes. The optional else clause can be used to specify a default action; that is, if the expression is false (equal to zero), then execute *statement2* rather than *statement1*.

The return Statement

The return statement terminates the execution of a function.

```
return [ expression ];
```

If specified, the value of *expression* is returned to the calling function. If *expression* is omitted, the return value is undefined.

The switch Statement

The switch statement is a multiple-branch decision statement. The syntax is:

```
switch (expression)
{
        case constant1:
              statements
              break;

        case constant2:
              statements
              break;
         . . .
        default:  statements
};
```

The value of *expression* is successively checked against a list of case constants. Program flow is transferred to the statement sequence whose case constant matches the value of the switch *expression*. If no case constant is equal to the value of the switch expression, the default *statements* are executed. If there is no default case, then none of the statements in the switch body are executed.

Execution begins at the selected statement and continues until a break statement is executed or the end of the compound statement is reached. If you do not end a case block with a break statement, program control "falls through" to any following statements.

There are two important rules to remember about switch statements:

- A switch statement can only test the expression for equality.
- No two case constants in the same switch statement can have identical values. You can, however, nest switch statements.

To associate a statement sequence with more than one case constant, omit the break statements between the case constants. The following example shows that case constants 1 through 3 are associated with the statement sequence that immediately follows them. Case constant 4 is associated with the statement sequence that immediately follows it. The syntax is:

```
switch (expression)
{
        case constant1:
        case constant2:
        case constant3:
              statements
              break;
        case constant4:
              statements
              break;
        default:
              statements
};
```

The while Statement

A while loop executes its associated statements zero or more times, based on the value of the entry expression. The syntax is:

```
while (expression)
        statement;
```

If the entry *expression* is true (evaluates to a nonzero value), the loop is executed. If the entry *expression* is false (evaluates to zero), the loop is skipped and program control passes to the next statement following the while loop.

Debugging Designs with Trace Messages

HV models generate trace messages to help you debug PCL programs and troubleshoot your circuit designs. Some HV models feature an adjustable detail level for trace messages, which lets you control the number and kind of messages generated by the model. Refer to the individual model datasheets for information on controlling message verbosity.

Here are some sample trace messages:

```
Trace: PCL Cmd: trace_on().
  (n=u9) (comp=C010) (loc=?) (lai=TMS370C010), at t=10001 (1000.1 ns).

Trace: PCL Cmd: enable_intr_level(0).
  Level 1 and 2 interrupts now enabled.
  (n=u9) (comp=C010) (loc=?) (lai=TMS370C010), at t=10001 (1000.1 ns).

Trace: PCL Cmd: set_trace_level(3).
  Trace level is now set to 3 (SPI + general messages).
  (n=u9) (comp=C010) (loc=?) (lai=TMS370C010), at t=10001 (1000.1 ns).

Trace: PCL Cmd: load_reg(0x1030,Ox09).
```

```
  (n=u9) (comp=C010) (loc=?) (lai=TMS370C010), at t=10001 (1000.1 ns).

Trace: PCL Cmd: load_reg(0x1031,Ox07).
  (n=u9) (comp=C010) (loc=?) (lai=TMS370C010), at t=10001 (1000.1 ns).

Trace: PCL Cmd: load_reg(0x1037,Ox00).
  SPIBUF is read only, command ignored.
  (n=u9) (comp=C010) (loc=?) (lai=TMS370C010), at t=10001 (1000.1 ns).

Trace: PCL Cmd: load_reg(0x1039,OxA0).
  (n=u9) (comp=C010) (loc=?) (lai=TMS370C010), at t=10001 (1000.1 ns).

Trace: PCL Cmd: load_reg(0x103D,Ox02).
  (n=u9) (comp=C010) (loc=?) (lai=TMS370C010), at t=10001 (1000.1 ns).

Trace: PCL Cmd: load_reg(0x103E,Ox22).
  (n=u9) (comp=C010) (loc=?) (lai=TMS370C010), at t=10001 (1000.1 ns).

Trace: PCL Cmd: load_reg(0x103F,Ox40).
  (n=u9) (comp=C010) (loc=?) (lai=TMS370C010), at t=10001 (1000.1 ns).

Trace: End of main PCL program.
  (n=u9) (comp=C010) (loc=?) (lai=TMS370C010), at t=10001 (1000.1 ns).
```

Running the PCL Compiler

In addition to compiling a PCL source file into a PCL program, the PCL compiler checks for errors and issues error messages specifying the location and nature of any errors it detects. You invoke the PCL compiler using compile_pcl.

For NT, invoke the compile_pcl program using the console command line. For more information, refer to "Running Console Applications on NT Platforms" on page 44.

Syntax

```
% compile_pcl source_file new_program [switches]
```

Arguments

source_file	Specify the name of the PCL source file that you want to compile. If '-', the compiler reads the source file from STDIN.
new_program	Specify the name for compiled program. If '-', the compiler writes the object file to STDOUT.

Switches

-C	Write source file comments to output.
-D=	Define a symbol with the given (optional) value.
-H	Display a help message and exit.
-I	Add a directory to the #include search list.
-M	Specify the name of the model being compiled.
-N	Do not predefine target-specific names.
-Stext	Specify sizes for #if sizeof.
-Usymbol	Undefine symbol.
-Xvalue	Set internal debug flag.

Example

The following example compiles the *MySrc.pl* program into object code and puts the output in the *MyObj.pcl* output file.

```
% compile_pcl MySrc.pcl MyObj.pcl
```

Note

Note that compile_pcl is a model-versioned tool, meaning that the model itself determines which version of the tool is used when you invoke the tool. You only need to select the model version.

Example PCL Program

This section shows a sample PCL program written for the Intel 80186 HV model. The program begins with the include statement that instructs the PCL compiler to include the model's command header (.cmd) file. The command header file contains the processor command definitions and #define directives. The program then defines a constant called error_addr and declares the variables addr and data_out as integers.

The main function declares an integer variable named data_in and then writes to and reads from successive memory locations. If the returned values do not equal the written values, the program displays an error message.

The last section of code defines the interrupt routines, one maskable and one nonmaskable.

```
#include "i80186.cmd"
#define error_addr                              0xffff
```

```
#define NMI                                     0x02
#define DMA0                                    0x0A
#define TIMER2                                  0x13

int addr, data_out;

main ()
{
       int data_in;
       addr= 0;
       data_out = 1;

       while (addr < 0x1000)
       {
             write_memory(4, addr, data_out);
             data_in = read_memory(4, addr);
             if (data_in != data_out)
                   write_io(4,error_addr,data_in);
             data_out = data_out << 1;
             if (data_out == 0)
                   data_out = 1;
             addr += 4;
       }
}  /* End of Main */

interrupt (vector)
int vector;

{
       switch (vector)
{
       case NMI:       /* NMI service */
             write_memory(2, 0x20C08, 0xBBC4);   /* push to stack */
             write_memory(2, 0x20C0A, 0x58DE);
             write_io(2, 0x00204, 0x20);
             break;

       case DMA0:      /* DMA channel 0 interrupt service */
             read_memory(2, 0x20C08);     /* read from stack */
             write_memory(2, 0x20C0A, 0x0070);
             read_io(2, 0x00204);
             break;

       case TIMER2:     /* timer 2 interrupt service */
             write_memory(2, 0x20C08, 0xBBC4);   /* push to stack */
             write_memory(2, 0x20C0A, 0x58DE);
             write_io(2, 0x00204, 0x20);
             write_register(0x2A, 0x0006);
                         /* write Priority Mask Register; */
```

```
                break; /* timer will be masked through priority */

        default:
                break;
        }
    }
```

8
User-Defined Timing

Introduction

The SmartModel Library includes standard, component-based timing files that have identical functionality, but different timing characteristics. (For example, the timing characteristics of the 74LS00 component differ from those of the 74F00 component.) In addition to these standard timing files, you can create custom, component-based timing files through user-defined timing (UDT). UDT is possible because a model's timing file is loaded at simulation startup. You can use UDT to:

- Create new timing versions that are not yet available in the SmartModel Library
- Develop a custom timing model using specifications from several possible manufacturers
- Represent your own custom timing or a special binding
- Scale timing to accommodate different design requirements
- Modify models to turn off their timing checks
- Modify memory models to turn off the access delay feature

You can also use UDT for instance-based timing; that is, you can specify timing characteristics for a single specific instance of a model. For example, you could use instance-based timing to back-annotate interconnect delays into a simulation or to specify a different interconnect delay for a model instance that is on the critical path.

Note

FlexModels support component-based UDT, but not instance-based UDT. For more information on FlexModels, refer to the *FlexModel User's Manual*.

Timing Files

The SmartModel Library contains both source and compiled timing files. When you want custom timing for models, you can either edit the source files supplied with the SmartModel Library or create your own. In either case you should create your own timing directory to keep your custom timing files in; otherwise, your work will be lost when you update your SmartModel Library.

You can store your custom timing files in any arrangement of directories that suits your needs. However, the models must know where to look for your custom timing files. The search rules for locating timing files are explained in the next section.

If you create new timing versions, you must support these files as you would any other library of simulation models. In particular, if the timing file format changes with time, you might receive an error message about the format of your file. In that case, you would need to rerun compile_timing.

Instance-Based Timing

Instance-based timing allows you to refine timing characteristics that might affect only a particular instance of a model. For example, you could use instance-based timing to back-annotate interconnect delays into a simulation.

As an example, consider a design that contains multiple ECL 10H101 gates, each with a typical propagation delay of 2 ns. The interconnect delay (printed circuit board trace length) may be 1.5 ns for instance U101 and 3.0 ns for instance U135. With instance-based timing, you could add the interconnect delay into your simulation by defining a U101 instance-based timing of 3.5 ns (2.0 ns from the gate, 1.5 ns from the interconnect) typical propagation delay, and a U135 instance-based timing of 5.0 ns delay.

Timing File Search Rules

Each model instance gets its timing information from a timing data file. Because you can have many timing data files, models follow prescribed search rules for locating a timing version. The rules are simple: a list of directories is searched until a match is found. That is, timing data files in the first listed directory are searched before timing data files in the second listed directory, and so on.

The timing information in any timing data file can be specified by component name and by instance name (both can exist in the same file). This is specified by the case selector in the timing data file (keyword is either COMP or INST). Timing data files are searched first for instance name, then for component name. If you use instance-based timing (case INST of), the value of the model's InstanceName parameter is compared with the strings in the timing data file. Alternately, if you use component-based timing (case COMP of), the value of the model's TimingVersion parameter is compared with the strings in the timing data file.

Each model instance searches for a matching value in the timing data file independently of all other model instances, and stops searching at the first match. Therefore, different instances of the same model can get their timing version information from different timing data files.

The timing data file search path can be set with the LMC_PATH environment variable. This allows each user to define their own timing file search path.

To define your timing file search path, set the LMC_PATH environment variable in your startup file. The syntax used to set this environment variable is the same as for the path variable; that is, you set LMC_PATH to a list of directories, where each directory listing is separated from others by a colon. For example, assuming you are using C shell:

```
% setenv LMC_PATH  "/usr/home/dir1:
                   /usr/home/dir2:
                   /usr/home/dir2/subdir1"
```

You must explicitly specify each directory (and subdirectory) containing custom timing files (the search is not recursive). For NT you must separate multiple entries for the LMC_PATH environment variable using a semicolon-separated list, not a colon-separated list as in UNIX.

Note that if you want to verify or diagnose where a model instance is getting its timing data from, you can use the TraceTimeFile command (issued through the SWIFT command channel). This command causes each model to display information about the timing data files it has searched, and which file produced a match.

For information on how to set environment variables and how to use the console application on the NT platform, refer to "Setting Environment Variables on NT Platforms" on page 43 and "Running Console Applications on NT Platforms" on page 44.

Creating New Timing Versions

To customize a model's timing, copy the supplied ASCII timing data file to your own timing directory using the Browser's Copy Timing File function. For instructions, refer to "Creating Custom Timing Versions" on page 164. After you copy the file, you can edit and compile it. Compiling the file builds the timing file the model uses in simulation. Figure 16 illustrates this process.

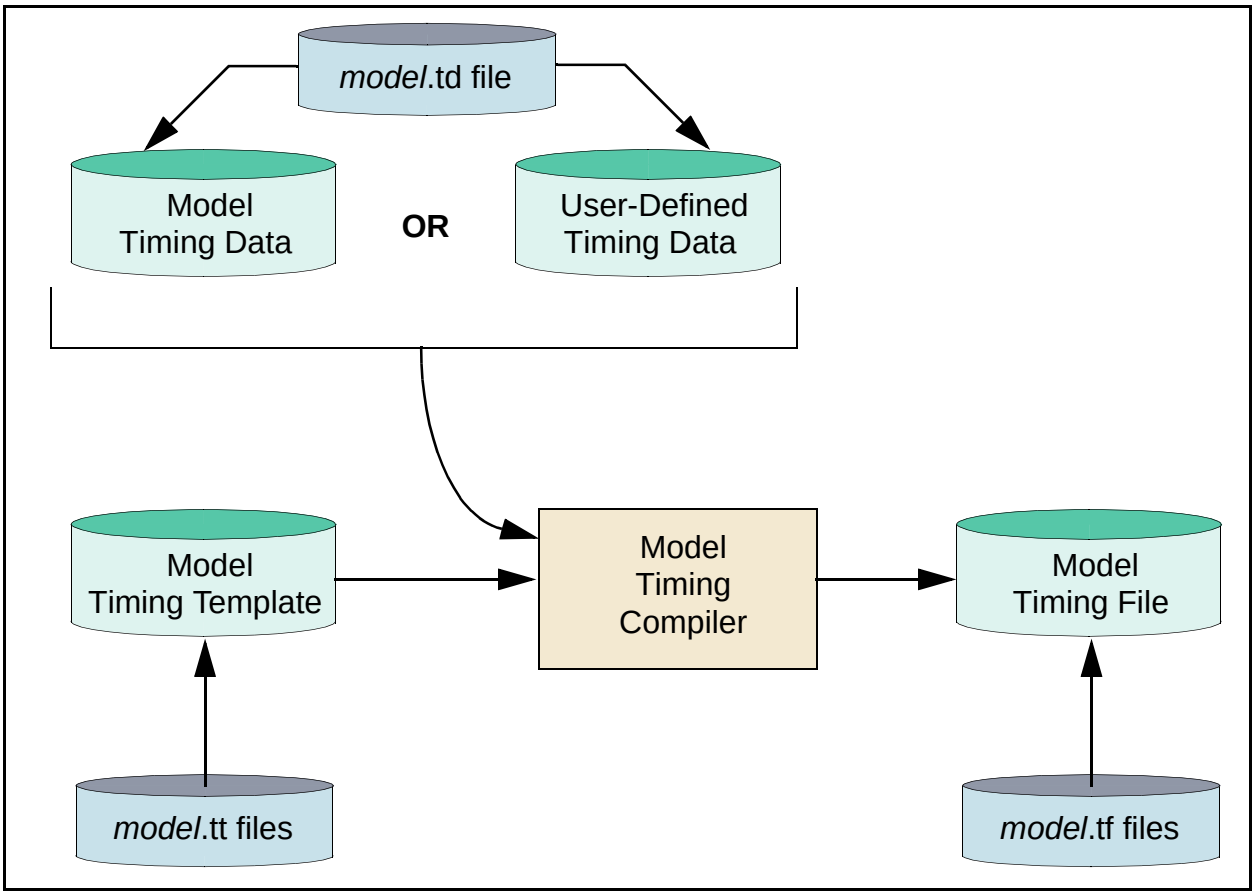


Figure 16: User-Defined Timing Process

A model timing file is created with a timing data file, a timing template, and the SmartModel Library timing compiler. You can use the timing data included with the library, or you can use your own model timing. In any case, a timing file is only generated once, and then (depending on the search rules) is available to the model in every simulation.

Note

Before you make significant changes to the timing data, you should be familiar with both the format and grammar of timing data files. For details, refer to "Timing Data File Format" on page 165 and "Timing Data File Grammar" on page 174.

User-Defined Timing Examples

Two examples of creating new timing are described here. In the first case, a timing version is added to reflect a new component that was not included in the standard timings for the model. In the second example, a worst-case timing version is generated from all the existing versions.

Example Timing Data File

Both examples are based on the timing data file for the am29520 model. Here is what the timing data file for this model looks like before any modifications have been made.

```
# Timing data file generated by Synopsys Logic Modeling, Inc.
# for use with the Synopsys Logic Modeling compile_timing tool.
# @(#) File generated: 8/19/93
# @(#) Tool versions: [ 1.1, 3 ]

# Timing descriptions:
# --------------------
# th_I_CLK_lh -- Hold time for I(instruction) to CLK(high)
# th_D_CLK_lh -- Hold time for D(inputs) to CLK(high)
# ts_I_CLK_lh -- Setup time for I(instruction) to CLK(high)
# ts_D_CLK_lh -- Setup time for D(inputs) to CLK(high)

# Range: Min/Typ/Max
model AM29520
  case comp
    of "AM29520-COM": # AMD, Bipolar Microprocessor Logic &
                      #Interface, AM29000...
    # Timing Label :     Min        Typ      Max # Vendor Label
    #-------------------------------------------------------
      pwmin_CLK          : 10.0                  ;# tpw
      th_I_CLK_lh        :  3.0                  ;# th
      th_D_CLK_lh        :  3.0                  ;# th
      ts_I_CLK_lh        : 10.0                  ;# ts
      ts_D_CLK_lh        : 10.0                  ;# ts
      tpd_S_Y            : {2.4},   12.0,    20.0;# tpdsel
      tpd_CLK_lh_Y(lh)   : {2.4},   12.0,    21.0;# tpd
      tpd_CLK_lh_Y(hl)   : {2.4},   12.0,    22.0;# tpd
```

```
      tpd_OE_Y(hz)       : {1.0},    5.0,    13.0;# tdis
      tpd_OE_Y(lz)       : {1.2},    6.0,    15.0;# tdis
      tpd_OE_Y(zh)       : {2.4},   12.0,    20.0;# tena
      tpd_OE_Y(zl)       : {2.6},   13.0,    21.0;# tena
    . . .
    of "L29C520M-1": # Logic Devices, Fast CMOS Data Book (1989)
    # Timing Label       : Min     Typ     Max # Vendor Label
    #-----------------------------------------------------------
      pwmin_CLK          : 10.0                 ;# tpw
      th_I_CLK_lh        :  3.0                 ;# th
      th_D_CLK_lh        :  3.0                 ;# th
      ts_I_CLK_lh        : 10.0                 ;# ts
      ts_D_CLK_lh        : 10.0                 ;# ts
      tpd_S_Y            : {2.9},   {14.7},  22.0;# tpdsel
      tpd_CLK_lh_Y(lh)   : {3.2},   {16.0},  24.0;# tpd
      tpd_CLK_lh_Y(hl)   : {3.2},   {16.0},  24.0;# tpd
      tpd_OE_Y(hz)       : {2.1},   {10.7},  16.0;# tdis
      tpd_OE_Y(lz)       : {2.1},   {10.7},  16.0;# tdis
      tpd_OE_Y(zh)       : {2.9},   {14.7},  22.0;# tena
      tpd_OE_Y(zl)       : {2.9},   {14.7},  22.0;# tena
  end case;
end model;
```

Adding a New Timing Version

In this example, you create a new timing version of a model. The modeled device is an Integrated Device Technology part whose component name is IDT29FCT520A-COM. To add a new timing version, follow these steps:

1. Create your own timing data directory. Do not store your modified timing files in the SmartModel Library directory structure.
2. Copy the original timing data file into your own timing data directory using the Browser. For instructions, refer to "Creating Custom Timing Versions" on page 164.
3. Use an ASCII editor to open the new file and duplicate one of the existing case statements. Edit the duplicated statement to make the new timing version you need, then delete all other case statements. In this example, the first case statement in the file was copied and edited to make the required Integrated Device Technology version.
4. Document your modifications by citing the specification's source in a comment following the component name.

5. Compile your new timing data file so it can be used by the model. The timing compiler creates an executable version of the timing file and checks your new file against the timing template, as described in "Using the Timing Compiler" on page 178. The compiler catches any typographical errors you might have made. Use the following command to invoke the timing compiler:

   ```
   % compile_timing am29520.td
   ```

   This command creates a timing file named am29520.tf. Move your new file to your user-defined timing directory. Now any "am29520" model in your design automatically finds the correct timing file at simulation startup.

6. Change the value of the SWIFT TimingVersion parameter in your model instantiating to the name you are using for your new timing version.

Example Timing Data File with New Timing Version

The following example shows a timing data file for the am29520 model that includes the new timing version.

```
# Timing data file generated by Synopsys Logic Modeling, Inc.
# for use with the Synopsys Logic Modeling compile_timing tool.
# @(#) File generated: 8/19/93
# @(#) Tool versions: [ 1.1, 3 ]
# added a component to support IDT -Bob/Rich, 8/19/93
# Timing descriptions:
# --------------------
# th_I_CLK_lh -- Hold time for I(instruction) to CLK(high)
# th_D_CLK_lh -- Hold time for D(inputs) to CLK(high)
# ts_I_CLK_lh -- Setup time for I(instruction) to CLK(high)
# ts_D_CLK_lh -- Setup time for D(inputs) to CLK(high)
# Range: Min/Typ/Max
model AM29520
  case comp
    of "IDT29FCT520A-COM":    # IDT, High Performance CMOS
                              #Data Book (1988)

    # Timing Label         :  Min      Typ      Max # Vendor Label
    #-------------------------------------------------------
      pwmin_CLK            :   7.0                    ;# tpw
      th_DIN_CLK_lh        :   1.0                    ;# th
      th_I_CLK_lh          :  1.0                     ;# th
      tpd_CLK_lh_Y(lh)     :  {2.4},   12.0,     21.0;# tpd
      tpd_CLK_lh_Y(hl)     :  {2.4},   12.0,     22.0;# tpd
      tpd_OE_Y(hz)         :  {2.4},   6.0,      12.0;# tdis
      tpd_OE_Y(lz)         :  {2.4},   6.0,      12.0;# tdis
      tpd_OE_Y(zh)         :  {2.4},   9.0,      15.0;# tena


      tpd_OE_Y(zl)         :  {2.6},   9.0,      15.0; tena
      tpd_S_Y              :  {2.4},   7.0,      13.0;# tpdsel
```

```
        ts_DIN_CLK_lh        :  5.0                    ;# ts
        ts_I_CLK_lh          :  5.0                    ;# ts
    end case;
end model
```

Creating Custom Timing Versions

In this example, a custom timing version of a model is created and placed in its own timing data file. This new file removes all specific component cases, replacing them with a single set of values. These values represent the minimums and maximums for all "commercial" timing in the timing data file created in the preceding example (that is, the original timing plus the Integrated Device Technology timing).

The timing component case in this file is named "29520" to differentiate it from any specific manufacturer's values. As before, the process is: copy the file to the user-defined timing directory, edit it, compile it, and place it in a directory for use by the "am29520" model at simulation time. The directory where you place the file must be in the timing file search path.

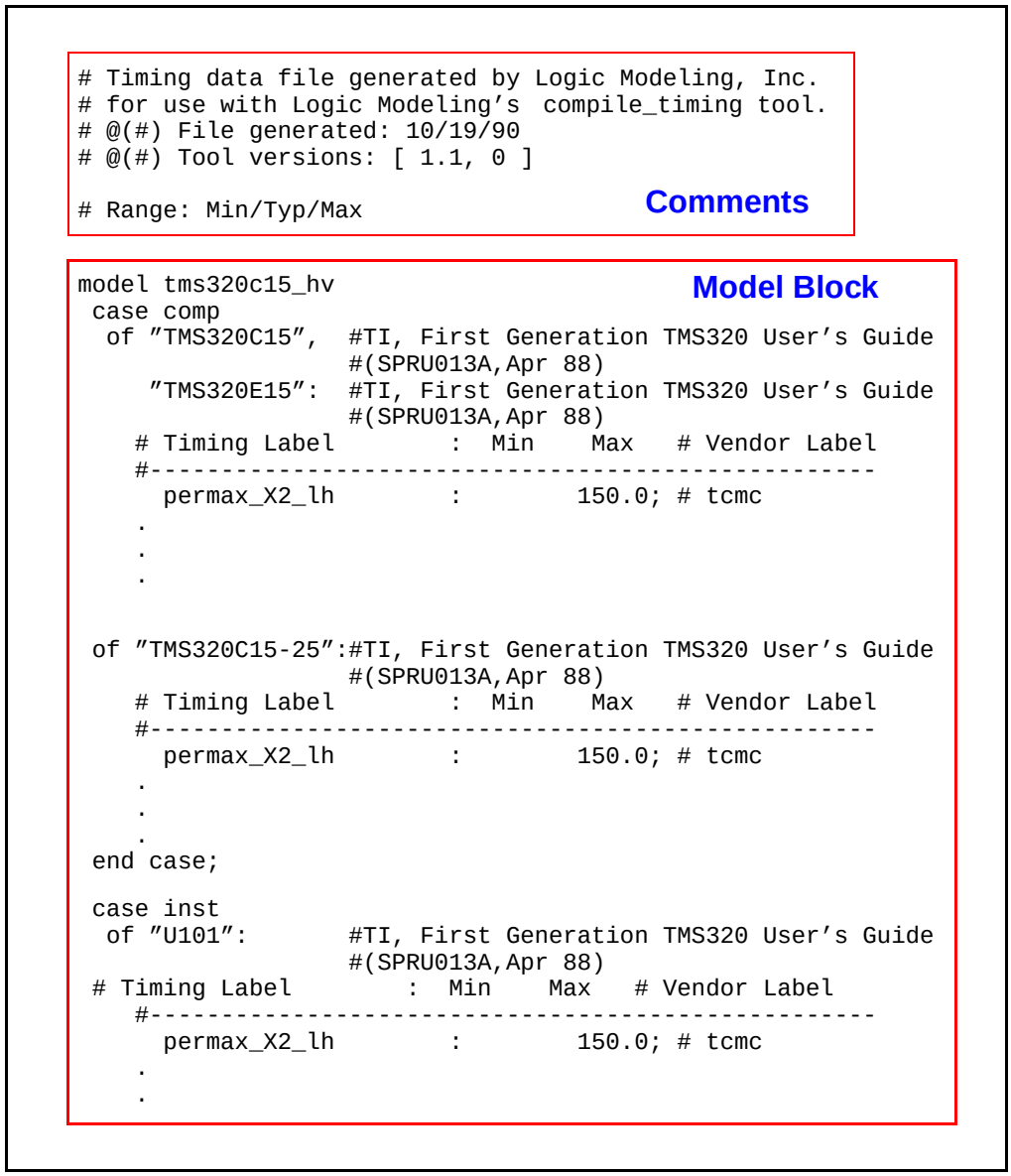
Example Timing Data File with Custom Component

The following example shows a timing data file for the am29520 model that includes a custom component.

```
# Timing data file generated by Synopsys Logic Modeling, Inc.
# for use with the Synopsys Logic Modeling compile_timing tool.
# @(#) File generated: 8/19/93
# @(#) Tool versions: [ 1.1, 3 ]
# modified 8/19/93 to create custom min-of-min and max-of-max
# cmpnt-Bob.
model AM29520
        case comp
              of "29520":  # Custom min-of-min/max-of-max component
              # Timing Label                        :MinTypMax# Vendor Label
              #-----------------------------------------------------
                    pwmin_CLK                       : 7.0    ;# tpw
                    th_DIN_CLK_lh                   : 3.0    ;# th
                    th_I_CLK_lh                     : 3.0    ;# th
                    tpd_CLK_lh_Y(lh)                : 2.4,11.3,25.0;# tpd
                    tpd_CLK_lh_Y(hl)                : 2.4,11.3,24.0;# tpd
                    tpd_OE_Y(hz)                    : 1.0,13.0,25.0;# tdis
                    tpd_OE_Y(lz)                    : 1.2,11.9,25.0;# tdis
                    tpd_OE_Y(zh)                    : 2.4,11.3,25.0;# tena
                    tpd_OE_Y(zl)                    : 2.6,11.2,25.0;# tena
                    tpd_S_Y                         : 2.4,11.3,25.0;# tpdsel
                    ts_DIN_CLK_lh                   :13.0    ;# ts
                    ts_I_CLK_lh                     :13.0    ;# ts
        end case;
end model;
```

Timing Data File Format

Timing data files are in ASCII format. The timing data files that come with the SmartModel Library are named *model*.td. You can access these timing data files using the Browser tool. Timing data files normally consist of comments and a model block, as shown in Figure 17.




```
# Timing data file generated by Logic Modeling, Inc.
# for use with Logic Modeling's  compile_timing tool.
# @(#) File generated: 10/19/90
# @(#) Tool versions: [ 1.1, 0 ]

# Range: Min/Typ/Max
```

```
model tms320c15_hv
 case comp
  of "TMS320C15",  #TI, First Generation TMS320 User's Guide
                   #(SPRU013A,Apr 88)
     "TMS320E15":  #TI, First Generation TMS320 User's Guide
                   #(SPRU013A,Apr 88)
    # Timing Label        :  Min    Max   # Vendor Label
    #-------------------------------------------------
      permax_X2_lh        :        150.0; # tcmc
    .
    .
    .


 of "TMS320C15-25":#TI, First Generation TMS320 User's Guide
                   #(SPRU013A,Apr 88)
    # Timing Label        :  Min    Max   # Vendor Label
    #-------------------------------------------------
      permax_X2_lh        :        150.0; # tcmc
    .
    .
    .
 end case;

 case inst
  of "U101":       #TI, First Generation TMS320 User's Guide
                   #(SPRU013A,Apr 88)
 # Timing Label        :  Min    Max   # Vendor Label
    #-------------------------------------------------
      permax_X2_lh        :        150.0; # tcmc
    .
    .
```

Figure 17: Timing Data File Elements

Assumed Propagation Delays

All models in the SmartModel Library support a range of timing delays, but manufacturers often supply only a single value for device delay parameters. Memories, for example, are often specified with only maximum delay values. In such cases, Logic Modeling duplicates the MAX delay value for the MIN and TYP fields of the timing data file. In this way, if you change the delay mode of your simulation, the delay value remains true to the manufacturer's specifications. Figure 18 shows what the timing data file for such a model would look like.

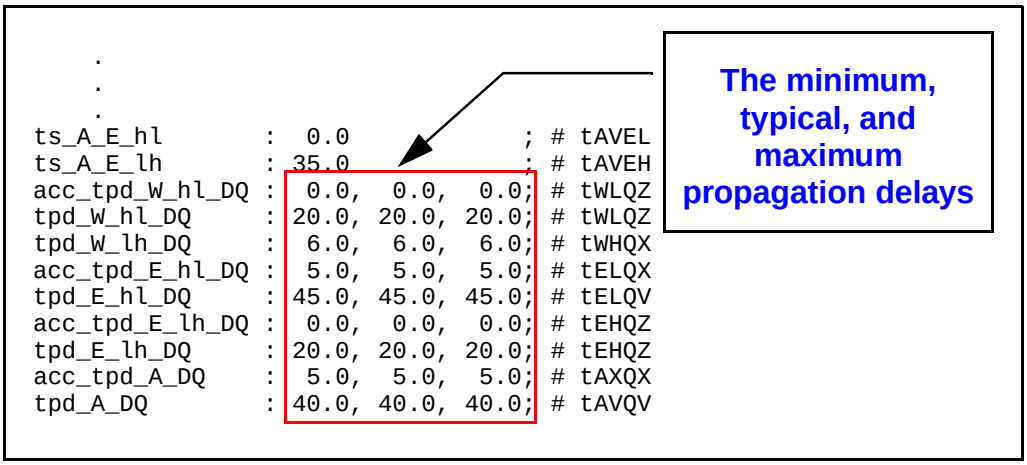


Figure 18: Assumed Propagation Delays

You can change any of the delay values by creating a custom timing version of the model.

Models With Vendor-Supplied Delay Ranges

When an IC manufacturer specifies a delay range for a part, the data is always modeled. Occasionally a manufacturer does not supply all three delay values for every parameter, in which case, the model uses derived values as shown in Table 15. If a manufacturer specifies a single propagation delay, then the specification is entered for all three values in the model.

Table 15: Derived Propagation Delay Values

Given	MIN	TYP	MAX
ALL	MIN	TYP	MAX
MIN & TYP	MIN	TYP	3/2 TYP
TYP & MAX	1/5 TYP	TYP	MAX

Table 15: Derived Propagation Delay Values (Continued)

Given	MIN	TYP	MAX
MIN & MAX	MIN	2/3 MAX	MAX
TYP	1/5 TYP	TYP	3/2 TYP
MAX	1/5 TYP	2/3 MAX	MAX

Note

A device is never specified with just a MIN spec, so that case does not appear in the table.

Calculated Propagation Delays

Occasionally, device manufacturers do not supply all three propagation delay values for every delay parameter. In such cases, Logic Modeling calculates the missing values. Figure 19 shows that calculated values are used for the TYP propagation delay (denoted by the braces).

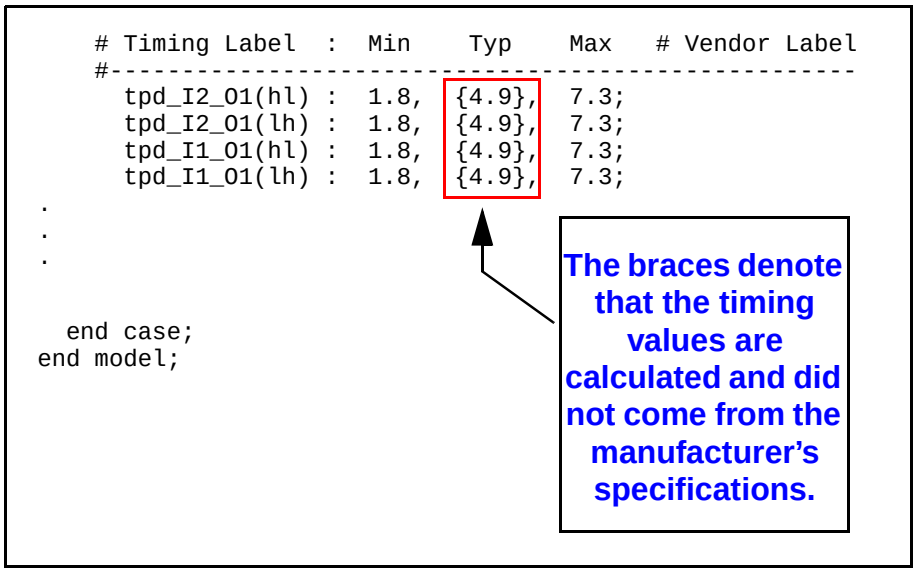


Figure 19: Calculated Propagation Delays

You can change any of the delay values by creating a custom timing version of the model.

Timing Data File Comments

Figure 20 illustrates the different types of comments that can appear in the timing data file header.

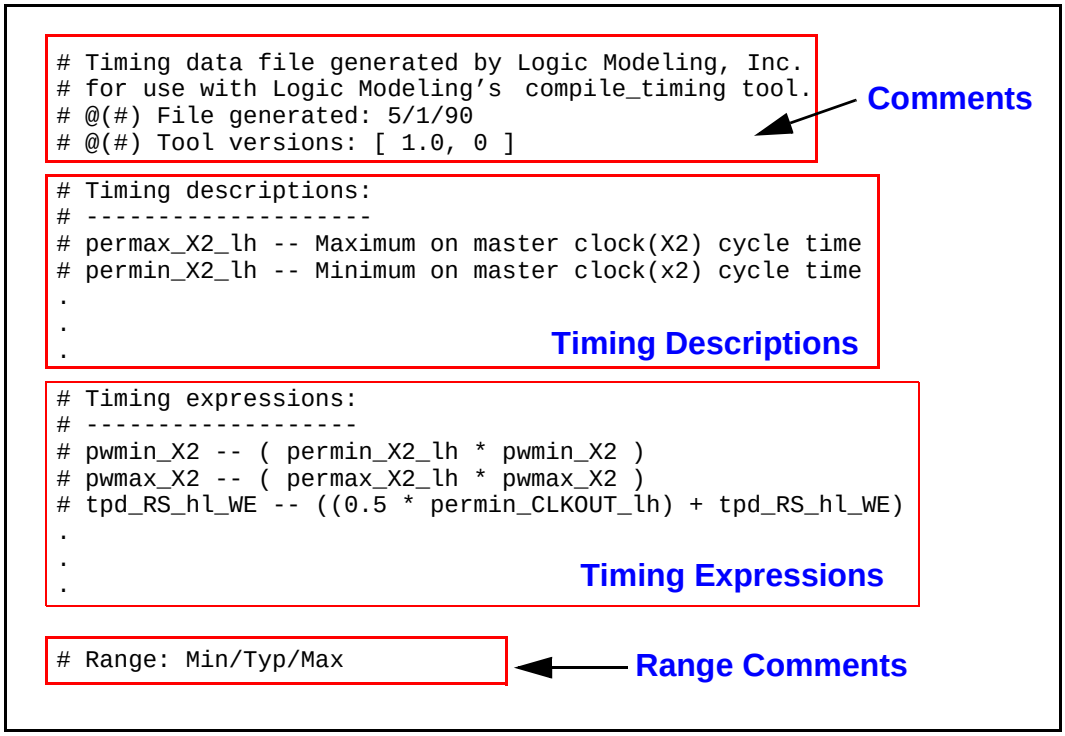


Figure 20: Timing Data File Comments

General Comments

General comments note the date the file was generated and the tool versions used.

Timing Description Comments

Timing descriptions provide information about the labels used in timing statements. If a label does not clearly describe the particular timing statement, a comment is included at the top of the timing data file that further describes it. The following example shows how comments are used to distinguish between timing descriptions that appear nearly identical.

```
# Timing descriptions:
# --------------------
# tpd_X2_lh_WE_1 -- WE(lh) delay time from X2(lh) with RS(hl)
# tpd_X2_lh_WE_2 -- Delay time X2(lh) to WE, (CLKOUT(hl) to WE(hl))
```

Timing Expression Comments

Timing expressions list the timing statement values that are calculated by substituting timing data numbers in an expression. These expressions are included as comments at the top of the timing data file. The following is an example of a delay that is half the period of a clock plus some constant:

```
# Timing expressions:
# -------------------
# tpd_RS_hl_WE -- ( ( 0.5 * permin_CLKOUT_lh ) + tpd_RS_hl_WE )
```

This means that the delay from RS(hl) to WE is calculated by substituting the timing data numbers into the expression. For example, you can supply the following numbers:

```
permin_CLKOUT_lh: 200.0;
tpd_RS_hl_WE: 50.0;
```

As a result, the actual number for the delay is:

```
( ( 0.5 * 200.0 ) + 50.0 ) = 150.0
```

Do not change the text in a timing expression comment. Timing expression comments serve only to document how calculations are performed.

There are several predefined functions that can appear in timing expressions:

MIN(value1, value2)
Returns the minimum of value1 and value2.

MAX(value1, value2)
Returns the maximum of value1 and value2.

ACTUAL(timing-name)
Evaluated during simulation.

The ACTUAL() expression is used to supply a value that is based on the actual clock period being used during the simulation. Before the simulation runs and the clock period can be used, the expression is evaluated to be equal to the value of timing-name.

Internal Pin Comments

Some manufacturers specify timing with respect to internal pins. This is often true for PLDs. All timing names dealing with internal pins are highlighted as comments at the top of the associated timing data file. The following example is for a PLD that has internal feedback pins:

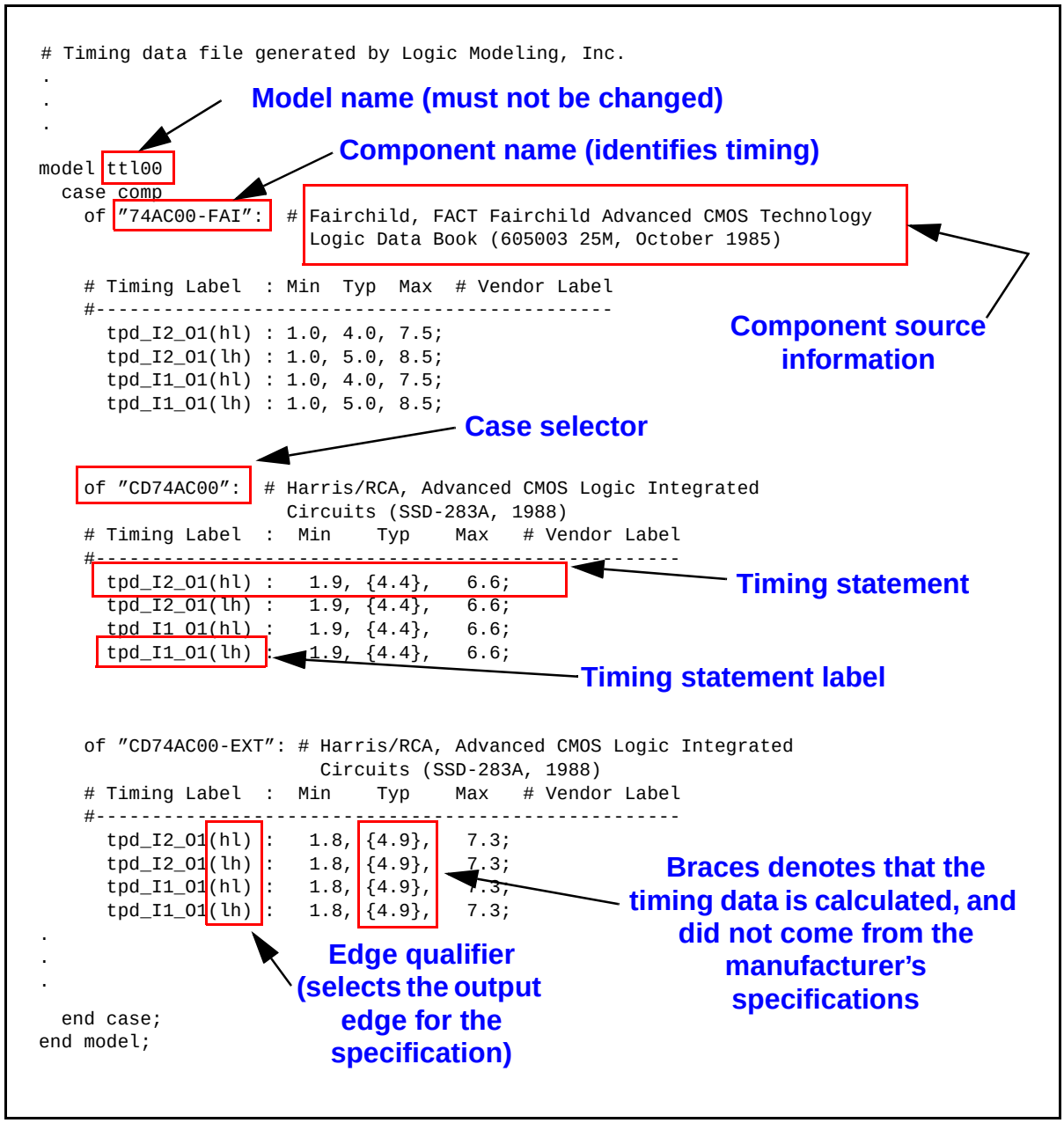
```
# Timing with internal pins:
# --------------------------
# tpd_CLK_lh_FB
```

Range Comments

Range comments describe the propagation delays used in the model: Min/Typ/Max.

Timing Data File Model Block

Figure 21 shows an annotated example of a model block in a timing data file.




```
# Timing data file generated by Logic Modeling, Inc.
.
.
.
model ttl00
  case comp
    of "74AC00-FAI":  # Fairchild, FACT Fairchild Advanced CMOS Technology
                        Logic Data Book (605003 25M, October 1985)

    # Timing Label  : Min  Typ  Max  # Vendor Label
    #-----------------------------------------------
      tpd_I2_O1(hl) : 1.0, 4.0, 7.5;
      tpd_I2_O1(lh) : 1.0, 5.0, 8.5;
      tpd_I1_O1(hl) : 1.0, 4.0, 7.5;
      tpd_I1_O1(lh) : 1.0, 5.0, 8.5;

    of "CD74AC00":  # Harris/RCA, Advanced CMOS Logic Integrated
                      Circuits (SSD-283A, 1988)
    # Timing Label  :  Min    Typ    Max   # Vendor Label
    #-------------------------------------------------
      tpd_I2_O1(hl) :   1.9, {4.4},   6.6;
      tpd_I2_O1(lh) :   1.9, {4.4},   6.6;
      tpd_I1_O1(hl) :   1.9, {4.4},   6.6;
      tpd_I1_O1(lh) :   1.9, {4.4},   6.6;


    of "CD74AC00-EXT": # Harris/RCA, Advanced CMOS Logic Integrated
                         Circuits (SSD-283A, 1988)
    # Timing Label  :  Min    Typ    Max   # Vendor Label
    #-------------------------------------------------
      tpd_I2_O1(hl) :   1.8, {4.9},   7.3;
      tpd_I2_O1(lh) :   1.8, {4.9},   7.3;
      tpd_I1_O1(hl) :   1.8, {4.9},   7.3;
      tpd_I1_O1(lh) :   1.8, {4.9},   7.3;
.
.
.
  end case;
end model;
```

Figure 21: Annotated Timing Data File Model Block

Timing Statement Format

Timing statements usually appear one to a line. When the same timing values apply to several timing labels, a list format can be used. There are two types of labels: delay labels and timing check labels. Both types are described below.

Delay Label Format

Delay labels have the following syntax:

```
Label = Type_FromPin[_FromEdge]_ToPin[_ToEdge][_Integer]
```

Delay Label Syntax Elements

Delay labels have the following syntax elements:

Type

One of the following values:

- tpd—Specifies a delay.
- acc_tpd—Specifies an access delay, meaning that the output goes to unknown after the delay.
- trigger—Specifies that the output makes the ToEdge transition after the delay.

FromPin

Specifies the input pin or bus that causes the delay.

FromEdge

Optional element whose value can be either lh (low-to-high) or hl (high-to-low). If you include a FromEdge value, the delay only occurs when the FromPin has the specified edge.

ToPin

Specifies the output pin or bus that changes as a result of the delay.

ToEdge

Optional element only used for triggers. The value of this element can be either al (any-to-low), ah (any-to-high), or az (any-to-Z).

Integer

Determines the correct timing parameter based on the internal state condition of the model. For example, transitions from high-to-Z may have different timings than transitions from high-to-low. Using this element you can differentiate between the timing values.

Timing Check Label Format

Labels for timing checks have the following syntax:

label = *Type_Pin1[_Edge][_Pin2][_Edge][_Integer]*

Timing Check Label Syntax Elements

Timing check labels have the following syntax elements:

Type

Specifies one of the following types of timing checks:

- ts—Setup check, where Pin1 before Pin2 must be greater than or equal to the specified value.
- recovery—Same as ts, but the error message is different.
- th—Hold, where Pin1 after Pin2 must be greater than or equal to the specified value.
- skewmin—Minimum skew, where Pin1 must come before Pin2
- skewmax—Maximum skew. This is the same as ts, but the error message is different.
- pwmin—Minimum pulse width, where Pin1 pulse width high or low must be greater than or equal to the specified value.
- pwlmin—Minimum pulse width low, where Pin1 pulse width low must be greater than or equal to the specified value.
- pwhmin—Minimum pulse width high, where Pin1 pulse width high must be greater than or equal to the specified value.
- pwmax—Maximum pulse width, where Pin1 pulse width high or low must be less than the specified value.
- pwlmax—Maximum pulse width low, where Pin1 pulse width low must be less than the specified value.
- pwhmax—Maximum pulse width high, where Pin1 pulse width high must be less than the specified value.
- fmin—Minimum frequency, where Pin1 frequency must be greater than or equal to the specified value.
- fmax—Maximum frequency, where Pin1 frequency must be less than the specified value.

- permin—Minimum period, where Pin1 period must be greater than or equal to the specified value.
- permax—Maximum period, where Pin1 period must be less than the specified value.

Pin1

Specifies the input pin or bus.

Edge

Optional element whose value can be either lh (low-to-high) or hl (high-to-low). The timing check occurs only if the pin makes the specified transition.

Pin2

Same as Pin1, except that it is used only for timing checks involving two pins.

Integer

Determines the correct timing parameter based on the internal state condition of the model. For example, transitions from high-to-Z may have different timing than transitions from high-to-low. Using this element allows you to differentiate between the timing values.

Timing Statement Format

Timing statements can appear one to a line, or when the same timing values apply to several timing labels, you can use a list to combine them, as shown in the following example. For the 74AC00-FAI component of the ttl00 model, a comma and new line separate the two timing labels from their mutual timing values. For the CD74AC00 component, all the timing labels use the same values, so they are listed with a comma and new line separating the four labels.

```
. . .
model ttl00
       case comp
             of “74AC00-FAI”:
             # Fairchild, FACT Fairchild Advanced CMOS Technology
             # Timing Label                    :MinTypMax# Vendor Label
             #-----------------------------------------------
                   tpd_I2_O1(hl)               ,
                   tpd_I1_O1(hl)               :1.0,4.0,7.5;
                   tpd_I2_O1(lh)               ,
                   tpd_I1_O1(lh)               :1.0,5.0,8.5;

             of “CD74AC00”:                    # Harris/RCA, Advanced
CMOS Logic Integrated
                                               # Circuits
             # Timing Label                    :MinTypMax Vendor Label
             #-------------------------------------------------
```

```
                    tpd_I2_O1(hl)              ,
                    tpd_I2_O1(lh)              ,
                    tpd_I1_O1(hl)              ,
                    tpd_I1_O1(lh)              :1.9,{4.4},6.6;
                    . . .
        end case;
end model;
```

Timing Data File Grammar

The information in timing data files conforms to a formal grammar that is documented in this section. In the descriptions below, terminals and reserved words are shown in roman type, *non-terminals* in italic type. The := symbol can be read as "is defined as." The vertical bar | delimits items in a list from which one item must be chosen. Square brackets [ ] enclose optional items. Braces enclose constructions that appear zero or more times. Left and right parentheses must enclose edge identifiers.

udt-data-file

udt-data-file := *model-block*

model-block

model-block := **model** *model-name*
case-statement
{ *case-statement* }
end model;

There can be one or more case statements within the model block. Case statements cannot be nested.

model-name

model-name := *identifier*

The model name is an identifier; it must match the name of the model that is using the timing data.

identifier

identifier := *letter* { *letter* | *digit* | _ }

letter := **a-z** | **A-Z**

digit := **0-9**

Timing data files are not case-sensitive; either uppercase or lowercase letters can be used.

case-statement

```
case-statement :=  case case-selector
                          of
                       case-tag   { , case-tag } : timing-statement ;
                                  { timing-statement ; }
                   end case ;
```

case-selector

```
case-selector := comp | inst
```

The value comp allows grouping of timing statements by manufacturer, speed version, and technology. The value inst identifies a specific instance of a model.

case-tag

```
case-tag := string-identifier
```

A case-tag is a text string identifying a specific value of the case selector.

string-identifier

```
string-identifier := " { letter | digit | general_symbols } "
```

All strings are quoted. Nonprintable characters (such as new line) are not allowed.

```
general_symbols    := _ | ! | @ | # | $ | % | ^ | & | * | ~ |
                . | , | ? | / | \ | < | > | : | - | +
```

timing-statement

```
timing-statement := timing-label { , timing-label  } : timing-values
```

If an output edge is used—and they are permitted only on delays—it must be surrounded by parentheses.

timing-label

```
timing-label := identifier [  ( output-edge{ output-edge}) ]
```

The timing-label is a required name used to identify the timing statement. Its value is fixed by the model and must be used.

output-edge

```
output-edge  := lh | lz | hl | hz | zl | zh | la | nl | ha |
                nh | za | nz | al | ah | az | aa | av | vv
```

The output-edge format requires parentheses. More than one edge can be specified. Use a space between elements of the list, as shown below:

```
. . .
of "CD74AC00": # Harris/RCA, Advanced CMOS Logic Integrated Circuits
# Timing Label       :   Min        Typ     Max # Vendor Label
#-----------------------------------------------------------
  tpd_I2_O1(hl lh)   :   1.9,       {4.4},  6.6;
  tpd_I1_O1(hl lh)   :   1.9,       {4.4},  6.6;
. . .
```

The output-edge is an optional identifier used to further qualify timing variations in propagation delays by denoting the output edge transition. If no edge is specified, then the timing values apply to all possible transitions on the output pin for the particular delay. Table 16 provides definitions for all of the possible output-edge values.

Table 16: Output-edge Values

Output-edge Value	Definition
lh	low-to-high
lz	low-to-high impedance
hl	high-to-low
hz	high-to-high impedance
zl	high impedance-to-low
zh	high impedance-to-high
la	low-to-any (lh, lz)
nl	any-to-not low (lh, lz, hz, zh)
ha	high-to-any (hl, hz)
nh	any-to-not high (hl, hz, lz, zl)
za	high impedance-to-any (zl, zh)
nz	any-to-not high impedance (lh, hl, zl, zh)
al	any-to-low (hl, zl)
ah	any-to-high (lh, zh)
az	any-to-high impedance (lz, hz)
aa	any-to-any (lh, lz, hl, hz, zl, zh)

Table 16: Output-edge Values (Continued)

Output-edge Value	Definition
av	any-to-not high impedance (lh, hl, zl, zh)
vv	not high impedance-to-not high impedance (lh, hl)

timing-values

timing-values := *timing-value* [, *timing-value* , *timing-value*] [*timing-unit*] | **NULL**

Timing-values is a list of one or three timing values, or the identifier NULL. The values represent timing for the minimum (min), typical (typ), and maximum (max) operating ranges. If a single value is used, it is assumed that it represents all three (min, typ, and max). If three values are specified, they represent min, typ, and max in that order. You can use the NULL identifier to disable timing checks and access delays.

timing-value

timing-value := *numeric-value*

The timing-value is either a number representing the delay or a timing check, depending on the statement.

numeric-value

numeric-value := [**{**] [**-**] *integer* | *real* [**}**]

The numeric-value can be a positive or negative integer or real. The value can also be scaled, using an optional timing-unit. The default unit for frequency timing statements is mhz, and ns is the default for all other timing statements. The minus sign denotes negative values. The braces used here are only found in files generated by Logic Modeling—they signify that the specifications were derived (not found on the manufacturer's datasheet).

real := *integer.integer*

integer := *digit* { *digit* }

timing-unit

timing-unit := **fs** | **ps** | **ns** | **us** | **ms** | **khz** | **mhz** | **ghz**

Table 17 provides definitions for all possible timing unit values.

Table 17: Timing Unit Values

Timing Unit	Definition
fs	femtoseconds
ps	picoseconds
ns	nanoseconds
us	microseconds
ms	milliseconds
khz	kilohertz
mhz	megahertz
ghz	gigahertz

comments

comments := { # | @ } [*comment_text*]

Comments begin with # or @ and continue to the end of the line.

Using the Timing Compiler

You use the timing compiler to compile timing data files for models. The timing compiler produces a timing file named *model*.tf in the current working directory. The tool also looks for the specified timing data file in the current directory. If the timing data file that you want to compile is not in the current working directory you must specify a full path name to the file.

In addition to producing a *model*.tf file that a model reads at simulator startup, the timing compiler also performs a series of checks on your timing data file.

Timing Compiler Checks

The timing compiler performs the following checks on the specified input source file:

- **Source Grammar Checks.** Verifies that the source file is written correctly. Ensures sensible source organization. For example, nested cases are not allowed.

- **Complete Source Checks.** Verifies that the timing source for a given model is completely specified. Partial timing specifications are not allowed. The timing statements in the source file must exactly match the template.
- **Global Value Checks.** Checks timing source values. While negative values are allowed in the timing source, they are not manageable in the model. Because negative propagation delays are not supported, they are identified as errors. Negative timing checks are not supported in the model runtime environment. They are identified and mapped to "0".
- **Valid Edge Checks.** Uses the timing template to verify correct output edge assignment in the timing source. If the model's output pin cannot be put into a high-impedance state, no Z-related edges are allowed in timing associated with that pin.
- **Tool and File Version Checks.** Checks tool versions and file format versions.

Running the Timing Compiler

The compile_timing tool takes an input timing data (.td) file as its only required argument and generates a compiled timing (.tf) file in the current working directory. You can optionally specify several switches, as shown in the following syntax description.

For NT, invoke the compile_timing program using the console command line. For more information, refer to "Running Console Applications on NT Platforms" on page 44.

Syntax

compile_timing [-Help] [-Messages] [-TTemplate *template-path*] *model*.td

Argument

model.td	Name of the input timing data file.

Switches

-H[elp]	Specify this switch for help using the compile_timing tool.
-M[essages]	Turn on user-defined timing messages in the model.
-TT[emplate] *template_path*	Used to specify a full path to a timing template input file. Do not use this switch. The tool automatically determines the correct version of the timing template to use.

Example

The following example invocation of compile_timing causes the tool to read a timing data file for the ttl00 model and generate a compiled timing file:

```
% compile_timing ttl00.td
```

You can use the -Messages switch to aid debugging efforts. When you compile a timing file specifying this switch, you enable generation of runtime messages. At simulation startup, each model compiled with messages enabled issues a message indicating the *model*.tf file being read and the instance or component being used.

9
Back-Annotating Timing Files

What is Backanno?

Different simulators back-annotate timing values from Standard Delay Format (SDF) files in different ways. To solve this problem for model users the Backanno tool extracts the relevant timing components from an SDF file and annotates SmartModel binary timing files with that information to normalize the model's behavior with different simulators.

This chapter describes how to use the Backanno tool to extract back-annotation timing data from SDF files and annotate them to a SmartModel format that the model can read. As part of this process, the Backanno tool creates:

- New SDF files that have the extracted timing commented out so that the delays are not back-annotated twice.
- Compiled time files (.tf) that contains the delays for the model.

The Backanno tool is controlled by a configuration file which:

- Identifies all the SmartModel instances to be back-annotated.
- Maps ports and delays.
- Identifies the SDF files and the hierarchical scope to which the data should be applied.

Process Overview

Figure 22 illustrates the process of back-annotating SmartModel timing files (.tf).

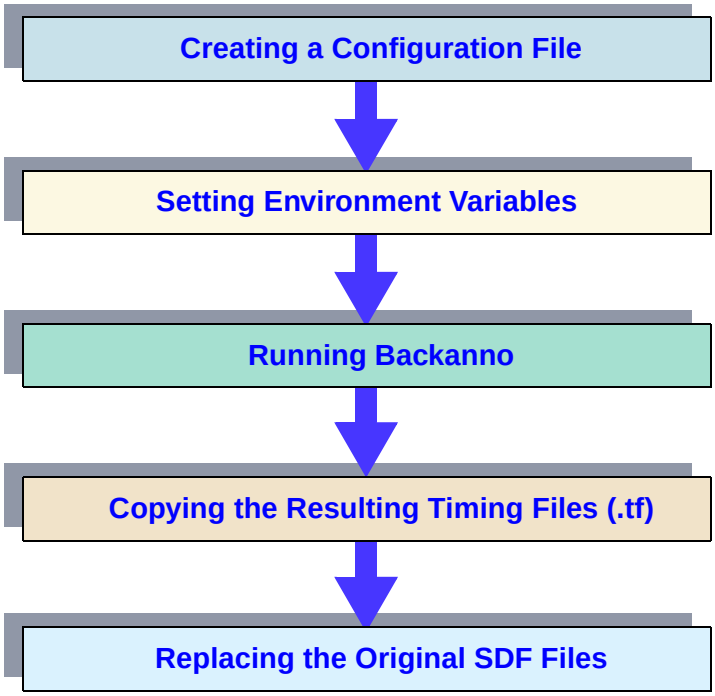


Figure 22: SmartModel Back-Annotation Process

Creating a Configuration File

You can create a configuration file to control the back-annotation process so that it is compatible with your simulator.

File Format

The configuration file format is similar in principle to Standard Delay Format. It allows C++ style comments and has the following general format:

```
{MODEL Section}*          // 0 or more MODEL Sections
{ANNOTATE Section}*       // 0 or more ANNOTATE Sections
{LMC_PATH Section}        // Optional LMC_PATH Section
```

Both Verilog- and VHDL-style hierarchy syntax are supported. For example, both "top.inst" and "/top/inst" are valid.

Both Verilog- and VHDL-style identifier syntax are supported. By default, the identifiers in the configuration file are treated as case-sensitive. However, the "-i" command line option causes the backanno tool to treat the identifiers as case-insensitive. Also, escape identifiers are allowed in both Verilog format – such as "\MY_MODULE," and VHDL format – such as "\MY_MODULE\".

Sample Configuration File

The following is an example of a configuration file. To see how the sections in the file tie together, you can refer back to this example as you read further.

This file is for a simulation session containing two SmartModel instances (I$1, I$4) of the ttl08. The model was generated from compiled Verilog with a vector input IN split into scalar ports I1 and I2.

```
(MODEL TTL08
( PORTMAP ( I1 IN[1] )
          ( I2 IN[2] ) )
      ( INSTLIST ( I$1 CD74AC08 )
                 ( I$4 SN74LS08  ) ) )

( LMC_PATH ./run16 )

(ANNOTATE new.sdf Top_Level
( DELAYSCALE MTM 1.0, 1.0, 1.0 )
( DELAYRANGE MTM )
( INTERCONNECT RCVR MAX )
( LOGFILE  sdf.log ) )
```

MODEL Section

The Model section consists of one or more model entries. There must be a model entry for each model in the design that is to be back-annotated. Each model entry can define optional port mapping and must declare one or more model instances.

Syntax

(MODEL *modelName* [*port_map*] *instance_list*)

Arguments

modelName

The SWIFT model name.

port_map

Consists of one or more port name mapping statements. It maps SWIFT port names to port names in the SDF file. The syntax for a port_map is:

(PORTMAP {(*SWIFTName DesignName*)})

SWIFTName

The model's SWIFT port name.

DesignName

The port name used by the delay calculator to generate the SDF. Unspecified ports are assumed to have identical names. In the case of duplicate entries, the last entry is used.

The *DesignName* is used when the model's port name doesn't match the SDF generated by the IC vendor's delay calculator. This can occur when:

- HDL compiler technology is used to generate a model.
- The design's port name is changed to create an HDL-independent model.
- The delay calculator has not been updated.

instance_list

Enumerates each model instance to have timing annotated and consists of one or more entries. The syntax is:

(INSTLIST {(*Instance Component*)})

Instance
The full hierarchical instance name of the model instance in the circuit.

Component
The SWIFT "TimingVersion" attribute; selects the base (unannotated) timing values shipped with the model.

ANNOTATE Section

The Annotate section consists of one or more individual SDF annotator entries. There is an SDF annotator entry for each SDF file that is to be parsed and applied to the timing of the simulation.

Verilog HDL implements this operation as $sdf_annotate task, while at least one VHDL implementation uses command line arguments. It appears that VHDL is less configurable and is a subset of the Verilog implementation. Most Verilog and VHDL capabilities are addressed.

Syntax

(ANNOTATE *SDFFile InstScope custom_parameters*)

Arguments

SDFFile
The complete path to the SDFFile.

InstScope
The hierarchical reference specifying the module instance (Verilog) or design region (VHDL) scope for application of the SDF file.

custom_parameters
Optional settings that customize the handling of timing values in the SDF file. These are:

- delay_scaling
- delay_selection
- delay_application
- process_reporting

The SDF format can contain either one or three (minimum, typical, and maximum) values. The model's compiled time files contain min, typ, and max values. Models can be configured by the simulator to use either min, typ, max, or a combination version (MTM) during simulation. As shown in Figure 23, delay scaling is applied to the SDF delay values prior to their usage. The scaled delay selection is applied to create the model's compiled time file. The model's DelayRange parameter selects the model's use of minimum, typical, or maximum timing delays during simulation.

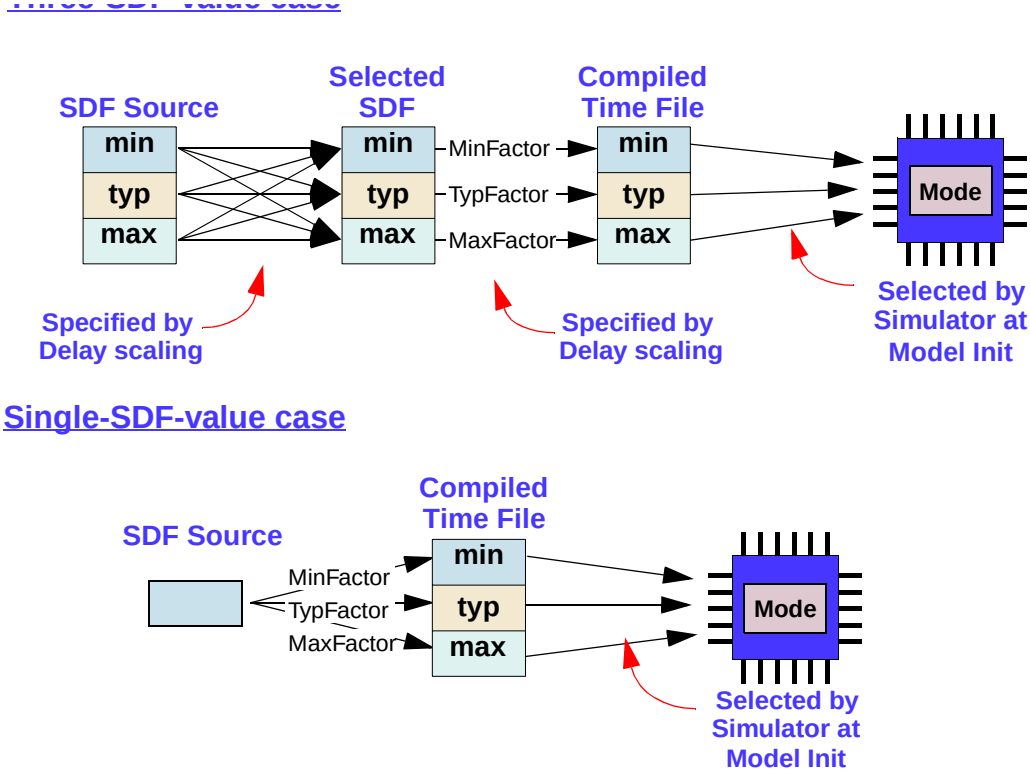


Figure 23: Delay Scaling Example

Delay_scaling

Delay_scaling allows a scale factor to be applied to the SDF value prior to its usage. The basic algorithm is:

> Any value that is mapped to a "min" scaled delay is multiplied by MinFactor, "typ" by TypFactor, and "max" by MaxFactor.

A separate scale factor is specified for each individual SDF delay range value. You must separate MIN, TYP, and MAX values with commas.

Syntax

(DELAYSCALE {MIN, | TYP, | MAX | MTM } *MinFactor TypFactor MaxFactor*)

Arguments

MinFactor
Floating point multiplier applied to minimum values.

TypFactor
Floating point multiplier applied to typical values.

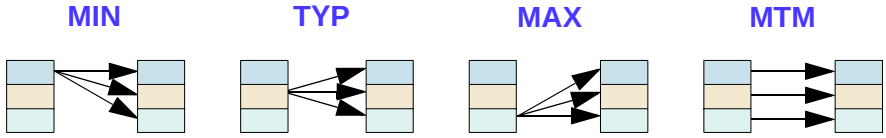
MaxFactor
Floating point multiplier applied to maximum values.

All three delay scale factors must be specified. If the entire delay_scaling is unspecified, then the following is assumed:

(DELAYSCALE MTM 1.0, 1.0, 1.0)

The application of the scale factors works such that the {MIN | TYP | MAX | MTM } construct specifies the selected SDF source and the scale factor triplet specifies the scaling that occurs for each SDF destination.

The mapping of SDF source to selected SDF in Figure 23 is controlled by the MIN, TYP, MAX, and MTM arguments as shown below.



Interconnect Statement

The Interconnect statement specifies where to place the SDF interconnect delay on the model. From the model's perspective, there are three possibilities: interconnect delays can be placed on model inputs, model outputs, or can be ignored.

Syntax

(INTERCONNECT { RCVR | DRVR [MIN | MAX])

Arguments

RCVR | DRVR
Specifies whether the interconnect delay is placed on the receiving (RCVR) or driving (DRVR) port.

RCVR MIN
Use the shortest path to any receiver, adding the delay to all receiving input ports.

RCVR MAX
Use the longest path to any receiver, adding the delay to all receiving input ports.

DRVR MIN
: Use the shortest path from any driver, adding the delay to all driving output ports.

DRVR MAX
: Use the longest path from any driver, adding the delay to all driving output ports.

If you omit the Interconnect statement, all interconnect delays will be ignored. This is the setting that is used when the simulator is configured to accurately simulate all interconnect delays.

For example, Figure 24 illustrates where InstC and InstD are SmartModels.

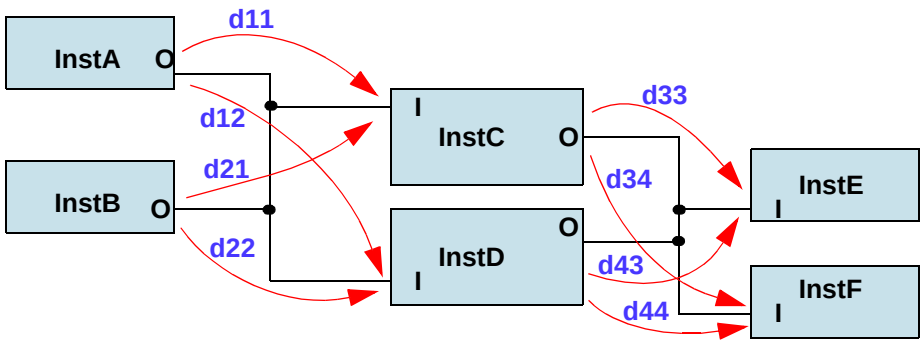


Figure 24: Interconnect Example

RCVR MIN. Places the shortest delay of d11 and d21 on InstC.I port and the shortest delay of d12 and d22 on InstD.I.

RCVR MAX. Places the longest delay of d11 and d21 on InstC.I port and the longest delay of d12 and d22 on InstD.I.

DRVR MIN. Places the shortest delay of d33 and d34 on InstC.O port and the shortest delay of d34 and d44 on InstD.O.

DRVR MAX. Places the longest delay of d33 and d34 on InstC.O port and the longest delay of d43 and d44 on InstD.O.

Process Output

Error Message Log

As delay values are annotated onto the model's timing, a log containing error messages from the annotator is written to a file you specify. By default, the log file is named ./ba.log. If more than one SDF file is used the messages are concatenated. The statement syntax is:

(LOGFILE *FileName*)

Model Time Files Location

As each model's timing is processed, compiled model time files are created. By default these files are placed in the current directory. You can specify a different destination directory with the following statement:

(LMC_PATH *pathname_1***:***pathname_2*: . . . **:***pathname_x*)

Setting Environment Variables

Set the $LMC_HOME environment variable to point to the directory that is the root of the SmartModel Library installation. You can use the $LMC_PATH variable to reference alternate locations to search for .td and .tt files for the compilation of SmartModel timing files (.tf).

Backanno Command Syntax

The syntax for the backanno command is as follows:

$LMC_HOME/bin/backanno *config_file* -ow -i

The following are the various sets of arguments for the backanno command:

-ow
: Overwrites SDF files in the current directory when creating new SDF files.

-i
: Causes case insensitivity. By default, backanno treats identifiers as case-sensitive, just like Verilog syntax, but the -i switch causes them to be case-insensitive, just like VHDL.

Running Backanno

Run the Backanno tool as shown in the following example:

```
% $LMC_HOME/bin/backanno configFile
```

Backanno creates the following files in the current working directory:

- The BAMODELS.LST file lists the back-annotated models.
- One .tf file for each model. Note that if you set the $LMC_PATH construct in the configuration file, the .tf files are written instead to the specified directory.
- An SDFFILES.LST file that maps the original SDF file name to the new SDF file name. Note that the new SDF file may have certain SDF constructs commented out to account for timing data that was extracted and back-annotated to the .tf files.

- SDF files that have the extracted timing values commented out.

Copying the Resulting Timing Files (.tf)

You can copy the new .tf files to the appropriate model directories or change your $LMC_PATH environment variable so that the correct .tf files are picked up by the simulator.

Note

If you copy back-annotated .tf files into the model directories, all users of the models will be using the back-annotated .tf files, whether that is their intent or not. If instead you place your back-annotated .tf files into a directory that you then reference with $LMC_PATH, only you are affected.

Replacing the Original SDF Files

To run your simulation with the back-annotated .tf files, replace the original SDF files in your design with the new SDF files (listed in the SDFFILES.LST).

If you use Verilog-XL HDL you can use Verilog-XL preprocessor directives to switch between your original SDF file and the new SDF file as follows:

```
‘ifdef USE_BA
    $sdf_annotate(“mySdf.sdf.new”);
‘else
    $sdf_annotate(“mySdf.sdf”);
‘endif
```

When you want the new “mySdf.sdf.new” to be active, add the following to your original simulation command line:

```
+define+USE_BA
```

10
Library Tools

Introduction

Many users will find that the only tools they need to make effective use of the SmartModel Library are the sl_admin and sl_browser tools, both available in $LMC_HOME/bin. Together, the Admin and Browser tools constitute the primary user interface to the SmartModel Library. That said, there remain some tasks that can only be accomplished using the command-line tools that are also provided with the library in the $LMC_HOME/bin directory. This chapter describes some of the command-line tools, what they are good for, and how to use them. Other command-line tools such as compile_timing are discussed in the context of broader discussions. For example, you can find more information about the compile_timing tool in the chapter about "User-Defined Timing" on page 157. Here's a preview of the miscellaneous tool topics that are covered in this chapter:

- Creating PortMap Files
- Copying Customizable Files with sl_copy
- Translating Memory Image Files
- Adding Back-Annotation
- Checking SmartModel Installation Integrity

For more information on the Admin tool, refer to the *SmartModel Library Administrator's Manual*. Detailed information about the Browser tool is provided in "Browser Tool" on page 43. Refer to "Browsing Your Design Using SmartBrowser" on page 106 for information on using the SmartBrowser command-line tool.

For information on setting environment variables and running command-line tools on NT, refer to "Setting Environment Variables on NT Platforms" on page 43 and "Running Console Applications on NT Platforms" on page 44.

Creating PortMap Files

A PortMap is an ASCII file that describes a SmartModel's interface requirements (for example, the pin porting between a symbol and the model it represents). PortMap files simplify the task of interfacing SmartModels with custom symbols. They are structured for easy parsing, thus providing a convenient source of information you can use in scripts or programs to create or verify custom symbols for use with SmartModels.

You can generate PortMap files using the ptm_make tool that comes with the SmartModel Library in $LMC_HOME/bin. Before running ptm_make, however, please note the following limitations. PortMap files generated from Synopsys data:

- Do not contain all of the data required to produce a high-quality visual symbol representation—they are not useful as a symbol generation database.
- Include only the pins that are used to define the functional description of the model.
- Do not contain printed circuit board (PCB) interface information such as data to drive a physical design system.

Note

The ptm_make tool is not supported for FlexModels.

Using the ptm_make Tool

The ptm_make tool generates PortMap files based on ModelMap data contained in the individual model directories. The tool uses the $LMC_HOME environment variable to locate the models. Models are user-versioned. The ptm_make tool selects model versions based on information in the default or custom .lmc file.

Syntax

PortMap files generated by the ptm_make tool are named *model*.ptm. Run ptm_make from the command line using the following syntax:

```
% ptm_make [model, ...] [-d , path_name] [-h]
```

Arguments

model	Use this optional argument to specify one or more model names. If you do not specify any model names, ptm_make generates PortMap files for all installed models.

Switches

-h[elp]	Specify this switch for help using the ptm_make tool.
-d[ir]	Use this switch to specify a destination directory for the generated PortMap file. If you do not use this switch, ptm_make puts the PortMap file output in the current working directory.

Examples

The following example generates a PortMap file for all models in the library:

```
% ptm_make
```

The following example generates specific PortMap files for the ttl00 and ttl821 models:

```
% ptm_make ttl00 ttl821
```

The following example places the generated PortMap files in the existing directory /user/drj/portmaps:

```
% ptm_make ttl00 ttl821 -d /user/drj/portmaps
```

PortMap File Format

A PortMap file contains a cross-reference of manufacturer pin-to-model signal names for all supported package types, and other data needed for a model to function with a custom symbol. You can use this data to provide the data link between a custom symbol and the corresponding SmartModel. In some environments you might use this data as an interface between a custom symbol and the model, or you could merge the data with your own symbol data to create new symbols.

PortMap files provide data in a format that can be parsed with UNIX tools such as awk and grep if you need to systematically extract information. File names for PortMap files consist of the model name and a .ptm extension. For example, the PortMap file of the ttl2151 model has the name ttl2151.ptm.

A PortMap file consists of a set of records that contain keywords and one or more value fields. Keywords and value fields are separated by vertical bars (|). Legal values for value fields depend on the keyword.

The following illustration shows the general format of a PortMap file. Brackets indicate that the enclosed item is optional; ellipses indicate that the preceding item can be repeated.

```
MODEL | model_name
VERSION | version
FUNCTION | function
SUBFUNCTION | subfunction
RANGE | range
MODELFILE | type [| type ]
. . .
PACKAGE | package_type
       PIN_COUNT | pin_count
       DEVICE/COMP | device | comp
       MODEL_PORT | pin_name | pin_type | pin_number [| pin_number ]
. . .
```

The MODEL, VERSION, FUNCTION, SUBFUNCTION, RANGE, and MODELFILE records can appear only once in a PortMap file. The PACKAGE, PIN_COUNT, DEVICE/COMP, and MODEL_PORT records can appear multiple times depending on how many packages are defined for the associated model. Note that the PIN_COUNT, DEVICE/COMP, and MODEL_PORT records are indented to indicate that they are subordinate to the PACKAGE record. Following are definitions for each of the PortMap file records:

MODEL | *model_name*
: Specifies the model to which the data corresponds. The *model_name* value is the name of a model (for example, ttl00).

VERSION | *version*
: Specifies the version number for the PortMap file. Note that PortMap files generated by the ptm_make utility do not have valid version records.

FUNCTION | *function*
: Specifies the name of a functional category for a model (for example, processor or memory).

SUBFUNCTION | *subfunction*
: Specifies the name of a subfunctional category for a model (for example, dram or sram).

RANGE | *range*
: Specifies the default timing range. Values for range can be MIN, TYP, MAX, or blank (no value). However, the ptm_make tool only generates range values of MAX.

MODELFILE | *type* [| *type*] ...

Specifies the type of additional configuration information the associated model requires. Models that require more than one type of configuration information are listed with multiple type values. Accepted values include PLD (JEDECFile), HVM (PCLFile), MEMORY (MemoryFile), and FPGA (SCFFile).

PACKAGE | *package_type*

Specifies the definition of a package. Accepted values include DIP, SOP, LCC, FLP, PGA, SMT, ZIP, RCC, SDP, and SOJ. The package type has a numeral appended to it to distinguish multiple descriptions of the same basic package type.

PIN_COUNT | *pin_count*

Specifies the total number of physical pins (including VCC, GND, NC, and so on) for the current PACKAGE description.

DEVICE/COMP | *device* | *comp*

device

Specifies the semiconductor vendor's device name (order number) as given in the manufacturer's data book. The device field may contain a vendor-specific extension if two or more vendors have identical ordering information.

comp

Specifies the timing version name. The value corresponds to one of the CASE statement values in the model's timing file.

MODEL_PORT | *pin_name* | *pin_type* | *pin_number* [| *pin_number*] ...

pin_name

Specifies the pin name used by the model to communicate with the simulator about the pin.

pin_type

Specifies the pin type. Accepted values include IN (input), OUT (output), IXO (bidirectional), and IYO (an output pin that must be seen as bidirectional). If the pin_type is IYO, it must be defined on the symbol as an IXO pin. Note that the ptm_make tool labels all IYO pins as IXO pins.

pin_number

Specifies the physical pin number for the current package type. Multiple pin_number values indicate a gate description where the values represent the pin numbers of each gate of the physical package.

Example PortMap File

The following sample shows part of a PortMap file generated by the ptm_make tool for the pal22v10 model. Ellipses (...) indicate places where information has been removed to conserve space. .

```
MODEL|pal22v10
VERSION|...
FUNCTION|prog_logic_devices
SUBFUNCTION|pal24
RANGE|MAX
MODELFILE|PLD
PACKAGE|DIP0
PIN_COUNT|24
DEVICE/COMP|AMPAL22V10-15DC|AmPAL22V10-15
DEVICE/COMP|AMPAL22V10-15DCB|AmPAL22V10-15
. . .
DEVICE/COMP|TICPAL22V10MJT|TICPAL22V10M
DEVICE/COMP|TICPAL22V10MNT|TICPAL22V10M
MODEL_PORT|COFB0|IXO|23
MODEL_PORT|COFB0|IXO|22
. . .
MODEL_PORT|IN8|IN|10
MODEL_PORT|IN9|IN|11
PACKAGE|FLP0
PIN_COUNT|24
DEVICE/COMP|AMPAL22V10-20BKA|AmPAL22V10-20
. . .
DEVICE/COMP|PALCE22V10H-30/BKA|PALCE22V10H-30
MODEL_PORT|COFB0|IXO|23
. . .
MODEL_PORT|IN9|IN|11
PACKAGE|LCC0
PIN_COUNT|28
DEVICE/COMP|AMPAL22V10-15JC|AmPAL22V10-15
. . .
DEVICE/COMP|TIBPAL22V10MFK|TIBPAL22V10M
MODEL_PORT|COFB0|IXO|27
. . .
MODEL_PORT|IN9|IN|13
```

Copying Customizable Files with sl_copy

You can use the Browser tool to copy customizable model source files such as model timing data files one at a time and then modify them to suit your needs. This method is explained in "Copy Customizable Files Dialog Box" on page 67. When you need to copy multiple model source files, you can use the sl_copy tool to do the job. Ordinarily, you could use a UNIX cp command to copy files and then modify them. But with the SmartModel Library, more than one version of the same model can exist in the same installed library ($LMC_HOME). It is therefore easy to inadvertently pick up the wrong version of a model source file. To solve this problem, Synopsys provides the Copy Customizable Files function with the Browser tool and the sl_copy tool for use on the command line.

By default, the sl_copy tool copies *model*.td files that you can then edit to create custom timing versions of models. For more information on creating custom timing versions, refer to "User-Defined Timing" on page 157. You can also use sl_copy to copy command header files (*model*.cmd) for Hardware Verification (HV) models. For more information on HV models, refer to "Processor Models" on page 129.

Syntax

You can run the sl_copy tool from the command line in two different ways, as shown in the following examples:

```
% sl_copy [switches] model_name new_file_name
% sl_copy [switches] model_name [model_name] directory_name
```

Arguments

model_name	Specify a model name whose source file you want to copy. If you specify an output directory using the directory_name argument, you can also specify multiple model names on the same command line.
new_file_name	Specify a full path and file name for the new file to be created if you are copying the source file from just one model.
directory_name	Specify a directory name where you want the copied source files from multiple models to be created if you list more than one model. The directory must already exist for this to work.

Switches

-td	Use this switch if you want to copy a model's timing data (.td) file. This is the file that the tool copies by default if you do not specify otherwise.
-cmd	Use this switch if you want to copy a Hardware Verification (HV) model's command header file (.cmd).
-v	This switch puts the tool in verbose mode, causing it to display the name of each copied file as it is being created.
-h	Use this switch to get a help message about using the tool.

Translating Memory Image Files

SmartModel Library memory models read memory image files (MIF) to configure themselves at simulation startup. The MIF file format does not match other memory image formats created by third parties; specifically, Intel Hex and Motorola S-record formats.

mi_trans

As a convenience, Synopsys supplies a command-line tool called mi_trans (memory image translator) that you can use to convert Intel Hex and Motorola S-record memory image files into the MIF format. The mi_trans tool replaces an earlier translator that you may have used in the past called mkmemimage. The mi_trans tool offers the following features:

- **Intel Hex** – For Intel Hex translations, mi_trans handles both extended segment address records and extended linear address records.
- **Motorola S-record** – For Motorola S-record translations, mi_trans can process input files containing mixed data lengths in different records. The tool recognizes S0, S5, S7, S8, and S9 records anywhere in the file.

Note

Although mi_trans is currently provided for backward compatibility, it has been replaced by an enhanced tool, cnvrt2mif. Use cnvrt2mif instead of mi_trans, for new designs. For syntax and usage, see "cnvrt2mif" on page 200.

Syntax

```
% mi_trans input_file output_image_width [-be] [-single] [-h]
```

Arguments

input_file	Specify a single input file either in Intel Hex or Motorola S-record format.
output_ image_width	Specify the desired width of data in the output file in terms of the number of nibbles, where each nibble equals four bits. The output data width must be less than or equal to the input data width.

Switches

-be	Use this switch if you want your output to be big-endian. If you do not set this switch, mi-trans creates little-endian output.
-single	Use this switch if you want to create a single output file with data in consecutive addresses. (This only applies to the Motorola S-record input format.)
-h	Specify this switch for help using the mi_trans tool.

The mi-trans tool generates one or more SmartModel MIF files named mem.1, mem.2, mem.3, and so on. The first MIF file (mem.1) contains the most significant bits of the data. If the input file data width is not a multiple of the output data width, mi_trans pads the MSB with zeros to make it fit.

Example #1—Input and output data widths match

```
Input data width:
       8 bits per address
Output data width:
       8 bits per address (2 nibbles)

% mi_trans input.data 2
```

In this case, the mem.1 output file data width corresponds exactly to the input data.

Example #2—Input data width a multiple of output data width

```
Input data width:
       16 bits per address
```

```
Output data width:
       8 bits per address (1 nibble)

% mi_trans input.data 1
```

In this case, mi_trans generates four files (mem.1 through mem.4). For each memory address, the tool writes out the most significant four bits in mem.1 and the least significant bits in mem.4.

Example #3—Input data width not a multiple of output data width

```
Input data width:
       18 bits per address
Output data width:
       16 bits per address (4 nibbles)

% mi_trans input.data 4
```

In this case, mi_trans generates a mem.1 file that holds the two MSBs padded with zeros. The mem.2 file contains the lower 16 bits of data.

cnvrt2mif

cnvt2mif replaces mi_trans for converting memory data files to MIF file format. Using cnvt2mif, you can optionally specify the number, name, and bitwidth of output files and perform endian conversion. In addition, you can select data from within the input file by specifying an address range for the conversion. You can also specify a base address for indexed addressing, and control the verbosity and extent of message display. cnvt2mif reads hexadecimal digits in either upper or lower case.

cnvt2mif translates both Intel Hex and Motorola S-record files, as follows:

- **Intel Hex-checksum** – For Intel Hex-checksum translations, cnvt2mif handles both extended segment address records and extended linear address records.
- **Motorola S-record** – For Motorola S-record translations, cnvt2mif can process input files containing mixed data lengths in different records. The tool recognizes S0, S1, S2, S32, S5, S7, S8, and S9 records anywhere in the file. Only S1, S2, and S3 records contain data to place in the output files.

cnvrt2mif returns execution status values as shown in Table 18 on page 201.

Table 18: cnvrt2mif Execution Status Values

Returned Value	Status
0	cnvrt2mif executed successfully with no error or warning messages.
1	cnvrt2mif executed successfully with no errors, but warning messages were generated.
2	cnvrt2mif failed to execute; errors were detected in the command line syntax, the file I/O, or the input file content.

Syntax

```
% cnvrt2mif input_file input_file(s) [ -o out_file_name ] [-ml]
[-devwidth device_width] [-devs num_devices] [-flip16] [-flip32]
[-flip64] [-range lo_adr hi_adr ] [-baseadr base_adr] [-noxsum]
[-wnodup] [-v ] [-V] [-mape] [-h[elp]] [-u[sage]] [-e[xamples]]
```

Arguments and options are order-independent.

Arguments

input_file(s) Specifies a list of one or more (up to 15) input files in Intel Hex-checksum or Motorola S-record format. You can mix input files of these two formats in the same command invocation, unless the files have different endianess. The endian option you use (-flip16, -flip32, -flip64, or none) applies to all input files specified in one command; you cannot specify endianism on a file-by-file basis. If files require different endian options, convert them individually, then concatenate them using (for example) the UNIX cat command, as follows:

```
% cat  file1.mif  file2.mif  file3.mif > final_output.mif
```

Options

-o *out_ file_name* Specifies the base filename of the MIF format output file; the extension .mif is automatically appended to the name. Multiple output files are named *out_ file_name*_00.mif, *out_ file_name*_01.mif, and so on. By default, the output file is named "mem.mif" if this option is omitted.

-ml	Indicates that ASCII hex values in the output file are to be lower case. (The letters "ml" stand for MIF Lowercase.) By default, upper case is used.
-devwidth *device_width*	A positive nonzero integer that specifies the bit width of each memory device, and therefore the bit width of each output file. The default is 16.

Note

Only byte-aligned widths (that is, multiples of 8 bits) are currently supported; for example, 8, 16, 24, 32 bit widths.

-devs *num_devices*	A positive nonzero integer that specifies the number of memory devices for which data files are to be generated. The default is 1.
-flip16	Indicates that a two-byte endian swap is to be performed on each 16-bit data word. Endian adjustments are made before splitting data into output-size words. By default, no endian adjustment is performed.
-flip32	Indicates that a four-byte endian swap is to be performed on each 32-bit data word. Endian adjustments are made before splitting data into output-size words. By default, no endian adjustment is performed.
-flip64	Indicates that an eight-byte endian swap is to be performed on each 64-bit data word. Endian adjustments are made before splitting data into output-size words. By default, no endian adjustment is performed.
-range *lo_adr hi_adr*	A pair of hexadecimal addresses, without a leading "0x", that specifies the limiting address range for data conversion. Only data within that address range, inclusive, is to be converted; data outside the range is ignored. For example, -range 2000 2fff indicates that only data in addresses 2000 through 2fff is to be converted.
-baseadr *base_adr*	A hexadecimal number, without a leading "0x", that specifies the input record address that corresponds to memory index 00000000 in the output files. For an example, see "Example #3—Input data width not a multiple of output data width" on page 200.
-noxsum	Suppresses the validation of checksums of input records. By default, these checksums are validated.

-wnodup	Suppresses the generation of warning messages when multiple input records occupy the same address. By default, these messages are generated.
-v	Indicates that status messages are to be moderately verbose.
-V	Indicates that status messages are to be highly verbose.
-mape	Indicates that error, warning, and status messages are to be issued on stdout instead of stderr. By default, messages are issued on stderr.
-h[elp]	Indicates that the syntax of cnvrt2mif is to be displayed.
-u[sage]	Indicates that basic usage of cnvrt2mif is to be displayed.
-e[xamples]	Indicates that examples of cnvrt2mif usage are to be displayed.

Example #1—Converting S-records into a 32-bit wide single output file with no endian conversion.

The input file srec4 contains two records with data from byte address 400 to 41f.

```
S3 15 00000400 101111231415161718191A1B1C1D1E1F 6E
S3 15 00000410 303132333435363738393A3B3C3D3E3F 5E
```

The following command specifies that the input file srec4 is to be converted to MIF format and written to a single 32-bit wide output file myoutput.mif, with no endian conversion.

```
% cnvrt2mif srec4 -devwidth32 -o myoutput
```

cnvrt2mif automatically recognizes the input file as being in Motorola S-record format. Because -noxsum was not specified, cnvrt2mif validates the checksums on each record. The output file myoutput.mif contains the following data:

```
100 / 10111213;
101 / 14151617;
102 / 18191A1B;
103 / 1C1D1E1F;
104 / 30313233;
105 / 34353637;
106 / 38393A3B;
107 / 3C3D3E3F;
```

Because this is a 4-byte (32-bit) device, the input record addresses (400-41f) have been adjusted to device indexes.

Example #2—Converting S-records into two 16-bit wide output files with endian conversion.

As in Example #1, the input file srec4 contains two records with data from byte address 400 to 41f.

```
S3 15 00000400 101112131415161718191A1B1C1D1E1F 6E
S3 15 00000410 303132333435363738393A3B3C3D3E3F 5E
```

The following command specifies that the input file srec4 is to be converted to MIF. The switch -devwidth 16 indicates that the output files are to be 16 bits wide; -devs 2 indicates that files are to be created for two devices (that is, two files are to be created). -flip32 indicates that there is to be a 32-bit (4-byte) endian conversion.

```
% cnvrt2mif srec4 -devwidth 16 -devs 2 -flip32 -o myoutput
```

The two output files are named myoutput_00.mif and myoutput_01.mif.

myoutput_00.mif contains the following data:

```
100 / 1312;
101 / 1716;
102 / 1B1A;
103 / 1F1E;
104 / 3332;
105 / 3736;
106 / 3B3A;
107 / 3F3E;
```

myoutput_01.mif contains the following data:

```
100 / 1110;
101 / 1514;
102 / 1918;
103 / 1D1C;
104 / 3130;
105 / 3534;
106 / 3938;
107 / 3D3C;
```

Because the two output files comprise a 4 byte wide memory image, the input record addresses (400-41f) have been adjusted to device indexes starting at 100.

Example #3—Converting S-records into one 8-bit wide output file with offset.

In this example, a memory device starts at 0x400 in the address space. The linker has generated an S-record file that contains the real addresses at which the processor in the design will address the contents of memory devices. The required outcome is that the indexes (that is, the internal memory relative addresses) must start at 0x0000, and only the first 8 bytes of the S-record file are to be placed in the MIF output file.

As in Examples #1 and 2, the input file srec4 contains two records with data from byte address 400 to 41f, as follows:

```
S3 15 00000400 101111231415161718191A1B1C1D1E1F 6E
S3 15 00000410 303132333435363738393A3B3C3D3E3F 5E
```

The following command uses the -baseadr 400 option to specify that the image is to be offset by 0x400 bytes, and the -range 400 407 option to specify that only data in the range of input addresses 0x400 and 0x407 is to be included in the output file. The -devwidth 8 option specifies an 8-bit data width for the output file.

```
% cnvrt2mif srec4 -devwidth 8 -baseadr 400 -range 400 407 -o output
```

The MIF format output file is named output.mif and contains the following data:

```
0 / 10;
1 / 11;
2 / 12;
3 / 13;
4 / 14;
5 / 15;
6 / 16;
7 / 17;
```

Notice that the offsets in the output file start at 0 and only the data from input addresses 0x400 to 0x407 is included.

Example #4 —Converting S-records into one 32-bit wide output file with conversion to lower case.

In this example, a post processing tool can read hexadecimal values of A through F only in lower case. Instead of using sed to filter the file, the -ml option of cnvrt2mif generates an output file in lower case.

Note

cnvrt2mif reads either upper or lower case hexadecimal characters from an input file record.

The input file infile2 contains the following data:

```
S1 13 0000 A0B1C2D3E4F5A6B7C8D9E0F1A2B3C4D5 7C
```

The following command generates a single 32-bit output file with alphabetic characters in lower case.

```
% cnvrt2mif infile2 -devwidth 32 -ml
```

Because no output filename was specified, the default filename, mem.mif, is used. The output file contains the following data:

```
0 / a0b1c2d3;
1 / e4f5a6b7;
2 / c8d9e0f1;
3 / a2b3c4d5;
```

Example #5 —Converting S-records into a single output file without checksum validation.

In this example, the input file was generated by a tool that did not compute a correct checksum for each record. In its default mode, cnvrt2mif reads the file and generates a large number of checksum warning messages, which could obscure other, more serious messages that might be generated.

The following command uses the -noxsum option to suppress the checksum validation and the accompanying warning messages.

```
% cnvrt2mif input_7.srec -noxsum -o myoutput
```

If invalid checksums are the only warning conditions detected, using the -noxsum option causes cnvrt2mif to return an execution status value of 0, whereas with checksum validation enabled, cnvrt2mif returns an execution status value of 1.

Note

Although this example shows the use of the -noxsum option, you should use it only sparingly, and only after you have attempted to locate and correct the problem in the S-record or Intel Hex file generator. Disabling checksum validation could result in invalid data being placed in the output .mif file and subsequently loaded into your memories.

Adding Back-Annotation

You can extract back-annotation timing data from Standard Delay Format (SDF) files using the Backanno tool and annotate them to the SmartModel timing data format. The Backanno tool creates new user SDF files and compiled timing files (.tf).

Syntax

Run the backanno tool from the command line as shown in the following example:

```
% $LMC_HOME/bin/backanno configFile
```

Argument

configFile	Specifies the configuration file that controls the back-annotation process.

For more information on using Backanno, including details about the configuration file format, refer to "Back-Annotating Timing Files" on page 181.

Checking SmartModel Installation Integrity

If you encounter unexplained problems while working with SmartModel Library models, it could be that the underlying cause is a faulty installation. To help diagnose such problems, Synopsys provides a tool called swiftcheck. The swiftcheck tool can identify many common installation problems by:

- Verifying that environment variables are properly set
- Checking that the SmartModel Library is properly installed
- Loading a user-specified model and exercising basic functionality

The swiftcheck tool reports the values of these environment variables:

- $LMC_HOME (required)
- $LMC_PATH
- $LMC_COMMAND
- $LM_LICENSE_FILE
- $LD_LIBRARY_PATH (SunOS only)

The swiftcheck tool verifies the values of these environment variables and produces an error message if it cannot find any of them. In particular, you will get a fatal error if the required variable $LMC_HOME is not set.

To verify the installation of your SmartModel product, swiftcheck searches for the necessary runtime utilities, using the $LMC_HOME environment variable for the path.

The swiftcheck tool enables you to specify a model that it will load and attempt to exercise. The swiftcheck tool loads all of the specified model's timing files and, if specified, custom configuration file(s). By loading and initializing the model, swiftcheck effectively tests the value of the $LMC_PATH environment variable and provides you with a simple way to test configuration files.

Because SmartModels do not require a configuration for initialization, swiftcheck loads and exercises the selected model even if you do not specify a configuration file. However, if swiftcheck cannot find the timing files for a model, it will not load or initialize that model.

The swiftcheck tool displays important messages (such as fatal errors) on your screen when the errors occur. In addition, swiftcheck places all messages, regardless of severity, in a log file named swiftcheck.out.

Syntax

Run the swiftcheck tool from the command line as shown in the following example:

```
% swiftcheck model [-switches]
```

Argument

model	Specify the installed model that you want the tool to load and exercise as a test of basic installation integrity.

Switches

-e[rrorlog] *filename*	Use this switch to a specify an output file for the error log other than the default of swiftcheck.out.
-h[elp]	Specify this switch for help using the swiftcheck tool.
-hh[elp]	Specify this switch to print a more detailed message about using the swiftcheck tool.
-j[edecfile] *filename*	Load the specified configuration file for the JEDEC model.
-m[emoryfile] *filename*	Load the specified configuration file for the memory model.
-n[omodels]	Use this switch if you want to run swiftcheck but do not want the tool to load and exercise a model.
-p[clfile] *filename*	Load the specified configuration file for the HV model.

-t *timing_version*	Load a particular timing version for the specified model.
-u[sage]	Another switch that you can specify for help using the swiftcheck tool.

Examples

The following example invocations show how to invoke the swiftcheck tool in several different ways. The first example causes swiftcheck to load and exercise the ttl00 model, which does not require a configuration file:

```
% swiftcheck ttl00
```

This next example causes swiftcheck to load and exercise the am2168 memory model using the MIF file called *my_memfile* to configure the model:

```
% swiftcheck -m my_memfile am2168
```

The next example causes swiftcheck to load and exercise the mc68332_hv hardware verification model using the MIF file called *my_memfile* and the PCL file called *my_pclfile* to configure the model:

```
% swiftcheck -m my_memfile -p my_pclfile mc68332_hv
```

A
Reporting Problems

Introduction

If you think a SmartModel is not working correctly, check with your System Administrator to see if you are using the latest version. It is possible that a more recent version of a model has the fix you need. Significant model changes are documented in the model history section at the end of each model's datasheet.

First, verify the version number of the model using the Browser tool ($LMC_HOME/bin/sl_browser) to access the model datasheet. The title banner at the top of all SmartModel datasheets lists the model's MDL version number. Then compare reported fixes for subsequent versions of that model by reading the model history section at the end of the latest datasheet on the Model Directory:

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

For more information on model history, refer to "Model History and Fixed Bugs" on page 214.

If you cannot find a more recent model version that solves the problem, contact Customer Support. For details on how to get in touch with Synopsys, refer to "Getting Help" on page 15.

Using Model Logging

Use this method of model logging if the model is a

- SmartModel
- FlexModel with the _fz suffix
- FlexModel with the _fx suffix that *also* has a model history entry with the reference number 54399 in its datasheet Model History section, and you or Technical Support know the specific instance that is causing the problem. If more than one instance might be the source of the problem, you should not use this logging method.

  All other FlexModels with the _fx suffix must have their problems logged using the procedure described in Creating FlexModel Log Files in the *FlexModel User's Manual*.

Before you contact Technical Support, generate a model logging file (mlog.log). Model logging captures all of a model's activity during simulation (that is, stimulus and response) in ASCII text format. Transmitting an mlog.log file to Technical Support will help ensure accurate diagnosis of the problem. Only one instance of one model can be logged at any one time and system performance degrades when you use model logging.

To generate a mlog.log file, create a file called mlog.cfg in the directory where you run the simulator. All models look for this file and read its contents, if it exists, to determine which model to log. If you need to log more than one model, see "Logging Multiple Instances" on page 213.

You can select a model for model logging in any of the following ways:

- Create an empty mlog.cfg file. If you do not specify a particular model, the first SmartModel loaded in a circuit is logged. This is handy if you have only one model in the design.
- Specify a model by its model name. Put a line in the mlog.cfg file that follows this case-sensitive convention:

  ```
  %m model_name
  ```

  For example:

  ```
  %m mc68030_hv
  ```

  This causes the first model of that name to be logged. This is a good method to use when the design has only one instance of a particular model type.

- Specify a model by its instance name. Put a line in the mlog.cfg file that follows this case-sensitive convention:

  ```
  %i <scope><instance_name>
  ```

  For example:

  ```
  %i u100
  ```

  The instance with instance name u100 is logged. Note that for some simulators the instance name includes the instance path.

During simulation, the specified model creates a file named mlog.log. This file contains all of the stimulus and response recorded at the model's ports during simulation.

Logging Multiple Instances

If you need to log more than one instance of a model in the design, reset the instance name specified in the mlog.cfg file and rerun the simulator for each instance you want to log. Remember to save the mlog.log output file to another location prior to running the simulator.

Sending the Log Files to Customer Support

After you rerun your simulation to generate the model log files, tar those files, zip the tarball up using gzip, and send the zipped log files to Customer Support as an e-mail attachment. Include your call number if you have one and a description of the problem in the body of your message.

Other Diagnostic Information

Depending on the type of model, the following information may be required in addition to the mlog.cfg file:

- **Hardware verification models.** Send the PCL source program, and any other files required for compilation.
- **PLD models.** Send the source JEDEC program files.
- **Memory models.** Send the memory image files.
- **SmartCircuit models**. Send the netlist description files necessary to create a CCN file, and your MCF file. Also provide the version numbers of any third-party tools you used, and the version number of smartccn.

Please use e-mail to send the test data described above for the type of model you are using. In all communications to the Synopsys Technical Support Center, please include a phone number where we can reach you.

Model History and Fixed Bugs

At the end of each SmartModel datasheet is a model history section detailing significant model changes that occurred during the past year. If the model has not changed significantly in a year, its datasheet does not contain any model history entries. Significant changes cause the model to behave differently in simulation. Of course, this includes all model bug fixes. For information about gaps in model version numbers, refer to "MDL Version Numbers and Model History" on page 26.

Each change entry in the model history includes the:

- Reference number
- MDL version of the model after the change
- MDL date of the change
- Problem and resolution descriptions

Model history entries look like the following example.

```
-----------------------------------------------------------------
Reference:: 41087

MDL Version:: 01002
MDL Date:: 13-June-1996

SRC Version:: v1.1

Problem::    The minimum high/low pulse width for CCLK in
             synchronous peripheral mode did not conform to revised
             vendor specifications.

Resolution:: Corrected the model.
-----------------------------------------------------------------
```

Model History Entry Field Descriptions

The "Reference::" Field

The "Reference::" field contains the internal number assigned to the specific change.

The "MDL Version::" Field

The "MDL Version::" field contains the model version after the change. Not all MDL version number changes are significant. Only changes such as bug fixes that affect model behavior are considered significant and generate model history entries. That's why the model MDL version number listed in the title banner on the first page of a datasheet can be a higher number than the MDL version number listed in the latest model history entry for a model.

The "MDL Date::" Field

The "MDL Date::" field contains the publication date for the corresponding MDL Version of a model.

The "SRC Version::" Field

The "SRC Version::" field contains the internal model source code version after the change. Because not all MDL Version changes for a model involve changes to the source code, the same SRC Version number can appear in multiple model history entries for different MDL Version numbers.

The "Problem::" and "Resolution::" Fields

The "Problem::" and "Resolution::" fields briefly describe the user-visible symptoms of the problem and, if appropriate, what was changed to correct it.

B
Glossary

Introduction

Following are definitions of some terms that have special meaning in the context of using the SmartModel Library.

Configuration. A platform-specific set of SmartModel Library models and user-versioned tools, with one version number specified for each.

Configuration (LMC) File. A file that contains a configuration; that is, a platform-specific set of SmartModel Library models and user-versioned tools, with one version number specified for each. Configuration files have .lmc extensions.

Custom Configuration. A user-specified, platform-specific set of SmartModel Library models and user-versioned tools, with one version number specified for each. Overrides model and tool versions specified in the default configuration.

Custom Configuration (LMC) File. A file that contains a custom configuration; that is, a user-specified, platform-specific set of SmartModel Library models and user-versioned tools, with one version number specified for each. Custom configuration files have .lmc extensions.

Datasheet. A document that describes a model in the SmartModel Library, including its sources, supported hardware components and devices, programming, use, timing parameters, and any differences between the model and the corresponding hardware part.

Default Configuration. A system-specified, platform-specific set of SmartModel Library models and user-versioned tools, with one version number specified for each; used if no other user-specified configuration exists.

Default Configuration (LMC) File. The file that contains the default configuration; that is, a system-specified, platform-specific set of SmartModel Library models and user-versioned tools, with one version number specified for each. Supplied with the SmartModel Library.

LD_LIBRARY_PATH . For Sun operating system only. An environment variable that contains the path to Sun libraries that are to be executed.

LMC_COMMAND. An environment variable that contains a semicolon-separated list of session commands to be used during simulation.

LMC_CONFIG. An environment variable that contains a colon-separated list of paths to user-specified configuration (LMC) files. For NT, path entries must be separated by semicolons.

LMC_HOME. An environment variable that contains the path to the SmartModel installation tree.

LMC_PATH. An environment variable that contains a colon-separated list of paths to user-specified model timing files. For NT. path entries must be separated by semicolons.

LM_LICENSE_FILE. An environment variable that contains the path to a FLEXlm license file. For NT, path entries must be separated by semicolons.

LMC or .lmc file. A configuration file. "LMC" stands for "List of Model Configurations". An LMC file must have the .lmc extension.

Model. A behavioral software representation of a standard integrated circuit.

Model Name. A string of alphanumeric characters that identifies a specific model in the SmartModel Library (for example, am2168, dflipflop, or i80c31).

Model Report. One of the three different reports that you can generate using the Browser tool.

Model Version. A string of numbers that identifies a specific version of a model in the SmartModel Library (for example, 01000, 01003, or 01012).

Model-versioned Tool. A tool whose version number is specified by the model and cannot be specified by the user. Examples of model-versioned tools include compile_pcl and compile_timing.

Platform. The workstation on which the SmartModel Library is to be installed (for example, hp700, sunSunOS, or pcnt).

Predefined Window Element. A window element created by Synopsys and supplied with a specific model.

SmartModel Library. A collection of behavioral simulation models of standard integrated circuits, designed to be used in EDA simulation environments that use the SWIFT interface.

.td file. A source timing version file.

.tf file. A timing version file that has been compiled and is ready for simulation.

Timing File. A file that contains timing parameters for a SmartModel.

Timing Version. A SmartModel representation that specifies model timing parameters. Each timing version has a unique name.

User-defined Window Element. A window element created by a user.

User-versioned Tool. A tool whose version number can be specified by the user, usually by placing it and its version number in a custom configuration file. Examples of user-versioned tools include ptm_make, mi_trans, and swiftcheck.

Window. A view through which you can access one or more of a model's internal registers.

Window Element. A window with a specific name, created to monitor a specific register or memory array.

Index

D

E

F

G

H

I

J

K

L

M

N

S

T