Open Simulation System Reference

Product Version ICADVM20.1
October 2020

Contents

Preface

Open Simulation System (OSS) provides access to the simulators that it supports and lets you integrate and customize new simulators into the Virtuoso design environment. This reference guide describes the components of OSS and explains how to integrate and customize simulators. The guides also explains how to customize hierarchical netlister (HNL), HNL net-based netlister, flat netlister (FNL), and post layout simulation. In addition, the guide explains the process to maintain a design library for simulation.

This reference guide is aimed at designers of integrated circuits and assumes that you are familiar with:

- The Virtuoso design environment and application infrastructure mechanisms designed to support consistent operations between all Cadence tools.
- The applications used to design and develop integrated circuits in the Virtuoso design environment, notably Virtuoso Layout Suite and Virtuoso Schematic Editor.
- The design and use of parameterized cells.
- Component Description Format (CDF), which lets you create and describe your own components for use with ADE.

This preface contains the following topics:

Scope

Unless otherwise noted, the functionality described in this guide can be used in both mature node (for example, IC6.1.8) and advanced node (for example, ICADVM18.1) releases.

Label	Meaning
(ICADVM18.1 Only)	Features supported only in the ICADVM18.1 advanced nodes and advanced methodologies release.
(IC6.1.8 Only)	Features supported only in mature node releases.

Licensing Requirements

Hierarchical and Flat Netlisting

You must have one of the following licenses to run OSS from simulation environment (using the `si` command or composer interface) or to integrate third-party netlisters, simulators, or formatters into OSS. If one of these licenses is not already checked out, the first available license will be checked out in the following order when you run OSS:

- 95100 Virtuoso® Schematic Editor L
- 95115 Virtuoso® Schematic Editor XL
- 206 Virtuoso® Simulation Environment

If you are using licensed Cadence tools, which use OSS functionality, then the OSS will not check out any license. If you have already paid for third party integration through other Cadence tools then also you do not require a separate SE license. For example, if you have the license for ADE integration for third-party simulators then you do not require the SE license.

prFlatten, auLvs, and auErc Flat Netlisting

You must have either of the following licenses for prFlatten, auLvs, and auErc flat netlisting. If one of these licenses is not already checked out, the first available license will be checked out in the following order when you run OSS:

- 95310 Virtuoso® Layout Suite XL

- 95323 Virtuoso® Layout Suite - GXL

For the 95323 Virtuoso® Layout Suite - GXL license, four tokens are required.

For Verilog netlisting in the simulation environment, (using the si command or composer interface), you need only the Virtuoso® Schematic Editor Verilog ® Interfaces license (license number 21400). A separate Schematic Editor license is not required to run Verilog netlister.

Queue-Based License Checks for Verilog Netlisting Processes

Using a license checking utility, you can queue Verilog netlist processes when all the available licenses are being used.

To enable the license checking utility:

➡ Set the environment variable `CDS_LIC_QUEUE_ENABLE` to `1`.

To enable the license checking utility when licenses are on multiple servers:

➡ Set the environment variable `CDS_LIC_QUEUE_POLL` to `1`.

The license checking utility waits for a minimum of ten minutes before checking the availability of licenses for queued processes. The process remains in queue only during the wait period. You can increase this wait period.

To increase the wait period of the license checking utility:

➡ Set the environment variable `CDS_LIC_QUEUE_MAXTIME_INT` to the wait period. For example, to increase the wait period to 15 minutes, set the environment variable to 15.

```
setenv CDS_LIC_QUEUE_MAXTIME_INT 15
```

Note: You can only increase the wait period from the default ten minutes. If you set `CDS_LIC_QUEUE_MAXTIME_INT` to a value less than ten, the license checking utility still waits for ten minutes, and not to the set value.

The license checking utility displays the following message every minute, till a license becomes available or the wait period is complete.

```
Waiting for license feature 21400...
```

The utility displays the following message when the wait period is over and if during this period no license became available to a process. After the wait period, the process is no longer in the queue.

```
Max Queue wait time reached for license feature  21400
```

Related Documentation

What's New and KPNS

- *Open Simulation System What's New*
- *Open Simulation System Known Problems and Solutions*

Installation, Environment, and Infrastructure

- *Cadence Installation Guide*.
- *Virtuoso Design Environment User Guide*.
- *Virtuoso Design Environment SKILL Reference*.
- *Cadence Application Infrastructure User Guide*.

SKILL Reference

- *Digital Design Netlisting and Simulation SKILL Reference*.

Virtuoso Tools

IC6.1.8 Only

- *Virtuoso Layout Suite L User Guide*
- *Virtuoso Layout Suite XL User Guide*
- *Virtuoso Layout Suite GXL Reference*

ICADVM18.1 Only

- *Virtuoso Layout Viewer User Guide*
- *Virtuoso Layout Suite XL: Basic Editing User Guide*
- *Virtuoso Layout Suite XL: Connectivity Driven Editing Guide*
- *Virtuoso Layout Suite EXL Reference*
- *Virtuoso Concurrent Layout User Guide*

- *Virtuoso Design Planner User Guide*
- *Virtuoso Multi-Patterning Technology User Guide*
- *Virtuoso Placer User Guide*
- *Virtuoso Simulation Driven Interactive Routing User Guide*
- *Virtuoso Width Spacing Patterns User Guide*
- *Virtuoso RF Solution Guide*
- *Virtuoso Electromagnetic Solver Assistant User Guide*

IC6.1.8 and ICADVM18.1

- *Virtuoso Abstract Generator User Guide*
- *Virtuoso Custom Digital Placer User Guide*
- *Virtuoso Design Rule Driven Editing User Guide*
- *Virtuoso Electrically Aware Design Flow Guide*
- *Virtuoso Floorplanner User Guide*
- *Virtuoso Fluid Guard Ring User Guide*
- *Virtuoso Interactive and Assisted Routing User Guide*
- *Virtuoso Layout Suite SKILL Reference*
- *Virtuoso Module Generator User Guide*
- *Virtuoso Parameterized Cell Reference*
- *Virtuoso Pegasus Interactive User Guide*

Additional Learning Resources

Video Library

The Video Library on the Cadence Online Support website provides a comprehensive list of videos on various Cadence products.

To view a list of videos related to a specific product, you can use the *Filter Results* feature available in the pane on the left. For example, click the *Virtuoso Layout Suite* product link to view a list of videos available for the product.

You can also save your product preferences in the Product Selection form, which opens when you click the *Edit* icon located next to *My Products*.

Virtuoso Videos Book

You can access certain videos directly from Cadence Help. To learn more about this feature and to access the list of available videos, see Virtuoso Videos.

Rapid Adoption Kits

Cadence provides a number of Rapid Adoption Kits that demonstrate how to use Virtuoso applications in your design flows. These kits contain design databases and instructions on how to run the design flow.

In addition, Cadence offers the following training courses of interest:

- Virtuoso Schematic Editor
- Virtuoso Analog Design Environment
- Spectre Circuit Simulator
- Spectre Simulations Using Virtuoso ADE
- Virtuoso Simulation for Advanced Nodes

To explore the full range of training courses provided by Cadence in your region, visit Cadence Training or write to training_enroll@cadence.com.

Note: The links in this section open in a separate web browser window when clicked in Cadence Help.

Help and Support Facilities

Virtuoso offers several built-in features to let you access help and support directly from the software.

- The Virtuoso *Help* menu provides consistent help system access across Virtuoso tools and applications. The standard Virtuoso *Help* menu lets you access the most useful help and support resources from the Cadence support and corporate websites directly from the CIW or any Virtuoso application.

- The Virtuoso Welcome Page is a self-help launch pad offering access to a host of useful knowledge resources, including quick links to content available within the Virtuoso installation as well as to other popular online content.

  The Welcome Page is displayed by default when you open Cadence Help in standalone mode from a Virtuoso installation. You can also access it at any time by selecting *Help – Virtuoso Documentation Library* from any application window, or by clicking the *Home* button on the Cadence Help toolbar (provided you have not set a custom home page).

For more information, see Getting Help in *Virtuoso Design Environment User Guide.*

Customer Support

For assistance with Cadence products:

- Contact Cadence Customer Support

  Cadence is committed to keeping your design teams productive by providing answers to technical questions and to any queries about the latest software updates and training needs. For more information, visit https://www.cadence.com/support

- Log on to Cadence Online Support

  Customers with a maintenance contract with Cadence can obtain the latest information about various tools at https://support.cadence.com

Feedback about Documentation

You can contact Cadence Customer Support to open a service request if you:

- Find erroneous information in a product manual
- Cannot find in a product manual the information you are looking for
- Face an issue while accessing documentation by using Cadence Help

You can also submit feedback by using the following methods:

- In the Cadence Help window, click the *Feedback* button and follow instructions.

- On the Cadence Online Support Product Manuals page, select the required product and submit your feedback by using the *Provide Feedback* box.

Typographic and Syntax Conventions

The following typographic and syntax conventions are used in this manual.

text	Indicates names of manuals, menu commands, buttons, and fields.
`text`	Indicates text that you must type exactly as presented. Typically used to denote command, function, routine, or argument names that must be typed literally.
`z_argument`	Indicates text that you must replace with an appropriate argument value. The prefix (in this example, *`z_`*) indicates the data type the argument can accept and must not be typed.
|	Separates a choice of options.
`{ }`	Encloses a list of choices, separated by vertical bars, from which you **must** choose one.
`[ ]`	Encloses an optional argument or a list of choices separated by vertical bars, from which you **may** choose one.
`[ ?argName` *`t_arg`* `]`	Denotes a *key argument*. The question mark and argument name must be typed as they appear in the syntax and must be followed by the required value for that argument.
`...`	Indicates that you can repeat the previous argument. Used with brackets to indicate that you can specify zero or more arguments. Used without brackets to indicate that you must specify at least one argument.
`,...`	Indicates that multiple arguments must be separated by commas.
`=>`	Indicates the values returned by a Cadence® SKILL® language function.
`/`	Separates the values that can be returned by a Cadence SKILL language function.

If a command-line or SKILL expression is too long to fit within the paragraph margins of this document, the remainder of the expression is moved to the next line and indented. In code excerpts, a backslash (\ ) indicates that the current line continues on to the next line.

1

Introducing the Open Simulation System (OSS)

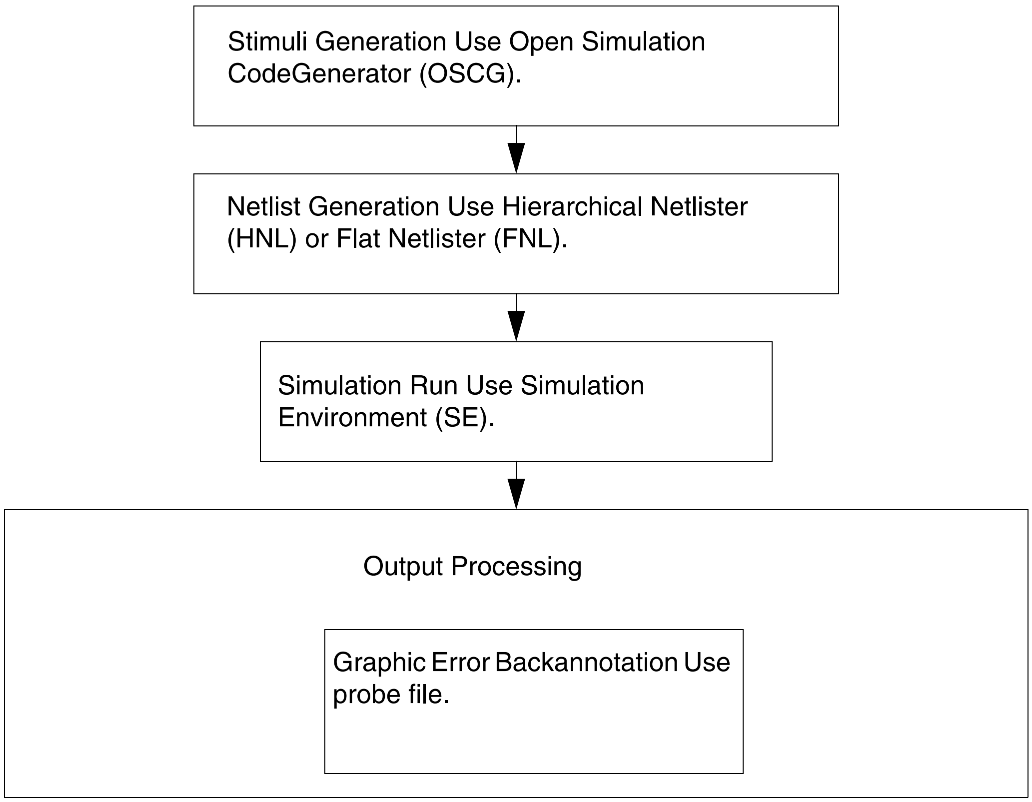
The Open Simulation System provides the foundation for Cadence’s simulation strategy that lets you integrate simulators into the Cadence system. Simulators integrated using OSS present a consistent user interface for controlling execution of simulation and generation of netlists and input vectors.

The *Open Simulation System Reference* manual provides the information needed by a CAD developer to integrate a simulator into the Cadence system. In this chapter you can find information on the following topics:

OSS Components Overview

The simulation process can be broken down into several steps. These steps and the tools provided in OSS to implement them are outlined in Figure 1-1 on page 24.

Figure 1-1 Simulation Process



Simulator Integration Overview

To integrate a simulator into the Cadence system, you must customize the tools required by your specific application. Depending on the application, you might not need all of the tools provided. For example, if your application require a stimuli, then, you need to develop the stimuli generator. Now onwards, OSS will not support any simulation test language. The following is a brief description of the simulation process stages and the tools provided in OSS to simplify the process.

1. Create an STL code generator for the simulator.

   The Open Simulation Code Generator (OSCG) functionality has been removed from OSS. The customer needs to develop its own code generator.

2. Create the appropriate netlist.

Before creating the appropriate netlist, you need to decide whether to create a flat or a hierarchical netlist. Accordingly, the Cadence HNL or FNL is used. To customize a netlist, each cell in the library needs to have a view of itself that guides the netlister in terms of the properties that need to be extracted and their format in the netlist.

3. Customize SE.

   SE controls the simulation execution, including invoking the simulator and the netlister as well as loading the data Therefore, you need to modify SE so that it recognizes the new simulator and creates the control files for the specific simulator.

4. Create an error backannotation file.

   A mechanism called probing is used throughout the design analysis process. Probing is a way of graphically highlighting nets (nodes) and instances (devices) in your display. A file format is provided for textually specifying which devices and nets should be probed. With this format, you can store errors in a file, associate any error text with each device, and display the errors graphically at a later time in the design analysis process. This file format is called the *probe file format* and is described in the “Probe File Format” section of the “Graphics Editor” chapter of the Virtuoso Design Environment SKILL Reference.

Designer and Developer Perspectives

The designer and the developer use OSS differently. The designer is primarily concerned with the user interface, the processes of running simulation, and studying simulation output. The developer needs to know substantially more about OSS internal architecture.

64-bit OSS

OSS is now available in the 64-bit version also. The 64-bit OSS is available in same hierarchy as the 32-bit OSS is. Similarly both versions (32/64 bit) of all the applications using OSS are available in the same hierarchy. Therefore, a change is required in the use model of these applications.

As per the new use model, there are two sub-directories called `32bit`and `64bit`present under the `bin` directory. The bin directory now contains a wrapper with the same name as that of the binary. You need to call this wrapper and depending upon the environment settings the appropriate version of the binary is invoked.

The directory structure is as follows:

wrapper to binary	**bin/<app>.exe**
32-bit version of binary	bin/32bit/<app>.exe
64-bit version of binary	bin/64bit/<app>.exe

64-bit si

The non graphics simulation environment `si` is also available on both the platforms namely, 32bit and 64bit. Under this new use model the configuration is as follows:

32/64-bit wrapper	**/bin/si**
32-bit binary	/bin/32bit/si
64-bit binary	/bin/64bit/si

The wrapper decides on the version of the application to be executed based on the following conditions.

```
IF the operating system supports 64-bit applications
AND a 64-bit version of the application exists
AND the user elects to use the 64-bit version
   THEN execute the 64-bit application,
OTHERWISE execute the 32-bit version.
```

The user elects to use the 64-bit version of an application by setting an environment variable $CDS_AUTO_64BIT.

The variable can be set to the following values:

ALL	**all applications are run as 64-bit where possible**
NONE	all applications are run as 32-bit
<list>	A list of case-sensitive application names (delimited by space, semi-colon or colon) and only these applications will be run as 64-bit.

Note: For information on using the `simIlSleep()` function in replay mode, see the *Running si in Replay Mode* section in Chapter 1 of Simulation Environment Help.

2

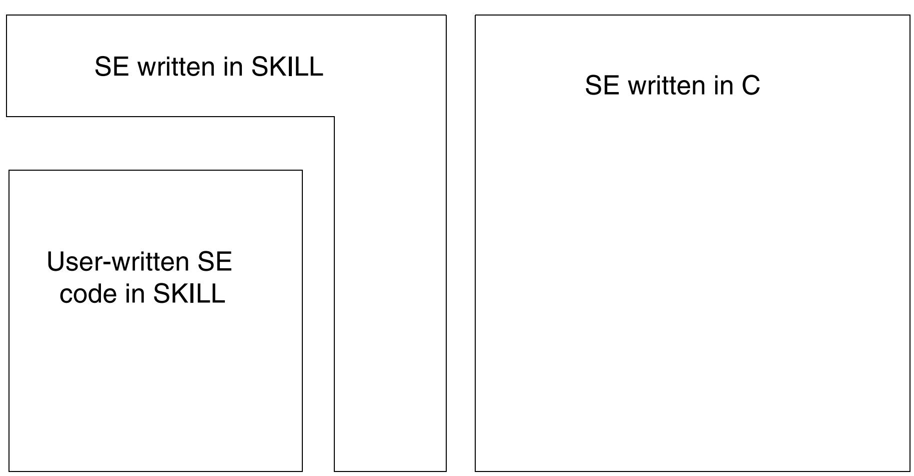
Integrating Simulators

In this chapter, you can find information about the following topics:

- Requirements on page 28
- Simulation Steps on page 29
- Integration Steps on page 35

From the user perspective, a simulation in the Cadence system can be run with a single command. Figure 2-1 shows the structure of the Simulation Environment (SE), which allows integrated simulators to have the same single-step simulation interface to your users. This simple interface can be provided by SE, the controlling environment used to run simulations.

Figure 2-1 The Simulation Environment (SE)



SE is a non-graphics program that uses the SKILL language, the same language and syntax known as SKILL™ in the Cadence graphics program, as its interface. The difference between SKILL in the Cadence graphics program and SKILL in SE is the extensions that have been made to them in each program. In SE, the SKILL language has been extended with several commands, or functions, that are specific to running simulations and generating netlists. In the Cadence graphics program, the SKILL language has been extended to contain commands for graphic access and database manipulation. These are not needed and not available in SE. The syntax of the language is the same in both programs, and the same basic commands are available in both.

Requirements

Before you attempt to integrate your simulator into the Cadence system, you must have certain background information.

SKILL

You must have a working knowledge of SKILL because many of the tools you are using can be modified using the Cadence standard language*,* SKILL. Before continuing, you should read the *Cadence SKILL Language User Guide*. Then, write some small SKILL programs to make sure you understand the language and its syntax. You can load and run your programs using SE or SKILL in the Cadence graphics environment.

Database Access Functions (SKILL Database Access)

You should have a good understanding of the Cadence database structure, as described in the Virtuoso Design Environment SKILL Reference*,* and the functions available to read the database.

Note: SE does not allow database modification.

The Cadence tools make it possible for you to interface most simulators without knowing any of the database structure. With database structure knowledge you can develop a better and more efficient interface. It is strongly recommended that you have such background knowledge before you attempt to integrate your simulator into the Cadence system. After you have read the database access functions documentation, write a few small functions to display textual information about a design you create. For example, open a design and make a parts list, or print the names and widths of all of the terminals and nets in the top-level design. You can load and run your programs using SE or SKILL in the Cadence graphics program.

Running Simulation Using SE

To be able to develop an interface, you must understand what you want the interface to look like and what tools are available to you. The simplest way to do this is to use an existing interface. First, read the *Simulation Environment Help*. Next, create a small schematic, simulate it, and analyze the results in text.

Creating Stimuli Using STL

Since the Simulation Test Language (STL) functionality has been removed from OSS, customers need to develop their own simulation test and data.

Developing a Simulator Interface

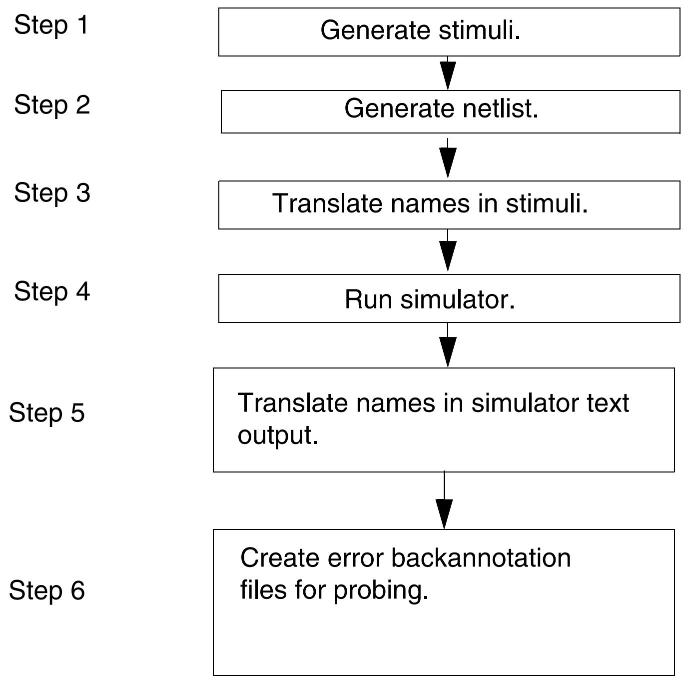
After you have read the recommended chapters and completed the suggested exercises, you should have enough background information to develop your simulator interface. The rest of this section explains the basic structure and interrelationship of the tools provided in the OSS and customizing each tool to your specific needs. When you develop an interface, you are not simply creating separate and distinct tools for your designers to use. Rather, you are creating an integrated design analysis system. Therefore, a description of how each tool can be modified is not enough. You need the steps to integrate these tools and produce a single unified interface. The remainder of this chapter explains the various simulation processes performed in the Cadence system and their interrelationships. Detailed explanations for each tool are in separate chapters in the remainder of this manual.

Simulation Steps

To decide which tools you want to use to develop your interface, you must first understand the operation, function, and features of each tool as it relates to simulation as well as the simulation flow in the Cadence system.

Figure 2-2 illustrates the steps performed during a simulation.

Figure 2-2 Simulation Steps



The input preparation and simulation steps must be performed in the order shown in Figure 2-2; however, the post-processing steps can be performed in any order. The following sections describe the function of the tools in performing each of the corresponding steps in the diagram. In addition to explaining what the tool does, there is an explanation of why the tool was designed to function in this manner. This background information should help you decide how best to use each of the tools.

Generating Stimuli

Most design analysis tools require a set of vectors or input patterns to drive the analysis or simulation, and these stimuli must be provided before you can run a simulation. The designer running a simulation can produce these manually.

Generating a Netlist

All design analysis tools require a description of the connectivity for the design to be analyzed. Since non-Cadence tools are not able to read the Cadence database directly, a textual description (or netlist) of the connectivity must be produced. In addition to connectivity, the netlist may contain information such as model descriptions for devices used (required by such simulators as SPICE™), delay information for devices used (such as rise and fall times), and node settings for constant nodes (such as *VDD* and *GND*).

Netlist Formatting

Each design analysis tool has its own input syntax for connectivity information. Therefore, as part of OSS, you are able to modify the output syntax produced by the tools that generate these textual connectivity descriptions. Two tools provided to generate netlists are the Flat Netlister (FNL) and the Hierarchical Netlister (HNL). Both of these tools generate textual network descriptions and can be modified in a similar manner. The difference between them is in the type of output produced. (Refer to the “Hierarchical and Flat Netlisting” section in this chapter.)

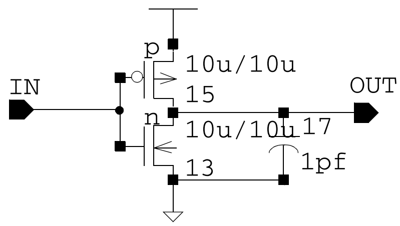
Both HNL and FNL allow you to format the netlist output using the Cadence standard language, SKILL. In addition to the basic commands available in SKILL, you can use the database access functions, as well as specific netlister-defined functions that simplify the netlisting process.

In addition to SKILL formatting, FNL provides its own formatting language. This is a fast and compact language consisting of predefined variable names and substitution expressions. It is a superset of the syntax used for interpreted labels in schematic entry and requires no database knowledge to use. You can customize the netlister output using either language (SKILL or FNL formatting), or a blend of both. This formatting ability is only available in FNL and is explained in detail in the “Customizing the Flat Netlister (FNL)” chapter in this manual.

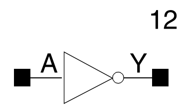
Hierarchical and Flat Netlisting

The Cadence design entry system allows the designer to enter designs hierarchically. This means that the designers only need to enter a schematic once and can then reference, or use, that schematic in subsequent designs. FNL “flattens” (or expands) the design hierarchy before outputting the connectivity information. By flattening the design, the netlister replaces all symbols with the connectivity in the schematics they represent. Refer to Figure 2-3 on page 32. If a schematic for an inverter consists of a pair of transistors and a capacitor (a), and if another schematic contains two symbol views (b) of these inverters (c), the flattened netlist produced by FNL will contain four transistors, two capacitors, and the connectivity describing their interconnections as represented by the schematic in (d).

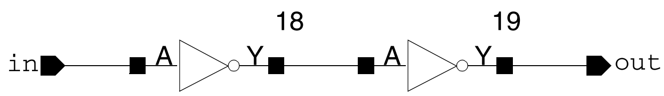
Figure 2-3 Example of Flattened Design



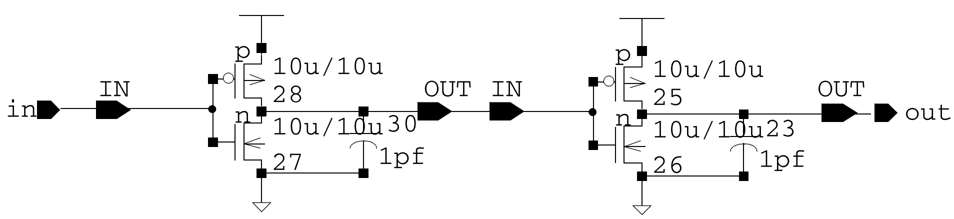
(a) inv/schematic



(b) inv/symbol



(c) inverter chain schematic



(d) inverter chain schematic as presented by the Flat Netlister (FNL)

HNL does not flatten the hierarchy before outputting the connectivity. Instead, it produces a hierarchical netlist. Therefore, it outputs the connectivity for low-level cells first, then the connectivity for cells that reference them. In the example in the previous paragraph, a description of the schematic for an inverter is output first, and then a description of the schematic that references the inverters. Instead of outputting the connectivity for the inverter for each reference to it, the connectivity is only output once, and this description is then referenced in subsequent uses of the inverter.

Not all simulators are able to read in a hierarchical netlist. If your simulator can only read in a flat network description, use FNL to produce the netlist for your simulator. If your simulator can read in either type of netlist, you can use either HNL or FNL. Though it is also possible to write your own netlister using SKILL, this is strongly discouraged because the rest of the Cadence system does not understand the name mapping you need to perform.

Netlister Name Mapping

The Cadence design entry system allows the designer great flexibility in naming instances (devices) and nets contained in a design. Most simulators are much more restrictive in the names they can accept. For example, some simulators require that names begin with an alphabetic character. Others have name length restrictions: For example, names may be required to be unique within or not longer than 12 characters. These restrictions usually force some form of name mapping. The names in the design cannot always be used in the netlist produced for a particular simulator. This is especially true in flat netlists. Since a particular schematic may be referenced several times in a higher level schematic, a name must be generated for each net and instance that is unique within the flattened design.This name is generated by putting together the name of each instance (reference) to a lower level schematic down to the most primitive level and separating them by slashes (/). These names quickly become too large for most simulator name spaces. To solve this problem, both netlisters provide a name-mapping mechanism that allows you to map names that are invalid for your simulator to names that are valid.

This name mapping is saved with the netlist for use by the rest of the Cadence system. That way, designers can still reference names placed in designs when generating stimuli for their simulators without being concerned if the name was mapped when the netlist was generated. Functions are provided to translate between the name found in the design and the name placed in the netlist. The users always enter the name they placed in the design. This name is then translated to the name in the netlist before it is given as input to the simulator. The simulator output is then parsed, and the names are translated back to those the user entered. This way, the user never needs to see the names that appeared in the netlist. Refer to the *Simulation Environment Help* for more information on name mapping.

Translating Stimuli Names

As explained in the “Netlister Name Mapping” section in this chapter, the names appearing in the netlist are not always the same ones entered by the designer. However, designers should always be able to refer to components in their designs using the names they specified. This is true when they generate their simulator input and when they analyze the output. Designers need to be aware that name mapping occurs during netlisting, but they do not need to know the mechanism or the mapped name.

For designers to use and see the names they enter in the design, there are name-mapping functions available in SE. When generating their stimuli, designers simply mark names used in their designs. As part of the simulation process, SE translates all names in the simulator input file before providing it as input to the simulator. From the designer’s perspective, the translation is a transparent part of the simulation process.

Running the Simulator

SE is the environment that controls all the simulation process steps. It provides you with the ability to manage the complex series of steps required to run a simulation and make simulation appear as a single-step process to the designer.

SE can be customized using the Cadence standard language, SKILL. This is the same language and syntax used to customize FNL and HNL output. The basic commands in SKILL have been enhanced with the addition of commands and functions specific to running simulations.

SE can be run interactively or batch from a stand-alone ASCII terminal, or it can be run from within the Cadence graphics environment.

Translating Names in Simulator Text Output

Since netlist names and input stimuli provided to the simulator may have been mapped from user-assigned names to simulator-acceptable names, the text output of the simulator must have these names translated back to the user-assigned names. Functions to do this have been provided in SE and can be run as a transparent part of the simulation process. These functions are explained in detail in the “Customizing the Simulation Environment (SE)” chapter in this manual.

Creating Backannotation Error Messages

To simplify locating errors in a design, you can graphically backannotate error messages into the schematic by means of *probes*. Probes are the same mechanism used to graphically select nets for waveform display. The information necessary for placing the probes is stored with the error messages in the Probe File. The syntax of this file is described in the “Probe File Format” section of the “Graphics Editor” chapter of the SKILL reference manuals. You can convert the error messages from your simulator as a transparent part of the simulation process using SE, or, preferably, modify your simulator to produce this syntax directly.

The designer can then display all errors found during simulation with a single command. The error regions are highlighted graphically, and the designer can step through each error and display the error message associated with that region. Errors can be associated with nets, instance, or terminals.

Integration Steps

The “Simulation Steps” section in this chapter describes the “what” of simulator integration while this section describes the “how to.” There are many steps to fully integrate a simulator into the Cadence system. You need not complete them all at once; the development can be done in stages. As each stage is completed, you can test that portion. Once that stage is functioning to your satisfaction, you can proceed to the next stage of integration. Some of the tools provided in OSS depend on others being functional first. For example, to make use of FNL, you should first set up an initialization file in SE. To ensure there are no missing dependent packages as you develop any portion of your interface, follow these steps:

1. Learn background information.
   - ❑ SKILL
   - ❑ database access functions
   - ❑ Running Simulations Using SE
2. Create basic SE command files to run the netlister to produce the netlist for your simulator.
   - ❑ Read the “Customizing the Simulation Environment (SE)” chapter in this manual.
   - ❑ Create the `*install_dir*/tools/dfII/local/si/caplib/simulator.ile` file.
3. Customize netlister output to produce syntax needed for your simulator.

   If you want to produce hierarchical netlists, use HNL; to produce flat netlists, use FNL.

- ❑ Customize HNL output:

  Read the “Customizing the Hierarchical Netlister (HNL)” chapter in this manual.

  Create HNL format functions in the *install_dir*/tools/dfII/local/hnl/simulator.ile file.

  Create library elements for netlister use.

- ❑ Customize FNL output:

  Read the “Customizing the Flat Netlister (FNL)” chapter in this manual.

  Create a global view whose cellname is `nlpglobals`, and whose viewname is the same as the name of your simulator.

  Create FNL format functions in the *install_dir*/tools/dfII/local/fnl/simulator.ile file.

  Create library elements for netlister use.

4. Customize SE to run your simulator in addition to generating netlists.
   - ❑ Modify the *install_dir*/tools/dfII/local/si/caplib/simulator.ile file.
   - ❑ Create a sed input file for simulator output name translation.
5. Customize STL output using the OSCG option of STL.

   The Open Simulation Code Generator (OSCG) functionality has been removed from OSS. The customer need to develop its own code generator.

Note: Modification of SE is performed at each step in the development of your interface because SE is the controlling program that invokes each step of the simulation process. As you add steps to this process, you must instruct SE to also run these steps. This is done to simplify the simulation process for your users.

3

Customizing the Simulation Environment (SE)

The Simulation Environment (SE) controls all steps of the simulation process. It lets you manage the complex series of steps required to run a simulation, making simulation appear to be a single-step process to the designer.

SE can be customized using the Cadence standard language, SKILL. This is the same language and syntax used to customize the output of the Flat Netlister (FNL) and the Hierarchical Netlister (HNL). In addition to the commands available in SKILL, database access functions are also available. These commands have been enhanced with additional commands and functions specific to running simulations.

This chapter explains the strategy behind SE, how it works, and how to customize it to run your simulator.

SE Goals

A designer typically uses several design analysis tools throughout the design cycle: tools to verify timing, simulate the design at various levels, and verify the physical against the logical design. In traditional computer-aided design (CAD) systems, each of these tools was separate, distinct, and had a separate non-standard user interface. In addition, running each design analysis tool frequently required several steps. These include generation of a netlist, sometimes translating the netlist from a standard format such as Electronic Design Interchange Format (EDIF) to the syntax required by the simulator, creating input stimulus in the syntax required by the analysis tool, and translating the results to either a user-readable format, or converting the output to a syntax required by other display tools. The designer needs to manually perform many of these steps.

SE simplifies this process. SE provides a single user interface designers can use to access all their design analysis tools. Cadence uses SE to provide the interface to all of the simulators it supports. With the Open Simulation System (OSS), you can use SE to interface your design analysis tool into the Cadence system, and provide your designers with the same consistent user-friendly interface.

Underlying the visible user interface is a simulation interface framework. This framework provides functions to perform the steps required in the simulation process. When you specify default values for the parameters used by these functions, the functions alter their behavior to perform the steps required by your application. Integrating your application into SE consists of specifying these parameters, which consist of a set of SKILL variables and functions. Designers never need to know the details of this framework because they see only the user interface. However, to add your simulator into this framework, you must thoroughly understand the framework.

This chapter explains how SE works, and then shows how to customize it to run your simulator. Before continuing, you need to do the following:

1. Familiarize yourself with the simulation user interface.

   Before you can customize SE, you should have an in-depth knowledge of running simulations using the Cadence system. To do this, first read the *Simulation Environment Help*. Then, design a few small circuits and simulate them using SE.

2. Acquire a working knowledge of SKILL.

   SE is customized using the Cadence standard language, SKILL. Before you attempt to customize SE, you must have a working knowledge of SKILL. First, read the *Cadence SKILL Language User Guide*. Then, write a few small programs in SKILL and run them in SE or using SKILL in the Cadence graphics environment.

3. Acquire a working knowledge of database access functions.

   Database access functions are the SKILL level database access functions available from Cadence. You do not need to understand the structure of the Cadence database to customize SE to run your simulator; however, you can use these features to develop advanced simulation interfaces. Read the *Virtuoso Design Environment SKILL Reference* to become familiar with the Cadence database structure and the functions available to access it. Then, write a small program to access the design database.

4. Customize HNL or the FNL netlisting.

   The first step to performing a simulation is generating a netlist suitable as input to your simulator. Before customizing SE, customize either HNL or FNL to generate a netlist in the syntax required by your simulator. This process is described in the “Customizing the Hierarchical Netlister (HNL)” and “Customizing the Flat Netlister (FNL)” chapters in this manual. These chapters also explain how and why name mapping is performed in the Cadence system. This knowledge is required to understand the name translation steps performed during a simulation.

5. Familiarize yourself with the probe file.

Error backannotation is performed using probes. You can store error messages from your simulator in a file that can later be graphically displayed on the design. The format of this file is documented in the “Probe File Format” section of the Graphics Editor chapter of the *Virtuoso Design Environment SKILL Reference*.

SE Steps

SE performs several steps during the simulation process. This section includes an explanation of the following major steps and their functions:

- Run Directory Initialization
- Netlisting
- Simulation Input Translation
- Running the Simulator
- Simulator Output Translation

Run Directory Initialization

When a simulation is run on the Cadence system, all inputs and outputs of the simulation process are contained in a single directory. This directory is called the Simulation Run Directory (or `run directory`). The first step in the simulation process is to ensure required files exist in this directory. Most design analysis tools require instructions as to what actions are to be performed. These instructions are stored in the `control` file. You can provide a default set of instructions in a file, and then whenever your designers request creation of a new run directory, SE copies this file into that directory. In addition, you can perform more sophisticated control file creation. For example, you could read in the design, and for every terminal in the top level of the design, create a default set of stimuli.

Netlisting

Netlisting is the process of converting the connectivity of a design into a textual description suitable as input to a design analysis tool. Netlisting is the most complex step in integrating your simulator into the Cadence system. Therefore, the details of these steps are contained in separate chapters. The “Customizing the Flat Netlister (FNL)” and “Customizing the Hierarchical Netlister (HNL)” chapters in this manual describe in detail how to customize the output of the Cadence-provided netlisters. In addition, the “Customizing the Flat Netlister (FNL)” chapter in this manual explains the modifications required to SE to run the netlister.

Name mapping is frequently performed by the netlisters. The Cadence system allows you flexibility in naming components of your design. These names are occasionally not valid in the syntax of the target design analysis tool. In addition, when generating a flat netlist, a new name must be generated to uniquely identify a device contained in a hierarchical design. The designer never needs to know what name was assigned by the system; only the designer-assigned names are of concern. Part of the SE functionality automatically translates between these names as needed by the application.

Simulation Input Translation

As mentioned in the previous “Netlisting” section in this chapter, names in the netlist are frequently not the names the designer placed in the design. However, the names in the netlist are the only ones the design analysis tool recognizes. To enable the designer to specify the same names in the design and the input to the simulator (stimulus and commands), the `control` file is translated before it is provided as input to the simulator. Any names that were mapped to a different name during the netlisting process are then substituted with the name used for the netlist. This way, the designer does not need to know the names used in the netlist, and the simulator is always presented a consistent set of names.

Running the Simulator

Once the input for the simulator has been prepared, the simulator is run. Tools are provided within SE to specify the UNIX® command to run your simulator. SE then runs the simulator, and waits for it to complete execution. When the simulator has completed, the simulator output needs to be prepared for analysis by the designer. The simulation output is in the textual form.

Simulation Output Translation

The simulator text output also requires translation. The names in the output file referencing the design are the same ones that appeared in the netlist. These may not be the same names as those entered in the design; therefore, the names need to be converted back to the names the designer entered.

How SE Works

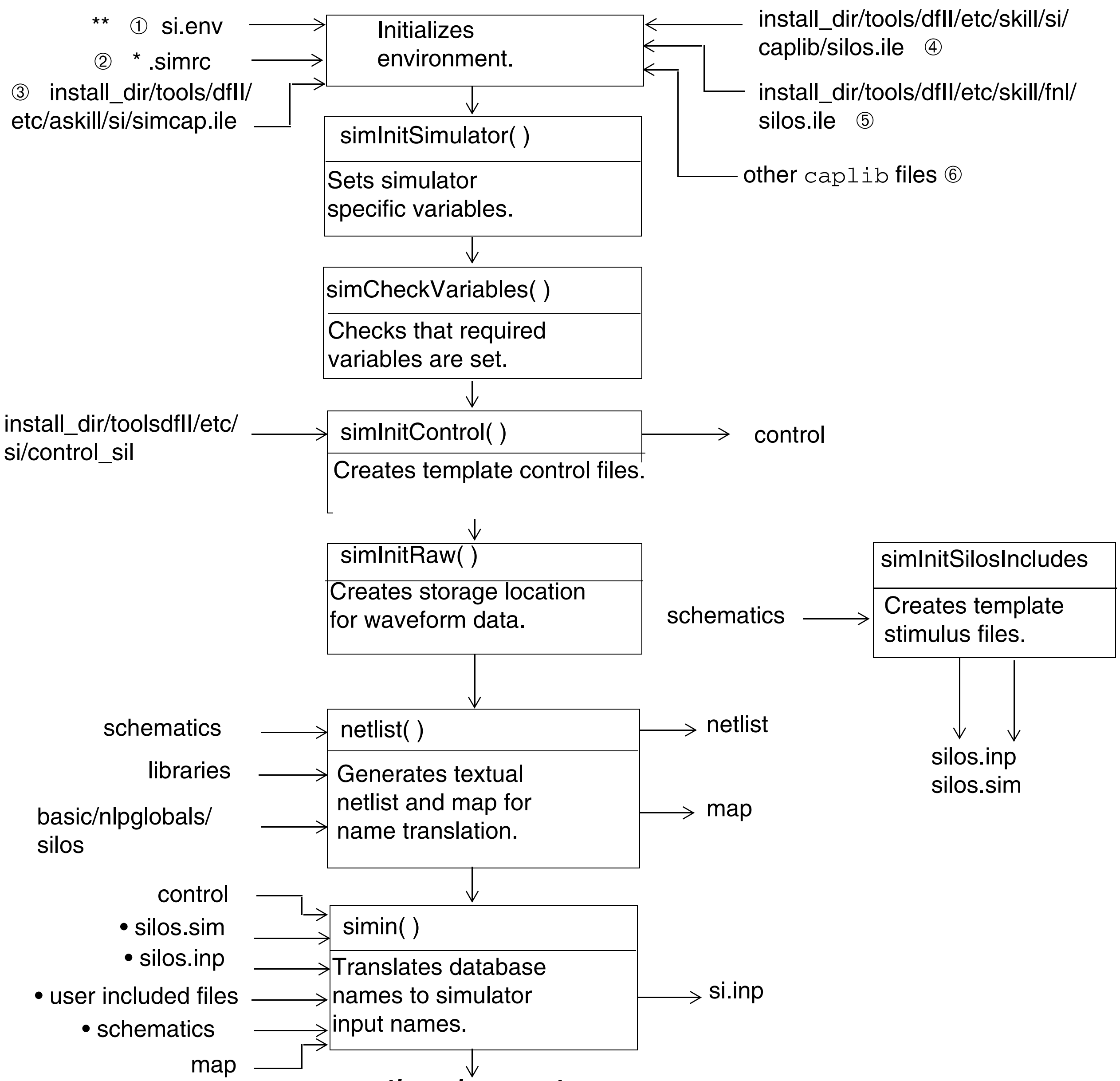
SE has both a graphics environment, Cadence graphics, and a non-graphics environment, Simulation Interface (SI). Designers can perform all simulation functions in either environment. For example, they can generate netlists, run name translations, and complete simulations by entering commands from menus in the Cadence graphics environment. They can also perform these tasks by starting SI in UNIX and entering commands.

In the Cadence graphics environment, designers can run simulation functions in the foreground or in the background after running the `Initialize` command from the Simulation menu. For example, if they run the `netlist` command in the foreground, the Cadence graphics environment is locked while the netlist is generated. If they run `netlist` in the background, they can continue to issue other commands.

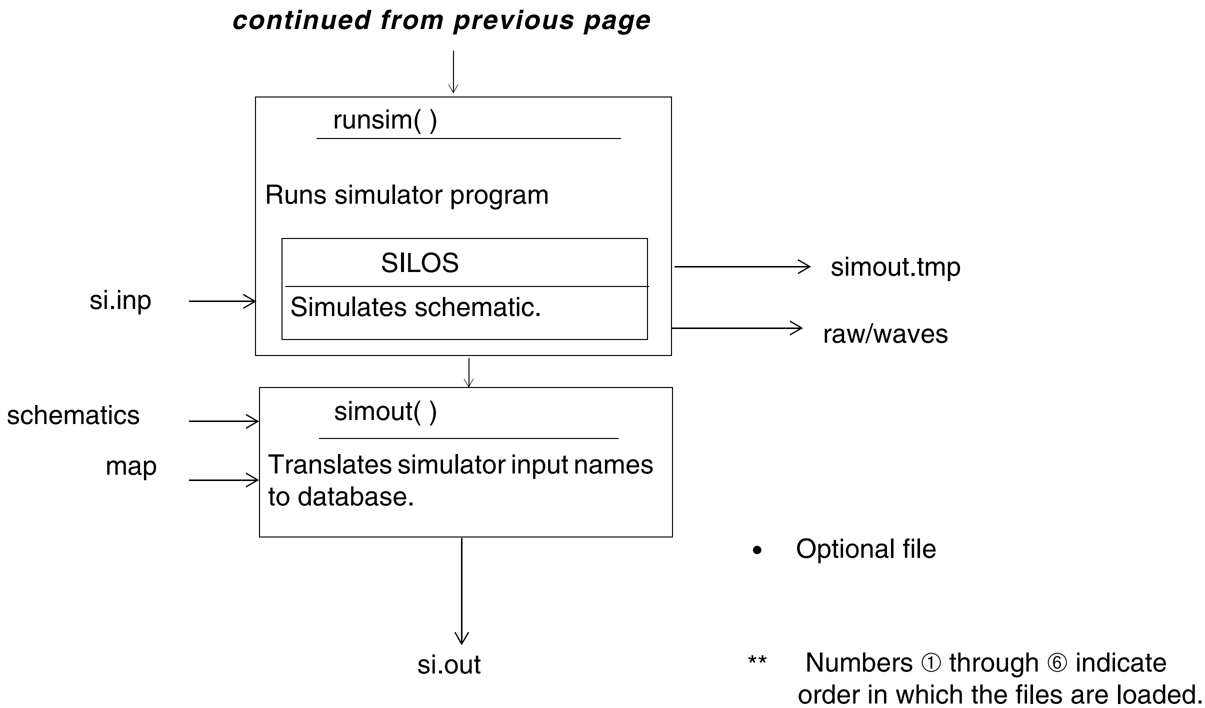
In SI, designers can run simulation in either batch or enter commands to a command interpreter. If they enter the `si` command with the `-batch` option, the program initializes itself and runs the single command they specify following the `-command` option. If they do not enter a command, SI runs the complete simulation process. If designers want to run the simulation and be prompted for commands, they enter the `si` command without the `-batch` option. A SKILL command interpreter is invoked when the program has been initialized. Designers can then run any SKILL or SE function. In this mode, unlike batch mode, designers can enter more than one command.

Figure 3-1 on page 42 shows the steps performed during a simulation using the SILOS® simulator. Included in the diagram are the files used as input to each step, and the resulting output files. The numbers next to the files in the initialization step specify the order in which the files are loaded.

Figure 3-1 Steps Performed During SILOS Simulation



continued on next page



SE Initialization

Figure 3-1 on page 42 shows the steps performed during SILOS simulation. When SE starts executing, the first step is to initialize the SKILL environment with the default functions and variable settings used for running a simulation. This is accomplished by loading various files that are written in SKILL syntax.

All the files provided as part of the simulation environment are stored in the `install_dir/tools/dfII/etc/skill/si/caplib` directory. This directory is relative to the hierarchy in which your Cadence software has been installed. For example, if your software has been installed in the `/usr1/cds` directory, these files would be located in the `/usr1/cds/etc/skill/si` directory. This directory also contains the `caplib` directory, the SE capabilities library.

In addition to the `simcap.ile` file in `install_dir/tools/dfII/etc/skill/si`, the files in the `caplib` subdirectory are all written in the SKILL language. The SKILL code is encrypted, but any SKILL files you create can be either encrypted or non-encrypted. If you are customizing SE, unencrypted versions of files included with SE are also provided. The file hierarchy is replicated in unencrypted form in the `install_dir/tools/dfII/group/oss/src/seIl` directory. SE never searches for files in this directory. The files in the `install_dir/tools/dfII/group/oss/src/seIl` directory are provided as a convenience so that you can read the SKILL source code that uses the functionality of SE. You can find the following files in unencrypted form in the `install_dir/tools/dfII/group/oss/src/seIl` directory:

```
getfterms.il
init.il
netlist.il
seAutoload.il
siminout.il
simulate.il
util.il
silos.il
```

Note: In the remainder of this chapter, directory files are specified as located in *`install_dir`*`/tools/dfII/etc`... The name of the directory is relative to where you install the Cadence software. When you see such a reference, replace the word *`install_dir`* with the file system pathname to where the software is installed. SE loads files in the following order to initialize the simulation environment:

`si.env`
`.simrc`
`simcap.ile`
simulator-specific SE customization
simulator-specific FNL customization
`caplib` files

How SE Locates Customization Files

All the customization files are searched through the Cadence Setup Search File mechanism (CSF) that uses the `setup.loc` file for customized search paths. The `setup.loc` file is an ASCII file that specifies the locations to be searched and the order in which they should be searched.

To specify the files that should be found through CSF, create an ASCII file, `csfLookupConfig`, in any directory. This directory should be listed in your setup.loc file. In the `csfLookupConfig` file, specify the files that should be found through CSF. An example of a `csfLookupConfig` file is as follows:

```
INCLUDE si/<simulator>.ile
INCLUDE hnl/<simulator>.ile
INCLUDE hnl.ile
INCLUDE .simrc
```

You can access sample `csfLookupConfig.sample` and the default `setup.loc` files from *`your_install_dir`*`/share/cdssetup` directory. The default `setup.loc` file specifies a default search order, which includes commonly-used storage locations.

The CSF search mechanism is used for the following files:

1. <*simSimulator*>.ile / <*simSimulator*>.il
2. hnl.ile and hnl.il
3. All those files which are loaded using simLoadNetlisterFiles (for FNL)

The following files are excluded from the CSF search mechanism because they expect a specific directory order. Also because other applications may have files with the same name as these are generic names.

- caplib/util.il
- caplib/getfterms.il
- caplib/init.ile
- caplib/siminout.il
- caplib/simulate.il
- caplib/remote.il
- caplib/netlist.il

For more information about the CSF search mechanism, the csfLookupConfig and setup.loc file format refer to *Chapter 3, Cadence Setup Search File: setup.loc* of the *Application Infrastructure User Guide*.

If you have not defined an entry for a global file, say .simrc, in the csflookupConfig file, then .simrc is searched for in the following order:

$SIMRC/.simrc
$ossSimUserSiDir/.simrc
dfII/local/.simrc
Current UNIX directory/.simrc
~/.simrc

The search stops when the first instance of the file is found. Other files are not loaded, unless the identified file loads them, allowing for tiered loading or for the local CAD group to alter or disallow the search mechanism. If it does not exist, no error is generated.

If you do not want to use the CSF search mechanism, then you can continue to set the shell environment variable $ossSimUserSiDir to point to $CDS_SITE/si or any other directory. You can also use the hnl and fnl specific shell environment variables, ossSimUserHnlDir and ossSimUserFnlDir to specify a different directory containing the .ile files.

The si.env File

The `si.env` file is read in from the simulation run directory. This file is used to instruct SE which design to simulate and which simulator to use. In addition to the default information SE stores in this file, you can instruct SE to save any application-specific variables here by using the variable `simSimulatorSaveVars`. The file is overwritten by the SE function `simPrintEnvironment()` each time a simulation is run, and any other information stored in this file that SE has not been informed of is lost. The `si.env` file is used to communicate environment information between SE and the Cadence graphics program.

The .simrc File

You can use the `.simrc` file to customize simulations. This file lets you set your defaults for the simulation variables. It overrides the contents of the `si.env` file. Therefore, the `.simrc` file provides a mechanism to set the defaults at the user or flow level. This file is optional and is loaded if it exists.

If you want to use another file, instead of `.simrc`, to customize simulations for a specific flow, use the UNIX environment variable `ossUserSimrc` to set the path of that custom file. For example, the following command sets `mysimrc` as the custom file.

```
$ setenv ossUserSimrc mysimrc
```

The functionality provided by `ossUserSimrc` lets you customize the simulation environment to suit the specific requirements of different flows, without changing your `.simrc` file. If `ossUserSimrc` is not set, then the default CSF mechanism is used to load the simulation customization file.

The simcap.ile File

The `simcap.ile` file is read in from the *`install_dir`*`/tools/dfII/etc/skill/si` directory. This file is written in SKILL syntax and contains general variable default settings and function definitions. It is provided by Cadence as part of the simulation environment, and should not require modification to integrate your simulator. In addition to setting defaults, at the end of this file are calls to functions defined in SE. These functions are executed as the file is loaded. The `simcap`, `caplib`, `hnl`, `fnl` code is searched for in the following order:

```
install_dir/tools/dfII/local/[si,hnl,fnl]
install_dir/tools/dfII/etc/skill/[si,hnl,fnl]
```

This order lets you to build an interface without modifying Cadence- provided software hierarchy.

The overall file loading process is described in more detail following these definitions of the initialization files.

Simulator-Specific SE Customization

A file with the same name as the simulator and the `.ile` suffix is loaded from the *`install_dir`*`/tools/dfII/local/si/caplib` or *`install_dir`*`/tools/dfII/etc/skill/si/caplib` directory. For example, when running a SPICE simulation, a file with the name `spice.ile` is loaded from that directory. This file contains all the default variable settings specific to running a SPICE simulation. This is the file that you must create for SE to recognize your simulator. The command to load this file is contained in the `simcap.ile` file, and is executed when the `simcap.ile` file is loaded.

Simulator-Specific Flat Netlister (FNL) Customization

A file with the same name as the simulator and the `.ile` suffix is loaded from the *`install_dir`*`/tools/dfII/local/fnl` or *`install_dir`*`/tools/dfII/etc/skill/fnl` directory. For example, when running a SPICE simulation, a file with the name `spice.ile` is loaded from that directory. This file contains all the default variable settings and function definitions used by FNL to output the netlist in your simulator's syntax. For more information on this file, refer to the "Customizing the Flat Netlister (FNL)" chapter in this manual. This file is optional and is loaded if it exists. If it does not exist, no error is generated.

caplib Files

In addition to the files previously described in this section, the `init.ile`, `netlist.ile`, `siminout.ile`, `simulate.ile` and a few other files are loaded from the *`install_dir`*`/tools/dfII/etc/skill/si/caplib` directory. These files are loaded in the following order:

1. *`install_dir`*`/tools/dfII/etc/skill/si/caplib/util.ile`
2. *`install_dir`*`/tools/dfII/etc/skill/si/caplib/getfterms.ile`
3. *`install_dir`*`/tools/dfII/etc/skill/si/caplib/init.ile`
4. *`install_dir`*`/tools/dfII/etc/skill/si/caplib/siminout.ile`
5. *`install_dir`*`/tools/dfII/etc/skill/si/caplib/simulate.ile`
6. *`install_dir`*`/tools/dfII/etc/skill/si/caplib/remote.ile`
7. *`install_dir`*`/tools/dfII/etc/skill/si/caplib/netlist.ile`

Besides defining global functions and default variable settings, the `simcap.ile` file contains calls to functions that complete the initialization of the simulation environment. These function calls are at the end of the file and are executed as the file is being loaded. The first function to be called is `simInitSimulator`. The `simInitSimulator` function calls a function that you must write to set certain variables to values appropriate for running your simulator. This function must have the same name as your simulator and should be defined in a file that also has the same name as your simulator. This file should be stored in the *`install_dir`*`/tools/dfII/local/si/caplib` directory. A complete description of what this function and file must contain is in the “Simulator-Specific SE Customization File” section in this chapter.

Next, the `simCheckVariables` function is executed. This function checks if the following variables have been defined:

```
simSimulator
simRunDir
simCellName
simLibName
simViewName
simNlpGlobalLibName
simNlpGlobalCellName
simViewList
simStopList
simSedFile
simCommand
```

These variables are required to enable SE to run a simulation. The first eight variables are defined in the `si.env` environment file and by SE. The remaining variables must be defined in the simulator-specific SE customization file you must write.

That completes the initialization sequence performed by SE. At this point, control is either passed to the top-level SKILL command interpreter, and SE can be instructed to run specific commands, or the command specified when SE was invoked is executed.

SE Execution

Once the simulation environment has been initialized, command execution begins. If SE is being run interactively, you are prompted to issue commands. You can then set variables and call any function defined in the simulation environment. If SE is being run in a batch mode, it executes a single command. You can optionally specify the command to be executed when you invoke SE (for more information refer to the *Simulation Environment Help*). This command can be any function defined in the simulation environment, for example, `netlist` or `simin`. If no command is specified, the default is to run a complete simulation, which is equivalent to executing the `sim` function.

SE runs a complete simulation by executing the `sim` function that is defined in the *`install_dir`*`/tools/dfII/etc/skill/si/caplib/simulate.ile` file. (A detailed description of this function is in the “SE Functions” section in this chapter.) A complete simulation consists of executing any steps required by the simulator being used including preparing the simulator input, running the simulator, and output post-processing. The order of steps performed is determined by the value of the `simActions` variable which is an ordered list of the functions to be executed when running a complete simulation. The default value is the following:

```
'(simCheckVariables()
simInitRunDir()
netlist()
simin()
runsim()
)
```

If this sequence is not suitable for running your simulator, you can change the default by setting the `simActions` variable in your simulator-specific SE customization file. If any of the functions specified in this list fails to complete successfully (does not return `t`), the simulation process is halted. Once all of the functions have been executed, SE stops executing, and the simulation has completed.

Following is a description of the steps performed by each of these functions:

simCheckVariables( )

Checks that certain variables required by SE are defined:

```
simSimulator
simRunDir
simCellName
simLibName
simViewName
simViewList
simStopList
simSedFile
simCommand
simNlpGlobalLibName
simNlpGlobalCellName
```

This function is also executed as part of the simulation environment initialization sequence and again as part of the simulation sequence because the designer has the ability to run SE interactively. In doing so, it is possible for the designer to damage the environment, for example, by setting the listed variables to a `nil` value. By calling this function again, we reduce the chance that the environment has been corrupted to the point that simulation is not possible.

simInitRunDir( )

Executes the initialization sequence specified by the simInitRunActions variable. By default, this is executing the simInitControl( ) and simInitRaw( ) functions. In addition, you can add new functions to create default input stimulus based on the design.

The simInitControl function copies the default simulator control file into the simulation run directory if a control file does not already exist. If you want to provide a default simulator input file (template) for your designers, you must write the template file and place it in the install_dir/tools/dfII/local/si directory. Then, in your simulator-specific SE customization file, you must tell SE the name of that file. These files are described in detail in the “Necessary Files to Integrate your Simulator” section in this chapter. The simInitRaw function creates the raw directory used for storage of simulator waveform output in the simulation run directory. The name of the raw directory cannot be altered because it is the standard location used by the Cadence waveform display program to locate waveform data when displaying waveform results.

netlist( )

Generates the textual netlist representing the connectivity of the design to be simulated. This netlist is later used as input to the simulator. Which design is netlisted, and how the hierarchy is traversed (including whether HNL or FNL is used), is determined by global variables. For running LVS in the batch mode using `si`, the command to be used is `sim` instead of `netlist()`. This is the case with formatters like auCdl. These variables can be set in your simulator-specific SE customization file.

simin( )

Translates the designer-specified names in the simulator input file into the names used in the netlist. Since the names to be translated are determined by the designer, you do not need to make any changes to use this function.

runsim( )

Runs the simulator. First the `simCommand` variable is evaluated. This variable defines the actions to be performed to run the simulator. It should be set in your simulator-specific SE customization file. When the simulator has finished executing, the `simOutWithArgs` function is called which translates the netlister-assigned names for nets and instances in the given input files to the corresponding designer-specified names in the given output files. Since there is no fixed syntax to simulator output, you must instruct SE with a `sed` input file how to determine which names require translation. The `sed` files are described in the “Files Needed to Integrate your Simulator” section in this chapter.

Files Needed To Integrate Your Simulator

To fully integrate your simulator using SE you must create the following files:

```
Simulator-Specific SE Customization File
Template Control File
Name Translation File
```

The following sections explain the purpose of each file, its contents, and its expected location.

Simulator-Specific SE Customization File

The first step in customizing SE to run a new simulator is to define the environment and steps to execute.

1. Create a file with the same name as the new simulator and the suffix `.ile`.
2. Place the file in the *`install_dir`*`/tools/dfII/local/si/caplib` directory.
3. Write the necessary SKILL procedures.
4. Set the needed variables for running SE.

Note: With the OSS system, you can define procedures and execute encrypted or non-encrypted SKILL code in SE. Any system equipped with the Simulation Environment (SE) product can run simulations using the interface you develop. To develop a new simulation interface, you can encrypt the SKILL file you write, by using the `encrypt` command. This command is available in your development version of SE before you distribute the SKILL file to your designers.

All files for the simulation are stored in the run directory. Because designers can start the Simulation Environment from any location in the file system, all accesses to files in the run directory must be made using full pathnames. If you cannot specify full pathnames for the files your application requires or produces (for example, if an output file from your application has a fixed name), you must make sure that the process that runs the application changes the current working directory to the run directory before the application is started. The examples in the following sections show you how to do this.

Example of Customizing SE To Run a New Simulator

As an example of how to customize SE to run a new simulator, follow the steps given in this section for the creation of the simulator-specific SE customization `spice.ile` file for the

SPICE simulator. The name of the file differs for each simulator integrated. For example, if you are integrating a simulator called SILOS, this file is called `silos.ile`.

Set the Switch and Stop Lists

This `spice.ile` file must contain the default values for the switch and stop lists used when netlisting. For a description of switch and stop lists, refer to “FNL Flattening Process” section in the “Customizing the Flat Netlister (FNL)” chapter in this manual. The `netlist` function uses the `simViewList` and `simStopList` variables. The designer should not be allowed to set these variables directly. Instead, the designer sets switch and stop lists on a per-simulator basis by setting variables for the lists whose names include the name of the simulator. This way the designer can set default values for switch and stop lists for several different simulators in the `.simrc` file. The following is an example of how to set the default values of these variables for SPICE:

```
; Set the default SPICE-specific view switch
;list for netlisting.
simSetDef( 'spiceSimViewList,'("spice" "schematic") )

; Set the default SPICE-specific stopping
;view list for netlisting.
simSetDef( 'spiceSimStopList,'("spice") )
```

Instead of setting these variables using an equals sign (=), the `simSetDef` function is called. This function sets the specified variable only if it has not already been set, or has been set to `nil`. This allows the designer to override the default by setting the variable in the `.simrc` file in the home login directory.

Next, you must set the variables used by the `netlist` function from the simulator-specific versions of these variables using the following commands:

```
; Set the view switch list used for netlisting.
simViewList = spiceSimViewList

; Set the stopping view list used for netlisting.
simStopList = spiceSimStopList
```

Now the equal sign is used instead of the `simSetDef` function. This is because the designer is not allowed to set these variables directly.

Set the Default Control File

Tell SE the name of the simulator input control file template. This file is copied into the run directory when the directory is being created by SE. The control file template is explained in the “Template Control File” section later in this chapter. The name of the template file is `control.spi`. The following example shows how the `simSetDef` variable is set:

```
; Set the name of the default SPICE control file ;template.
simSetDef('simDefaultControl "control.spi")
```

Set the sed Input Filename

SE must be told the name of the `sed` input file used for simulator output name translation. This is done by setting the `simSedFile` variable to the name of the `sed` file to be used. In the following example, the file is called `sed.spi`. What this `sed` file should contain is explained in the “Name Translation File” section later in this chapter. Following is an example of how to set the `simSedFile` variable:

```
; Set the sed script filename to trigger name ;translations.
simSedFile = prependInstallPath("local/si/sed.spi")
```

The `simSedFile` variable is set to the return value of the `prependInstallPath` function. This function prepends the install path of the Cadence software to the filename given as argument, thereby creating a full file system pathname. The filename the `simSedFile` variable is set to must be a full pathname.

Set the global cellview

SE must also be instructed which global cellview to use when running FNL. The global cellview is a database view that instructs FNL how the netlist is to be formatted. Refer to the “Customizing the Flat Netlister (FNL)” chapter in this manual. The name of the cell is determined by the `simNlpGlobalCellName` variable. The name of the view is usually the name of your simulator. This information is stored in the `simNlpGlobalLibName`, `simNlpGlobalCellName` and `simNlpGlobalViewName` variables. The first two are set to defaults by SE; the `simNlpGlobalViewName` variable must be set by you. The following is an example of how to set this variable for later use by the `netlist` function.

```
; Set the default name of the global view
;used for netlisting.
simSetDef('simNlpGlobalViewName "spice")
```

Set the Simulator Function

All the variable settings so far described are placed inside the `spice.ile` file, outside any function so that the variables are defined as the file is loaded.

The `spice.ile` file must also contain one function, which must have the same name as the simulator. In the following example, the function must therefore be called `spice`. This function is called by SE during the initialization phase and again before a simulation is run. Because SE can be run in an interactive mode, the designer can change at any time the values of the simulator-specific variables described in the “Simulator-Specific SE Customization File” section in this chapter. To ensure that the designer's changes also alter the variables used by SE, about which the designer is not informed (for example, `simViewList`), this function must reset those variables from their simulator-specific equivalents. The following two lines should be placed in the `spice` function:

```
simViewList = spiceSimViewList
simStopList = spiceSimStopList
```

In addition to resetting these variables, the `spice` function must set the `simCommand` variable. This variable is evaluated by the `runsim` function to run the simulator. The value of this variable must be a SKILL expression. It should be a command that runs the simulator, and returns `t` if the simulator completed successfully, or `nil` if the simulator detects errors. You can use the `simExecute` function to execute a UNIX command. This function returns `t` if the command given as argument completes with an exit status of zero (0), otherwise it returns `nil`. The UNIX command you execute can then invoke your simulator. If your simulator takes its input from `stdin`, you can redirect its input from the file containing the input stimulus for your simulator.

The simulator input file is called `si.inp`. The textual output of the simulator should be redirected to a file named `simout.tmp`, which is then translated by the `runsim` function. In addition, the `simCommand` variable can include any run line arguments required by your simulator.

The following is an example of how to set the `simCommand` variable to run the SPICE simulator:

```
simCommand = 'prog( (tmpCmd )
; Run the SPICE simulator
sprintf(tmpCmd "cd %s; exec spice -r raw/waves.tmp
   < st.inp > simout.tmp 2>&1" simRunDir)
return(simExecute(tmpCmd))
)
```

Note: The command to run the simulator includes the `cd simRunDir` command. This causes the child process that runs the simulator to change the current working directory to the simulation run directory. If the simulator reads in a file or writes a file using a relative pathname, such as `netlist`, that file will be in the simulation run directory.

The `simCommand` variable can also perform other required preprocessing or postprocessing. For example, it can remove old waveform data before the simulator is run. This ensures that old results are not confused with a new simulation run. Set the `simCommand` as follows:

```
simCommand = 'prog( (tmpCmd)
; Remove old waveform file
simDeleteRunDirFile("raw/waves")
simDeleteRunDirFile("raw/waves.tmp")
; Run the SPICE simulator
sprintf(tmpCmd "cd %s; exec spice -r raw/waves.tmp
   < st.inp > simout.tmp 2>&1" simRunDir)
return(simExecute(tmpCmd))
)
```

An example of how to use the `simActions` variable to do this can be found in the complete `spice.ile` file in Figure 3-4 on page 58.

Set the Unbind Variables

To enable designers to simulate in the foreground, the SKILL simulation environment must be loaded into the Cadence graphics environment. This includes defining all SKILL functions and variables used in a simulation interface. The names of these functions and variables may conflict with those used in another simulation interface. For example, both the SPICE and the SILOS interfaces can define the `simActions` variable. If designers use SPICE and then use SILOS, they will get error messages when the simulator-specific code for the SILOS interface is loaded because the `simActions` variable has been defined for SPICE. To avoid this problem, SE unbinds all functions and variables it defines. Then it reloads the appropriate interface files when designers change simulators. The unbinding and loading of files occurs when designers execute the `Initialize` command from the Simulation menu.

All functions and variables defined by Cadence-supplied interfaces and SE will be unbound and redefined as needed. You must provide a list of all of the functions and variables you define. There are separate lists for functions and variables defined in the Hierarchical Netlister, the Flat Netlister, and SE. The following sections describe the variables you must define for SE.

simSimulatorUnbindVars

The `simSimulatorUnbindVars` variable specifies the variables you define in your `spice.ile` file that must be unbound when simulators are switched. You should include any global variable you set in this list. Of primary importance are those that you set using the `simSetDef` function. If you do not include a variable in this list, designers will get warning messages indicating this variable has already been defined, and the correct value for the variable will not be used when they switch to a different simulator.

Set this variable to a list containing the names of all of the variables you define. Define this variable in the `caplib` file outside of any function definition. The following is an example of how to set this variable for SPICE:

```
simSetDef('simSimulatorUnbindVars, '(spiceSimViewList
   spiceSimStopList
   simDefaultControl
)
)
```

simSimulatorUnbindFuncs

The `simSimulatorUnbindFuncs` variable specifies the functions you define in your `spice.ile` file that must be unbound when simulators are switched. You should include any functions you define in this list. Of primary importance are those that you define using the `simIfNoProcedure` function. If you do not include a function in this list, the correct function may not be used when designers switch simulators.

Set this variable to a list containing the names of all of the functions you define. Define the variable in the `caplib` file outside of any function definition. The following is an example of how to set this variable for SPICE:

Figure 3-2 Using simCommand To Run SPICE and the Waveform Translator

```
simSetDef( 'simSimulatorUnbindFuncs, '(spice) )

simCommand = 'prog( (cmd fileName)
   ; Remove old waveform file
   simDeleteRunDirFile("raw/waves")
   simDeleteRunDirFile("raw/waves.tmp")


   ; Run the SPICE simulator
   sprintf(cmd "cd %s;exec spice -r raw/waves.tmp <si.inp>
                     simout.tmp 2>&1" simRunDir)

   if( simExecute(cmd) == nil then
      simPrintError( "si: Spice did not complete without errors." )
      return(nil)
   )

   ; If a non-empty waveform file was created, then translate it
   ; into WSF syntax using the program spi2wsf
   sprintf(fileName "%s/raw/waves.tmp "simRunDir)
   if( isFile(fileName) then
      if( fileLength(fileName) != 0) then
         sprintf(cmd "cd %s;exec spi2wsf raw/waves.tmp > raw/
            waves "simRunDir)
         if( ! simExecute(cmd) then
            simPrintError( "si: Can't translate raw/waves.tmp
               into wsf.\n" )
            return(nil)
         )
      )
      simDeleteRunDirFile("raw/waves.tmp")
   )

   return(t)
)
```

Complete Sample SPICE Customization File

The last thing that must be placed in the `spice` function is a call to the `simPrintEnvironment` SE function. This function writes the current environment to the `si.env` file.

Figure 3-3 is a complete example of a simulator-specific SE customization file for the SPICE simulator.

Figure 3-3 Complete Example of SPICE SE Customization File

```
;
; This file contains the function that defines the sequence of
; steps for performing a spice simulation.
;
; Set the default SPICE-specific view switch list for netlisting.
simSetDef( 'spiceSimViewList,'("spice" "schematic") )
; Set the default SPICE-specific stopping view list for netlisting.
simSetDef( 'spiceSimStopList,'("spice") )
; Set the view switch list used for netlisting.
simViewList = spiceSimViewList
; Set the stopping view list used for netlisting.
simStopList = spiceSimStopList
; Set the name of the default SPICE control file template.
simSetDef('simDefaultControl "control.spi")
; Set the sed script filename to trigger name translations.
simSedFile = prependInstallPath("local/si/sed.spi")
; Set the default view name of the global view used for netlisting.
simSetDef('simGlobalViewName "spice")
;Set the lists of functions and variables that must be unbound when
;environments (simulators) are changed.
simSetDef('simSimulatorUnbindFuncs '(spice))
simSetDef('simSimulatorUnbindVars '(spiceSimViewList
                  spiceSimStopList simDefaultControl)
)
;
; spice() -
; This function sets the simViewList and simStopList variables to
; the simulator-specific values in "spiceSimViewList" and
; "spiceSimStopList" respectively. Then the simCommand variable
; is set to the commands required to run a SPICE simulation.
; This includes running the SPICE simulator and waveform
; translation. The simulator is not really run by
; this function, this function only sets the command to run it.

; Example:

; spice()
simIfNoProcedure( spice()
   ; Set the switch and stop lists from the simulator-specific
   ; variables for SPICE in case the user has modified them when
   ; running SE interactively.

simViewList = spiceSimViewList
simStopList = spiceSimStopList
simCommand = 'prog( (cmd fileName)
   ; Remove old waveform file
      simDeleteRunDirFile("raw/waves")
      simDeleteRunDirFile("raw/waves.tmp")
```

```
   ;    Run the SPICE simulator
sprintf(cmd "cd %s;exec spice -r raw/waves.tmp < si.inp > simout.tmp 2>&1"
simRunDir)
if( ! simExecute(cmd) then
   simPrintError( "si: Spice did not complete without errors.")
   return(nil)
)
; If a non-empty waveform file was created, then translate it
; into WSF syntax using the program spi2wsf
sprintf(fileName "%s/raw/waves.tmp "simRunDir)
if( isFile(fileName) then
   if( fileLength(filename) ! = 0) then
      sprintf(cmd "cd %s;exec spi2wsf raw/waves.tmp>raw/waves "simRunDir)
      simPrintError( "si: Can't translate raw/waves.tmp into wsf.\n" )
      return(nil)
   )
   simDeleteRunDirFile("raw/waves.tmp")
)
return(t)
)
; Rewrite the environment file to ensure that it reflects the
; values used to run the simulation in case any values have
; been modified by the user while running SE interactively.
simPrintEnvironment()
)
```

Alternate SPICE caplib File

Figure 3-4 on page 58 is another example of the same file and is functionally equivalent to Figure 3-1 on page 42. The waveform translation step is different in implementation. Instead of encoding the waveform translation step in the `simCommand` variable, the `simActions` variable is now set. This is done because you want a set of functions other than the default to be executed when running a SPICE simulation. Most of the functions are the same; the only difference is that the `SpiceTranslateWaves` function, which performs the waveform translation, has been added after the `runsim` function. Use of the `simActions` variable allows you a great amount of flexibility. With it, you can select the functions to be executed when running a simulation, without ever modifying any Cadence-supplied code. In addition, the sequence of functions you select affect only SPICE (in this case) simulations. No other simulations are affected.

Figure 3-4 Performing a SPICE Simulation

```
; This file contains the function that defines the sequence of
; steps for performing a SPICE simulation.
;
; Set the default SPICE-specific view switch list for
; netlisting.
simSetDef( 'spiceSimViewList,'("spice" "schematic") )
; Set the default SPICE-specific stopping view list
; for netlisting.
simSetDef( 'spiceSimStopList,'("spice") )
; Set the view switch list used for netlisting.
simViewList = spiceSimViewList
```

```
; Set the stopping view list used for netlisting.
simStopList = spiceSimStopList
; Set the name of the default SPICE control file template.
simSetDef('simDefaultControl "control.spi")
; Set the sed script filename to trigger name translations.
simSedFile = prependInstallPath("local/si/sed.spi")
; Set the default name of the global view used for
; netlisting.
simSetDef( 'simNlpGlobalViewName "spice")
; Set the default actions to be performed when running a SPICE
; simulation.
simSetDefWithNoWarn( 'simActions,'(simCheckVariables()
                     simInitRaw()
                     simInitRunDir
                     netlist()
                     simin()
                     runsim()
                     SpiceTranslateWaves()
                  )
)

;Set the lists of functions and variables that must be unbound when
;the environments (simulators) are changed.
simSetDef('simSimulatorUnbindFuncs'(SpiceTranslateWaves))
simSetDef('simSimulatorUnbindVars'(SpiceSimViewList piceSimStopList
            simDefaultControl)
)
;
; SpiceTranslateWaves() -
; This function translates the SPICE waveform raw/waves.tmp
; output file into WSF syntax, and places it in the raw/waves
; file for use by waveform display.
;
; Example:
; SpiceTranslateWaves()
simIfNoProcedure( SpiceTranslateWaves()
   prog( (fileName cmd)
      ; If a non-empty waveform file was created, then translate
      ; it into WSF syntax using the program spi2wsf
      sprintf(fileName "%s/raw/waves.tmp "simRunDir)
      if( isFile(fileName) then
         if( fileLength (fileName) != 0) then
            sprintf(cmd "cd %s;exec spi2wsf raw/waves.tmp>raw/
                     waves "simRunDir)
            if( ! simExecute(cmd) then
               simPrintError( "si: Can't translate raw/
                     waves.tmp into wsf.\n" )
               return(nil)
            )
         )
         simDeleteRunDirFile("raw/waves.tmp")
      )
   )
)
; This function sets the simViewList and simStopList variables to
; the simulator-specific values in "spiceSimViewList" and
; "spiceSimStopList" respectively. Then the simCommand variable
; is set to the commands required to run a SPICE simulation. The
```

```
; simulator is not really run by this function, this function
; only sets the command to run it.
;
; Example:
; spice()
simIfNoProcedure( spice(cmd)
   ; Set the switch and stop lists from the simulator-specific
   ; variables for SPICE in case the user has modified them when
   ; running SE interactively.
simViewList = spiceSimViewList
simStopList = spiceSimStopList
simCommand = 'prog( ()
      ; Remove old waveform file
      simDeleteRunDirFile("raw/waves")
      simDeleteRunDirFile("raw/waves.tmp")
      ; Run the SPICE simulator
   sprintf(cmd "cd %s;exec spice -r raw/waves.tmp < si.inp >
            simout.tmp 2>&1"simRunDir)
   if( ! simExecute(cmd) then
      simPrintError( "si: Spice did not complete without
            errors.\n" )
      return(nil)
   )
   return(t)
)
; Rewrite the environment file to ensure that it reflects the
; values used to run the simulation in case any values have
; been modified by the user while running SE interactively.
simPrintEnvironment()
)
```

Template Control File

When the simulation run directory is created, part of the process of initializing the directory is copying in a template control file. The control file is the simulator input file which is translated by the `simin` function as part of the simulation process to produce the `si.inp` file. This translation is required because of the name mapping performed during the netlisting process. The `si.inp` file is then used as input to the simulator. You are not required to create a template control file, but it gives your designers a starting point when creating their simulator stimulus.

When `simin` translates the control file, all text in the control file is copied to the `si.inp` file, unless it is surrounded by square brackets ([ ]). An opening square bracket ([ ) signals that the following text up to the closing square bracket ( ]) is to be interpreted. The entire expression is replaced by the resulting interpreted value. Following the opening square bracket ([ ) should be one of the following command characters:

`#  [#netname]`	Replace the [#netname] expression with the netlister-assigned net name for `netname`.

`$`	`[$instname]`	Replace the [$instname] expression with the netlister-assigned instance name for `instname`.
`!`	`[!filename]`	Replace the [!filename] expression with the contents of the `filename` file. If you want to include a file from a directory other than the simulation directory, you must use the full file system pathname. If the file does not exist, an error is generated. The contents of the new file are also parsed with the same translations being performed on the contents of the included file.
`?`	`[?filename]`	Replace the expression [?filename] with the contents of the `filename` file. If you want to include a file from a directory other than the simulation directory, you must use the full system pathname. If the file does not exist, *no* error is generated.
`n!`	`[n!filename]`	Same as `[! filename]`, except that the contents of the new file are not parsed, and square-bracketed expressions are *not* interpreted.
`n?`	`[n?filename]`	Same as `[? filename]`, except that the contents of the new file are not parsed, and square-bracketed expressions are *not* interpreted.

You can use any of these six expressions in your template control file. With the exception of the above substitution expressions, the control file you create must be in the syntax required by your simulator. The `simin` function does not translate syntax. Its main purpose is to allow designers to enter the same names they did in their designs when generating simulator input. Using this translation mechanism, designers never need to know the netlister-assigned names.

Figure 3-5 shows a sample control file template for SPICE.

Figure 3-5 Sample SPICE Control File Template

```
* Spice template control file
.options acct opts nopage limpts=1000
.width in=80 out=80
[ !spice.inp ]
[ !spice.sim ]
[ !netlist ]
.end
```

The name of this file must be the same as the name you assigned to the `simDefaultControl` variable in your simulator-specific SE customization file. In Figure 3-

3 on page 57, the file is called `control.spi`. It is customary to name the `control` file `control`, suffixed with part of the simulator name. When you have finished writing this file, place it in the `/cds/local/si` directory.

Name Translation File

The textual simulator output contains the netlister-assigned names. To make the output more readable for the designer these names must be translated back to the names assigned in the schematic. This translation is performed by the `simout` function which is the inverse of `simin`. The `simout` function translates the netlister-assigned names in the `simout.tmp` file back to the designer-assigned names, and places the result in the `si.out` file. The syntax required to trigger these translations is the same as for `simin`. All text in the `simout.tmp` file is copied verbatim to the `si.out` file, unless text is surrounded by square brackets ([ ]). The opening square bracket ([ ) signals that the following text up to the closing square bracket ( ]) is to be interpreted. The entire expression is replaced by the resulting interpreted value.

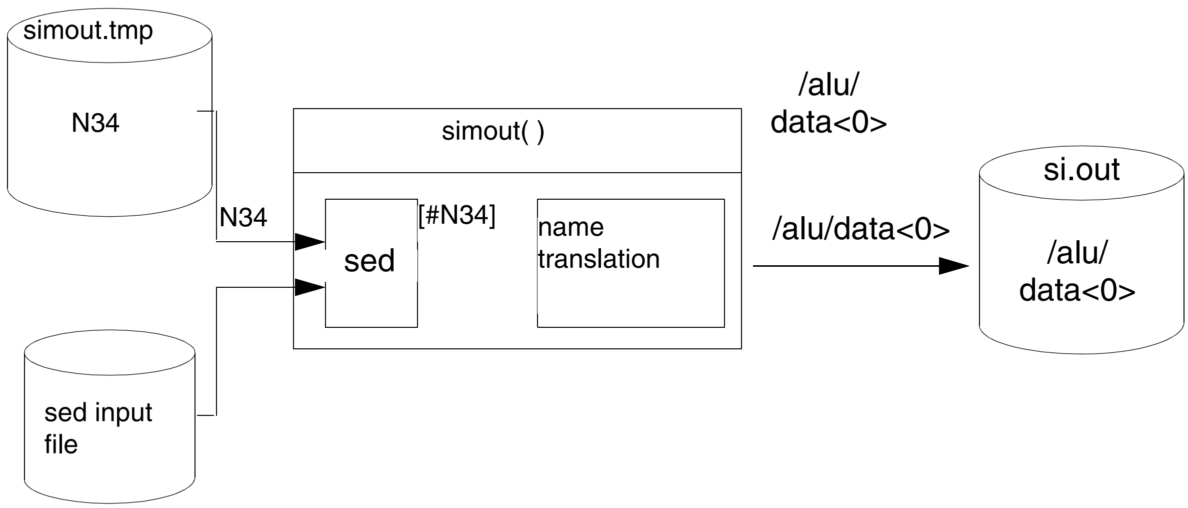
Following the opening square bracket ([ ) should be one of the following command characters:

`#`	`[#netname]`	Replace the [#netname] expression with the designer-assigned net name for netlister-assigned name `netname`.
`$`	`[$instname]`	Replace the [$instname] expression with the designer-assigned instance name for the netlister-assigned name `instname`.

Because names requiring translation are not output by the simulator surrounded by square brackets, you must insert these into the file so that the required names are translated. This is done using the UNIX `sed` function. You must provide the `sed` input file that inserts the correct substitution characters. For a description of the `sed` command language, refer to the UNIX manuals that come with your computer. Your `sed` script must parse the simulator's text output, and detect any names requiring translation. These names must then be surrounded by square brackets ([ ]), and the correct command character must be inserted after the opening square bracket ([ ). For example, if the name “netname” must be translated back to the designer-assigned name for the net, the `sed` script must replace the word “`netname`” with “`[#netname]`” so that the `simout` function translates it.

Figure 3-6 on page 64 shows how the `sed` script is used in the output name translation process.

Figure 3-6 Output Name Translation



Depending on the output syntax of your simulator, and the netlister-assigned names it requires, the `sed` script can be either simple or complex.

For example, the following is the complete `sed` script required for the text output of the SILOS simulator:

```
s/\(N [ 0-9 ] [ 0-9 ]*\ )/[#\1]/g
s/\(I [ 0-9 ] [ 0-9 ]*\)/[$\1]/g
```

Figure 3-7 is a partial `sed` script for SPICE output.

Figure 3-7 Partial sed Script for SPICE Output

```
/INPUT LISTING/,/^\.END/{
   /^\./{
      s/V(([0-9][0-9]*\)*,*\([0-9][0-9]*\)*)/V([#\1],[#\2])/g
      s/V((*\[0-9][0-9]*\)*)/V([#\1])/g
      s/V\([RIMP]\)(*\([0-9][0-9]*\)*,([0-9]
            [0-9]*\)*)/V\1([#\2],[#\3])/g
      s/V\([RIMP]\)(*\([0-9][0-9]*\)*)/V\1([#\2])/g
      s/VDB(*\([0-9][0-9]*\)*,*\([0-9][0-9]*\)*)/
            VDB([#\1],[#\2])/g
      s/VDB(*\([0-9][0-9]*\)*)/VDB([#\1])/g
}
/^[K\*]/b
/^[CLRFHVID]/s/\([0-9][0-9]*\)*\([0-9][0-9]*\)/[#\1][#\2]/
/^[JQ]/s/\([0-9][0-9]*\)*([0-9][0-9]*\)*\([0-9]
[0-9]*\)/[#\1][#\2][#\3]/
}
/(*\([0-9][0-9]*\))/s/(*\([0-9][0-9]*\))/([#\1])/g
/*$#[0-9][0-9]*/s/$#\([0-9] [0-9]*\)/[$1]/g
```

You can test your `sed` script by taking an existing textual output file from your simulator and running your `sed` script on it by issuing the following UNIX command:

```
sed -f sed_file_name < simulator_output_file_name
 > file_name
```

Then edit the *`file_name`* file and make sure that all names requiring translation are surrounded by square brackets ([ ]) and have the correct command character following the opening square bracket ([ ).

The name of this `sed` file should be the same as the one used to set the `simSedFile` variable in the simulator-specific SE customization file (*spice*). In Figure 3-7 on page 64, this file name is the full file system path to the `sed.spi` file. It is customary to name this `sed` file `sed`, suffixed with the first three letters of the name of the simulator used, for example, `sed.spi`.

If you can modify the simulator itself to produce names in the output file with the necessary name translation syntax, then you do not need to write a `sed` file.

SE Naming Conventions

SE follows the Cadence standard naming conventions for SKILL code. The first letter of all variable and function names is lowercase. The remainder of the name is also lowercase except for the first letter of each word which is uppercase.

To ensure you do not use any of the same names used within SE, it is recommended that you begin all variable and function names with an uppercase letter. Then make the remainder of the name lowercase, except for the first letter of each word. This makes variable names effortless to read and ensures that SE does not alter the value of your variables.

SE Variables

The following is a list of global variables which are defined in SE:

simActions

Default list of functions to be executed in order to run a complete simulation. If these functions do not provide the proper sequence of steps to be performed for a particular simulator, the variable can be set in the simulator-specific file stored in the `local/si/caplib` directory with the same name as the simulator. You can set this variable outside any function or in the function of the same name as the simulator.

```
Defined in: local/si/caplib/simulate.ile
Default: ( simCheckVariables()
      simInitRunDir()
      netlist()
      simin()
      runsim()
)
```

simAlwaysAddPrefixInInstName

Always prefixes the instance name with the `namePrefix` of the device type. If set to `nil`, and if the instance name already starts with the name prefix, then the instance name is not changed. For example, if a resistor instance name is `R0`, and the `namePrefix` is `R`, the instance name is not changed to `RR0`.

When set to true, all instance names are always prefixed with the `namePrefix`. OSS maintains a table mapping the original name to the new names. Therefore, OSS stores the instance name mapping for `R0` as `R0 RR0`. Since OSS keeps a record of old and new instance names, it is also possible to backannotate, which is not possible if the netlister generates the prefixed instance names.

```
Defined In: .simrc, libInit.il, or a customer SKILL file loaded in a Virtuoso session

Default value: nil

Example: simAlwaysAddPrefixInInstName = t
```

You can override the `simAlwaysAddPrefixInInstName` variable to always add the prefix for hierarchical instances but not for primitives, by using the following procedure:

```
procedure(auCdlAlwaysAddPrefixInInstName()
    ;;do not always add prefix to primitives
    if(hnlIsAStoppingCell( hnlCurrentMaster ) then
      simAlwaysAddPrefixInInstName = nil
    else
    ;;always add prefix to hierarchical instances
      simAlwaysAddPrefixInInstName = t
    )
)
```

simCapUnit

Scaling factor for capacitance. Refer to the netlister documentation for details of how the value is used.

```
Defined in: etc/skill/si/simcap.ile
Default: 1.0e-15
```

simCellName

Name of the top-level cellname of the design.

```
Defined in: si.env
Default: myCell
```

simCellViewModifiedAction

When this variable is set to `warning` or `ignore`, netlisting continues when some edits are made to a writable schematic that does affect the connectivity of the design. If however, this variable is set to `error` or is not specified in the `.simrc`, an error message is displayed and netlisting discontinues.

```
Defined in: .simrc
```

simCleanFileList

List of files to delete from the current run directory when the `simCleanRun()` function or clean run menu command is executed. The `simCleanFileList` variable should be set to a list of file names, for example,

```
simCleanFileList='("save.cmm" "save.sim")
Defined in: simulator-specific file in local/si/caplib directory.
Default: nil
```

simCommand

Actions to perform to run the simulator used for this simulation. This variable is used by the `runsim()` function to run the simulator.

```
Defined in: simulator-specific file in local/si/caplib directory.
Default: None
```

Example:

```
simCommand = 'prog( (cmd)
simDeleteRunDirFile("raw/waves")
sprintf( cmd "cd %s" "exec pacsim si.inp 2>&1 >simout.tmp"simRunDir)
if( ! simExecute(cmd) then
simPrintError("si: Pacsim did not complete without errors. \n" )
return(nil)
)
return(simPacsimRawToWaves( "waves.tmp" ) )
)
```

simCompleteMessage

Default completion message printed by the `sim()` function. This message is printed if the function does *not* have an error during execution.

```
Defined in: etc/skill/si/simcap.ile
Default: "Simulation completed successfully."
```

simControlFile

Full pathname to control file template usually stored in the `etc/si` directory.

```
Defined in: etc/skill/si/simcap.ile
Default: nil
Example: simControlFile = "/usr2/companyDefaults/etc/si/control.sil"
```

simDefaultControl

Relative pathname to the default control file template.

```
Defined in: etc/skill/si/simcap.ile
Default: nil
Example: simDefaultControl = "control.sil"
```

simDefaultRunDir

Relative pathname to the default simulation run directory.

```
Defined in: etc/skill/si/simcap.ile
Default: nil
Example: simDefaultControl = "silos.run1"
```

simDefaultSimulator

Name of the default simulator. You can use this variable to set the name of the default simulator in your site.

```
Defined in: etc/skill/si/simcap.ile
Default: nil
Example: simDefaultControl = "silos"
```

simDoNetlist

Non-nil if a new netlist should be generated.

```
Defined in: etc/skill/si/simcap.ile
Default: t
```

simDoNotForkNetlist

If set to `nil`, the netlist function will be forked and run under a new process. This variable is only used when netlisting does not run in the graphics program. Netlisting in the graphics program is never run under a forked process.

```
Defined in: etc/skill/si/simcap.ile.
Default: nil
```

simFailedMessage

Default completion message printed by the `sim()` function. This message is printed if the function has an error during execution.

```
Defined in: etc/skill/si/simcap.ile
Default: "Simulation did not complete."
```

simGenWarnings

Variable to indicate whether to generate warnings for user overrides of simulator initialization. This variable is used by the `simSetDef()` function.

```
Defined in: etc/skill/si/simcap.ile
Default: t
```

simHost

Host name of the computer on which the simulator is to be run. This variable is used only for remote simulation.

```
Defined in: etc/skill/si/simcap.ile
Default: simHost = "localhost"
```

simHostDiffers

Variable is `nil` if the remote host used for simulation has an identical binary storage format; otherwise, it is `t`. This variable is used only for remote simulation.

```
Defined in: etc/skill/si/simcap.ile
Default: nil
```

simIgnoreTerm

OSS reads the `nlAction` property on terminals and ignores the terminals while netlisting if the `simIgnoreTerm` variable is set to `true` and the `nlAction` property is set to `ignore`.

If the `simIgnoreTerm` variable is not set or is set to `nil`, OSS does not read the `nlAction` property on the terminals.

```
Defined in: simulator-specific file in local/si/caplib directory or .simrc file.
Default: nil
```

simInitRunActions

List of functions to call when initializing a new run directory. If you want a different sequence of steps when a run directory is initialized for your tool, you must redefine this variable in your tools `caplib` file.

```
Defined in: etc/skill/si/caplib/init.ile
Default: simInitRunActions = '(simInitControl()simInitRaw())
```

simInstNamePrefix

String prefix that should be used when outputting FNL-assigned instance names. If `nil`, only numbers are output.

```
Defined in: etc/skill/si/simcap.ile
Default: nil
Example: simInstNamePrefix = "I"
```

simLibName

The name of the library containing the top-level cellview of the design.

```
Defined in: si.env
Default: "myLib"
Example: simLibName = "testLib"
```

simMaxNetlistErrors

Specifies the maximum number of errors that can be encountered during netlisting before netlisting aborts. Currently, FNL, not HNL, observes this variable.

```
Defined in: etc/skill/si/simcap.ile
Default: 25
Example: simMaxNetlistErrors = 50
```

simModelNamePrefix

String prefix used when outputting FNL-assigned model names. If this is set to `nil`, only numbers are output.

```
Defined in: etc/skill/si/simcap.ile
Default: nil
Example: simModelNamePrefix = "Model"
```

simNetlistHier

Non-nil if the netlist should be hierarchical.

```
Defined in: etc/skill/si/simcap.ile
Default: nil
```

simNetNamePrefix

String prefix that should be used when outputting FNL-assigned node names. If `nil`, only numbers are output.

```
Defined in: etc/skill/si/simcap.ile
Default: nil
Example: simNetNamePrefix = "N"
```

simNlpGlobalCellName

Name of the cell containing the global netlist format property definitions for FNL.

```
Defined in: etc/skill/si/simcap.ile
Default: "nlpglobals"
```

simNlpGlobalLibName

Name of the library containing the global netlist format property definitions for FNL.

```
Defined in: etc/skill/si/simcap.ile
Default: "basic"
```

simNlpGlobalViewName

Name of the view containing the global cellview netlist format property definitions for FNL.

```
Defined in: simulator-specific file in local/si/caplib or etc/skill/si/caplib
Default: None
```

simNotIncremental

When set to `t`, HNL will try to netlist in non-incremental mode if the target simulator formatter will allow it to.

```
Defined in: etc/skill/si/simcap.ile
Default: nil
```

simPcellPrefix

String prefix for all Pcells. If set to `nill` or not defined, there will be no impact on the existing Pcell names.

```
Defined in: .simrc
Default: nill
Example: Pcell name Xpcell246Y is renamed to myPrefix_Xpcell246Y in the netlist.
```

simPinGlobals

When set to `t`, the top-cell sub-circuit header as well as other cells list all global nets that are physically connected to at least one instance terminal and all resolved inherited terminals. All these new terminals propagate upwards until the top cell. Global nets that are floating or not connected to any instance terminal (but may be connected to pins only) are not listed in either the top cell or the sub-circuit of the current cell.

```
Defined in: .simrc
Default: nil
Example: simPinGlobals='t
```

simReNetlistAll

When set to `t`, will force HNL (when running in incremental mode) to renetlist all cells used in the design.

Note: When HNL is not running in incremental mode, all cells are always renetlisted.

```
Defined in: etc/skill/si/simcap.ile.
Default: nil
```

simViewName

Name of the top-level view of the design.

```
Defined in: si.env in the simulation run directory
Default: "myView"
Example: simViewName = '("verilog")
```

simViewList

List of views to attempt to open for each cell when traversing the design hierarchy during netlisting and name translation (`simin()`, `simout()`). This variable must be set by the function of the simulator name, for example, `silos( )`, to be the simulator-specific view list. For example, the `silos()` function sets it as follows:

```
simViewList = silosSimViewList
```

```
Defined in: etc/skill/si/simcap.ile
Default: "si: no view list has been specified."
```

simRunDir

Directory in which SE data is stored.

```
Defined in: bin/si and also in the Cadence graphics program
Default: depends on simulator and design
Example: simRunDir = "/mnt2/dave/alu_simulations/silos1"
```

simRunningInSi

Variable to indicate SKILL code is being executed in `bin/si`. If this variable is set to `nil`, then you can make use of SKILL functions that require graphics (such as `hiDisplayForm( )`). By testing the value of this variable, you can write SKILL code that can be run in both SE and the Cadence graphics program.

```
Defined in:  bin/si and also in the Cadence graphics program
Default: t in bin/si; nil  in the Cadence graphics program
```

simSedFile

Name of the `sed` input script used by the `simout()` and `simOutWithArgs()` simulator output name translation functions. This `sed` input script must surround each name that requires translation in the simulator output file with square brackets ([ ]) and insert the correct command character after the opening bracket ([ ). For example, if the name `netname` must be translated back to the designer-assigned name for the net, the `sed` script must replace the word `netname` with `[#netname]` so that the `simout` function translates it. For more information, refer to the `simout()` or `simOutWithArgs()` functions.

```
Defined in: simulator-specific file in local/si/caplib
Default: depends on simulator
Example: simSedFile = "/cds/local/si/sed.sil"
```

simSimulatorSaveVars

List for symbols whose values you want written to the environment file, in addition to the default variables. The type of the value will be checked, and the proper format will be printed into the `si.env` file.

```
Defined in: simulator-specific file in local/si/caplib
Default: nil
Example: simSimulatorSaveVars = '(compareSchematic compareLayout)
```

simSimulatorUnbindFuncs

Specifies the functions in your simulator-specific `caplib` file that must be unbound when designers switch simulators. Define this variable in the `caplib` file outside of any function definition. Set it to a list of the names of all functions you define. If you do not include a function in this list, the correct function might not be used when designers switch simulators.

```
Defined in: simulator-specific file in local/si/caplib
Default: depends on simulator
Example: setting this variable in the caplib/silos.ile file.
simSetDef( 'simSimulatorUnbindFuncs,'( silos simInitSilosIncludes ))
```

simSimulatorUnbindVars

Specifies the variables in your simulator-specific `caplib` file that must be unbound when designers switch simulators. Define this variable in the `caplib` file outside any function definition. Set it to a list of the names of all global variables you define. If you do not include a variable in this list, designers will get warning messages indicating this variable has already been defined, and the correct value for the variable will not be used when they switch to a different simulator.

```
Defined in: simulator-specific file in local/si/caplib
Default: depends on simulator
Example: setting this variable in the caplib/silos.ile file.


simSetDef( 'simSimulatorUnbindVars, '(silosSimViewList
   silosSimStopList
   simDefaultControl silosBinary
   simSilosSave simNetNamePrefix
   simInstNamePrefix
   simModelNamePrefix
   )
)
```

simStopList

List of views that are valid stopping points for expansion used during netlisting. This variable must be set by the function of the simulator name, for example, `silos()`, to be the simulator-specific stopping list. For example, the `silos( )` function sets it as follows:

```
simStopList = silosSimStopList
```

```
Defined in: simulator-specific file in the local/si/caplib directory.
Default: None
Example: simStopList = '("silos")
```

simTimeUnit

Scaling factor for delay times. Refer to the “Customizing the Flat Netlister (FNL)” chapter in this manual for details of how the value is used.

```
Defined in: etc/skill/si/simcap.ile
Default: 1.0e-9
```

simSimulator

```
Defined in: si.env in the simulation run directory
Default: "spice"
Example: simSimulator = "spectre"
```

simSupportDuplicatePorts

Determines whether to remove duplicate ports from a netlist. The value of this variable is `t` by default and the simulator accepts duplicate ports. If the variable is set to `nil`, the netlister removes duplicate ports from a netlist.

Note: The removal of duplicate ports might slow OSS down for a large design.

```
Defined in: User customization file,.simrc or Virtuoso® Design Environment
workbench if running in foreground mode
Default: ‘t
Example: simSupportDuplicatePorts = nil
```

simSymbolModifiedAction

Indicates whether to report an error, or to generate a warning or ignore, in case the symbol master of an instance is newer than the time the instance was last changed.

The following table lists the valid values of `simSymbolModifiedAction`, along with the actions that the netlister takes for these values.

Value	Netlister Action
`ignore`	Continue generating the netlist without displaying any message. This is also the default action that the netlister performs when the variable is not set.
`warning`	Continue generating the netlist, and display a warning message about the symbol master being newer than the corresponding instance.
`error`	Stop generating the netlist, and display an error message about the symbol master being newer than the corresponding instance.

```
Defined in: etc/skill/si/simcap.ile
Default: ignore
Example: simSymbolModifiedAction="warning" ;
```

SE SKILL Functions

The SKILL functions defined by the simulation environment (SE) let you simplify the integration of your simulator. These functions are in both the Cadence graphics program and the SE program.

For details on the SE SKILL functions, see *OSS Functions* in the *Digital Design Netlisting and Simulation SKILL Reference*.

SE Graphics Variables

You can use the following SE graphics variables to determine what SE does at the end of a batch job:

```
simPostAnalysisProcessingFunc
simDoNotDisplayDialogBox
```

These variables let you register a function to be called after a batch job started in SE completes. For example, you could have SE clean up the run directory or generate required files for the next steps in the design analysis process.

To use this feature, you must write your post-analysis processing function and register your function with SE through the `simPostAnalysisProcessingFunc` variable. SE uses this variable to call your registered function.

The following example shows you how to register a function for SE to call at the end of a batch job:

```
procedure( b( paramA )
   println( paramA->status )
    println( paramA->rundir )
    t
)
simPostAnalysisProcessingFunc = 'b
```

Function `b` is the function you want SE to call. It must accept one parameter, which is a property list with two properties, `status` and
`rundir.paramA>status` returns a string indicating the status of the completed batch job: `succeeded`, `failed`, or `killed`. `paramA->rundir` returns the pathname of the run directory in which the batch job was started.

The assignment statement `simPostProcessingFunc = 'b` is where your function is reassigned.

By default, SE brings up a dialog box at the end of a batch job indicating the completion status of the job. If you do not want SE to bring up the dialog box, set `simDoNotDisplayDialogBox`, to `t`.

simDoNotDisplayDialogBox

If set to `t`, stops SE from displaying a dialog box that notifies the designer that the batch job has finished.

Defined: `etc/skill/si/simcap.ile`
Default: `nil`

simPostAnalysisProcessingFunc

Used to call the postprocessing function you registered with SE. This postprocessing function is called after a batch job completes.

Defined: `etc/skill/si/simcap.ile`
Default: `nil`

4

Customizing the Interactive Simulation Environment (ISE)

The Interactive Simulation Environment (ISE) enables designers to run simulation interactive Simulation Environment (ISE) enables designers to run simulation interactively while remaining in the Cadence graphics environment.

With an interface to a simulator developed using ISE, designers can

- Interact directly with the simulator.
- Use simulator menus.
- Point at the design to interact with the simulator.

Use OSS and SE to develop an interactive simulation interface, just as you develop a batch interface. You can customize ISE using SKILL. All of the functionality described in this manual is available to you when you develop an interactive interface.

This chapter explains the strategy behind ISE, the way designers use an interactive interface, the way ISE works, and the way you can customize ISE to run your simulator.

Understanding Interactive Simulation

Because ISE is an extension of SE and OSS, all information in this manual on generating netlists and stimulus, and integrating a simulator applies to ISE. This chapter describes extensions and functionality that relate only to interactive simulation.

Interactive and Batch Simulation

Interactive simulation differs from batch simulation. A sequence of steps for running batch simulations can be effortlessly defined. The steps performed during an interactive simulation are determined by the designer. Developing an interactive interface using ISE differs from developing a batch interface using SE because the sequence of steps cannot be determined in advance. In addition to developing the netlist interface, you must also generate a menu and the SKILL functions necessary to run the simulator interactively. This menu may contain many simulator-specific commands. Depending on the application program being integrated, the overlap of these commands with another application can be small.

Only the framework required to develop an interactive interface, or the functional primitives, are provided in ISE because the commands available in an interactive application can vary drastically. You can use these functions with SE and SKILL to create higher level commands used by designers to run interactive simulations.

Creating an Interactive Interface

ISE provides the functionality not available in SE to develop an interactive simulation interface. By building on SE, it is possible to simplify creating an interactive environment and provide consistency between a batch and an interactive interface to the same application. All the basic concepts apply to both interfaces. A netlist is required to communicate the connectivity of the design to the simulator, a single directory called the `simulation run` directory is used to store all of the inputs and outputs used and produced by the simulator, and functions are provided to translate names used in the design to names acceptable to the simulator.

You can integrate a new simulator into the Cadence environment as an interactive simulator, or you can enhance an existing batch interface to provide both batch and interactive abilities. This migration is an evolutionary process, not a complete redesign. It is simplest first to develop the batch interface so you develop a thorough understanding of SE; then extend this interface so it also functions interactively. Virtually all of the batch interface you develop will be common to the batch and the interactive environments. If you developed the batch interface as described in the “Customizing the Simulation Environment (SE)” chapter in this manual, the extension to an interactive environment involves developing the set of SKILL functions and menus necessary for issuing commands directly to the simulator.

ISE provides you with all of the functions you need to create and maintain the windows used to run a simulator interactively, communicate between windows, translate names interactively, and issue commands to the simulator.

Using an Interactive Simulation Interface

An interactive interface allows designers to interact with the simulator while remaining inside the Cadence graphics environment. It is possible to issue commands to the simulator directly or through menus and interaction with the graphic representation of the design.

Initializing the Interactive Environment

Before invoking an interactive simulation, the designer must execute the simulation environment `Initialize` command. Once the simulation environment has been initialized, the designer can invoke an interactive session. If no netlist exists in the current simulation run directory, the designer is prompted to determine if he would like one generated before starting the session.

Creating Windows for Interactive Simulation

Next, the windows required to run an interactive simulation are created. By default, the schematic and simulation windows appear. However, you can disable the schematic window.

The schematic window is opened to edit the design, and the design is automatically edited in that window.

The simulation window lets the designer interact with the simulator. This is a text window that displays text output from the simulator. The designer can use this window to issue commands directly to the simulator. The simulator specified is automatically started and linked to this window.

All of the initialization steps are controlled by SKILL variables. The default is to initialize the environment as specified above. However, a designer can modify the default initialization at this point. For example, a designer could alter the window arrangement.

Running the Simulation

Once the environment has been set up, the way the simulation is run is under the control of the designer. There are no predetermined steps. For example, a designer can modify the design in one window and instruct ISE to update the connectivity for the simulator. ISE netlists the design in the foreground and instructs the simulator to read in the netlist. Once these steps are completed, control is returned to the designer.

The designer can instruct ISE to update the simulator with new stimulus for running the simulation. This may require the user to create the input stimulus or simply translate a file of designer-entered stimulus. The translation of the input file is done to convert the designer-assigned names to names assigned by the netlister. Once the stimulus is prepared for input to the simulator, ISE automatically instructs the simulator to read in the stimulus.

The designer can instruct the simulator to simulate for a period of time. ISE issues the appropriate command to the simulator and returns control to the designer. The designer can continue to work while the simulator is running the simulation. As the simulation is running, text output from the simulator is continuously being updated in the simulation window. This display is updated until the designer requests the updating to stop. Updating is also suspended while a menu is on the screen.

Once the simulation stops, the designer can instruct ISE to set a specified node to a specific value. The designer selects the command and points at the node to be set in the schematic window. ISE determines the netlister-assigned name of the selected node and issues the appropriate instruction to the simulator. Next, the designer can instruct the simulator to continue the simulation by using an ISE command or by typing directly into the simulation window.

Ending the Session

At any point, the designer can end the simulation session by selecting the `Finish Interactive` entry from the menu. ISE issues the appropriate commands to terminate the simulator and closes all of the windows it opened.

When the designer ends the session, the environment is the same as it was before starting ISE. The designer must instruct ISE to terminate the session instead of closing the windows manually, or all of the ISE processes will not be properly terminated until the designer leaves the graphics environment.

Customizing ISE

To fully utilize ISE, you must understand the environment that has been created and make use of the features in a structured and ordered manner. The following sections describe different aspects of ISE that you can use while developing your interface:

ISE Interfacing Steps

Describes the steps to perform when developing an interface using ISE.

Foreground Simulation Environment

Describes how you can use SE commands when developing your interactive interface.

Window Environment

Describes the windowing environment and how it can be used by an interactive interface.

Menu Commands

Describes how to create menus that make use of the windowing environment and issue instructions to the appropriate application.

ISE Variables

Describes the variables defined by ISE that you can use in the Cadence graphics environment to develop an interactive interface to your simulator.

ISE Functions

Describes the functions defined by ISE that you can use in the Cadence graphics environment to develop an interactive interface to your simulator.

ISE Interfacing Steps

When you create an interactive simulation interface, you perform these steps:

- Generate a batch interface
- Design the interface for graphic and nongraphic execution
- Create an interactive menu and SKILL commands

Generate a Batch Interface

Before you can generate an interactive interface to your simulator, you should generate a batch interface. Aside from the flexibility this affords your designers, the interface you develop will be the starting point for your interactive interface. To develop a batch interface to your simulator, follow the steps outlined in the “Integration Steps” section of the “Integrating Simulators” chapter of this manual.

Design the Interface for Graphic and Nongraphic Execution

After developing a batch interface to your simulator, make sure it also executes as a foreground SKILL command in the graphics environment. The main differences between the graphic and nongraphic environments are in displaying messages and executing UNIX commands. If you follow the instructions in the “Customizing the Simulation Environment (SE)” chapter in this manual, your interface should already be capable of executing in both environments. To ensure that messages can be displayed in both environments, always use the `simPrintMessage()` and `simPrintError()` functions instead of writing directly to the *stdout* and *stderr* ports. To ensure that any programs you invoke as part of the simulation process will function in both environments, always use the `simExecute()` function instead of `system()`, `csh()`, or `sh()`. Never use any SKILL functions that require a graphic interface unless you are certain the command is invoked only in the graphics environment. You can test which environment the function is executing in by testing the value of the `simRunningInSi` variable.

Create an Interactive Menu and SKILL Commands

The last step in creating an interactive interface is creating the interactive menu and corresponding set of SKILL commands your designers will use when running an interactive simulation. Creating menus is described in the *Virtuoso Design Environment User Guide*. You can code the menus in SKILL and they will invoke the SKILL commands you specify. It is not possible to describe the commands you need to create for your simulator. However, there are basic features common to most simulators. For example, a command to update the netlist would invoke the `iseUpdateNetlist()` ISE function. Another common command is to force a node to a specific value. To do this, you can write a function that will get the simulator input name of the node the designer is pointing at by calling the `iseGetExtName()` ISE function. You then create a command in your simulator’s input syntax using that name to force the node and issue the command to the simulator using the `isePrintSimulatorCommand()` ISE function. For a full description of the functions and variables available to you in ISE (in addition to the SE functions described in the “Customizing the Simulation Environment (SE)” chapter in this manual), refer to the “ISE Variables” and “ISE Functions” sections at the end of this chapter.

Foreground Simulation Environment

Most of the functionality required to create an interactive simulation interface is provided by the foreground SE capabilities. To generate an interactive interface, you must be able to reference all global environment variables used by SE, as well as being able to netlist and translate names as a programmable part of your interface. Depending on the completion status of any particular command, you may need to execute special functions. These capabilities are all available in SE.

Global Environment Variables

All global environment variables defined in batch SE are also available directly from SKILL once the `Initialize` menu command has been executed.

SE Functions

All functions defined in SE are directly available to you from SKILL once the `Initialize` menu command has been executed. The same functions used to develop a batch simulation interface can be called by a SKILL procedure you write. For example, you can use the `netlist()` function provided by SE to generate a netlist. This function executes in the same manner in the graphics environment as in the standalone, nongraphic SE program. The function does not return control until it has completed. Once it has completed, it will return `t` on success, or `nil` on failure. Your SKILL procedure can use this return status to determine whether to issue an instruction to the simulator to input the netlist or load a probe file into the schematic window to highlight the error regions in the design that caused the netlister to fail.

All SE variables and functions are described in the “Customizing the Simulation Environment (SE)” chapter in this manual. Most of these functions are not available until the `Initialize` command has been executed.

Differentiating Between Graphic and Nongraphic Environments

Some functions you develop may behave differently depending on the availability of a graphic interface. For example, when graphics are available, you might want to display a form that enables the designer to verify the parameters used to invoke the simulator. This is only possible in the graphics environment. In addition, both the ISE and the SE code used to invoke batch simulations reside in the same simulator-specific `caplib` file described in the "Customizing the Simulation Environment (SE)" chapter in this manual. The `simRunningInSi` variable enables you to determine in which environment a procedure is invoked. You should never set this variable. In the graphics environment, it is set to `nil`. In the non-graphics environment, it is set to `t`.

Window Environment

With ISE you can issue commands to the simulator, and the Graphics Editor directly. If one of the windows ISE uses has been closed, ISE will not execute the functions that interact with that window.

The functions provided by ISE to interact with the windowing environment are documented in the "ISE Functions" section at the end of this chapter.

Menu Commands

After developing a batch interface to your simulator, you can convert it to an interactive interface by creating the commands and menus required to interact with the simulator and design. You can customize the menus for the schematic and simulation windows. ISE does not bring up any default menus except the Probe menu in the schematic window and the System menu in the simulator window.

To create a menu, define it (as a pull-down menu) using the menu creation routines (such as `hiCreateMenuItem`, `hiCreatePulldownMenu`, etc.) described in *User Interface SKILL Functions*. One menu handle is returned per call to the routine `hiCreatePulldownMenu`.

To instruct ISE to bring up the menus you define, use the `iseSchematicMenuHandle` and `iseSimulatorMenuHandle` variables. The `iseSchematicMenuHandle` variable is the SKILL list of menu handles for the schematic window. The `iseSimulatorMenuHandle` variable is the SKILL list of menu handles for the simulation window.

For example, to bring up menus in the schematic window with the handles, A and B, and menus in the simulation window with the handles, C and D, write the following:

```
iseSchematicMenuHandle = list( A B )
iseSimulatorMenuHandle = list( C D )
```

You must create these menus before you can call the `iseInitSimWindow` and `iseInitSchematicWindow` functions.

The best place to put menu-generation code is the simulator-specific `caplib` file. Enclose the code by an `if-then` statement, where the `if` condition specifies whether the current simulation session is invoked in the graphics or nongraphics environment. An example of the menu-creation code is shown here:

```
if( ! simRunningInSi then
   menuItem1 = hiCreateMenuItem(
      ?itemText "start simulator"
      ?callback "iseStartSimulator()"
      ?name 'menuItem1)
   menuItem2 = hiCreateMenuItem(
      ?itemText "exit simulator"
      ?callback "simVerilogExitSimulator()"
      ?name 'menuItem2 )

   myPulldownMenuHandle = hiCreatePulldownMenu(
      'myPulldownMenuHandle
      "simulator commands"
      list(       'menuItem1
            'menuItem2 )
      "this is a help message"

   iseSimulatorMenuHandle = list( myPulldownMenuHandle )
)
```

If your menu has several pull-down menus, create them similarly. Then, put the menu handles together into a SKILL list and pass it to ISE through the variable `iseSimulatorMenuHandle`. For example:

```
iseSimulatorMenuHandle = list( myPulldownMenuHandle1
   myPulldownMenuHandle2
   ........ )
```

The commands you create depend on the capabilities of your simulator.

Filtering Simulator Inputs and Outputs

Use the `iseFilterInputFunc` and `iseFilterOutputFunc` variables to filter inputs and outputs to and from the simulator. These variables let you register functions to be called when there are inputs (simulation commands) to or outputs from the simulator. Then you can filter this data before passing it back to ISE to send to the simulator or print to the simulation window. To pass the filtered input to ISE to send to the simulator, call the `iseCommToSimulator(text)` function. To pass the filtered output to ISE, call the `iseSendOutputToEncapHistory(text)` function.

Creating User Specified Window Placement

Use the `iseSchWinAttrId`, `iseSimAttrId`, and `iseWaveAttrId` variables to specify default window size and placement of the windows opened by ISE in your `.Xdefaults` file. You can use these variables to specify the attribute string name you use in your `.Xdefaults` file to identify the defaults for each window.

To specify the desired window size in the `.Xdefaults` file, the keyword `Opus` must to be added in front of the window attribute string (example: `Opus.userSchSize: 100x200+10+400`). When you change the `.Xdefaults` file, you need to use `xrdb .Xdefaults` to read the change into the X resource database. Otherwise, the new window sizes won't be used.

ISE Variables

This section describes the variables defined in ISE.

iseDontOpenSchematicWindowIfOneExists

If set to `t`, ISE searches all windows and makes the ISE schematic window the first window it finds with the appropriate design that was opened in append mode. If no such window exists, ISE searches all windows and makes the ISE schematic window the first window it finds with the appropriate design that was opened in read or write mode. If no such window is found, ISE opens a new schematic window with the appropriate design opened in append mode.

Default: `nil`

iseExitSimulator Command

Command issued to terminate the simulator. You must set this variable to a SKILL string that is the exit command for the simulator.

Default: `nil`

iseFilterInputFunc

Registers your function so that any command string typed into the simulation window (up to the last carriage return/line feed character) is returned to you by the ISE function `iseGetInputFromEncapWindow()`. You can then filter this simulator command before you pass it back to ISE to send it to the simulator.

Default: `nil`

iseFilterOutputFunc

Registers a function to be called whenever ISE receives outputs from the simulator. You can then filter these outputs if needed before you pass them back to ISE for printing to the output portion of the simulator window.

Default: `nil`

iseInitSchWindowFunc

Name of the function for initializing the schematic window.

Default: `iseInitSchematicWindow()`

iseInitSimWindowFunc

Name of the function for initializing the simulator window.

Default: `iseInitSimWindow()`

iiseInputNetlistCommand

Command for reading the netlist into the simulator.

Default: `nil`

iseInputStimulus Command

Command for reading stimulus into the simulator.

Default: `nil`

iseInvokeSimulatorFunc

Defines the routine you want to call for preprocessing before invoking the simulator. Before the routine returns, you must define the variable `iseRunSimulatorCommand`. The value of this variable must be the command ISE needs to invoke your simulator. The `iseInvokeSimulatorFunc` variable takes precedence over `iseRunSimulatorCommand`. If `iseInvokeSimulatorFunc` is defined, it will be used first; that is, the preprocessing routine will be called. When the preprocessing routine returns, the variable `iseRunSimulatorCommand` must be defined.

Default: `nil`

iseOpenWindowsFunc

Default function for opening windows used by ISE.

Default: `iseOpenWindows()`

iseReleaseFunc

Name of the function for defining and sending the command to release a node from a preset value.

Default: `nil`

iseRunSimulator Command

Command to invoke the simulator. You must set this variable to a SKILL string that is the invocation command for the simulator. Define this variable in your `.simrc` file, or if you want to do preprocessing before invoking the simulator, define it in the preprocessing function (using the variable `iseInvokeSimulatorFunc`). The variable `iseRunSimulatorCommand` must be set and returned from the preprocessing routine before the simulator can be invoked (see also `iseInvokeSimulatorFunc`).

Default: `nil`

iseSchematicMenu Handle

A SKILL list of the handle of the menu (or menu handles if you have more than one menu) you want to display in the schematic window. You can create this menu by writing SKILL code and using Cadence's menu-creation routines, such as `hiCreateMenuItem`, `hiCreatePulldownMenu`, etc.

Default: `nil`

iseSetFunc

Name of the function for printing the set node command in the simulator window.

Default: `nil`

iseSimulateFunc

Name of the function for printing the simulate command in the simulator window.

Default: `nil`

iseSimulatorMenuHandle

A SKILL list of the handle of the menu (or menu handles if you have more than one menu) you want to display in the simulation window. You can create this menu by writing SKILL code and using Cadence's menu-creation routines, such as `hiCreateMenuItem`, `hiCreatePulldownMenu`, and so forth.

Default: `nil`

iseStartSimulatorFunc

Name of the function for invoking the simulator.

Default: `iseStartSimulator()`

iseSchWinAttrId

Name of the identifier used in the `.Xdefaults` file to specify the placement and size of the schematic window opened by ISE. ISE will only use `.Xdefaults` for window placement if the user preference for window placement is set to the default value. For example, if the following line appeared in your `.Xdefaults` file:

```
Opus.userSchSize: 100x200+10+400
```

set this variable:

```
iseSchWinAttrId = "userSchSize"
```

Default: `nil`

iseSimWinAttrId

Name of the identifier used in the `.Xdefaults` file to specify the placement and size of the simulation window opened by ISE. ISE will only use `.Xdefaults` for window placement if the user preference for window placement is set to the default value. For example, if the following line appeared in your `.Xdefaults` file:

```
Opus.userSimSize: 100x200+10+400
```

set this variable:

```
iseSimWinAttrId = "userSimSize"
```

Default: `nil`

ISE Functions

OSS provides various ISE SKILL functions. For details on the ISE SKILL functions, see *OSS Functions* in the *Digital Design Netlisting and Simulation SKILL Reference*.

Remote Interactive Simulation

The interactive simulation environment (ISE) supports remote and local interactive simulation. Remote interactive simulation is starting and running an analysis tool on another machine somewhere in the network. ISE has two different modes of support for remote interactive simulation: network file system (NFS) mode and copy mode.Chk the following text wrt waveform and suggest.

You can dynamically interact with the analysis tool in either remote interactive simulation mode. In NFS-mount mode, the local run directory is mounted (either through manual mount or automount) onto the remote node where your analysis tool runs. Using this mode, you can use your integrated tool to dynamically display waveforms. In copy mode, relevant data is copied to the remote node for analysis, thus preventing the dynamic display of waveforms.

NFS-Mount Mode

NFS allows transparent access to disks located on different machines in the network. The machine where you are running the Cadence environment is referred to as the local node. The machine where your analysis tool is running is referred to as the simulation node.

Automount and manual mount are two different types of directory access under NFS. Automount automatically mounts a disk on another machine in the network when the local machine accesses a directory on the remote disk. Manual mount mounts a disk remotely, either through an entry in the `/etc/fstab` file or by using the UNIX `mount` command. ISE supports both types of directory access.

Usually, you want to take advantage of NFS mounting to run simulation on a remote machine, while pointing to your data on the local machine. With NFS-mount, the online dynamic display of waveforms is possible with any display tool that is integrated into OSS and ISE.

Copy Mount Mode

If NFS-mount is not available, you can still run remote interactive simulation using ISE. In copy mode, ISE copies all data, such as netlists and stimuli, from the local run directory to the remote machine for use in remote simulation.

Assigning the Path

In the following examples, the local node is `localmachine`, the remote node is `remotemachine`, and the local run directory in `localmachine` is

```
/usr/mySimRunDir.
```

When using NFS-mount mode, the variable `iseRemoteDiskFullPathName` provides the full path name of the mounted directory on the remote node. You can use the ISE variables to incorporate remote interactive simulation. For example, the disk "/usr" on `localmachine` is mounted onto `remotemachine` and is named `/localmachineUsr`. The simulation run directory carries the full path name `/localmachineUsr/mySimRunDir`.

- In NSF-mount mode, assign the full path name to the variable `iseRemoteDiskFullPathName` as shown.

```
iseRemoteDiskFullPathName = "/localmachineUsr/mySimRunDir"
```

Note: If you are using automount, the full path name might be `/net/localmachine/usr/mySimRunDir`, where `net` is the identification of the network. This network identification can differ from one network to another, so consult your system administrator for the precise format.

In copy mode, the variable `iseRemoteDiskFullPathName` is the full path name of the run directory on the remote node `remotemachine`. You select the location of this run directory on `remotemachine`. Relevant data, such as netlists and stimuli, are copied from the local node to this remote node for use by the analysis tools. Analysis results reside temporarily in the directory `iseRemoteDiskFullPathName` until the end of analysis. At that time, all data, including the analysis results, is copied back from the directory `iseRemoteDiskFullPathName` on `remotemachine` to the local run directory in `localmachine`.

➤ In copy mode, assign the full path name to the variable `iseRemoteDiskFullPathName` as shown.

```
iseRemoteDiskFullPathName = "/usr/disk1/mySimRunDir"
```

There is no default value for this variable. You must bind this variable if you want to run remote interactive simulation using ISE.

Selecting the Mode

The string variable `iseRemoteMode` identifies the mode of remote simulation you intend to use.

- In NFS-Mount mode, set the value of `iseRemoteMode` to `mount`:

  ```
  iseRemoteMode = "mount"
  ```

- In copy mode, set the value of `iseRemoteMode` to `copy`:

  ```
  iseRemoteMode = "copy"
  ```

There is no default value for this variable. You must bind this variable if you want to run remote interactive simulation using ISE.

Copying the Simulation Files

The variable `iseFilesForRemoteSimulation` identifies the files on `localmachine` that you want copied to the run directory on `remotemachine`. This variable is a SKILL list of fully qualified file names.

➤ In copy mode, set the variable `iseFilesForRemoteSimulation` to a list of files to copy to the remote node.

```
iseFilesForRemoteSimulation = '("/usr/mySimRunDir/si.inp"
"/usr/mySimRunDir/stimuli.file1"
"/usr/mySimRunDir/stimuli.file2")
```

The `si.inp` file is automatically included in the files to be copied to the remote node.

➤ If you do not want to copy the `si.inp` file, set the variable `iseUserWantCompleteFileCopyControl` to `t`. (The default value is `nil`.)

```
iseUserWantCompleteFileCopyControl = t
```

OSS System Requirements

For remote simulation using OSS/SI/ISE, the following system requirements must be met:

- On the local node, the executable *serv* must be accessible through your UNIX search path. This is a Cadence-supplied executable and can be found in `install_dir/tools/dfII/bin`.
- On the remote node, you must have a login directory, and the executable `serv` must be accessible through your UNIX search path.

5

Customizing the Hierarchical Netlister (HNL)

Most simulators and design analysis tools require a textual description of the design to be analyzed as input. This textual description can be either a flat or a hierarchical description of the design. A hierarchical description contains *sub-circuit* or *macro* definitions for each level of the design hierarchy and then references these descriptions in higher level portions of the design. The Hierarchical Netlister (HNL) is the Cadence-provided tool to simplify creation of a hierarchical network description. HNL traverses the design database and simplifies the connectivity information for use by the output-formatting instructions. You provide the output functions written in the Cadence standard language, SKILL, to write the connectivity information to the netlist file in the syntax required by your simulator.

This chapter contains the following sections.

- How the Netlister Works on page 98

  Provides information for the CAD developer to customize the netlister to generate the desired output.

- Support for Inherited Connections on page 121

  Provides information on how OSS handles inherited connections and supply sensitivity information.

- Support for Iterated Instances on page 131

  Provides information on how to netlist iterated instances without any expansion.

- Writing a Formatter on page 133

  Describes the order in which netlister functions are executed and guides you through the functions you must write to format the netlister output in a new syntax.

- HNL Global Variables on page 150

  Provides a list of the global variables defined by the netlister.

- HNL Access Functions on page 174

Provides a list of access functions divided into the following categories: property, database, print, miscellaneous, name-mapping.

- Incremental Hierarchical Netlisting on page 175

  Overview of incremental hierarchical netlisting.

- Writing an Incremental Netlist Formatter on page 179

  Shows you how to design a formatter that netlists incrementally.

How the Netlister Works

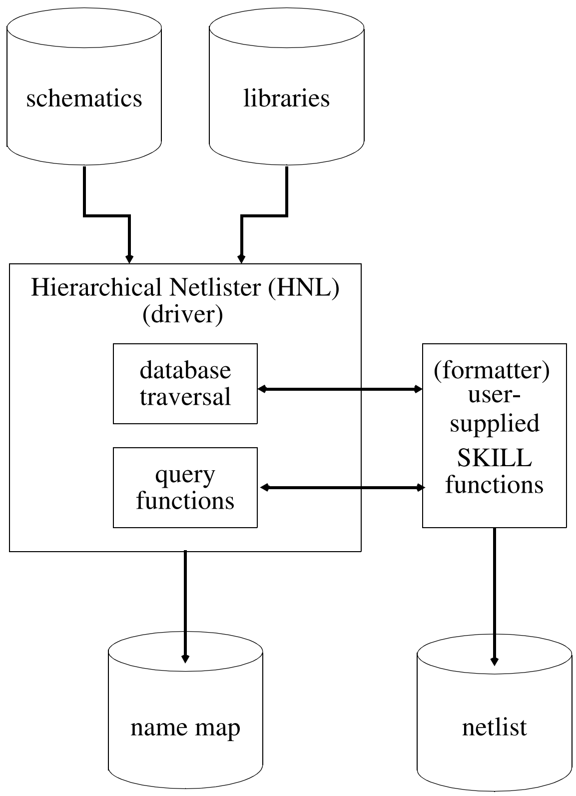
This section details the following topics:

- Introduction
- Defaults Setup
- Handling Designs with Instances of Different Place Masters Having the Same Switch Master
- Property Inheritance Warning
- Instance Ignore Conventions
- Skipping Terminals
- Output Formatting
- Map File
- Naming Conventions
- Source Code

Introduction

HNL, the hierarchical netlister, consists of two parts. The first part is the database traversal routines, or the driver; the second part is the output formatter. Figure 5-1 on page 99 is a diagram of the relationship between the two parts.

Figure 5-1 The Hierarchical Netlister (HNL) Structure



Because Cadence cannot support all the netlist syntaxes that customer simulators may require, the company provides the driver functions for the database traversal and supports output formats (formatters) for a few of the popular simulators used in the integrated circuit computer-aided design (IC CAD) industry. To address non-Cadence-supported netlist syntaxes, you can customize the netlister output by providing SKILL functions to output the textual netlist. These output functions are referred to as the *output formatter*. SKILL provides you with the ability to do flexible formatting; however, the database traversal functions required to netlist designs generated with the Cadence schematics system can be extremely complex. As a result, Cadence provides you with the database traversal and lets you write the formatting instructions to output the netlist.

The new Hierarchy Editor lets you specify instance-specific view lists to use for instances in the design. Two instantiations of the same master can carry different view lists which define how the hierarchy below should be traversed. In the past the hierarchical netlister needed to look at a unique cellview once. Now it might need to look at the same cell view more than once since the hierarchy below it can be different in a different branch of the design. To do that, HNL uses the view list (in the simplest term) as the distinguishing factor.

The traversal functions provided in the netlister are generic and suffice for most netlist syntaxes. To provide for any potential exceptions, you can override any of the functions used to traverse the database. In most cases, to modify the netlister output, you need to write the output functions for each portion of the netlist, but you do not need to write, or even modify, any of the database traversal functions. This significantly reduces the time required to write "your own netlister." In addition, little knowledge of the Cadence database, or design storage within it, is required.

For each aspect of a design, the traversal functions call an output-formatting function to handle outputting that portion of the design. If the target netlist syntax does not require any information on that aspect, the function can return a SKILL `true(t)` and not output anything. For example, some simulators require a header and a trailer for the netlist. As a result, the driver functions call a formatter function at the beginning and end of netlist generation. When netlisting for SILOS, the function that outputs netlist header information prints out a GLOBAL statement for the nets connected globally by name throughout the design. The output functions that format the end of the netlist for SILOS automatically generate clock statements for the global nets *vdd!* and *gnd!* if they are used in the design. Similarly, output-formatting functions are called at the beginning and end of the description for each cell used in the design and for each instance (reference) to a cell or gate.

The output-formatting functions also have access to certain information used or generated by the traversal functions, for example, the name of the current instance (device), the current instance, and the master of the current instance. For a full description of each variable available to the formatting functions and for a detailed description on how to customize netlister output, refer to "HNL Global Variables" on page 150" and "HNL Access Functions" on page 174 in this chapter.

To further simplify the process of writing output-formatting functions, a library of frequently needed functions is included as part of the netlister. These functions can be called by any of the output-formatting functions, and all have access to the current environment and status of the traversal functions. Included in this library are functions to determine the name of the net attached to a given terminal, a function to get the value of a property of given name, and functions to map names used in the design that may not be valid in the target netlist syntax. Using this library of functions, and the supplied database traversal functions, the output functions do not need to differentiate between busses and single-wire nets, one of the most difficult aspects of netlister development.

Defaults Setup

The netlister only sets variables and defines functions that have not been defined when it is loaded. This enables you to set variables and define any functions you want the netlister to use before loading or running it. For example, you can set default values for netlisting in the `.simrc` file in your home login directory. Then, when you load the netlister, a warning is generated if you have set any variables the netlister normally defines. This is only a warning to remind you that you have modified the defaults; it does not hinder the running of the netlister. Replacing netlister functions is considered an advanced capability and should only be done after you have a thorough understanding of how the netlister works and of the SKILL language. If you define netlister functions, the netlister uses those definitions instead of its own internal definitions for the functions. No warning is generated when you run the netlister if you define any of these functions, a feature which allows CAD developers to customize the netlister functionality to your site.

Setting defaults for the netlister in SE is the same as setting other SE defaults. For information on setting defaults in SE, refer to the *Simulation Environment Help*.

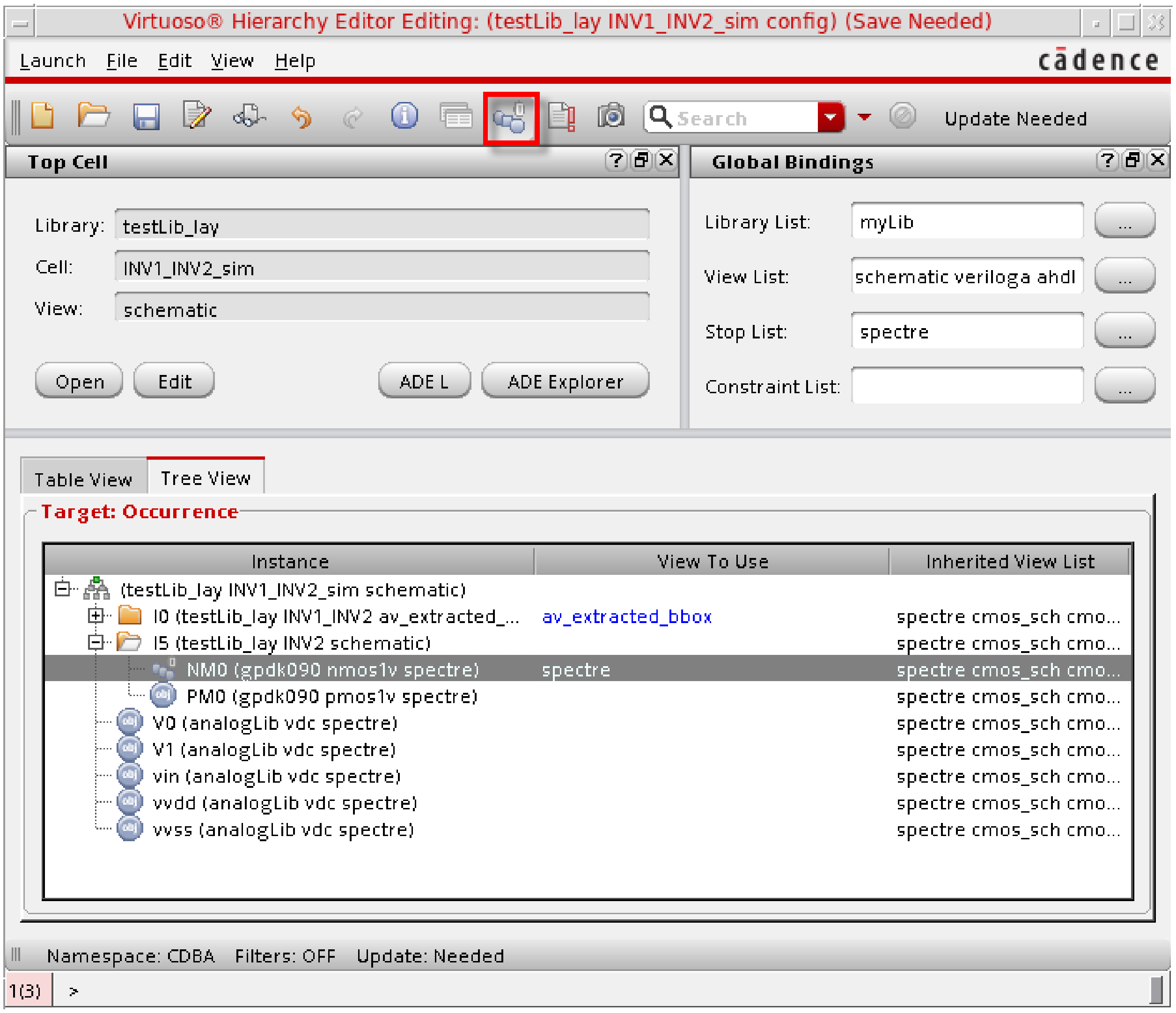
Handling Designs with Instances of Different Place Masters Having the Same Switch Master

It is possible that a design has instances of different place masters that are bound to the same switch master. If such place masters have the same terminal representations, the OSS-based netlister continues to generate the netlist correctly. If the place masters have different terminal representations, the netlist generation process stops because it can result in an incorrect netlist. For example, the netlist can be incorrect if the netlist generation process continues when one of the place masters has descending bus terminal, while the other has split bus terminals. If such a situation is encountered, apply occurrence binding to one of the instances of the switch master and regenerate the netlist.

Note: Occurrence bindings are configuration rules defined at the occurrence level. An occurrence is an instance defined by the full path from the top-level design to the instance. Therefore, occurrence bindings apply to a single object at a specific path in the design. For details, see *Occurrence Bindings* in *Virtuoso Hierarchy Editor User Guide*

Design example: Consider that a design has instances `/I5` and `/I0/I2` with different place master and the same switch master. `/I5` has bus terminals, while `/I0/I2` has flattened terminals. When you generate the netlist, the netlist generation process reports the issue and stops generating the netlist.

To netlist the design correctly, apply occurrence binding using Hierarchy Editor, as illustrated below, so that instance `/I5/NM0` is bound to `spectre`.



Property Inheritance Warning

Because of the inherent differences in hierarchical and flat netlist structure, HNL and FNL provided in SE handle simulation properties differently. Whereas properties placed throughout the design hierarchy can be used to set property values lower in the hierarchy for flat netlists, this is not possible in HNL.

Note: Only properties placed on an instance of, or in a simulation primitive (a stopping cell) appear in the hierarchical netlist.

In all other respects, the default libraries provided for FNL and HNL are identical. The same delay properties, time scale values, and gates are supported in each. The difference is the syntax of the commands used to format the netlist. Whereas FNL supports its own compact substitution expressions as well as SKILL formatting, HNL provides the full power of SKILL to netlist formatting instructions but does not support the substitution expression syntax supported by FNL.

Instance Ignore Conventions

This section details the following topics on removing devices:

- Ignoring Devices (using the `nlAction` property)
- Removing Devices with Two Terminals (using the `lxRemoveDevice` property)
- Removing Devices with Multiple Terminals (using the `hnlHonorLxRemoveDevice` and `hnlUserShortCVList` SKILL variables)

Video

For details on removing devices, view the video *Removing Devices from Netlists*.

You can also ignore devices using the following SKILL variable and properties:

- `hnlUserIgnoreCVList` SKILL variable

  Set this SKILL variable to specify the list of user-specified cellviews (instances having place master or switch master in the specified library and cell), which OSS must ignore when netlisting a design. For details, see "hnlUserIgnoreCVList" on page 173.

- `nlIgnore` property

  Set this property on an instance, or its place master or switch master, to ignore that instance when netlisting for a particular simulator. For example, when you set the property as shown below, the instance will be ignored by the `auCdl` and `vhdl` netlisters. For details, see "nlIgnore" on page 147.

  ```
  nlIgnore = "vhdl auCdl"
  ```

- `lvsIgnore` property

  Set this property to `TRUE` on an instance or its place master that you want to ignore when using the CDL netlister only.

  ```
  lvsIgnore = "TRUE"
  ```

Ignoring Devices

Sometimes you may want to place instances of symbols that do not represent components of the circuit. For example, your company might have a policy that requires its corporate logo to appear on all schematics drawn by its designers. The logo symbol can be stored in a cellview in a library along with the symbols of the circuit components such as a NAND gate or an INVERTER.

Clearly, the logo should not appear as a component in any netlists generated from the schematic, so the netlister must be told to ignore the symbol. Do this by adding a property whose name is `nlAction` on the instance, or its place master or switch master, that you want the netlister to ignore. The `nlAction` property must have a property type of `string` and the value `ignore`.

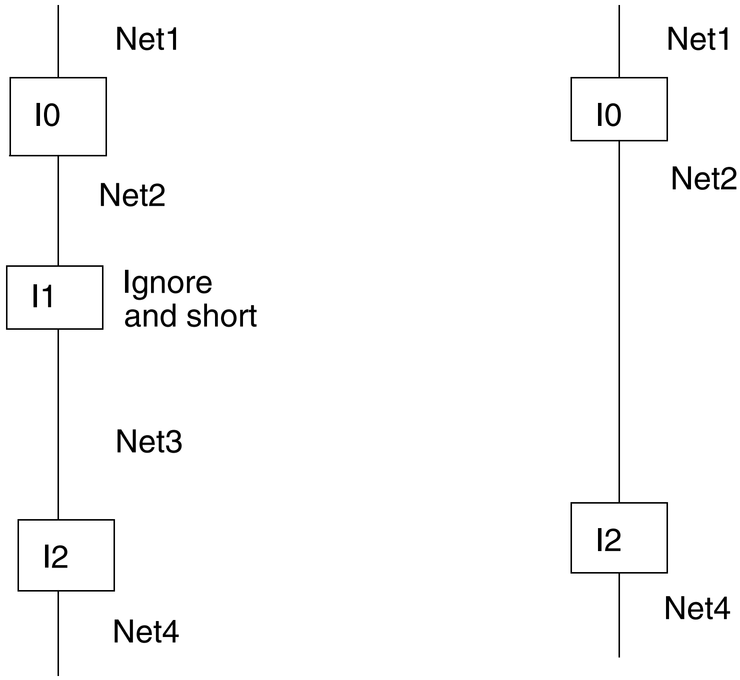
For example, suppose you want to generate a SILOS netlist from a schematic containing the logo symbol. You must have a cell called *logo* in your library with both a symbol view and a SILOS view. The SILOS view must have the `nlAction` property set to the string value of `ignore`, as shown below.

```
nlAction="ignore"
```

Removing Devices with Two Terminals

The `lxRemoveDevice` is a property which when defined on an instance with two or more terminals in the schematic, shorts the instance terminals and replaces the same with a surviving net. You can use this property to remove devices, like parasitic devices.

The following figure describes the scenario and the outcome after implementing the changes.



The above figure depicts the diagrammatic representation of how the instance is removed. It shows `Net2` as the surviving net. Instance `I1` is the parasitic device which has the property `lxRemoveDevice` specified on it with any value which is not equivalent to an empty string.

The netlister determines the surviving net based on certain rules.

Note: The above functionality is similar to the `nlAction=ignore` setting. The only difference will be the replacement of the removed instance by the surviving net.

Note: You can also use the `hnlUserShortCVList` SKILL variable to specify the cellview names in a list to remove the devices that have two terminals.

Error Handling

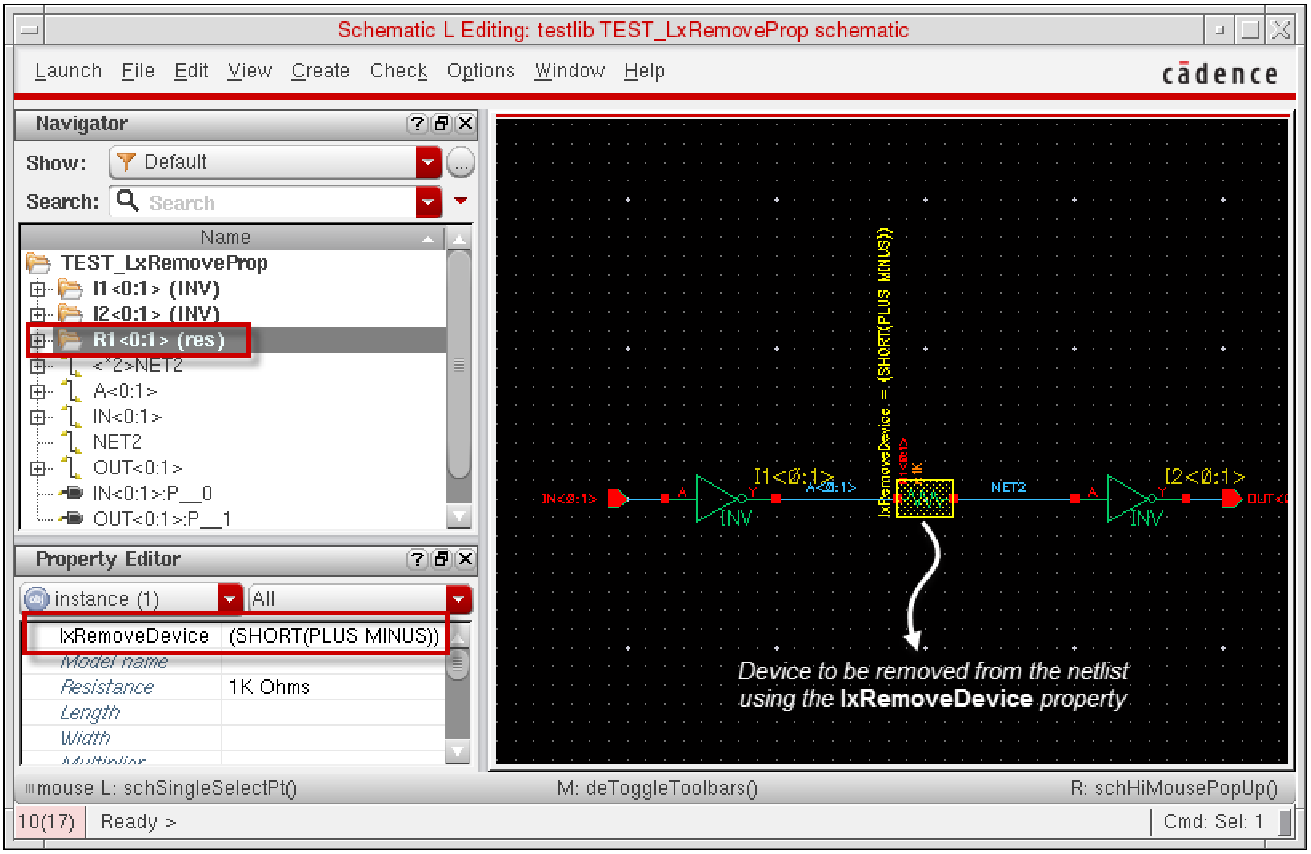
While using this functionality, you encounter some warnings in the following scenarios:

1. Invalid value of instance parameter `lxRemoveDevice` which are either NULL or empty "" strings.
2. `lxRemoveDevice` has a valid string value, but the instance has more than two terminals. In this case, ensure that the shorting rule for shorting multiple terminals is specified correctly.
3. `lxRemoveDevice` has a valid string value and the instance has two terminals. However, the nets connected to the terminal do not follow the defined criteria. It is possible that both

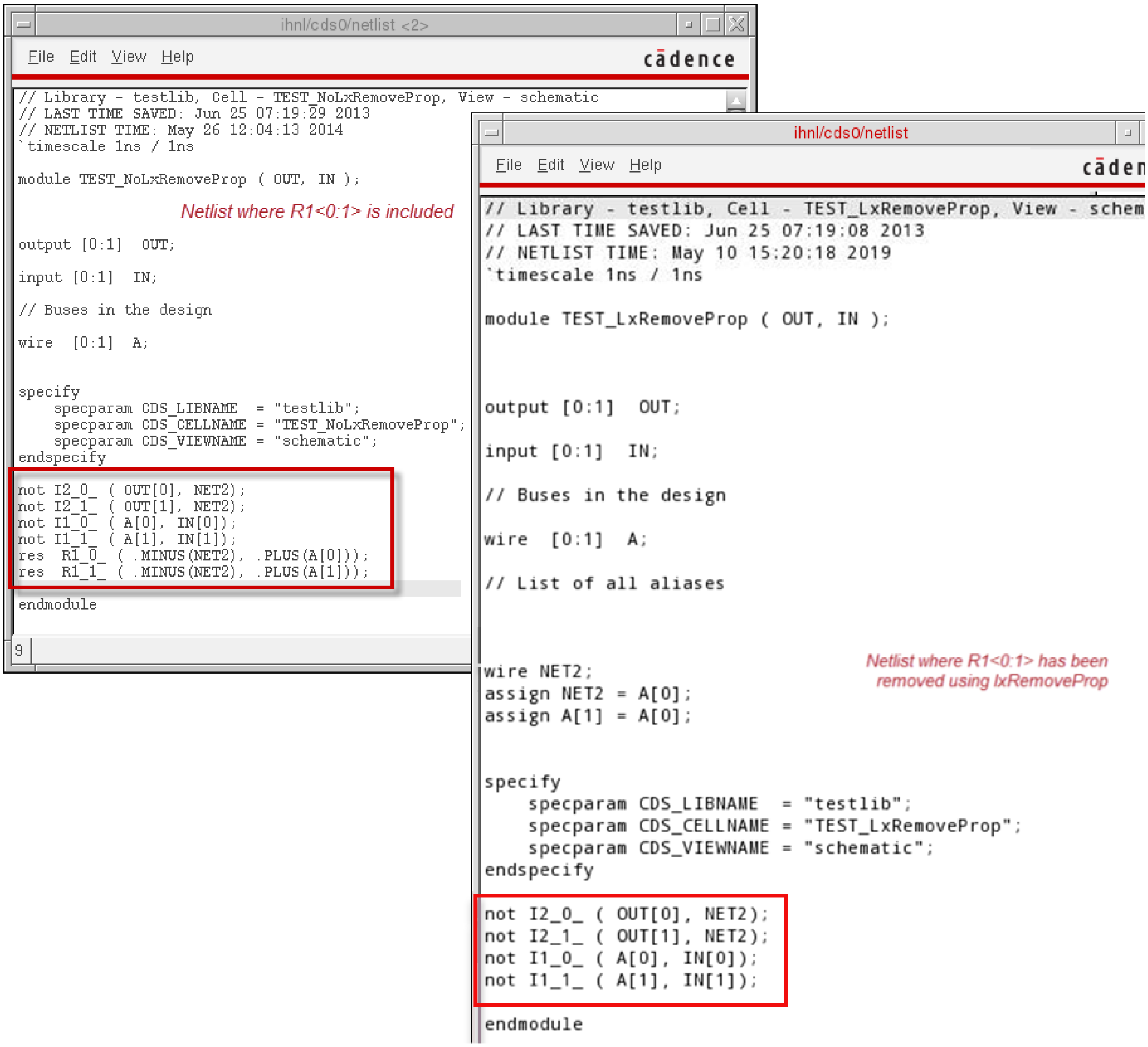
the nets are global or connected to I/O pins or one is global and one is connected to an I/O pin.

Example of Removing a Device

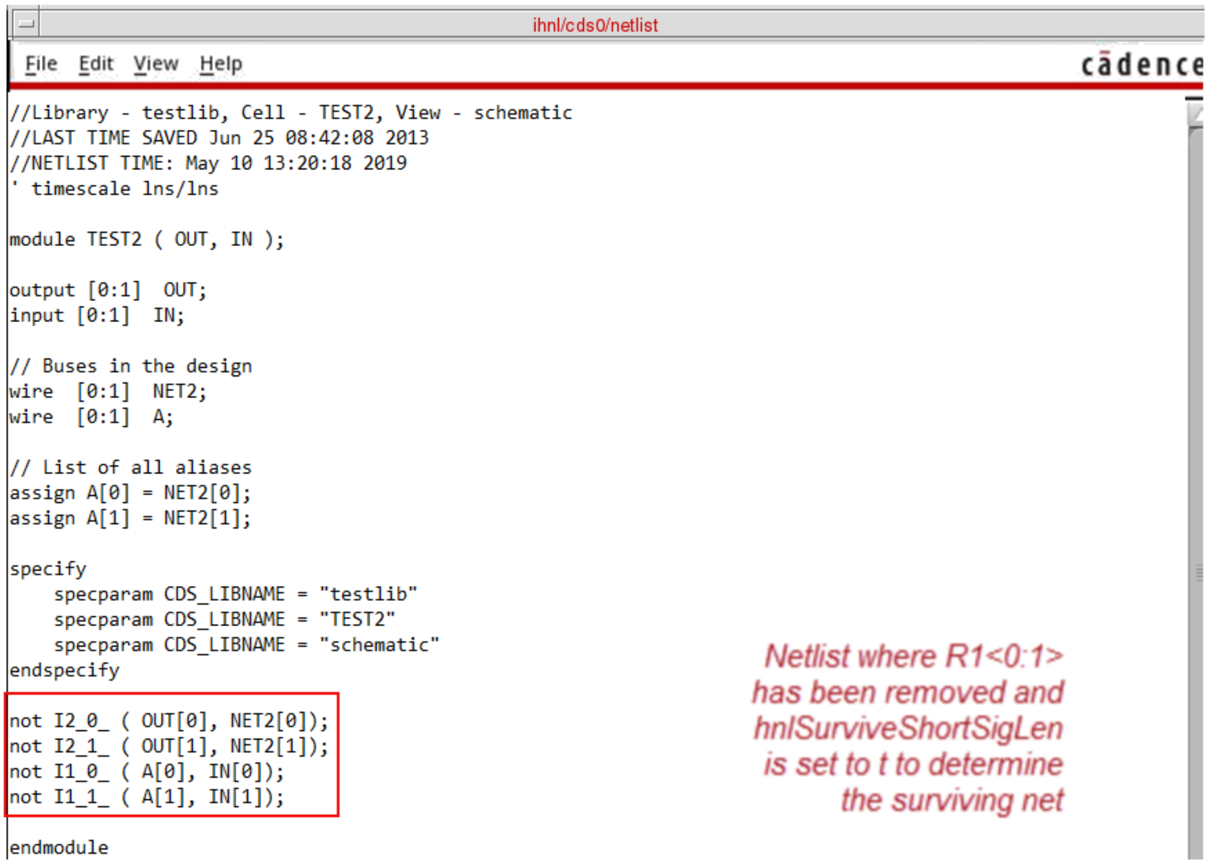
The following figure illustrates a schematic design where instance `R1<0:1>` is configured to be removed from the netlist using the `lxRemoveDevice` property.



When you netlist the design using the NC-Verilog Integration Environment, `R1<0:1>` is removed from the resulting netlist. The scalar net `NET2` becomes the surviving net for the shorted connection between `I1<0:1>` and `I2<0:1>`, as illustrated in the following figure.



If `hnlSurviveShortSigLen` was set to `t` in `.simrc` or Virtuoso CIW, before the netlist was generated, the surviving net would be `A<0:1>` because it has the shorter length. See the following figure.

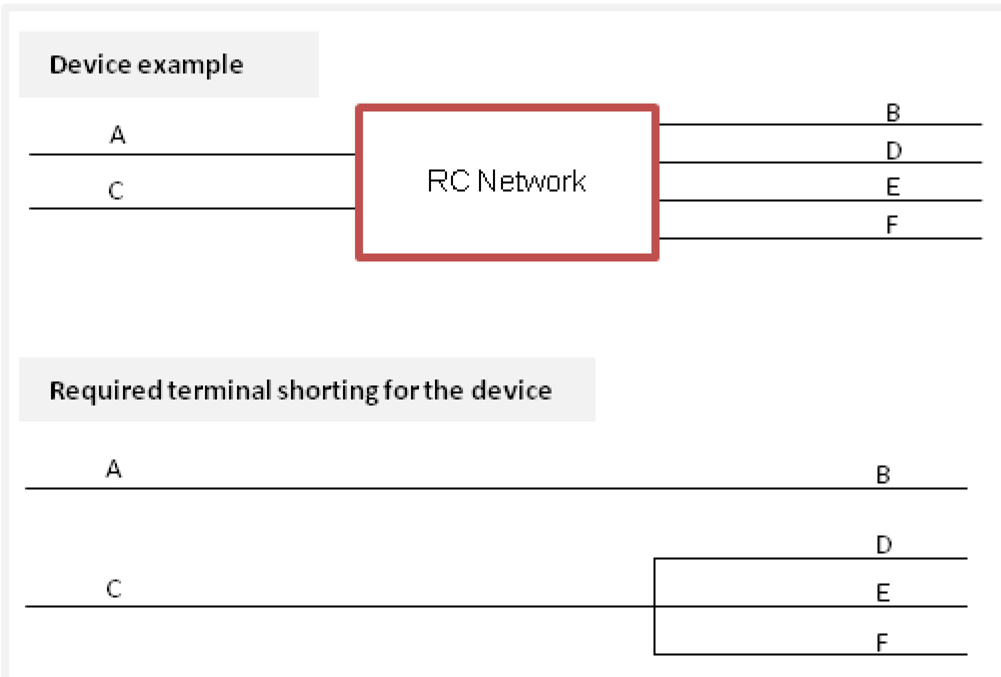



```
//Library - testlib, Cell - TEST2, View - schematic
//LAST TIME SAVED Jun 25 08:42:08 2013
//NETLIST TIME: May 10 13:20:18 2019
' timescale lns/lns

module TEST2 ( OUT, IN );

output [0:1]  OUT;
input [0:1]  IN;

// Buses in the design
wire  [0:1]  NET2;
wire  [0:1]  A;

// List of all aliases
assign A[0] = NET2[0];
assign A[1] = NET2[1];

specify
    specparam CDS_LIBNAME = "testlib"
    specparam CDS_LIBNAME = "TEST2"
    specparam CDS_LIBNAME = "schematic"
endspecify

not I2_0_ ( OUT[0], NET2[0]);
not I2_1_ ( OUT[1], NET2[1]);
not I1_0_ ( A[0], IN[0]);
not I1_1_ ( A[1], IN[1]);

endmodule
```

For details on the NC-Verilog Integration Environment, see *Virtuoso NC-Verilog Environment User Guide*.

Removing Devices with Multiple Terminals

During hierarchical netlisting, you can short the terminals of a device. For example, you can remove parasitic devices by shorting terminals. The following figure illustrates how you can short the terminals of a device.



To short the terminals of a device, set the `hnlHonorLxRemoveDevice` SKILL variable to `t` and use one of the following methods.

- Set the `lxRemoveDevice` string property at the instance level.

  Example:

  ```
  lxRemoveDevice(short(A B) short(C D E F))
  ```

  In this example:

  - `A` is shorted to `B`.
  - `C` is shorted to `D`, `E`, and `F`.

The following figure illustrates how you use the `lxRemoveDevice` property in the Property Editor assistant of Schematic Editor.

Note: `lxRemoveDevice` does not work for read-only libraries. Therefore, for read-only libraries, use `hnlUserMultiTermShortCVList` or `hnlUserShortCVList` in the `.simrc` file.

- Use the `hnlUserMultiTermShortCVList` SKILL variable to specify the cellview names and pin information in a list in the following syntax:

```
hnlUserMultiTermShortCVList  = `(("lib" "cell" "view" "(short(A B) short(C D
E)" ))
```

You must provide values for all the fields of this variable. Otherwise, the netlister reports an error.

Examples:

- Multiple cells with multiple terminals

```
hnlUserMultiTermShortCVList = `(("sample" "dffpp_c_" "symbol" "(short(in1 out3
out4))") ("testLib" "bottom" "symbol" "(short(in1 out2) short(in2 out3 out4
out5))"))
```

- A cell with two terminals

```
hnlUserMultiTermShortCVList = `(("sample" "dffpp_c_" "symbol" "(short(in1
out5))"))
```

- Use the `hnlUserShortCVList` SKILL variable in the following syntax to specify the cellview names in a list to remove the devices that have two terminals only.

```
hnlUserShortCVList = list(
                  ;;; all cells from this library
                  "libN"
                  ;;; cell1, cell2 and cell3 from lib1
                  list("lib1" "cell1" "cell2" "cell3")
                  ;;; all cells from this library
                  list("libM")  )
)
```

Note: The list should have only one entry for each library.

You can specify any of the elements in the list and keep the remaining elements blank.

Examples:

```
hnlUserShortCVList  = `("sample" "dffpp_c" "symbol")
hnlUserShortCVList  = `("sample" "dffpp_c" "")
```

If your shorting rule for a device results in the shorting of two pins, the netlister displays a warning and continues to generate the netlist without removing the device. If you want to remove the device in such cases, set `hnlEnableTerminalShort` to `t`.

When you remove a device using any of the specified methods, the application uses the following criteria to determine the surviving net:

1. If `hnlEnableDriverLoadBasedShortRule` is set to `t`, the shorting is done on the basis of the load and the driver net. The driver net is always displayed on the right.

   For example, consider that you remove device `A` in the following design.

   In this case, the shorting is as indicated below because `net1` is the driver net.

   ```
   net2 <= net1
   ```

2. If `hnlEnableDriverLoadBasedShortRule` is set to `t` and the load and driver net cannot be determined, then the net with the shorter name survives. In case, `hnlSurviveShortSigLen` is set to `nil`, the net with the smaller ASCII values of the name survives.

   Note: The smaller ASCII value is determined by SKILL `strcmp()`.

3. If `hnlEnableDriverLoadBasedShortRule` is not set:

   a. Nets connected to I/O pins or global nets survive.

   b. If the nets are at the same level of the hierarchy and `hnlSurviveShortSigLen` is set to `t` in Virtuoso CIW or the `.simrc` file, then the net with the shorter name survives. If the length of the names is also the same, then the net with the smaller ASCII values of the name survives. For example, between the following net pairs, the first net in bold text survives when the `hnlSurviveShortSigLen` is set to `t`:

      ("**net1<0:1>**" "net00<1:0>")
      ("**A<1:0>**" "<*2>net2")
      ("**net7<1:0>**" "netF<0:1>")
      ("**net3<0:1>**" "netE<1:0>")

c. If `hnlSurviveShortSigLen` is not set and the nets are at the same level of the hierarchy, then the nets with the shorter name is selected. If the length is also the same, their alphabetic sorting is done to find the surviving net.

Notes:

- Cadence recommends that you use `hnlUserMultiTermShortCVList` instead of `hnlUserShortCVList`. If required, use `hnlUserShortCVList` for devices that have only two terminals.
- OSS gives precedence to `hnlUserMultiTermShortCVList` or `hnlUserShortCVList` over `lxRemovedDevice`.

The netlister performs the following checks when processing a design that uses `lxRemoveDevice`, `hnlUserMultiTermShortCVList`, or `hnlUserShortCVLis` to remove devices:

- Property syntax.
- Mismatch in the terminals on the place master and property values.

  The netlister performs shorting on terminals that are set on the property and map to a master. It ignores other terminals.

If any of the checks fail, the netlister processes the design using the severity defined using the variable `simCheckShortCVMismatchAction`. An example of the variable definition is as follows:

```
simCheckShortCVMismatchAction="error"
```

The `simCheckShortCVMismatchAction` variable can have the following values:

- `ignore`

Check Failure Type	Action
Syntax check failure	Stop the netlisting process and raise an error.
Mismatch check failure	Do not ignore the property for the specific instance or cellview. Short the terminal sets that are found on both the place master and the property value without any messages.

- `warning`

Check Failure Type	Action
Syntax check failure	Stop the netlisting process and raise an error.
Mismatch check failure	Do not ignore the property for the specific instance or cellview. Short the terminal sets that are found on both the place master and the property value. Print warning messages for mismatch check failure. By default, the `ignoresimCheckShortCVMismatchAction` environment variable is set as `warning`.

- `error`

Check Failure Type	Action
Syntax check failure	Stop the netlisting process and display an error message. The error message specifies the instance or cellview with the issue.
Mismatch check failure	Stop the netlisting process and display an error message. The error message specifies the instance or cellview with the issue.

Important

Regardless of the method used to short the terminals of device, ensure that the `hnlHonorLxRemoveDevice` SKILL variable is set to `t` in the `.simrc` file. By default, `hnlHonorLxRemoveDevice` is not set.

The auCdL netlister is an exception and honors the `lxRemoveDevice` property even if the `hnlHonorLxRemoveDevice` SKILL variable is not set. If you do not want auCdL to honor the `lxRemoveDevice` property, set `hnlHonorLxRemoveDevice` to `nil`.

Note: The `nlAction` property directs the netlister to perform specific actions, such as ignoring instances of discreet components to prevent them from being netlisted. This property has a higher priority than the `lxRemoveDevice` property. Therefore, if `nlAction` is set to ignore, the devices are not shorted even when the `lxRemoveDevice` property is set.

Skipping Terminals

If there is a mismatch in the number of terminals or the direction of any terminal between the switched master and its placed master, you can choose to ignore or drop such terminals from the netlist. The following points explain how the netlister behaves in such cases and how you can ignore such terminals:

- If there are extra terminals on the switched master or the schematic view that are not on the placed master or symbol, the netlister ignores those extra terminals and does not print them in the netlist. This behavior cannot be changed through any variable.
- If there are terminals on the placed master or symbol view but not in the switched master or schematic view, the netlister action depends on the value of the `simCheckTermMismatchAction` environment variable. The following table lists the values that this variable can have, along with the netlister action.

Value of the Variable	Netlister Action
`ignore`	Drops the mismatched terminal from the netlist.
`ignoreall`	Ignores all the instances with unbound instance terminals. **Note:** When the variable is set to `ignore`, the netlister prints an error message for an unbound instance terminal and stops generating the netlist. When the variable is set to `ignoreall`, the netlister ignores such an instance without printing any error message and continues to generate the netlist.
`warning`	Drops the mismatched terminal from the netlist and prints a warning message.
`error`	Stops the netlist generation and prints an error message.

The formatter is usually configured to set the value of `simCheckTermMismatchAction`. You can also set the value of this variable in `.simrc`, `si.env`, or Virtuoso CIW.

If the value of this variable is not set, OSS uses the default value `error`. In this case, the netlist generation is stopped and an error message is displayed. To continue netlist generation and ignore such terminals from the netlist, set the `nlAction` string property for such terminals to `ignore` and set `simCheckTermMismatchAction` to `ignore` or `warning`.

Note: Some terminals on placed masters are created with the `physOnly` attribute. These terminals are used to implement connectivity model and are by default ignored by an OSS-based flat netlister.

- If the direction of a terminal on the place master and its corresponding terminal on the switch master does not match, by default, the netlister prints the terminal in the netlist. This is because, by default, the `simCheckTermDirectionMismatch` environment variable is set as `ignore` and the netlister ignores the mismatch in the terminal directions.

  If you do not want to ignore the mismatch in the terminal directions, set the `simCheckTermDirectionMismatch` variable to any of the following values:

  - `warning`: Drops the terminal with mismatched directions from the netlist and prints a warning message.
  - `error`: Stops netlisting when a terminal with mismatched direction is found and prints an error message.

Output Formatting

Netlister output formatting is produced by SKILL functions either written by you or released as part of the standard library with the purchase of an interface for the target simulator. These functions consist of user-defined function calls and references to documented netlister variables and functions.

Figure 5-1 on page 99 is an example of the formatting functions used by the netlister to generate a hierarchical netlist. The example consists of the formatting function for a two-input AND gate and the needed support functions. The `and2` cell and `silos` view contain a `hnlSilosFormatInst` property whose value is the `hnlSilosPrintLogGate("AND")` string. To produce format strings for `and3`, `and4`, `and5`, and `and*` cells you place the same property on `silos` views of the corresponding cells. To produce XNOR*, NOR*, OR*, NAND*, and INV formats add the same property on each of the primitives for those devices and change the name argument to the name of each device.

Figure 5-1 on page 99 illustrates the basic types of formatting functions that you must write to customize HNL output. (For simplicity, macro references are excluded from this example; they are explained in the “Writing a Formatter” section in this chapter.)

Functions not defined in this example are defined in the hierarchical netlist driver. For details, refer to “HNL Global Variables” on page 150 and “HNL Access Functions” on page 174 in this chapter.

When a two-input AND gate is encountered in the schematic being netlisted, the following type of entry results in the netlister output file by execution of the functions in Figure 5-1 on page 99.

```
andout  .AND  2  3  in1  in2
```

Figure 5-2 Formatting Functions for Hierarchical Netlisting

```
; The following functions would be defined in the formatting file
; for the target simulator. For SILOS this would be
; install_dir/tools/dfII/etc/skill/hnl/silos.ile.
;
; Set the lists of variables and functions that must be
; unbound when environments (simulators) are switched.
hnlFormatterUnbindVars = nil
hnlSetDef('hnlFormatterUnbindFuncs '(hnlSilosPrintTimeProp
   hnlSilosPrintMarginalProp
   hnlSilosPrintDelays
   hnlSilosPrintGateParams
   hnlSilosPrintLoad
   hnlSilosPrintOutputs
   hnlSilosPrintInputs
   hnlSilosPrintLogGate
   hnlSilosPrintStoppingRef
   hnlPrintInst
   )
)
; This procedure uses the hnlScaleTimeUnit function to locate
; and scale the named property. If found, then prefixString is
; printed if not null, followed by the integer value of the
; property. t is returned if the property was printed, else nil.
;
procedure( hnlSilosPrintTimeProp( propName prefixString )
   prog( (valuetmp)
      value = hnlScaleTimeUnit( propName )
      if( value != nil then
         if( prefixString == nil then
            sprintf(tmp "%d" value)
         else
            sprintf(tmp "%s%d" prefixString value)
         )
         hnlPrintString(tmp)
         return( t )
      )
      return( nil )
   )
)
; This procedure uses the hnlScaleMarginalDelay function to
```

```
; locate and scale the named property. If found, then
; prefixString is printed if not null, followed by the float
; value of the property, with one digit after the decimal. t is
; returned if the property was printed, else nil.
;
procedure( hnlSilosPrintMarginalProp( propName prefixString )
   prog( (value tmp)
      value = hnlScaleMarginalDelay( propName )
      if( value != nil then
         if( prefixString == nil then
            sprintf(tmp "%.1f" value)
         else
            sprintf(tmp "%s%.1f" prefixString value)
         )
         hnlPrintString(tmp)
         return( t )
      )
      return( nil )
   )
)
; This function prints out the delays for a basic logic gate in
; SILOS netlist syntax.
; Sample:
;   tr,mr tf,mf
;   tr,trs tf,tfc
;   tr tf
;
procedure( hnlSilosPrintDelays()
   hnlSilosPrintTimeProp("tr" " ")
   if( ! hnlSilosPrintMarginalProp("mr" ",") then
      hnlSilosPrintMarginalProp("trc" ",")
   )
   hnlSilosPrintTimeProp("tf" " ")
   if( ! hnlSilosPrintMarginalProp("mf" ",") then
      hnlSilosPrintMarginalProp("tfc " ",")
   )
)
; This function prints out the strength followed by the delays
; for a basic logic gate in SILOS netlist syntax.
;
procedure( hnlSilosPrintGateParams()
```

```
    let( (propVal)
        propVal = hnlGetPropVal("strg" hnlCurrentMaster
            hnlCurrentInst)
        if( propVal != nil then
            hnlPrintString("/")
            hnlPrintString(propVal)
        )
        hnlSilosPrintDelays()
    )
)
; Print a blank, followed by the load factor for the
; terminal of given name if it is defined.
;
procedure( hnlSilosPrintLoad( termName )
    let( ( propName propVal tmp)
        sprintf( propName "lf%s" termName )
        propVal = hnlGetPropVal(propName hnlCurrentMaster
            hnlCurrentInst)
        if( propVal != nil then
            if( floatp( propVal ) then
                sprintf(tmp " %.1f*" propVal)
                else
                sprintf(tmp " %d*" propVal)
            )
            hnlPrintString(tmp)
        else
            hnlPrintString(" ")
        )
    )
)
; This function writes the list of output names for the current
; instance to the netlist file. Names are automatically mapped to
; be valid for simulator input,
; the line is automatically continued if its length would exceed
; the simulator limit.
;
procedure( hnlSilosPrintOutputs()
    let( (name)
        foreach( name hnlCurrentOutputs
            hnlPrintString( hnlMapName( name ))
            hnlPrintString( " " )
```

```
      )
   )
)
; This function writes the list of input names for the current
; instance to the netlist file. Names are automatically mapped to
; be valid for simulator input, the line is automatically
; continued if its length would exceed the simulator limit.
;
procedure( hnlSilosPrintInputs()
   let( (termlist)
      foreach( termlist hnlSortTermsToNets( hnlCurrentInTerms )
         hnlSilosPrintLoad( car(termlist) ~> name )
         hnlPrintString(hnlMapName(cadr(termlist)))
      )
      hnlPrintString("\n")
   )
)
; This function prints the SILOS syntax connectivity for a basic
; logic gate:
;
; output_net_names .name/strength delay_values input_net_names
;
; The line is automatically continued if the line's length would
; exceed the simulator limit.
;
procedure( hnlSilosPrintLogGate( name )
   hnlSilosPrintOutputs()
   hnlPrintString(" .")
   hnlPrintString(name)
   hnlSilosPrintGateParams()
   hnlSilosPrintInputs()
)
; This function prints out the connectivity for a stopping
; cellview reference in the netlist file. The connectivity
; for the device is formatted by evaluating the
; hnlSilosFormatInst property if it is found on the master of
; the current instance. If no such property exists, an error is
; printed, and hnlError is set to t.
;
procedure( hnlSilosPrintStoppingRef()
   let( ( foundProp )
```

```
        foundProp = hnlCurrentMaster~>hnlSilosFormatInst
        if( foundProp != nil then
            evalstring( foundProp )
        else
            hnlPrintString("\n")
            sprintf(errorMessage"Netlister: No format property for
                '%s'\n" hnlCurrentType)
            println(errorMessage)
            hnlError = t
        )
    t
)
)
; This function writes the connectivity for an instance in the
; design to the netlist file. The instance can be either a
; primitive, or a macro reference.
;
; This function must be defined, and must be called
; hnlPrintInst(). It is called by the hierarchical
; netlister to format the reference to each instance.
;
procedure( hnlPrintInst()
    prog( ()
    ; Macro references and primitives must be formatted differently.
    if( hnlCurrentMaster != nil then
        if( hnlIsAStoppingCell(hnlCurrentMaster) then
            return( hnlSilosPrintStoppingRef() )
        else
            ;
            ; Output connectivity for a macro references.
            ;
            )
        )
    )
)
```

Map File

Full path names to an instance or a net are not required in a hierarchical netlist. Therefore, the names generated by the netlister do not become as long as they can with FNL. Names in HNL are usually the same as those you enter. This reduces, but does not eliminate, the problem of name lengths acceptable to the simulator. There is also a second problem created by names containing characters not valid in names for certain target simulators. This problem arises from the following two sources:

- The designer enters a character not valid for the target simulator, for example, “+” in the name of an instance or a net in the schematic.
- The designer does not name a net or instance; the schematics extractor or graphics editor names it with a number. Many simulators do not allow a name to begin with a digit.

These two sources, in addition to names that exceed a maximum limit, force name modification in the netlister output. Invalid characters are handled by a fast character-mapping array where invalid characters are mapped to unused (or seldom used) valid strings. When a valid name cannot be generated by character mapping, a unique number is assigned and optionally prefixed by a string of alphabetic characters to produce a valid name.

Since the netlist is hierarchical, you do not need to have a unique name for each node in the hierarchy. In HNL a simple name-mapping table is used. If a name is invalid, a valid name is created. Both the new and old names are added to the name-map table. Then, if the same invalid character string is encountered elsewhere in the hierarchy, even if the name is for a different node or instance, the same mapped name is reused. The simulator creates a unique name for each node in the hierarchy when it flattens the design internally. This makes the name cross-reference table (or map) compact.

Naming Conventions

The names of all variables and functions not declared local to a particular function are prefixed with the package name `hnl`. The remainder of the name is lowercase except for the first character of each word, which is uppercase.

Source Code

Source code is only provided for the HNL entry-point functions. All other code is provided in a separate file, which is encrypted and not user-readable. The interfaces and the functionality of these encrypted functions are described in the “Access Functions” section in this chapter.

Support for Inherited Connections

Inherited connections in the design provide you the flexibility to specify power and ground signals at a higher level of the hierarchy and inherit these signals as connections at a lower level using netExpressions and netSet properties. This way you need not create explicit power and ground terminals at the macro level instead use inherited connections.

OSS-based hierarchical netlisters resolve these connections at the time of netlisting by introducing dummy ports or psuedo ports. These dummy ports are created to maintain connectivity across modules and their instantiations. These dummy ports introduce unwanted nets in the hierarchy in place of netExpressions and netSet properties. This results in a loss of data while sharing design information across the flow. In addition, these dummy ports cause problems on a round trip when you want to edit the design specified at the time of design entry. There is also a loss of ground and power sensitivity information when using tools, such as ASSURA which can interpret logical and physical connectivity. This occurs because the *.GROUNDSENSITIVITY and *.POWERSENSITIVITY statements for power and ground terminals do not get added to the netlist.

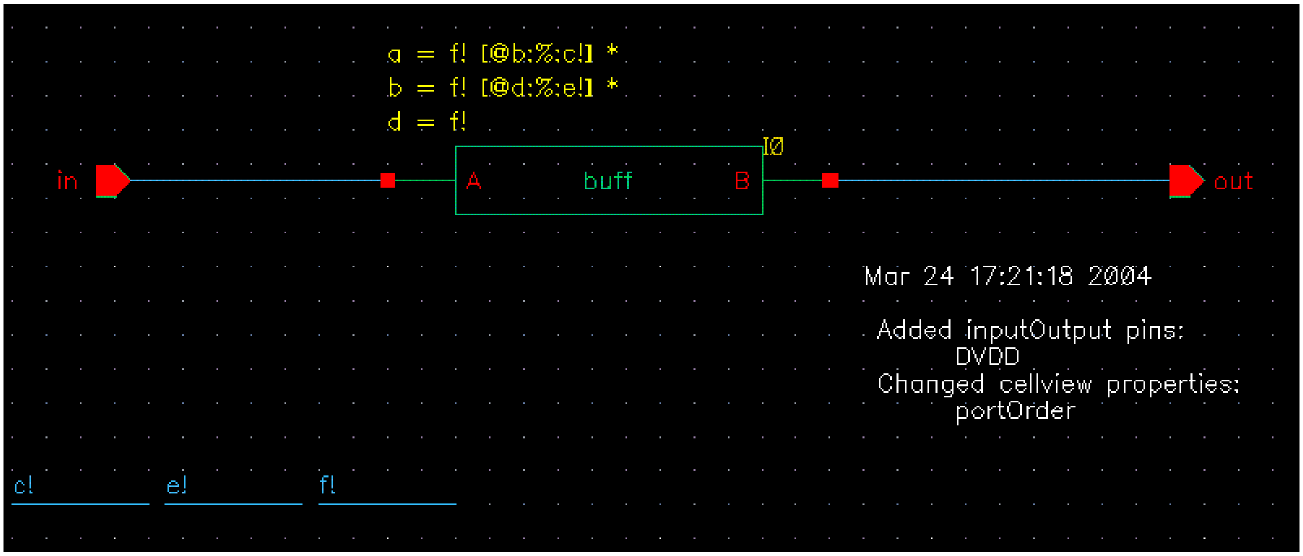
The SKILL environment variable, `simPrintInhConnAttributes`, lets you prevent the creation of pseudo ports and get the inherited connections information in the netlists which you started off with at the time of the design entry.

`simPrintInhConnAttributes` is a boolean variable and the default value is `nil`. You can set this environment variable in the `.simrc` file. The syntax of `simPrintInhConnAttributes` is as follows:

```
simPrintInhConnAttributes = t/nil
```

As the default value is `nil` by default OSS creates pseudo terminals for intermediate levels of the hierarchy.

The following figure shows an example of a buffer that uses explicit terminals with inherited connections for the power and the ground supplies. This example illustrates the changes in the netlist due to the `simPrintInhConnAttributes` flag.

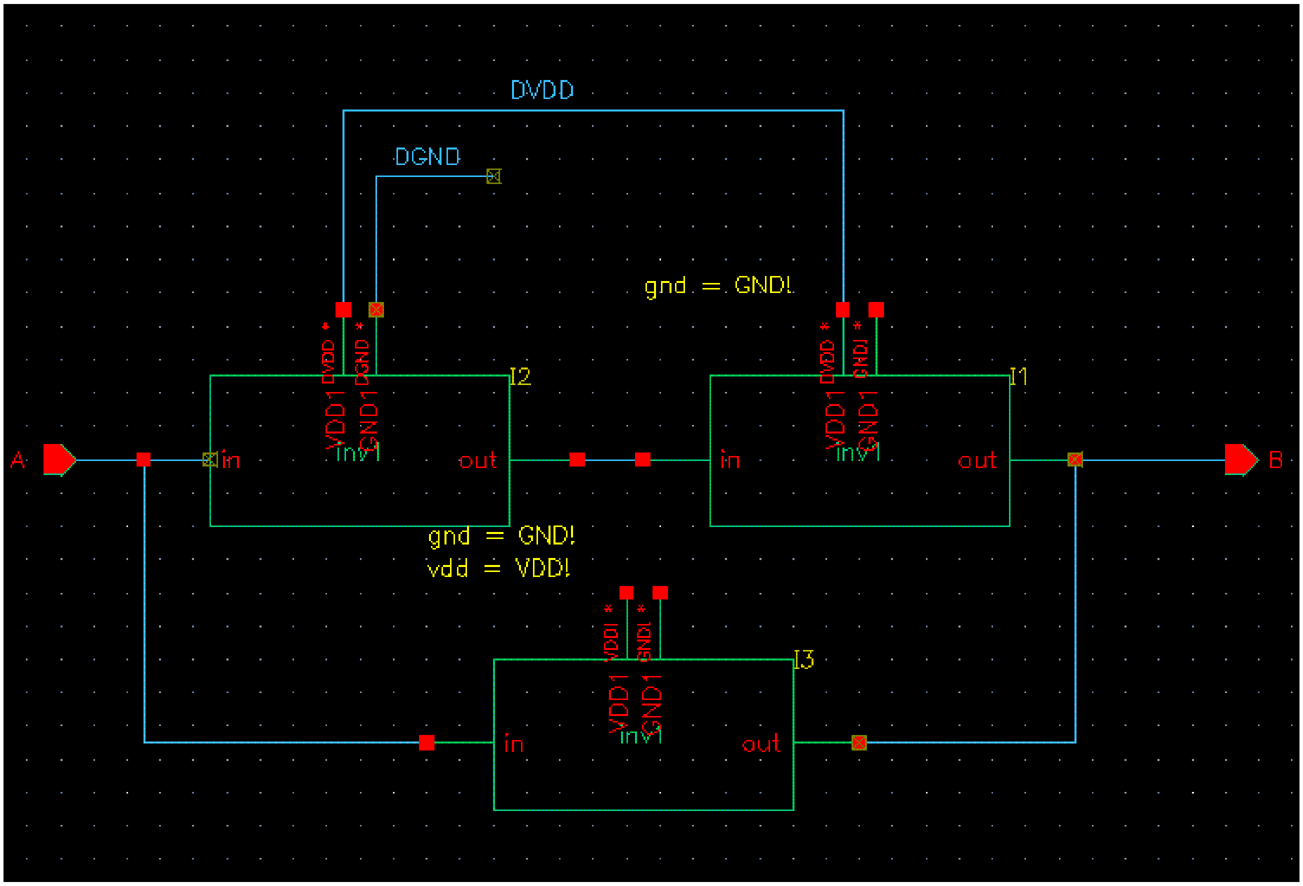


The above figure shows the top cell of buff with the following netSet properties:

```
a = [@b:%:c!] *
b = [@d:%:e!] *
d = f!
```

In the above example, a, b, and d will all resolve to `f!`.

The following figure shows the buff schematic has 3 instances of inverters.



In the case of I1, one terminal is overridden and resolves to DVDD while one terminal is resolved using netSet property as follows.

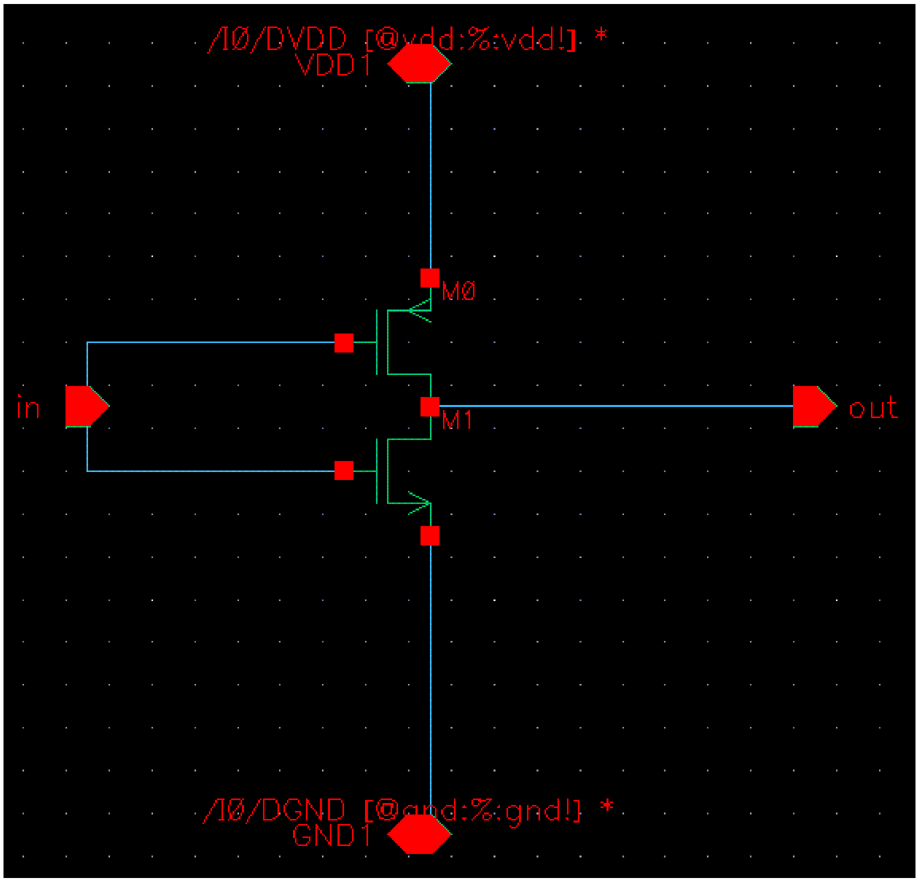
```
gnd = GND!
```

In the case of I2, both the terminals are physically overridden in the buff schematic by explicit connections to DVDD and DGND.

In the case of I3, both the terminals are resolved using netSet expressions as follows.

```
gnd = GND!
vdd = VDD!
```

In the following figure, inv1 from the buff schematic has explicit connections to the power and the ground supplies. Both the explicit terminals are inherited with the power (VDD1) terminal having the netExpression `[@vdd:%:vdd!]` and the ground terminal (GND1) having the netExpression `[@gnd:%:gnd!]`. These types of terminals are referred to as Inherited Ports.



The symbol which is created now has 4 terminals instead of 2 and this symbol is used in I1, I2, and I3.

When you netlist top/test cellview using auCdl, you get the following netlist. The pseudo ports created due to the presence of implicit terminals (inherited connections) are highlighted in bold.

```
************************************************************************
* auCdl Netlist:
*
* Library Name: test
* Top Cell Name: top
* View Name: schematic
* Netlisted on: Jun 28 15:50:13 2005
************************************************************************
*.EQUATION
*.SCALE METER
*.MEGA
.PARAM
```

```
*.GLOBAL GND!
+ VDD!
+ f!
+ e!
+ c!

*.PIN GND!
*+ VDD!
*+ f!
*+ e!
*+ c!

************************************************************************
* Library Name: test
* Cell Name: inv1
* View Name: schematic
************************************************************************

.SUBCKT inv1 in out inh_gnd inh_vdd
*.PININFO in:I out:O inh_gnd:B inh_vdd:B
MM1 out in inh_gnd inh_gnd NM
MM0 out in inh_vdd inh_vdd PM
.ENDS

************************************************************************
* Library Name: test
* Cell Name: buff
* View Name: schematic
************************************************************************

.SUBCKT buff A B
*.PININFO A:I B:O
XI3 A B GND! VDD! / inv1
XI1 net6 B GND! DVDD / inv1
XI2 A net6 DGND DVDD / inv1
.ENDS

************************************************************************
* Library Name: test
* Cell Name: top
* View Name: schematic
************************************************************************

.SUBCKT top in out
*.PININFO in:I out:O
XI0 in out / buff
.ENDS
```

When you set `simPrintInhConnAttributes` to `t` the netlist produced is as follows:

```
************************************************************************
* auCdl Netlist:
*
* Library Name: test
* Top Cell Name: top
* View Name: schematic
* Netlisted on: Oct 14 12:25:05 2004
************************************************************************
```

```
*.EQUATION
*.SCALE METER
*.MEGA
.PARAM

*.GLOBAL GND!
+ VDD!
+ f!
+ e!
+ c!

*.PIN GND!
*+ VDD!
*+ f!
*+ e!
*+ c!

************************************************************************
* Library Name: test
* Cell Name: inv1
* View Name: schematic
************************************************************************

.SUBCKT inv1 gnd! vdd! in out
*.PININFO in:I out:O gnd!:B vdd!:B
*.PINMAP GND1:gnd! VDD1:vdd!
*.GROUNDSENSITIVITY gnd! GND1
*.POWERSENSITIVITY vdd! VDD1
*.NETEXPR vdd vdd! vdd!
*.NETEXPR gnd gnd! gnd!
MM1 out in gnd! gnd! NM
MM0 out in vdd! vdd! PM
.ENDS

************************************************************************
* Library Name: test
* Cell Name: buff
* View Name: schematic
************************************************************************

.SUBCKT buff A B
*.PININFO A:I B:O

XI3 GND! VDD! A B / inv1 $NETSET vdd="VDD!" gnd="GND!"
XI1 GND! DVDD net6 B / inv1 $NETSET gnd="GND!"
XI2 DGND DVDD A net6 / inv1
.ENDS

************************************************************************
* Library Name: test
* Cell Name: top
* View Name: schematic
************************************************************************

.SUBCKT top in out
*.PININFO in:I out:O

XI0 in out / buff $NETSET a="b c!" d="f!" b="d e!"
.ENDS
```

Support for Supply Sensitivity

The OSS-based netlisters netlist SUPPLYSENSITIVITY information along with *.`NETEXPR` and $`netSet` statements when Verilog modules interface with CDL sub-circuits. You need to add this information on the terminals of the cellview by using the SUPPLYSENSITIVITY constructs, *.POWERSENSITIVITY and *.GROUNDSENSITIVITY. These constructs have the following syntax:

```
*.POWERSENSITIVITY netName value
*.GROUNDSENSITIVITY netName value
```

These statements can be entered in either upper, lower or even mixed case.

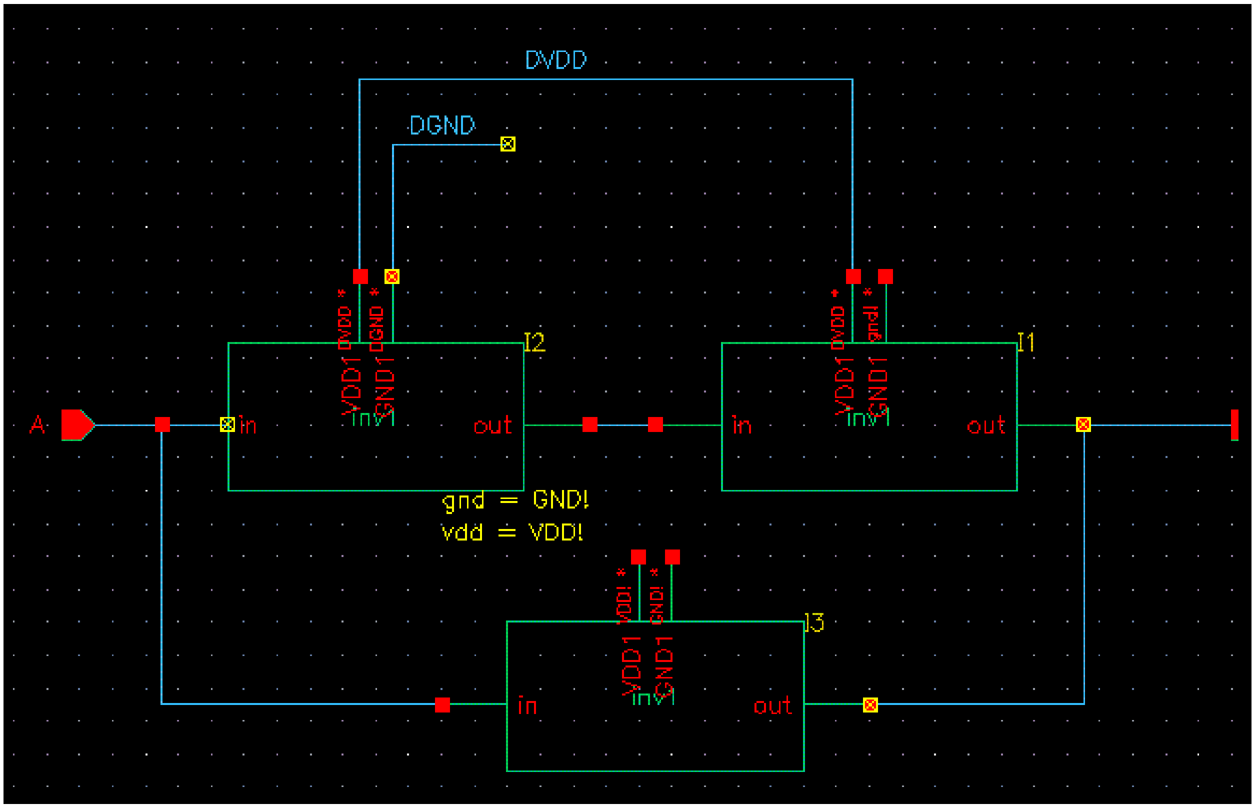
Following is an example to explain this feature.

```
.SUBCKT OR2 A B Z PWR GND
*.POWERSENSITIVITY A PWR
*.GROUNDSENSITIVITY A GND
*.POWERSENSITIVITY B PWR
*.GROUNDSENSITIVITY B GND
.ENDS
```

These terminals and their sensitivity information is stored as terminal properties and is included in the netlist. The names of the properties are `groundSensitivity` and `powerSensitivity`. These properties are of type `String`. For example, Terminal A has a property value pair on a terminal `SUP` as `groundSensitivity = gnd_io7` and `powerSensitivity = pwr_io7` then the CDL netlist contains following statements:

```
*.POWERSENSITIVITY SUP pwr_io7
*.GROUNDSENSITIVITY SUP gnd_io7
```

In situations when some of the inherited terminals are overridden at the instance level in a macro and some are not, then the `$PINS` statement is used to define the termOrder. Refer to the Instance I1 in the buff schematic shown in the following figure.



I1 has only one terminal which is overridden. Therefore all the terminals are indicated in the netlist using the $PINS statement for the instance I1.

Note: $PINS statement is not used in the Verilog format since the Verilog language has existing constructs to define NULL ports. Also by using the explicit netlist option in Verilog Netlister, you can specify the port and its corresponding net in the netlist.

The netlist for the above example is as follows:

```
************************************************************************
* Library Name: test
* Cell Name: buff
* View Name: schematic
************************************************************************

.SUBCKT buff A B
*.PININFO A:I B:O

*.NETEXPR gnd gnd! gnd!
XI3 GND! VDD! A B / inv1 $NETSET vdd="VDD!" gnd="GND!"
XI1 / inv1 $PINS VDD1=DVDD GND1=gnd! in=net6 out=B
```

```
XI2 DGND DVDD A net6 / inv1
.ENDS

************************************************************************
* Library Name: test
* Cell Name: top
* View Name: schematic
************************************************************************

.SUBCKT top in out
*.PININFO in:I out:O

XI0 in out / buff $NETSET a="b c!" d="f!" b="d e!"

.ENDS
```

Notice in the above netlist the `$PINS` statement indicates all the terminals and their resolutions as well as the unresolved terminal names, in this case `in=net6` and `out=B`.

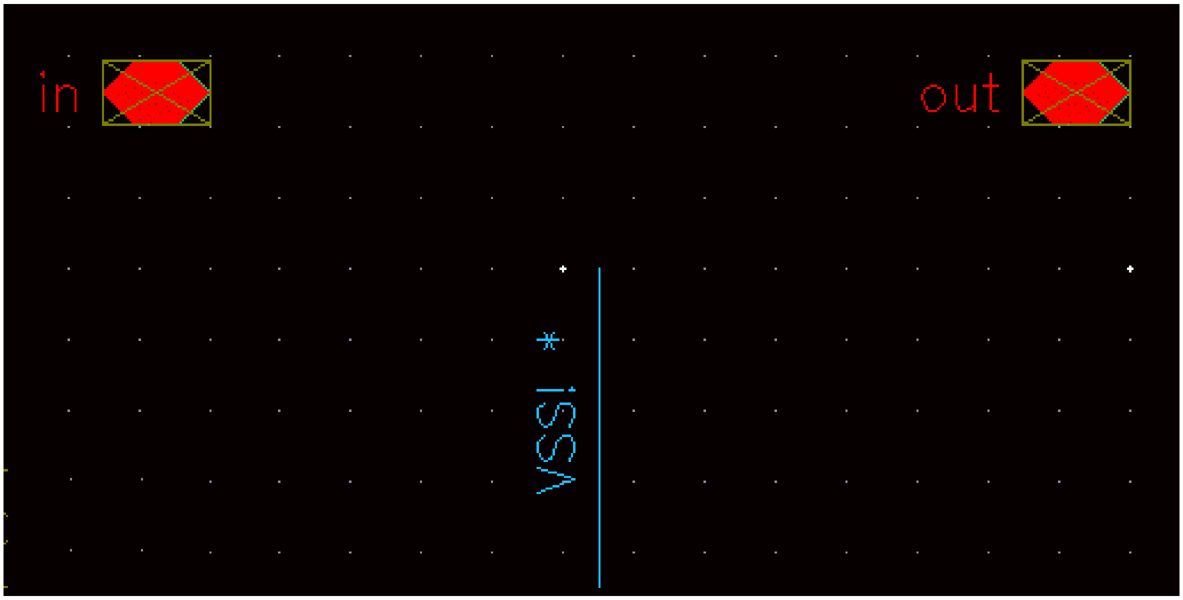
Ignoring Mismatch in Inherited Connections

The OSS-based netlisters, do not support mismatch in inherited connection between the placed and switched masters of an instance. If one of the masters of an instance has a terminal with net expression, which is a case of explicit inherited connection, and the other has a wire with net expressions, which is a case of implicit inherited connection, OSS flags an error and stops netlisting.

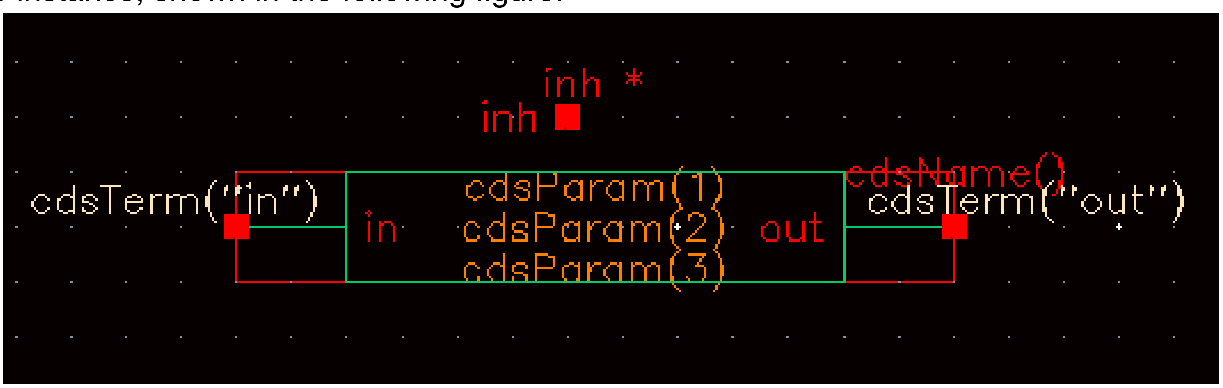
A SKILL environment variable, hnlAllowExplicitImplicitInhMismatch, lets you ignore this mismatch while printing netlist. `hnlAllowExplicitImplicitInhMismatch` is a boolean variable with a default value as `nil`. You can set this variable in the `.simrc` file or in the CIW as follows:

```
hnlAllowExplicitImplicitInhMismtach = t
```

The following figure shows an example of an instance that has an implicit connection, `VSS!`, in the switched master or the schematic view.



This connection is taken as an explicit connection in the placed master or the symbol view of the instance, shown in the following figure.

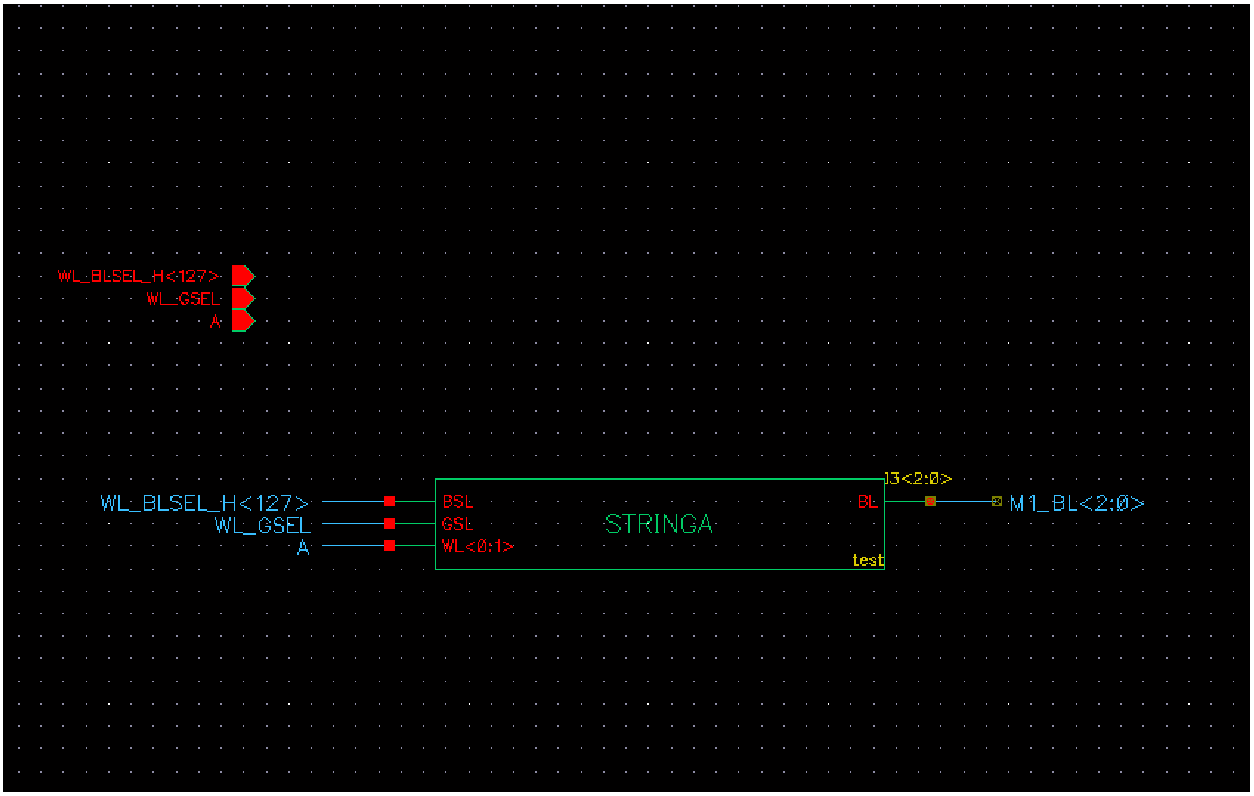


By default, OSS does not support such mismatch. You can use the `hnlAllowExplicitImplicitInhMismatch` variable to ensure that OSS generates netlist for such connections.

Support for Iterated Instances

OSS supports netlisting of iterated instances without any expansion. An iterated instance refers to an array of multiple instances. This feature specifies that there is one to one mapping between a composer design and a netlist with respect to iterated instances. Iterated instances feature reduces netlist size, which further results in significant decrease in the time taken to generate a netlist. For example, in a Verilog design with an iterated instance of 8K range, a netlist is generated in approximately 4 minutes. This is an improvement over the 9 hours taken earlier without iterated instances support.

The following figure shows a design with iterated instance, `I3<2:0>`.



Earlier, each iterated instance was expanded to a list of individual instances and netlisted separately. The following example shows an iterated instance, `I3<2:0>`, which is expanded as three individual instances, `I3_2_`, `I3_1_`, and `I3_0_`:

```
STRINGA I3_2_(M1_BL[2], WL_BLSEL_H[127], WL_GSEL, {A,A});
STRINGA I3_1_(M1_BL[1], WL_BLSEL_H[127], WL_GSEL, {A,A});
STRINGA I3_0_(M1_BL[0], WL_BLSEL_H[127], WL_GSEL, {A,A});
```

Note: This example uses Verilog syntax.

The iterated instance netlisting feature is enabled by default. You can also set this feature.

To enable the iterated instance netlisting, set the `hnlSupportIterInst` flag to true in the `.simrc` file or on CIW command line:

```
hnlSupportIterInst = t
```

Note: To enable the iterated instance netlisting in Verilog, set the `simVerilogPrint2001Format` flag to true in the `.simrc` file or on CIW command line:

```
simVerilogPrint2001Format = t
```

The following example shows an array of iterated instances, which is netlisted as a single instance, `I3<2:0>` when you set the variable to `t`:

```
STRINGA I3[2:0](M1_BL[2:0], WL_BLSEL_H[127], WL_GSEL, {A,A});
```

Note: This example uses Verilog syntax.

To disable the iterated instance netlisting and print the iterated instances in the expanded form, set the `vlogExpandIteratedInst` flag to `t` in the `.simrc` file or at CIW as:

```
vlogExpandIteratedInst = t
```

If a design module has multiple iterated instances with the same base name, the netlister prints them in the expanded form, even if the `logExpandIteratedInst` flag is not set to `t`. For example, consider a schematic that contains split iterated instances `I0<2:0>` and `I0<3>`. In this case, the netlister prints the instances in the expanded form, as illustrated in the following Verilog netlist snippet:

```
    dummy I0_2_ ( cds_globals.gnd_, net3);
    dummy I0_1_ ( cds_globals.gnd_, net3);
    dummy I0_0_ ( cds_globals.gnd_, net3);
    dummy I0_3_ ( net2, cds_globals.gnd_);
```

If there are no aliases in a design, you can stop alias processing to further improve OSS performance. For example, if you set the `hnlIgnoreAlias` flag to true in a Verilog design with an iterated instance of 8K range, a netlist is generated in approximately 1 second. This is a further improvement over the 4 minutes taken when the flag value is false. The `hnlIgnoreAlias` flag is set to true as:

```
hnlIgnoreAlias = t
```

Writing a Formatter

Several output functions must exist for the netlister to run. Depending on the output syntax required, the functions may or may not produce any output. This section describes the order in which data is presented by the netlister traversal functions and the output functions expected. These functions are

- stored in a file, whose name is that of the simulator concatenated with the string `.ile`
- placed in the *`install_dir`*`/tools/dfII/local/hnl` directory, where *`install_dir`* is the directory in which the Cadence software has been installed

This file must be encrypted for HNL to execute correctly on a non-development system.

Set Output Variables

When the netlister starts to execute, it calls an `hnlSetOutputVars()` formatter function. This function takes no arguments and sets any target format-specific variables. This function is called before the netlister starts traversing the database.

Create Netlister Data Structures

After setting the output variables, the netlister traverses the database and constructs a list of all the cellviews (schematics) that need to be placed in the netlist. At the same time, a list of global signals used throughout the design is created. No format functions are called until all the data has been read in and checked for basic integrity. Although the formatter is not called at this point, it is possible to override certain netlister functions. Refer to the “Access Functions” section in this chapter for a description of these functions, their names, and their argument lists to help determine how to modify the netlister. Most of these functions do not need to be modified, as they simply assemble and verify the data. However, certain ones can be replaced to drastically modify the netlister.

Print Netlist Header

Once all the data has been collected, the netlister calls the `hnlPrintNetlist()` netlister-defined function. This function, in turn, calls `hnlPrintNetlistHeader()`, a function which must be defined as one of the formatting functions to print any header information at the beginning of the netlist file.

Print Connectivity for Subcircuits

Next, another netlister-defined function `hnlDoInstBased()` is called. This function sets the following global variables:

```
hnlCurrentCell
hnlCellInputs
hnlCellOutputs
hnlCellOthers
hnlCellOutTerms
hnlCellInTerms
hnlCellOtherTerms
hnlCellNetsOnTerms
```

These variables are set for each macro definition to be output to the netlist as well as for the top-level design. The `hnlDoInstBased()` function calls output-formatter-defined functions for each cell needed to define the netlist. First, format functions are called for all of the devices used in the design that are not stopping cells. These devices are frequently called macros or sub-circuits. Then, format functions are called to output the top-level design being netlisted. The functions called are determined by the `hnlMacroCellFuncs` and `hnlTopCellFuncs` global variables. These lists can be modified to alter the order of the output functions called to eliminate unneeded portions of the netlist or to add calls to the formatter functions you define.

Print Subcircuit Header

For each device that is not the top-level design, the `hnlCurrentCell` global variable is set to the current device being expanded. Then the `hnlPrintDeviceHeader()` formatter-defined function is called. This function typically outputs a macro or subcircuit statement.

Print External Connections for Subcircuit

Next, formatter functions are called to print any needed declarations for external connections to this device. All of these functions must be defined, although they do not need to output anything. The `hnlPrintPorts()` function, which takes no arguments, is then called to output any port declarations for this cell that may be required by the target simulator. To help output these declarations, the following global variables have been set prior to calling this function:

```
hnlCurrentCell
hnlCellInputs
hnlCellOutputs
hnlCellOthers
hnlCellOutTerms
hnlCellInTerms
hnlCellOtherTerms
hnlCellNetsOnTerms
```

Of particular importance is the `hnlCellNetsOnTerms` variable. This is a list of the nets attached to the terminals/ports of this cell sorted alphanumerically by terminal name. These nets have been separated into single-wire nets before sorting. This list can be used to output the ports in order so that they positionally correspond to references to this cell that may be output later in the netlist. If the `hnlCurrentTermsOnInst` global variable is used, when later outputting references to macros, the order of the nets attached to the macro definition and the macro reference match.

Print Connectivity for Instances in Subcircuit

After the external connections section of the netlist has been completed, the `hnlPrintDevices()` netlister-defined function is called. This function sets the following global variables for use by the format function:

```
hnlCurrentOutTerms
hnlCurrentInTerms
hnlCurrentOutputs
hnlCurrentInputs
hnlCurrentOtherTerms
hnlCurrentOthers
hnlCurrentType
hnlCurrentInst
hnlCurrentInstName
hnlCurrentIteration
hnlCurrentTermsOnInst
hnlCurrentMaster
```

For each iteration of each device used in the current cell being expanded, these variables are set, and the `hnlPrintInst()` formatter function is called. This function takes no arguments. It has access to the listed global variables and various netlister query functions. These variables and functions provide information needed to output the connectivity for this instance (reference) to the device. The `hnlPrintInst()` function is probably the most complex that you must write. It must print the connectivity for each device used in the design, and must be able to differentiate and handle both macro/subcircuit references and the formatting of primitive devices. It is the core entry point for the output formatter, and is also later called for each device referenced in the top-level cell being netlisted.

Print Subcircuit Footer

When the connectivity for each instance (reference) in this cell has been output, the `hnlPrintDeviceFooter()` formatter function is called to output any macro completion information required. This information is typically an end-of-macro or subcircuit statement.

Print Connectivity for Top Cells

Once the definitions for each level of the hierarchy have been output, the `hnlDoInstBased( )` function calls formatter functions to output the connectivity for the top-level schematic as specified by the `hnlTopCellFuncs` variable.

Print Top Cell Header

First, the `hnlPrintTopCellHeader()` function is called. This function handles any description signaling the beginning of the connectivity for the top-level design.

Print Connectivity for Instances in Top Cell

Next, the `hnlPrintDevices()` netlister-defined function is called as it was for each subcircuit in the design.

Print Top Cell Footer

When all the connectivity has been output, the `hnlPrintTopCellFooter()` formatter function is called to signal the end of connectivity for the top-level design.

Print Netlist Footer

The last formatter function to be called is `hnlPrintNetlistFooter()`. This function outputs any information needed to complete the netlist.

Unbind Variables

So designers can switch between two different simulators in the Cadence graphics environment, you must specify the functions and variables you define in your formatting instructions for HNL. To do this, define the `hnlFormatterUnbindVars` and the `hnlFormatterUnbindFuncs` variables in your simulator-specific HNL formatting file.

hnlFormatterUnbindVars

Specifies the variables you define in your simulator-specific `hnl` file that must be unbound when simulators are switched. You must include any variables you define in this list. If you do not include a variable in this list, the correct value may not be used when the designer switches to a different simulator.

Set this variable to a list containing the names of all of the variables you define. Define the variable in the `hnl` file for your netlist formatter outside of any function definition. The following is an example of how to set this variable in the `hnl/silos.ile` file for the SILOS netlist formatter:

```
simSetDef( 'hnlFormatterUnbindVars, ' (hnlMaxNameLength
      hnlMapIfFirstChar
```

```
      hnlMapIfInName
      hnlHierarchyDelimeter
   )
)
```

hnlFormatterUnbindFuncs

Specifies the functions you define in your simulator-specific `hnl` file that must be unbound when simulators are switched. You must include any functions you define in this list. If you do not include a function in this list, the correct function may not be used when the designer switches to a different simulator.

Set this variable to a list containing the names of all of the functions you define. Define this variable in the `hnl` file for your netlist formatter outside of any function definition. The following is an example of how to set this variable in the `hnl/silo.ile` file for the SILOS netlist formatter. This example is only a partial list of what is used in the `silos` interface.

```
hnlSetDef('hnlFormatterUnbindFuncs,'(hnlSilosPrintInputs
      hnlPrintInst
      hnlSilosPrintMacroRef
      hnlPrintPorts
      hnlSetOutputVars
      hnlPrintTopCellFooter
   )
)
```

Required Formatter Functions and Variables

The following is the complete list of all the functions that must be defined in any output formatter written for the hierarchical netlister:

```
hnlPrintNetlistFooter( )
hnlPrintNetlistHeader( )
hnlSetOutputVars( )
hnlPrintTopCellHeader( )
hnlPrintTopCellFooter( )
hnlPrintPorts( )
hnlPrintDeviceHeader( )
hnlPrintDeviceFooter( )
hnlPrintInst( )
```

In addition, the following variables should be defined by the formatter:

```
hnlHierarchyDelimiter
hnlDriverWillPrint
hnlCommentStr
hnlLinePostfix
```

```
hnlLinePrefix
hnlMaxLineLength
hnlMaxNameLength
hnlMapIfInName
hnlMapIfFirstChar
hnlHierarchyDelimeter
hnlFormatFuncsLoaded
hnlFormatterUnbindVars
hnlFormatterUnbindFuncs
```

Instead of setting the `hnlMapIfInName` and `hnlMapIfFirstChar` variables, you may want to set the following variables, depending on the name mapping functions you use:

```
hnlMapInstFirstChar
hnlMapInstInName
hnlMapModelFirstChar
hnlMapModelInName
hnlMapNetFirstChar
hnlMapNetInName
hnlMapTermFirstChar
hnlmapTermInName
```

For an explanation of the meaning of each of these variables, refer to “HNL Global Variables” on page 150 in this chapter.

Figure 5-3 on page 143 is a sample set of formatting functions to textually output a design in human-readable form. Figure 5-3 on page 143 is the output generated when the sample formatter functions are loaded into the netlister and the netlister is run on a 74LS169A schematic.

```
; This file contains the functions to format a sample netlister
; output.
;
; Set so this file is only loaded the first time the netlister is
; run.
hnlFormatFuncsLoaded = t
; Set the lists of variables and functions that must be unbound
; when environments (simulators) are switched.
hnlFormatterUnbindVars = nil
hnlSetDef('hnlFormatterUnbindFuncs '    ( hnlPrintNetlistHeader
                           hnlPrintNetlistFooter
                           hnlSetOutputVars
                           hnlPrintTopCellHeader
                           hnlPrintTopCellFooter
                           hnlPrintPorts
```

```
                           hnlPrintDeviceHeader
                           hnlPrintDeviceFooter
                           UserSamplePrintMacroRef
                           UserSamplePrintStoppingRef
                           hnlPrintInst
                         )
)
; This function prints the netlist header to the netlist file.
; Currently, it writes the global statement for all global nets
; used in the design.
;
hnlIfNoProcedure( hnlPrintNetlistHeader()
   let( (name)
      if( hnlAllGlobals != nil then
         hnlPrintString("The following global nets are used:")
         foreach( name hnlAllGlobals
            hnlPrintString(" ")
            hnlPrintString(name)
         )
         hnlPrintString("\n")
      )
      t
   )
)
; This function prints the netlist footer to the netlist file.
; None is needed.
;
hnlIfNoProcedure( hnlPrintNetlistFooter()
   t
)
; This function sets the simulator-specific output variables. None
; are needed for this syntax.
;
hnlIfNoProcedure( hnlSetOutputVars()
   hnlDriverWillPrint = t
   t
)
; This function outputs any special information to designate the
; beginning of the top-level schematic.
;
hnlIfNoProcedure( hnlPrintTopCellHeader()
   let( (tmp)
```

```
      sprintf(tmp "\nBegin description of schematic '%s'\n"
            hnlTopCell~>cellName )
      hnlPrintString(tmp)
      t
   )
)
; This function outputs any special information to designate the
; end of the top-level schematic.
;
hnlIfNoProcedure( hnlPrintTopCellFooter()
   let( (tmp)
      sprintf(tmp "End description of schematic '%s'\n"
            hnlTopCell~>cellName )
      hnlPrintString(tmp)
      t
   )
)
; This function outputs a blank followed by the name of the net
; attached to each bit of the terminals of this macro. The
; ordering is matched to macro references because the names in
; hnlCellNetsOnTerms are used. The function
; then outputs a new line.
;
hnlIfNoProcedure( hnlPrintPorts()
   let( ( name )
      hnlPrintString( " External connections =")
      ; Print connections in sorted order.
      foreach( name hnlCellNetsOnTerms
         hnlPrintString("   ")
         hnlPrintString(name)
      )
      hnlPrintString("\n")
      t
   )
)
; This function writes a cell definition header to the netlist
; file.
;
hnlIfNoProcedure( hnlPrintDeviceHeader()
   sprintf(tmp"\nBegin description of cell '%s'\n"
         hnlCurrentCell~>cellName )
```

```
   hnlPrintString(tmp)
   t
)
; This function completes a macro definition in the netlist file.
;
hnlIfNoProcedure( hnlPrintDeviceFooter()
   let( (tmp)
      sprintf(tmp"End description of cell '%s'\n"
            hnlCurrentCell~>cellName )
      hnlPrintString(tmp)
      t
   )
)
; This function prints out the connectivity for a macro reference
; in the netlist file.
;
hnlIfNoProcedure( UserSamplePrintMacroRef()
   let( ( name tmp)
      sprintf(tmp" Instance '%s' references cell '%s'\n"
            hnlCurrentInstName hnlCurrentType)
      hnlPrintString(tmp)
      hnlPrintString( " Connections = ")
      ; Print connections in sorted order.
      foreach( name hnlCurrentTermsOnInst
         hnlPrintString("   ")
         hnlPrintString(name)
      )
      hnlPrintString("\n")
      t
   )
)
; This function prints out the connectivity for a stopping
; cellview(primitive) reference in the netlist file.
;
hnlIfNoProcedure( UserSamplePrintStoppingRef()
   let( (tmp)
      sprintf(tmp" Instance '%s' is a '%s'\n"
            hnlCurrentInstName hnlCurrentType)
      hnlPrintString(tmp)
      hnlPrintString("\n Outputs = ")
      foreach( name hnlCurrentOutputs
```

```
         hnlPrintString("   ")
         hnlPrintString(name)
      )
      hnlPrintString("\n Inputs = ")
      foreach( name hnlCurrentInputs
         hnlPrintString("   ")
         hnlPrintString(name)
      )
      hnlPrintString("\n")
      t
   )
)
; This function writes the connectivity for an instance in the
; design to the netlist file. If the instance is a
; stopping cellview ( determined by hnlIsAStoppingCell() ) then
; UserSamplePrintStoppingRef() is called, otherwise
; UserSamplePrintMacroRef() is called. The netlist driver expects
; this function to exist with the given name and to output the
; needed connectivity for each instance.
;
hnlIfNoProcedure( hnlPrintInst()
   prog( ()
      ; Macro references and primitives must be formatted
      ; differently.
      if( hnlCurrentMaster != nil then
         if( hnlIsAStoppingCell(hnlCurrentMaster) then
            return( UserSamplePrintStoppingRef() )
         else
            return( UserSamplePrintMacroRef() )
         )
      )
   )
)
```

Figure 5-3 Output From Sample Formatter Functions

Begin description of cell '@/fflop'

```
External connections = CLK D Q Q*
   Instance '14' is a 'nand2'
      Outputs = 14.Y
      Inputs = 11.Y D
```

```
Instance '7' is a 'nand2'
   Outputs = Q
   Inputs = 2.Y Q*
Instance '1' is a 'nand2'
   Outputs = 1.Y
   Inputs = 14.Y 2.Y
Instance '8' is a 'nand2'
   Outputs = Q*
   Inputs = Q 11.Y
Instance '2' is a 'nand2'
   Outputs = 2.Y
   Inputs = 1.Y CLK
Instance '11' is a 'nand3'
   Outputs = 11.Y
   Inputs = 2.Y CLK 14.Y
```

End description of cell '@/fflop'

Begin description of schematic '74LS169A'

```
Instance '32' is a 'inv'
   Outputs = 32.Y
   Inputs = 48.Y
Instance '2' is a 'inv'
   Outputs = 2.Y
   Inputs = CLK
Instance '25' is a 'inv'
   Outputs = 25.Y
   Inputs = UP
Instance '6' is a 'inv'
   Outputs = 6.Y
   Inputs = P*
Instance '43' is a 'inv'
   Outputs = 43.Y
   Inputs = 25.Y
Instance '11' is a 'inv'
   Outputs = 11.Y
   Inputs = 2.Y
Instance '52' is a 'inv'
   Outputs = 52.Y
   Inputs = T*
Instance '48' is a 'inv'
   Outputs = 48.Y
   Inputs = LOAD*
Instance '39' is a 'inv'
   Outputs = 39.Y
   Inputs = 56.Y
Instance '55' is a 'and2'
   Outputs = 55.Y
   Inputs = 48.Y A
Instance '15' is a 'and2'
   Outputs = 15.Y
   Inputs = 48.Y D
```

```
Instance '58' is a 'and2'
   Outputs = 58.Y
   Inputs = 48.Y B
Instance '40' is a 'and2'
   Outputs = 40.Y
   Inputs = 48.Y C
Instance '9' is a 'and2'
   Outputs = 9.Y
   Inputs = 43.Y 50.Q*
Instance '36' is a 'and2'
   Outputs = 36.Y
   Inputs = 25.Y QD
Instance '28' is a 'and2'
   Outputs = 28.Y
   Inputs = 43.Y 14.Q*
Instance '3' is a 'and2'
   Outputs = 3.Y
   Inputs = 25.Y QC
Instance '41' is a 'and2'
   Outputs = 41.Y
   Inputs = 43.Y 35.Q*
Instance '16' is a 'and2'
   Outputs = 16.Y
   Inputs = 25.Y QB
Instance '53' is a 'and2'
   Outputs = 53.Y
   Inputs = 43.Y 20.Q*
Instance '42' is a 'and2'
   Outputs = 42.Y
   Inputs = 56.Y 20.Q*
Instance '19' is a 'and2'
   Outputs = 19.Y
   Inputs = 25.Y QA
Instance '44' is a 'and3'
   Outputs = 44.Y
   Inputs = 27.Y 32.Y QD
Instance '56' is a 'and3'
   Outputs = 56.Y
   Inputs = 52.Y 6.Y LOAD*
Instance '7' is a 'and3'
   Outputs = 7.Y
   Inputs = 32.Y 38.Y QC
Instance '54' is a 'and3'
   Outputs = 54.Y
   Inputs = 35.Q* 17.Y 56.Y
Instance '33' is a 'and3'
   Outputs = 33.Y
   Inputs = 32.Y 34.Y QB
Instance '49' is a 'and3'
   Outputs = 49.Y
   Inputs = 32.Y 39.Y QA
```

```
Instance '57' is a 'or3'
   Outputs = 57.Y
   Inputs = 49.Y 55.Y 42.Y
Instance '10' is a 'or3'
   Outputs = 10.Y
   Inputs = 44.Y 15.Y 31.Y
Instance '51' is a 'or3'
   Outputs = 51.Y
   Inputs = 33.Y 58.Y 54.Y
Instance '37' is a 'or3'
   Outputs = 37.Y
   Inputs = 7.Y 40.Y 26.Y
Instance '12' is a 'nor2'
   Outputs = 12.Y
   Inputs = 9.Y 36.Y
Instance '22' is a 'nor2'
   Outputs = 22.Y
   Inputs = 28.Y 3.Y
Instance '47' is a 'nor2'
   Outputs = 47.Y
   Inputs = 41.Y 16.Y
Instance '17' is a 'nor2'
   Outputs = 17.Y
   Inputs = 53.Y 19.Y
Instance '14' references cell '@/fflop'
   Connections = 11.Y 37.Y QC 14.Q*
Instance '20' references cell '@/fflop'
   Connections = 11.Y 57.Y QA 20.Q*
Instance '50' references cell '@/fflop'
   Connections = 11.Y 10.Y QD 50.Q*
Instance '35' references cell '@/fflop'
   Connections = 11.Y 51.Y QB 35.Q*
Instance '26' is a 'and4'
   Outputs = 26.Y
   Inputs = 47.Y 14.Q* 17.Y 56.Y
Instance '27' is a 'nand4'
   Outputs = 27.Y
   Inputs = 17.Y 56.Y 47.Y 22.Y
Instance '31' is a 'and5'
   Outputs = 31.Y
   Inputs = 47.Y 22.Y 50.Q* 17.Y 56.Y
Instance '24' is a 'nand5'
   Outputs = RCO
   Inputs = 47.Y 22.Y 12.Y 17.Y 52.Y
Instance '38' is a 'nand3'
   Outputs = 38.Y
   Inputs = 56.Y 17.Y 47.Y
Instance '34' is a 'nand2'
   Outputs = 34.Y
   Inputs = 56.Y 17.Y
```

End description of schematic '74LS169A'

HNL Specific Properties

nlIgnore

When this property is set on an instance, the instance will be ignored while netlisting for a particular simSimulator.

Example:

```
nlIgnore = "vhdl auCdl"
```

In this example, the instance, on which the `nlignore` property is set, will be ignored during `vhdl` and `auCdl` netlisting.

HNL Variables

hnlEmptySwitchMasterAction

Use the `hnlEmptySwitchMasterAction` variable to define how you want to generate the netlist for an instance containing an empty switch master. The following table describes the possible values of this variable.

Value	Description
`ignore`	HNL ignores the instance during netlist generation.
`honor`	HNL honors the instance during netlist generation.
`error`	OSS generates an error for the instance and does not generate the netlist.

Default: `ignore`

Example:

```
hnlEmptySwitchMasterAction="honor"
```

hnlComparePmAndInstTermMisMatch

A design can have an instance where the representation of the instance terminals (instTerm) and the terminals of the place master of that instance are different. For example, the instance instTerm can be of the type scalar, while the corresponding place master terminal can be a bus.

To generate the netlist correctly in such cases, set the `hnlComparePmAndInstTermMisMatch` variable to `t` in the simulation configuration file, such as `.simrc`, or Virtuoso CIW.

Default: `nil`

Example:

```
hnlComparePmAndInstTermMisMatch = t
```

hnlPseudoTermSortOnNet

When set to `t`, sorts the inherited terminals as described below and prints the nets attached to these terminals according to the sorting.

- Sort explicit inherited terminals based on the `term` name
- Sort implicit inherited terminals based on the net expression

Default: `nil`

Example:

```
hnlPseudoTermSortOnNet = t
```

hnlResolveBusRangeByTermRange

When set to `t`, declares the bus range on the basis of the pin direction instead of the internal bus, if their range is equal.

Default: `nil`

Example:

```
hnlResolveBusRangeByTermRange = t
```

If bus `in[7:0]` is connected to a pin and bus `in[0:7]` is an internal bus, then the bus range will be defined as `in[7:0]`.

hnlRetainInstanceNetsInShorting

When a device with the lxRemoveDevice component (For example, `basic/cds_thru`) is removed during netlisting, the instance connected to this device can have connectivity according to the two nets of the two terminals of this component. Only one net survives, while the other is eliminated.

When `hnlRetainInstanceNetsInShorting` is set to `t`, the instance will have connectivity according to the net in which it is connected to the lxRemoveDevice component. For example, if instance `'I1'` is connected to the lxRemoveDevice component `'via net net'`, then in netlist, instance '`I1'` will have connectivity `'via net1'`.

If `hnlRetainInstanceNetsInShorting` is set to `nil`, then in the netlist, instance `'I1'` will have connectivity according to the survived net. Consider two terminal lxRemoveDevice components with nets `'net1'` and `'net2'`. If `'net2'` is the survived net, then instance `'I1'` will have connectivity `'via net2'`.

Note: When `hnlRetainInstanceNetsInShorting` is set to `t`, then the netlist shorting connectivity will be shown by the assign statement in the verilog and vhdl netlist and `via *.CONNECT` statement in CDL netlist.

Default: `t`

Example:

```
hnlRetainInstanceNetsInShorting = nil
```

hnlSortPseudoTermsOnDefSigName

When set to `t`, sorts the inherited terminals based on their default signal names in the net expression, and prints the nets attached to these terminals according to the sorting.

Default: `nil`

Example:

```
hnlSortPseudoTermsOnDefSigName = t
```

Note: When none of these variables (`hnlPseudoTermSortOnNet` and `hnlSortPseudoTermsOnDefSigName`) are set, the inherited terminals are sorted based on the property name in the net expressions, and the attached nets are printed according to this sorting.

HNL Global Variables

The following are global variables you can access. Which variables are set depends on the formatter functions being used.

The contents of certain HNL variables need to be changed to support inherited connection, and are noted below. For inherited net expression, there is not sufficient information to determine the drive direction of the net, so all pseudo connections created for inherited signals are assumed to be *inout* connections. What this means is that these pseudo connections will be found in the HNL variable `hnlCellOthers`.

For any inherited terminal, the drive direction of the terminal is given. Thus inherited terminals can be found in any of the three HNL variables: `hnlCellInputs`, `hnlCellOutputs`, and `hnlCellOthers`.

Additionally, inherited terminals and inherited signals present another problem. As the default clause of the inherited net expression implies, the signal connected to the inherited terminal, if there is no inherited connection, is global by definition. However, when the cellview is being netlisted, the default signal name cannot be used when netlisting the connection to an inherited terminal. This is because if there is an inherited connection and the inherited signal is another global signal then we are allowing two globals to short together as a result. This is illegal in OA connectivity data and would generate wrong connection in most netlist syntaxes. To resolve this, a netlister-generated name is created for this connection with each inherited terminal. The same can be said for inherited signals. For readability, the netlister-generated names are derivatives of the property names specified in the inherited net expressions.

Note: Since terminal connections can be inherited, it is of paramount importance to check whether the connections to the terminals are inherited before using the terminal ids. returned by HNL through hnlCellInTerms, hnlCellOtherTerms, or hnlCellOutTerms in operations such as *`termId~>net~>name`*. For inherited signals, the same precaution must be taken before operations such as sigId~>name are performed.

hnlAllGlobals

List of all of the global signals used in the design.

Default: `nil`

hnlAllowExplicitImplicitInhMismatch

Allows the placed and switched masters of an instance to have mismatch in terms of explicit and implicit inherited connections if the net expressions on terminal and wire match with each other.

Default: `nil`

hnlInhConnUseDefSigName

Prints pseudo-ports for the inherited connections with the default net names of the NetExpression.

Default: `nil`

hnlCellInTerms

List of the terminals on the current cell being expanded whose `term ~> io == input`. The current cell being expanded can be either a macro or the top cell. This variable is valid throughout evaluation of the functions specified by the `hnlTopCellFuncs` and `hnlMacroCellFuncs` variables. This list is alphabetically sorted by the terminal names.

Inherited terminals can be found in this list but since these are real terminals there are db id. associated with them. Here for inherited terminals the relationship with the entries in *`hnlCellInputs`* are db id. to *`netlister-generated`* names. These *`netlister-generated`* names are the same ones that appeared in *`hnlCellInputs`*.

Example

For module *`buffer`,* the value is list(db_id_IN).

Default: `nil`

hnlCellInputs

List of the signal names attached to the input terminals of the current cell being expanded. This cell can be either a macro or the top cell. This variable is valid throughout evaluation of the functions specified by the *`hnlTopCellFuncs`* and *`hnlMacroCellFuncs`* variables. This list is alphabetically sorted by the terminal names.

If inherited terminal is present, a *`netlister-generated`* name is inserted into this list for each inherited terminal to distinguish it from a normal (non-inheriting) terminal.

Example

For module *`buffer`*, the value is list(“IN”)

Default: `nil`

hnlCellNetsOnTerms

List of the signals attached to all of the bits of the terminals on the current cell being netlisted. The list is sorted alphabetically by terminal name, *not* by signal name. The current cell being expanded can be either a macro or the top cell. This variable is valid throughout evaluation of the functions specified by the *`hnlTopCellFuncs`* and *`hnlMacroCellFuncs`* variables.

As a result of inherited connections and inherited terminals, *`netlister-generated`* names can be found in this SKILL list.

Example

Default: `nil`

hnlCellOtherTerms

List of the terminals on the current cell being expanded whose `term ~> io != input` and `term ~> io != output`. The current cell being expanded can be either a macro or the top cell. This variable is valid throughout evaluation of the functions specified by the `hnlTopCellFuncs` and `hnlMacroCellFuncs` variables. This list is alphabetically sorted by the terminal names.

Logically there is no db id. for pseudo port created as a result of inherited signal. A *`netlister-generated-name`* generated by HNL is used as a place-holder instead. Inherited terminals can be found in this list but since these are real terminals there are db id. associated with them. Here for inherited terminals the relationship with the entries in *`hnlCellOutputs`* are db id. to *`netlister-generated`* names. These *`netlister-generated`* names are the same ones that appeared in *`hnlCellOthers`*.

Example

For module *`buffer`*, there are no db ids for the pseudo ports created but *`pseudo-names`* generated by HNL are used as the place holders for the inherited connections. The value is list (“inh_gnd” “inh_vdd”)

Default: `nil`

hnlCellOthers

List of the signal names attached to the terminals of the current cell being expanded that are not inputs or outputs, for example, `bidirectional` and `inputOutput` terminals. The current cell being expanded can be either a macro or the top cell. This variable is valid throughout evaluation of the functions specified by the `hnlTopCellFuncs` and `hnlMacroCellFuncs` variables. This list is alphabetically sorted by the terminal names.

Pseudo ports may be inserted into this list as a direct result of inherited connection. If inherited terminal is present, a netlister generated name is also inserted into this list for each inherited terminal to distinguish it from a normal (non-inheriting) terminal.

Example

For module *`buffer`*, the value is list(“inh_gnd” “inh_vdd”). These two are pseudo ports due to inherited connection.

Default: `nil`

hnlCellOutTerms

List of the terminals on the current cell being expanded whose `term ~> io == output`. The current cell being expanded can be either a macro or the top cell. This variable is valid throughout evaluation of the functions specified by the `hnlTopCellFuncs` and `hnlMacroCellFuncs` variables. This list is alphabetically sorted by the terminal names.

Inherited terminals can be found in this list but since these are real terminals there are db id. associated with them. Here for inherited terminals the relationship with the entries in *`hnlCellOutputs`* are db id. to *`netlister-generated`* names. These *`netlister-generated`* names are the same ones that appeared in *`hnlCellInputs`*.

Example

For module *`buffer`*, the value is list(db_id_OUT).

Default: `nil`

hnlCellOutputs

List of the signal names attached to the output terminals of the current cell being expanded. This cell can be either a macro or the top cell. This variable is valid throughout evaluation of the functions specified by the `hnlTopCellFuncs` and `hnlMacroCellFuncs` variables. This list is alphabetically sorted by the terminal names.

If inherited terminal is present, a *`netlister-generated`* name is inserted into this list for each inherited terminal to distinguish it from a normal (non-inheriting) terminal.

Example

For module *buffer*, the value is list(“OUT”).

Default: `nil`

hnlCommentStr

String used by the simulator to indicate a comment. This variable is used by the `hnlPrintString()` function to determine if the current line being output to the netlist is a comment. If a comment exceeds the maximum line length of the target simulator, the comment is split into two or more single-line comments with the appropriate comment string at the beginning of each line.

Default: `nil`

hnlConfigMissingViewAction

Specifies the action to be taken if the view to be used, specified in the Hierarchy Editor, is missing from library. If the variable is not defined in the *si.env* file or is set to `warning`, the netlister displays a warning message and uses the next available view. If the variable is set to `error`, the netlister displays an error and stops further processing.

Default: `nil`

hnlCurrentInTerms

List of the terminals on the current device whose `term ~> io == input`. This list is alphabetically sorted by the terminal names.

Logically there is no db id for the pseudo port created as a result of inherited connection. A *netlister-generated* name is used as a place-holder instead. This name is the same one used in *hnlCurrentInputs.*

Example

For instance *I1* in module *top,* the value is `list( db_id_A )`.

Default: `nil`

hnlCurrentInputs

List of the signal names attached to the input terminals of the current instance. This list is alphabetically sorted by the terminal names.

As a result of inherited terminal, a *`netlister-generated`* name used to propagate the connection out of the module can be included if the connection of the inherited terminal cannot be resolved in the current instance. In the case when the connection for the inherited terminal is resolved in this instance then the real signal name (mapped if illegal) is used.

Example

For instance *`I1`* in module *`top`*, the value is `list( "IN" )`.

Default: `nil`

hnlCurrentInst

Current instance being netlisted.

Default: `nil`

hnlCurrentInstName

Name of the current instance. This name may differ from `hnlCurrentInst ~> name`, as in the case of iterated instances, where `hnlCurrentInstName` is the member name of the current iteration.

Default: `nil`

hnlCurrentIteration

Iteration number of the current instance being netlisted.

Default: `nil`

hnlCurrentMaster

Cellview that is the “view switched” master of the current instance. For example, if an instance of the `and2` symbol gate were placed in the current schematic being traversed by the netlister, this variable might be set to the `dbCellViewId` of the `and2` silos cellview (depending on the settings of the view list used during netlisting).

Default: `nil`

hnlCurrentOtherTerms

List of terminals on the current device whose `term ~>io != input` and `term ~> io != output`. The list also includes the terminals when simPinGlobals is set to `t`. This list is alphabetically sorted by the terminal names.

Logically there is no db id. for the pseudo port created as a result of inherited connection. A *`netlister-generated`* name is used as a place-holder instead. This name is the same one used in `hnlCurrentOthers`.

Example

For instance *I1* in module *top,* there are no db ids for the pseudo ports but *`netlister-generated`* by HNL are used as the place holders for the inherited connections. The value is list(“myagnd!” “avdd!”).

Default: `nil`

hnlCurrentOthers

List of the signal names attached to the terminals of the current instance that are not inputs or outputs, for example, `bidirectional` and `inputOutput` terminals. This list is alphabetically sorted by the terminal names.

As a result of inherited terminal or inherited connection, a netlister-generated name can be included if the connection of the inherited terminal cannot be resolved in the current instance. In the case when the connection for the inherited terminal is resolved in this instance then the real signal name (mapped if illegal) is used.

Example

For instance *`I1`* in module *`top`,* the value is list (“myagnd!” “avdd!”). These two are added due to inherited connections found in the *`schematic`* view of cell *`inv`* and the inherited terminals in the *`spice`* views of the cells *`pmos`* and *`nmos`*.

Default: `nil`

hnlCurrentOutTerms

List of the terminals on the current device whose `term ~> io == output`. This list is alphabetically sorted by the terminal names.

Logically there is no db id. for the pseudo port created as a result of inherited connection. A *`netlister-generated`* name is used as a place-holder instead. This name is the same one used in *`hnlCurrentOutputs`*.

Example

For instance *`I1`* in module *`top`*, the value is list(db_id_Y).

Default: `nil`

hnlCurrentOutputs

List of the signal names attached to the output terminals of the current instance. This list is alphabetically sorted by the terminal names.

As a result of inherited terminal, a *`netlister-generated`* name can be included if the connection of the inherited terminal cannot be resolved in the current instance. In the case when the connection for the inherited terminal is resolved in this instance then the real signal name (mapped if illegal) is used.

Example

For instance *`I1`* in module top, the value is list("net1").

Default: `nil`

hnlCurrentCell

`dbCellViewId` of the current cellview being netlisted.

Default: `nil`

hnlCurrentTermsOnInst

List of the signals attached to all of the terminals on the current instance being netlisted. The list is sorted alphabetically by terminal name, *`not`* by net name.

For inherited terminals and inherited connections, *`netlister-generated`* names are inserted. These names are the same names found in *`hnlCurrentInputs`*, *`hnlCurrentOutputs`*, and *`hnlCurrentOthers`*.

Example

Default: `nil`

hnlCurrentType

`cellName` of the master of the current instance.

Default: `nil`

hnlDriverWillPrint

Boolean flag that indicates whether the `hnlPrintString` function is used. If set to `nil`, the file specified by `hnlNetlistFileName` is opened, and the `hnlNetlistFile` variable is set to the port. If this variable is set to `t`, the HNL-supplied print functions are initialized instead of opening the port. The formatting instructions cannot use the `hnlNetlistFile` port to write to the netlist file; use the `hnlPrintString` function instead.

Default: `nil`

hnlError

Global flag to signal an error during netlisting. Set this flag to `t` if your formatter detects an error.

Default: `nil`

hnlEnableDriverLoadBasedShortRule

Global flag to short a terminal on the basis of the load and the driver net when removing a device. For details, see the related rules.

Default: `nil`

hnlEnableTerminalShort

Global flag to enable the shorting of terminals when removing a device.

If this flag is not set, the OSS netlister does not let two terminals to short. It displays a warning and continues to generate the netlist without removing a device.

Default: `nil`

hnlFormatFuncsLoaded

Global flag to signal that the output-formatting functions have been loaded. Set this flag to `t` in your simulator-specific `hnl` file outside of any function definitions.

Default: `nil`

hnlFormatterUnbindFuncs

List of functions you define in your simulator-specific `hnl` file that must be unbound when the designer switches simulators. Define this variable in the `hnl` file for your netlist formatter outside of any function definition.

Default: none

hnlFormatterUnbindVars

List of variables you define in your simulator-specific `hnl` file that must be unbound when the designer switches simulators. Define this variable in the `hnl` file for your netlist formatter outside of any function definition.

Default: none

hnlHierarchyDelimeter

Character used by the target simulator for delimiting levels of hierarchy when it flattens the hierarchical netlist. For the SILOS simulator, this is an opening parenthesis(( ). This variable must be set by the output formatter. It is used by the name translation functions.

Default: none

hnlHonorInhConnEscapeName

Supports the escape name mapping for inherited connection names. When set to `t` along with `simVerilogEnableEscapeNameMapping`, this variable allows the inherited connection names to also be the escaped name.

Default: `nil`

hnlHonorLxRemoveDevice

Specifies if the `lxRemoveDevice` property of instances should be honored to remove devices. This variable must be set to `t` to honor the `lxRemoveDevice` property.

Default: none

hnlIgnoreEditsOnReadOnlyCells

Specifies whether the edits made to the schematic of a read-only design that does affect the connectivity of the design should be ignored. This variable must be set to `t` to ignore the edits and continue netlisting.

Default: `nil`

hnlInstNameDifferentFrom

List of name spaces from which the Instance name should be different.

Example:

```
hnlInstNameDifferentFrom = ' ("net" "terminal" "model")
```

Default: none

hnlInvalidInstNames

List of strings that are invalid instance names when you use the `hnlMapInstName()` function to map names. This list can contain strings or lists. If the list contains a string, the string must be the name that is invalid as an instance name. If the invalid name is used, the netlister replaces the invalid name with a new name. If the list contains a sublist, the first element of the sublist must be a string that is the invalid name. If the sublist does not contain a second element, or the second element is `nil`, the netlister replaces the invalid name with a new name. If the second element of the sublist is a string, this string replaces the string that was specified as the first element. The second string can contain one or more characters.

Example:

```
hnlInvalidInstNames = list('("begin" "begin_") '("end" "end_") "input" "output")
```

Default: `nil`

hnlInvalidModelNames

List of strings that are invalid model names when you use the `hnlMapModelName()` function to map names. This list can contain strings or lists. If the list contains a string, the string must be the name that is invalid as a model name. If the name is used, the netlister replaces the invalid name with a new name. If the list contains a sublist, the first element of the sublist must be a string that is the invalid name. If the sublist does not contain a second element, or the second element is `nil`, the netlister replaces the invalid name with a new name. If the second element of the sublist is a string, this string replaces the string that was specified as the first element. The second string can contain one or more characters.

Example:

```
hnlInvalidModelNames = list('("begin" "begin_") '("end" "end_") "input" "output")
```

Default: `nil`

hnlInvalidNames

List of strings that are invalid names when you use the `hnlMapName()` function to map names. This list can contain strings or lists. If the list contains a string, the string must be the name that is invalid. If the name is used, the netlister replaces the invalid name with a new name. If the list contains a sublist, the first element of the sublist must be a string that is the invalid name. If the sublist does not contain a second element, or the second element is `nil`, the netlister replaces the invalid name with a new name. If the second element of the sublist is a string, this string replaces the string that was specified as the first element. The second string can contain one or more characters.

Example:

```
hnlInvalidNames = list('("begin" "begin_") '("end" "end_") "input" "output")
```

Default: `nil`

hnlInvalidNetNames

List of strings that are invalid net names when you use the `hnlMapNetName()` function to map names. This list can contain strings or lists. If the list contains a string, the string must be the name that is invalid as a net name. If the name is used, the netlister replaces the invalid name with a new name. If the list contains a sublist, the first element of the sublist must be a string that is the invalid name. If the sublist does not contain a second element, or the second element is `nil`, the netlister replaces the invalid name with a new name. If the second element of the sublist is a string, this string replaces the string that was specified as the first element. The second string can contain one or more characters.

Example:

```
hnlInvalidNetNames = list(`("begin" "begin_") `("end" "end_") "input" "output")
```

Default: `nil`

hnlInvalidTermNames

List of strings that are invalid terminal names when you use the `hnlMapNetName()` function to map names. This list can contain strings or lists. If the list contains a string, the string must be the name that is invalid as a terminal name. If the name is used, the netlister replaces the invalid name with a new name. If the list contains a sublist, the first element of the sublist must be a string that is the invalid name. If the sublist does not contain a second element or the second element is `nil`, the netlister replaces the invalid name with a new name. If the second element of the sublist is a string, this string replaces the string that was specified as the first element. The second string can contain one or more characters.

Example:

```
hnlInvalidTermNames = list('("begin" "begin_") '("end" "end_") "input" "output")
```

Default: `nil`

hnlListOfAllCells

List of cells which are not stopping points. This is a list of lists. The first element of each sublist is the cell (dbId). If configuration is used, remaining members of sublist are a string which is list of views separated by spaces, occurrence path string, a boolean parameter indicating if the cell is a top cell and configuration. The occurrence path string is visible only when an occurrence binding is created on an instance in the configuration.

Example:

When hierarchical HDB configuration is used, the syntax is as follows:

```
hnlListOfAllCells = list(list(cvId1 "spice spectre schematic" nil "TRUE"
"cofigId1")
                        list(cvId2 "spice schematic" "FALSE" "") . . . )
```

When only Top level HDB configuration is used:

```
hnlListOfAllCells = list(list(cvId1 "spice spectre schematic") list(cvId2 "spice
schematic")...)
```

Default: `nil`

Caution

More than one cellview with the same cellview ~> cellName might appear in this list if the cellviews come from different libraries.

hnlLinePostfix

String placed at the end of the current line when a line output to the netlist exceeds the maximum line length of the simulator and must be split into two lines. This variable is used by the `hnlPrintString` function.

Default: `nil`

hnlLinePrefix

String placed at the beginning of the next line when a line output to the netlist exceeds the maximum line length of the simulator and must be split into two lines. This variable is used by the `hnlPrintString` function.

Default: `nil`

hnlListOfAllStopCells

List of the cellviews that are stopping points. This is a list of lists. The first element of each sublist is the cellview. This list can be used to output model statements for simulators such as HSPICE during printing of the netlist header.

Default: `nil`

Caution

More than one cell with the same* cell ~> cellName *might appear in this list if the cellviews come from different libraries.

hnlMacroCellFuncs

Functions to call to output the connectivity for each macro or subcircuit. This variable can be modified to alter the order of output functions called and, thus alter the appearance of the netlist. Extra output functions may also be added to this list.

```
("hnlPrintDeviceHeader()"
"hnlPrintPorts()"
```

```
"hnlPrintDevices()"
"hnlPrintDeviceFooter()"
)
```

hnlMapDirName

Name of the directory containing the map file.

Default: `"map"`

hnlMapFileName

Name of the map file.

Default: `"current"`

hnlMapIfFirstChar

List of characters that are invalid as the first character of a name when the `hnlMapName()` function is used to map names. This list can contain either strings or lists. If the list contains a string, the string must contain a single character that indicates the character is invalid. If the character is used, the name is replaced by a new netlister-generated name. If the list contains a sublist, the first element of the sublist should be a string containing a single character that is the invalid character. If the sublist does not contain a second element or the second element is `nil`, the character is deleted when used as the first character of a name. If the second element of the sublist is a string, the string will replace the character specified as the first element. The second string can contain one or more characters.

Example:

```
hnlMapIfFirstChar = list("0" "1" "2 " "3" "4 " "5" "6" "7" "8"
("@" "_"))
```

This variable must be set by the output formatter.

Default: `nil`

hnlMapIfInName

List of characters that are invalid internal to a name when the `hnlMapName()` function is used to map names. This list can contain either strings or lists. If the list contains a string, the string must contain a single character that indicates the character is invalid. If the character is used, the name is replaced by a new netlister-generated name. If the list contains a sublist, the first element of the sublist should be a string containing a single character that is the invalid character. If the sublist does not contain a second element or the second element is `nil`, the character is deleted when used internal to a name. If the second element of the sublist is a string, the string will replace the character specified as the first element. The second string can contain one or more characters.

Example:

```
hnlMapIfInName = list(("/" "|") ("@" "_") "(" ")")
```

This variable must be set by the output formatter.

Default: `nil`

hnlMapInstFirstChar

List of characters that are invalid as the first character of a name when the `hnlMapInstName()` function is used to map names. This list can contain either strings or lists. If the list contains a string, the string must contain a single character that indicates the character is invalid. If the character is used, the name is replaced by a new netlister-generated name. If the list contains a sublist, the first element of the sublist should be a string containing a single character that is the invalid character. If the sublist does not contain a second element or the second element is `nil`, the character is deleted when used as the first character of a name. If the second element of the sublist is a string, the string will replace the character specified as the first element. The second string can contain one or more characters.

Example:

```
hnlMapInstFirstChar = list("0" "1" 2" "3" "4" "5" "6" "7" "8" "9" '("@" "-"))
```

Default: `nil`

hnlMapInstInName

List of characters that are invalid internal to a name when the `hnlMapInstName( )` function is used to map names. This list can contain either strings or lists. If the list contains a string, the string must contain a single character that indicates the character is invalid. If the character is used, the name is replaced by a new netlister-generated name. If the list contains a sublist, the first element of the sublist should be a string containing a single character that is the invalid character. If the sublist does not contain a second element or the second element is `nil`, the character is deleted when used internal to a name. If the second element of the sublist is a string, the string will replace the character specified as the first element. The second string can contain one or more characters.

Example:

```
hnlMapInstInName = list( '("/" "|") '("@" "-") "("  ")")
```

Default: `nil`

hnlMapTermInName

List of characters that are invalid internal to a name when the `hnlMapTermName()` function is used to map names. This list can contain either strings or lists. If the list contains a string, the string must contain a single character that indicates the character is invalid. If the character is used, the name is replaced by a new netlister-generated name. If the list contains a sublist, the first element of the sublist should be a string containing a single character that is the invalid character. If the sublist does not contain a second element or the second element is `nil`, the character is deleted when used internal to a name. If the second element of the sublist is a string, the string will replace the character specified as the first element. The second string can contain one or more characters.

Example:

```
hnlMapTermInName = list( '("/" "|") '("@" "-") "("  ")")
```

Default: `nil`

hnlMapTermFirstChar

List of characters that are invalid for the first character of a name when the `hnlMapTermName( )` function is used to map names. This list can contain either strings or lists. If the list contains a string, the string must contain a single character that indicates the character is invalid. If the character is used, the name is replaced by a new netlister-generated name. If the list contains a sublist, the first element of the sublist should be a string containing a single character that is the invalid character. If the sublist does not contain a second element or the second element is `nil`, the character is deleted when used as the first character of a name. If the second element of the sublist is a string, the string will replace the character specified as the first element. The second string can contain one or more characters.

Example:

```
hnlMapTermFirstChar = list("0" "1" 2" "3" "4" "5" "6" "7" "8" "9" '("@" "-"))
```

Default: `nil`

hnlMapModelFirstChar

List of characters that are invalid as the first character of a name when the `hnlMapModelName()` function is used to map names. This list can contain either strings or lists. If the list contains a string, the string must contain a single character that indicates the character is invalid. If the character is used, the name is replaced by a new netlister-generated name. If the list contains a sublist, the first element of the sublist should be a string containing a single character that is the invalid character. If the sublist does not contain a second element or the second element is `nil`, the character is deleted when used as the first character of a name. If the second element of the sublist is a string, the string will replace the character specified as the first element. The second string can contain one or more characters.

Example:

```
hnlMapModelFirstChar = list("0" "1" 2" "3" "4" "5" "6" "7" "8" "9" '("@" "-"))
```

Default: `nil`

hnlMapModelInName

List of characters that are invalid internal to a name when the `hnlMapModelName()` function is used to map names. This list can contain either strings or lists. If the list contains a string, the string must contain a single character that indicates the character is invalid. If the character is used, the name is replaced by a new netlister-generated name. If the list contains a sublist, the first element of the sublist should be a string containing a single character that is the invalid character. If the sublist does not contain a second element or the second element is `nil`, the character is deleted when used internal to a name. If the second element of the sublist is a string, the string will replace the character specified as the first element. The second string can contain one or more characters.

Example:

```
hnlMapModelInName = list( '("/" "|") '("@" "-") "("  ")")
```

Default: `nil`

hnlMapNetFirstChar

List of characters that are invalid for the first character of a name when the `hnlMapNetName()` function is used to map names. This list can contain either strings or lists. If the list contains a string, the string must contain a single character that indicates the character is invalid. If the character is used, the name is replaced by a new netlister-generated name. If the list contains a sublist, the first element of the sublist should be a string containing a single character that is the invalid character. If the sublist does not contain a second element or the second element is `nil`, the character is deleted when used as the first character of a name. If the second element of the sublist is a string, the string will replace the character specified as the first element. The second string can contain one or more characters.

Example:

```
hnlMapNetFirstChar = list("0" "1" 2" "3" "4" "5" "6" "7" "8" "9" '("@" "-"))
```

Default: `nil`

hnlMapNetInName

List of characters that are invalid internal to a name when the `hnlMapNetName( )` function is used to map names. This list can contain either strings or lists. If the list contains a string, the string must contain a single character that indicates the character is invalid. If the character is used, the name is replaced by a new netlister-generated name. If the list contains a sublist, the first element of the sublist should be a string containing a single character that is the invalid character. If the sublist does not contain a second element or the second element is `nil`, the character is deleted when used internal to a name. If the second element of the sublist is a string, the string replaces the character specified as the first element. The second string can contain one or more characters.

Example:

```
hnlMapNetInName = list( '("/" "|") '("@" "-") "("  ")")
```

Default: `nil`

hnlMaxLineLength

Maximum number of characters that a line output to the netlist file can contain, when using the `hnlPrintString` function. If the number of characters exceeds the specified limit, the line splits and a line continuation character is placed at the end of the line. In case you have longer expressions to be evaluated by a simulator, you can use the *`hnlSoftLineLength`* variable to prevent split of a line. You need to set the *`hnlMaxLineLength`* variable to nil while using the *`hnlSoftLineLength`* variable.

Default: `72`

Note: While using AMS OSS, if you need to evaluate long expressions, ensure that the value of the *`hnlMaxLineLength`* variable is set equal to the AMS OSS default value or a greater value. A value less than the AMS OSS default value 4000 is ignored.

hnlSoftLineLength

Maximum length of a line of output after which folding and continuation of the line needs to be considered. If 0, this is ignored and the behavior is the same as before taking into account only the *`hnlMaxLineLength`*.

Default: `0`

Note: While using AMS OSS, if you need to format long expressions, ensure that the value of the *`hnlSoftLineLength`* variable is set to the default 72 characters, which is also the

AMS OSS default value, or greater. A value less than 72 is ignored and the AMS OSS default value is set.

hnlMaxNameLength

Maximum number of characters allowed in a name. This variable must be set by the output formatter. It is used by the name translation functions.

Default: none

hnlMaxInstNameLength

Maximum number of characters allowed in an instance name. This variable must be set by the output formatter. It is used by the name translation functions.

Default: none

hnlMaxNetNameLength

Maximum number of characters allowed in a net name. This variable must be set by the output formatter. It is used by the name translation functions.

Default: none

simHnlDropUnusedInheritedPorts

Removes all unused inherited connections from a netlist. This is a boolean variable and can set in the `.simrc` file. Set the value of the variable as t to remove the unused inherited connections.

Default: nil

hnlModelNameDifferentFrom

List of name spaces from which the Model name should be different.

Example:

```
hnlModelNameDifferentFrom = ‘ (“parameter” “model”)
```

Default: none

hnlNamePrefix

String used to prefix system-generated names for names requiring complete remapping when using the `hnlMapName` function.

Default: `"hnl_"`

hnlNetlistFile

Output port for the netlist file if the `hnlDriverWillPrint` variable was set to `nil`.

Default: `nil`

hnlNetlistFileName

Name of the text netlist file.

Default: `"netlist"`

hnlNetNameDifferentFrom

List of name spaces from which the Net name should be different.

Example:

```
hnlNetNameDifferentFrom = ' ("terminal" "model")
```

Default: none

hnlViewList

List of strings that are the names of the valid views to switch into when expanding a cell.

Default: `nil`

hnlStopList

List of strings that are the names of the valid stopping view names.

Default: `nil`

hnlTermNameDifferentFrom

List of name spaces from which the terminal name must be different.

Example:

```
hnlTermNameDifferentFrom = ‘ (“net” “model”)
```

Default: none

hnlParamNameDifferentFrom

List of name spaces from which the Parameter name must be different.

Example:

```
hnlParamNameDifferentFrom = ‘ (“net” “terminal” “model”)
```

Default: none

hnlPrintInhConAtTop

Boolean flag that indicates whether the inherited connection is printed at the top cell or not.

Default: `nil`

hnlTopCell

`dbCellViewId` of the top cellview being netlisted.

Default: `nil`

hnlTopCellFuncs

List of functions to call to output the connectivity for the top-level schematic. This list can be modified to alter the order of functions called and, thus, alter the appearance of the netlist. Extra output functions may also be added to this list.

Default:

```
"hnlPrintTopCellHeader()"
"hnlPrintDevices()"
"hnlPrintTopCellFooter()"
```

hnlUserMultiTermShortCVList

Use this variable to short the terminals of a device during hierarchical netlisting. You can specify the cellview names and pin information in a list in the following syntax:

```
hnlUserMultiTermShortCVList  = '(("lib" "cell" "view" "(short(A B) short(C D
E)" ))
```

You must provide values for all the fields of this variable. Otherwise, the netlister reports an error.

Examples:

- Multiple cells with multiple terminals

```
hnlUserMultiTermShortCVList = '(("sample" "dffpp_c_" "symbol" "(short(in1 out3
out4))") ("testLib" "bottom" "symbol" "(short(in1 out2) short(in2 out3 out4
out5))"))
```

- A cell with two terminals

```
hnlUserMultiTermShortCVList = '(("sample" "dffpp_c_" "symbol" "(short(in1
out5))"))
```

For details, see “Removing Devices with Multiple Terminals” on page 109.

hnlUserShortCVList

Use this variable to short the terminals of a device during hierarchical netlisting. Use the variable in the following syntax to specify the cellview names in a list to remove the devices that have two terminals only.

```
hnlUserShortCVList = list(
                    ;;; all cells from this library
                    "libN"
                    ;;; cell1, cell2 and cell3 from lib1
                    list("lib1" "cell1" "cell2" "cell3")
                    ;;; all cells from this library
                    list("libM")  )
)
```

You can specify any of the elements in the list and keep the remaining elements blank.

Examples:

```
hnlUserShortCVList  = '("sample" "dffpp_c" "symbol")
hnlUserShortCVList  = '("sample" "dffpp_c" "")
```

For details, see “Removing Devices with Multiple Terminals” on page 109.

hnlUserStopCVList

List of user specified cellviews, which OSS treats as stop views while netlisting a design. You can specify this list in the `.simrc` file. Although instances of such a cellview appear in a netlist, the cellview module is not printed in the netlist.

Example:

```
hnlUserStopCVList = list
(
          ;;; all cells from this library
          "libN"
          ;;; cell1, cell2 and cell3 from lib1
          list("lib1" "cell1" "cell2" "cell3")
)
```

In this example, all the cellviews in the `libN` library will be treated as stop views. However, in the `lib1` library, only the `cell1`, `cell2`, and `cell3` cellviews will be treated as stop views.

Note: The list should have only one entry for each library, listing all the cellviews that need to be treated as stop views.

hnlUserIgnoreCVList

List of user specified cellviews, which OSS ignores while netlisting a design. You can specify this list in the `.simrc` file. If the list contains an entry of a cell, the instance or `subckt` of the cell is not printed. When an instance is ignored, it is in the open state.

Example:

```
hnlUserIgnoreCVList = list
(
          ;;; all cells from this library
          "libN"
          ;;; cell1, cell2 and cell3 from lib1
          list("lib1" "cell1" "cell2" "cell3")
)
```

In this example, all the cellviews in the `libN` library will be ignored when a design is netlisted. However, in the `lib1` library, only the `cell1`, `cell2`, and `cell3` cellviews will be ignored.

Note: Each library should only have a single entry listing all the cellviews that need to be ignored during netlisting.

simCapUnit

Refer to the “Customizing the Simulation Environment (SE)” chapter in this manual.

simTimeUnit

Refer to the “Customizing the Simulation Environment (SE)” chapter in this manual.

hnlNmpRemoveGlobalNetSuffix

Specifies that the `!` character would be removed from a global net name before mapping the map name through nmp. However, the `!` character is treated as a part of the global net name if the value of this variable is `nil`. A formatter should set the value of the variable to `t` while netlisting. If formatter does not set it, you can set the value of the `hnlNmpRemoveGlobalNetSuffix` variable in the `.simrc` file.

Example:

```
hnlRemoveGlobalNetPrefix = nil
```

```
glob!
```

In this example, `glob!` is treated as an identifier and the `!` character is not removed from the net name, `glob` before mapping. As a result, the `glob!` net name is mapped to `\glob!\`. If you set the value of the `hnlRemoveGlobalNetPrefix` variable to t , OSS removes the `!` character from the global net name, `glob!` and the mapped name will be `glob`.

Default: `nil`

HNL Access Functions

HNL supports various property, database, and print SKILL functions. For details on the HNL SKILL functions, see *OSS Functions* in the *Digital Design Netlisting and Simulation SKILL Reference*.

Incremental Hierarchical Netlisting

With the Incremental Hierarchical Netlister (IHNL) you can design a formatter that netlists incrementally. An incremental formatter checks each cellview in the design and netlists only the cellviews that the designer has modified since the previous netlisting. IHNL is simply an option to HNL; it is not a separate netlister. It is important that you read the basic HNL documentation first, before reading this section, which describes only additional functionality to allow your formatter to be used in the incremental netlisting mode. For simplicity, IHNL will be referred to as the hierarchical netlister running with the incremental feature.

How IHNL Works

IHNL automatically sets up an incremental netlisting directory whose name is stored in a variable named `hnlIncrementalDirectory`. The incremental netlisting directory is a subdirectory of the current netlisting directory. The incremental netlisting directory contains a subdirectory for each cellview (including the root cellview) referenced by the design. Each subdirectory contains a `netlist`, `map` and `control` file.

IHNL checks the `control` file to determine the cellviews for which it must create a new `netlist` file. IHNL netlists a cellview in the following cases:

- The designer has modified the cellview since the last netlisting.
- The designer has modified the symbol for an instance contained in the cellview.
- IHNL cannot find the `netlist`, `map`, or `control` file for the cellview.

IHNL creates new control and map files for the new netlist file. IHNL also creates the map/current name mapping file and an include file in the current netlisting directory. The include file lists the names and locations of the netlist files that the simulator must read for the simulation, or, if required by your simulator, the include file can contain a merged netlist of all the netlists for the sub-cells. Figure 5-4 on page 177 shows the file structure IHNL creates.

Important

It is possible that a design change impacts the hierarchy elaboration of a configuration-based design. For example, when you remove or set `nlAction=stop` for a component in a configuration-based design, the design hierarchy changes. However, the changes in the design hierarchy are not reflected in the design configuration, which can result in the generation of an incorrect netlist. To ensure that the updated design configuration is used, recompute the hierarchy and save the updates before netlisting the configuration-based design.

For details on using Hierarchy Editor to work with configuration-based designs and recompute the design hierarchy, see *Virtuoso Hierarchy Editor User Guide*.

Terms You Need To Know

The terms `cell name` and `module name` mean the following:

`cell name`	Name of a `design/macro` in Cadence schematic form.
`module name`	Name of the netlist of a `design/macro`, which is netlisted for a target simulator.

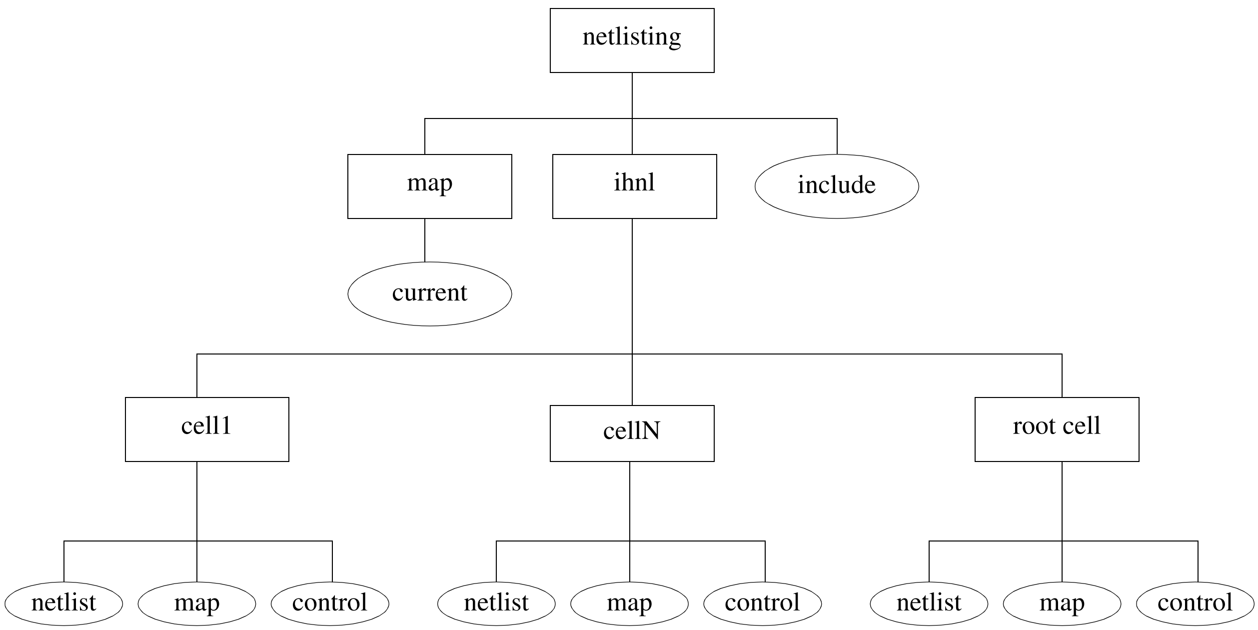
Name Mapping in IHNL

IHNL has all of the name mapping features of HNL. IHNL also performs name creation and character substitution in the same way as HNL. However, IHNL treats the following differently than HNL:

- Netlist module names
- Global nets

IHNL makes the names of all cells and global nets globally accessible. This means that you can use an IHNL function (such as `hnlMapCellModuleName` or `hnlGetGlobalMappedName`) to get the name of any cell or global net during an IHNL run, even if it is not in the cell you are netlisting. IHNL names netlist modules after their cell names, maps globally accessible names automatically, and records them in a global name table. IHNL makes subsequent references to globally accessible names from this table. The names of instances are locally accessible. This means you can only get the name of an instance when you netlist the cell in which it exists.

Figure 5-4 File Structure for an IHNL Run



Note: IHNL may map instances with identical names from different cells to different names. For example, if the instance *a* exists in two cells, IHNL may map the instances to two different names. When HNL runs in the non-incremental mode, it will always map these instances to the same name.

Netlist Module Names

You use the `hnlModulePrefix`, `hnlMapModelFirstChar`, and `hnlMapModelInName` variables to create the names of the netlist modules. When running in incremental mode, HNL uses the `hnlModulePrefix`, `hnlMapModelFirstChar`, and `hnlMapModelInName` variables to map invalid names to valid module names using the same name mapping rules as all other HNL name mapping functions such as `hnlMapNetName()`. Refer to the `hnlMapName` and the `hnlMapIfFirstChar` variable descriptions or to the `hnlMapName()` and `hnlInitMap()` function descriptions for additional information.

You cannot determine the order in which your formatter maps module names. IHNL automatically maps module names during design tree traversal. Your formatter can access the netlist module name of a design cell by calling the `hnlMapCellModuleName()` function.

Global Signals

You use the `hnlGlobalNetPrefix`, `hnlMapNetFirstChar`, and `hnlMapNetInName` variables to map global signals during design tree traversal. (The `hnlGlobalNetPrefix` lets you set the prefix string used when creating new names.) When running in incremental mode, IHNL uses the `hnlGlobalNetPrefix`, `hnlMapNetFirstChar`, and `hnlMapNetInName` variables to map invalid names to valid names using the same name mapping rules as all other HNL name mapping functions. Refer to the `hnlMapName` and the `hnlMapIfFirstChar` variable descriptions or to the `hnlMapName()` and `hnlInitMap()` function descriptions for additional information.

When you specify a prefix for a globally accessible name, make sure that prefix is different from prefixes for names that are not globally accessible. If you do not do this, your incremental formatter may create redundant names because it netlists only part of the design. Also, we recommend that you use only simple substitutions with IHNL (for example, substituting uppercase for lowercase letters). If you use complex substitutions (such as string substitutions) that are harder for you to remember, you may accidentally use the same prefix for globally accessible names and names that are not globally accessible. Use the `hnlGetGlobalMappedName()` function to access the results of the mapping.

Known Problems

If you redesign the netlist formatter for a simulator, the previous netlist may not reflect the new design of the formatter.

To correct this problem, purge the existing IHNL directory (`hnlIncrementalDirectory`) for the design before you netlist again, or you can set `simReNetlistAll` to `t` and netlist again.

Writing an Incremental Netlist Formatter

To write an incremental netlist formatter, you must

1. Decide how your formatter will netlist global nets and order netlist files, and how the target simulator will read the netlist files.
2. Write an HNL formatter.
3. Convert the HNL formatter to also function in incremental mode.

Configuring an Incremental Netlist Formatter

When you design an incremental formatter, you must decide the following issues:

- How the formatter netlists global nets

  We recommend that you netlist the global nets in the include file or in one file. You can also write global nets to a netlist file. However, if you do this, the netlist file that contains the global net definition may not be recreated in a subsequent IHNL run. This may result in name collision if the global net is renamed.

- How the formatter orders netlist files

  IHNL automatically determines the order of the netlist files. The netlist files are ordered so that a netlist filename appears before it is instantiated in other netlists. IHNL stores the names of all the netlist files in the `hnlNetlistFileList` variable. If you want to reverse the order of these files, use the `reverse` SKILL function.

- How the target simulator reads the netlist files

  For most simulators it is best to design your formatter so that it generates an `include` file that tells the simulator to use all the netlist files. You can also design your formatter so that the `include` file combines all of the netlist files in one file before passing it to the simulator.

Writing the HNL Formatter

Write your HNL formatter as described in “Writing a Formatter,” but do not redefine the `hnlFindAllCells()` function.

Converting to an IHNL Formatter

To convert your HNL formatter to an IHNL formatter, do the following:

1. Add the following four statements to your HNL formatter:

```
hnlSetDef( 'hnlIncremental t )
hnlSetDef( 'hnlModulePrefix "prefix1" )
hnlSetDef( 'hnlGlobalNetPrefix "prefix2" )
hnlSetDef( 'hnlIncludeFileName "name" )
hnlSetDef( 'hnlIncrementalOnly t )
```

   where

   `prefix1` is the prefix string for a mapped module name.

   `prefix2` is the prefix string for a mapped global net name.

   `name` is the name of the `include` file `hnlGenIncludeFile()` creates.

2. Define `hnlMapNetFirstChar`, `hnlMapNetInName`, `hnlMapModelFirstChar`, `hnlMapModelInName`, `hnlMapTermFirstChar`, and `hnlMapTermInName` if they are not already defined.

3. Convert any formatter code that prints out a declaration of global nets. Example: convert the following code for a formatter for the SILOS simulator:

```
if( hnlAllGlobals != nil the
      hnlPrintString( ".GLOBAL" )
   foreach( name hnlAllGlobals
      hnlPrintString( " " )
      hnlPrintString( hnlMapName( name )
   )
```

   to the following, where the hnlGetGlobalMappedName( ) function is used to find the mapped names of all global nets:

```
if( hnlAllGlobals != nil then
   fprintf( hnlIncludeFile ".GLOBAL" )
   foreach( name hnlAllGlobals
      fprintf( hnlIncludeFile " %s"

hnlGetGlobalMappedName( name ) )
   )
```

   This code netlists global nets in an `include` file. Remember that `hnlIncludeFile` can only be written by `fprintf()` because the `hnlPrintString()` function writes to the netlist file for the currently active cell being output.

4. Add code to instruct the simulator to read each of the netlist files. Example: use the following code to make SILOS read all netlist files by following the instructions in the `include` file:

```
foreach( name hnlNetlistFileList
```

```
   fprintf( hnlIncludeFile "!INPUT %s/%s/%s\n" hnlNetlistDirectory
hnlIncrementalDirectory name )
)
```

5. Change calls to `hnlMapModelName()` to `hnlMapCellModuleName()` for all references to a model (or module name of a netlist). However, this should not be done for stopping cells.

   Note: When you write an incremental netlister, remember these points:

   - Do not write a formatter that redefines the `hnlFindAllCells()` function.
   - Be aware of the name restrictions for your simulator. Do not use prefixes that will result in illegal names for your simulator.
   - Never use the `hnlGetGlobalMappedName()` function to find a netlist module name by using its cell name. You cannot find the module name with the cell name because IHNL supports multiple definitions of cell names (by using multiple reference libraries). Use `hnlMapCellModuleName()` instead.
   - If you plan to use remote simulation, all of the netlist files generated must be accessible on the file server. For Cadence standard remote simulation, this means the entire netlist for all subcircuits as well as the top-level cell view must be in a single file called `netlist`. The following sample does not do this, and, therefore, could not be used in conjunction with remote simulation.
   - If you simply use the replacement functions discussed, your netlist formatter will work only in incremental mode. You can write a formatter that works in basic HNL mode as well as in incremental, but you must perform extra checking. For example, you would only use the `hnlMapCellModuleName()` function when running in incremental mode, and continue to use `hnlMapModelName()` or `hnlMapName( )` in non-incremental mode.

   This could be done as follows:

```
if (hnlIncrementalMode then
   hnlPrintString (hnlMapCellModuleName (hnlCurrentMaster))
else
     hnlPrintString (hnlMapName (hnlCurrentMaster~>cellName))
)
```

Example

This example shows you how to convert an existing HNL SILOS formatter to an incremental formatter. The first section of the example shows you the sections of the existing HNL formatter that you must modify for incremental netlisting. The second part shows you the changes you must make.

HNL Code To Change

This HNL formatter netlists the .MACRO definition for a netlist module with the `hnlPrintDeviceHeader()` function. It netlists the .EOM (end of macro) definition of the module with the `hnlPrintDeviceFooter()` function and netlists references to all macros with the `hnlSilosPrintMacroRef( )` function. It netlists global nets (.GLOBAL) with the `hnlPrintNetlistHeader()` and the power net definition with `hnlPrintNetlistFooter()`. The formatter records name mapping in the `generic` section of the name mapping file.

```
; This procedure prints the .GLOBAL statement as the first entry
; in the netlist file.
;
hnlIfNoProcedure( hnlPrintNetlistHeader()
   let( ( name )
      if( hnlAllGlobals != nil then
      hnlPrintString( ".GLOBAL" )
      foreach( name hnlAllGlobals
         hnlPrintString( " " )
         hnlPrintString( hnlMapName( name ) )
      )
       )
   )
)

; This procedure prints the VDD and GND definitions
; at the end of the netlist file.
;
hnlIfNoProcedure( hnlPrintNetlistFooter()
   let( ( name )
      foreach( name hnlAllGlobals
         if( name == "vdd!" then
hnlPrintString( hnlMapName(name) )
hnlPrintString( " .CLK 0 S1\n" )
)
if( name = "gnd!" then
hnlPrintString( hnlMapName(name) )
hnlPrintString( " .CLK 0 S0\n" )
)
)
)
)
)
```

```
; This procedure prints the netlist module definition in the
; form of .MACRO cellName.
; If cellName is multiply defined, use the full path name.
hnlIfNoProcedure( hnlPrintDeviceHeader()
hnlPrintString( "\n.MACRO " )
if( hnlMultipleCells( hnlCurrentCell ~> cellName ) then
hnlPrintString( hnlMapName( hnlCurrentCell ~> fullPathName ) )
else
hnlPrintString( hnlMapName( hnlCurrentCell ~> cellName ) )
)
)
; This procedure prints the netlist module definition in the
; form of .EOM cellName.
; This procedure also verifies the multiplicity of the
; cell name.
hnlIfNoProcedure( hnlPrintDeviceFooter()
hnlPrintString( "\n.EOM " )
if( hnlMultipleCells( hnlCurrentCell ~> cellName ) then
hnlPrintString( hnlMapName( hnlCurrentCell ~> fullPathName ) )
else
hnlPrintString( hnlMapName( hnlCurrentCell ~> cellName ) )
)
)

; This procedure prints the module reference in the form of
; (instName moduleName.
hnlIfNoProcedure( hnlSilosPrintMacroRef()
hnlPrintString( "(" )
hnlPrintString( hnlMapName( hnlCurrentInstName ) )
hnlPrintString( " " )
if( hnlMultipleCells( hnlCurrentMaster ~> cellName ) then
hnlPrintString( hnlMapName( hnlCurrentMaster ~> fullPathName ) )
else
hnlPrintString( hnlMapName( hnlCurrentMaster ~> cellName ) )
)
)
```

IHNL Code To Add

In this example, the `include` file is called `netlist`. The prefix for mapping netlist module names is `SILOS`, and the prefix for global net mapping is `SNG`. The `hnlMapModelFirstChar`, `hnlMapModelInName`, `hnlMapNetFirstChar`, and `hnlMapNetInName` variables are set to values acceptable for formatting a netlist for the SILOS simulator. The global and power net definitions are written to the `include` file.

Add the following statements to your formatter:

```
hnlSetDef( 'hnlIncremental t )
hnlSetDef( 'hnlIncrementalOnly t )
hnlSetDef( 'hnlModulePrefix "SILOS" )
hnlSetDef( 'hnlGlobalNetPrefix "SNG" )
hnlSetDef( 'hnlMapModelFirstChar "0" "1" "2" "3" "4" "5" "6" "7" "8" "9" "*" ("a"
"A"))
hnlSetDef( 'hnlMapModelInName "-" "*")
hnlSetDef( 'hnlMapNetFirstChar "0" "1" "2" "3" "4" "5" "6" "7" "8" "9" "*" ("a"
"A") ("b" "B"))
hnlSetDef( 'hnlMapNetInName "*" "-" )

; This procedure prints no header for IHNL.
;
hnlIfNoProcedure( hnlPrintNetlistHeader()
t
)

; This procedure prints no footer for IHNL.
;
hnlIfNoProcedure( hnlPrintNetlistFooter()
t
)

; This procedure prints the netlist module definition in the
; form of .MACRO cellName.
hnlIfNoProcedure( hnlPrintDeviceHeader()
hnlPrintString( "\n.MACRO " )
hnlPrintString( hnlMapCellModuleName( hnlCurrentCell ) )
)

; This procedure prints the netlist module definition in the
; form of .EOM cellName.
hnlIfNoProcedure( hnlPrintDeviceFooter()
```

```
hnlPrintString( "\n.EOM " )
hnlPrintString( hnlMapCellModuleName( hnlCurrentCell ) )
)
```

Note: You do not have to check if the cell name is multiply defined when running in incremental mode.

```
; This procedure prints the module reference in the form of
; (instName moduleName.
hnlIfNoProcedure( hnlSilosPrintMacroRef()
hnlPrintString( "("
hnlPrintString(  hnlMapName( hnlCurrentInstName ) )
hnlPrintString( " " )
hnlPrintString( hnlMapCellModuleName( hnlCurrentMaster ) )
)

; This procedure creates the include file.  The .GLOBAL and
; power net definitions are written into the include file as the
; first and last commands.
; Create the !INPUT SILOS command to read in all
; the netlist files.
; Use full path names.
; You must define this procedure to create a file instructing
; the simulator how to read in the netlist files for each
; cellview.
hnlIfNoProcedure( hnlGenIncludeFile( ()
   let( ( name )
      if( hnlAllGlobals != nil then
         fprintf( hnlIncludeFile ".GLOBAL" )
         foreach( name hnlAllGlobals
            fprintf( hnlIncludeFile " %s" hnlGetGlobalMappedName( name ) )
      )
   )
      foreach( name hnlNetlistFileList
      fprintf( hnlIncludeFile "!INPUT %s/%s/%s\n"
         hnlNetlistDirectory hnlIncrementalDirectory name )
   )
   if( hnlAllGlobals != nil then
      if( name == "vdd!" then
         fprintf( hnlIncludeFile "%s .CLK 0 S1\n"
            hnlGetGlobalMappedName( name ) )
      )
```

```
      if( name == "gnd!" then
         fprintf( hnlIncludeFile "%s .CLK 0 S0\n" hnlGetGlobalMappedName( name ) )
      )
      )
   )
)
```

Variables and Functions for Incremental Netlisting

Use the following variables and functions when you write an incremental netlister. The meanings of some HNL variables and functions change when the formatter is running in incremental mode.

Definable Variables

The following are the variables that you can define in the netlist formatter or the designer can define through SKILL or with a form at run time.

hnlGlobalNetPrefix

Specifies the string the formatter should use as the prefix for all global nets in a netlist.

The netlister uses no prefix by default when it creates mapped global net names.

Default: `nil`

hnlIncludeFileName

Name of the `include` file. Use the `include` file to instruct the simulator to read all netlist files from the design during simulation. This variable should be defined in the global section of the `caplib` file for the tool. The designer should not change this variable.

Default: `nil`

hnlIncremental

Boolean flag specifying that the formatter can function in incremental mode. If your formatter has the necessary functions for incremental netlisting, then set this variable to `t` in your formatter.

Default: `nil`

hnlModulePrefix

Specifies the string the formatter uses as the prefix for a mapped module name in a netlist. A SILOS formatter uses the string `cds` for this variable. Example: `.MACRO cds12 <pins>`.

The netlister uses no prefix by default when it creates mapped module names.

Default: `nil`

hnlReNetlistAll

Specifies that IHNL should netlist all cellviews used in the design when set to `t`. Do not write your formatter to set this variable. It is a run time option that the designer can set. The designer can set this variable through SKILL or with a form.

Default: `nil`

hnlIncrementalOnly

Global flag to indicate that your formatter can only function in incremental mode, and is not capable of functioning in non incremental mode. Set this variable to `t` in your formatter if the formatter can only run in incremental mode, and the user setting of the `simNotIncremental` variable will be ignored, enforcing incremental netlisting. If you set this variable to `t`, you should also set the `hnlIncremental` variable to `t`.

Default: `nil`

hnlReservedNameList

Usually, when you use reserved words or keywords of a simulator as design variable names, then OSS automatically remaps these design variables. These variables then appear as mapped in expressions and parameter lists in the netlist. If you want to prevent the automatic name mapping, then use the SKILL list variable, `hnlReservedNameList`, to specify a list of simulator-specific reserved words. For example, in the case of hspice, if you are using temper, Temper, or TEMPER in the design, you would need to include the valid mapping of the word. In this case, it would be the word, `temper`.

Default: `nil`

Variables IHNL Defines

The following are the variables that the formatter can use to extract information from the design for the netlist. IHNL defines these variables. Do not redefine them. Many of these variables do not have defaults because the values change from run to run.

hnlIncludeFile

File pointer to the include file. You must use fprintf in your formatter to write the include file. IHNL opens this file after it traverses the design and closes it at the end of the netlisting session.

hnlIncrementalDirectory

Name of the incremental directory, which is a subdirectory of the current netlisting directory and contains the output of IHNL. The `hnlIncrementalDirectory` is a relative path name in the current netlisting directory.

hnlIncrementalMode

Signals if a formatter is in fact running in incremental mode. Because you can design your formatter to netlist in incremental and/or non incremental mode, you must use this variable to decide if your formatter is in fact running in incremental mode. If you design your formatter to run in incremental mode only, this variable is always set to `t`.

Default: the value of this variable is derived using the following expression:

```
hnlIncremental && (hnlIncrementalOnly||simNotIncremental==nil)
```

hnlListOfAllCells

List of all the cellviews in the design that need to be renetlisted.

hnlMacroBlockFuncs

Defines the procedures to execute when netlisting a macro cell. When you define this variable, call the `hnlSetCellFiles()` and `hnlCloseCellFiles()` functions. If you do not do this, your netlister will be unable to netlist incrementally. The `hnlSetCellFiles()` and `hnlCloseCellFiles()` functions simply return if your netlister is not running in incremental mode.

Default:

```
hnlSetDef( 'hnlMacroCellFuncs
             '( "hnlSetCellFiles( )"
                "hnlPrintMacroCellHeader( )"
                "hnlPrintDevices( )"
                "hnlPrintMacroCellFooter( )"
                "hnlCloseCellFiles( )"
              )
          )
```

hnlNetlistDirectory

The full path name of the current netlisting directory.

hnlNetlistFile

File pointer for writing the netlist. Use `hnlMakeNetlistFileName` to get the relative path name of this file. In HNL, you use the `hnlNetlistFileName` variable to specify this file pointer.

hnlNetlistFileList

List of all the netlist file names. The file names in this list are the relative file names in `hnlIncrementalDirectory`. The list is ordered so that a netlist file name appears before a file name that references the cell it describes.

hnlCellSeenList

List of all the cellviews in the design. The list is ordered so that a cellview appears before the cellview that references the cellview it describes.

hnlTopCellFuncs

Defines the procedures to execute when netlisting the top-level cellview. When you define this variable, call the `hnlSetCellFiles()` function before other formatting procedures. Call the `hnlCloseCellFiles()` function after any other formatting procedures. If you do not do this, your netlister will be unable to netlist incrementally. The `hnlSetCellFiles()` and `hnlCloseCellFiles( )` functions simply return if your netlister is not running in incremental mode.

Default:

```
hnlSetDef( 'hnlTopCellFuncs
          '( "hnlSetCellFiles( )"
            "hnlPrintTopCellHeader( )"
            "hnlPrintDevices( )"
            "hnlPrintTopCellFooter( )"
            "hnlCloseCellFiles( )"
         )
      )
```

6

Customizing the HNL Net-Based Netlister

In addition to instance-based netlisting capability, HNL supports net-based netlisting to facilitate the interface to net-based simulators and tools.

Similar to the instance-based capability, HNL traverses a given design, retrieves the relevant information, and sets up the environment for a formatter to generate the required netlist. Most processes are performed automatically by HNL, but generating a netlist in the targeted syntax is performed by the formatter.

The design of the formatter is similar to an instance-based one. When designing a net-based netlist formatter, the designer MUST set the HNL variable `hnlInstBased` to `nil`, which signals that the net-based capability is desired.

Flow of Net-Based HNL

When running in the net-based mode, HNL first traverses all models (cells) in the given design, identifying all global signals while preparing for the netlist generation steps. The design traversal and global signals identification processes are identical for both net-based and instance-based HNL, but are different from the incremental HNL. The net-based traversal cannot be used with the hierarchical netlister in incremental mode.

After the initial design traversal, HNL prints the required netlist header as instructed, then goes through all referred models in the design for netlist generation, beginning with call `hnlPrintNetlistHeader()`, then going through call `hnlDoNetBased()` and call `hnlPrintNetlistFooter()`.

The `hnlPrintNetlistHeader()` and `hnlPrintNetlistFooter()` procedures are explained in the HNL chapter of this manual. The `hnlDoNetBased()` procedure addresses each cell, prepares the internal data structures, and calls the procedures specified by the HNL variable, `hnlTopCellNetFuncs`, if the cell being netlisted is the top design cellview, or `hnlMacroCellNetFuncs` otherwise.

By default, `hnlTopCellNetFuncs` will activate the procedures `hnlPrintTopCellHeader`, `hnlPrintSignal`, and `hnlPrintTopCellFooter` while `hnlMacroCellNetFuncs` activates `hnlPrintDeviceHeader`, `hnlPrintPorts`, `hnlPrintSignal`, and `hnlPrintDeviceFooter`. The `hnlPrintSignal` procedure is responsible for netlisting each signal and its connection while the procedures with the name "Header" are responsible for netlisting the header of a cellview and netlist that are to appear before the signals and their connections. Similarly, "Footer" procedures are for netlisting all netlists that are to appear after the connections and the footer.

The default procedure, `hnlPrintSignal`, goes through all signals in `hnlCurrentCell`, finds the connectivity to the signal, then assigns the proper values to the supported variables for netlist formatting. For each signal, the procedures `hnlPrintSignalHeader`, `hnlPrintInst`, and `hnlPrintSignalFooter` are called to generate the netlist.

For descriptions of the above-mentioned procedures, refer to the "HNL Procedures for Net-Based Netlisting" section in this chapter.

The default flow of `hnlPrintSignal` is

```
for each signal in the cell being netlisted
     call hnlPrintSignalHeader( )

     for each instance connect to current signal
        prepare variables for netlisting
        call hnlPrintInst( )
    endfor
    call hnlPrintSignalFooter( )
 endfor
```

For details on the variables available for the netlist formatter, refer to the "HNL Variables for Net-Based Netlisting" section in this chapter.

HNL Variables for Net-Based Netlisting

The following HNL variables are available for use by the formatter for net-based netlisting. Some of these variables can be redefined by the user, and some are only defined during the evaluation of certain functions.

All variables used for managing the name mapping function are the same as for instance-based HNL. All variables used for performing output formatting are also identical to instance-based HNL. All names provided by HNL for performing net-based netlisting are single bit names, which means that signal names are always used instead of net names, and the name of each iteration of an instance is used for iterated instances.

hnlInstBased

The default is `t`. You must set the `hnlInstBased` variable to `nil` to run HNL in net-based mode.

hnlListOfAllCells

The list of all cellviews used in the given design. Defined after the initial design traversal is completed.

hnlListOfAllStoppingCells

The list of all primitives (stopping cellviews) used in the given design. Defined after the initial design traversal is completed.

hnlCurrentCell

The `dbCellViewId` of the current cell being netlisted. Defined by `hnlDoNetBased()`.

hnlAllTerms

The list of all terminals in `hnlCurrentCell`. Defined by `hnlDoNetBased()`.

hnlAllTermNames

The list of the names of all of the bits of all of the terminals in `hnlCurrentCell`. The names in this list are flattened (reduced to single bit names) and stored in the same order as `hnlAllTerms`. Defined by `hnlDoNetBased()`.

hnlCellInTerms

The list of all input terminals in `hnlCurrentCell`. Defined by `hnlDoNetBased()`.

Inherited terminals can be found in this list but since these are real terminals there are db id. associated with them. Here for inherited terminals the relationship with the entries in `hnlCellInputs` are db id. to *netlister-generated* names. These *netlister-generated* names are the same ones that appeared in `hnlCellInputs`.

Example

For module *buffer,* the value is list(db_id_IN).

hnlCellInTermName

The list of the names of all the bits of all input terminals in `hnlCurrentCell`. They are flattened and stored in the same order as `hnlCellInTerms`. Defined by `hnlDoNetBased()`.

hnlCellOutTerms

The list of all output terminals in `hnlCurrentCell`. Defined by `hnlDoNetBased()`.

Inherited terminals can be found in this list but since these are real terminals there are db id. associated with them. Here for inherited terminals the relationship with the entries in `hnlCellOutputs` are db id. to *netlister-generated* names. These *netlister-generated* names are the same ones that appeared in `hnlCellInputs`.

Example

For module *buffer,* the value is list(db_id_OUT).

hnlCellOutTermNames

The list of the names of all the bits of the output terminals in `hnlCurrentCell`. They are flattened and stored in the same order as `hnlCellOutTerms`. Defined by `hnlDoNetBased()`.

hnlCellOtherTerms

The list of all terminals in `hnlCurrentCell` that are neither input nor output terminals. Defined by `hnlDoNetBased()`.

Logically there is no db id. for pseudo port created as a result of inherited signal. A *netlister-generated-name* generated by HNL is used as a place-holder instead. Inherited terminals can be found in this list but since these are real terminals there are db id. associated with them. Here for inherited terminals the relationship with the entries in `hnlCellOutputs` are db id. to *netlister-generated* names. These *netlister-generated* names are the same ones that appeared in `hnlCellOthers`.

Example

For module *buffer,* there are no db ids for the pseudo ports created but *pseudo-names* generated by HNL are used as the place holders for the inherited connections. The value is list(“inh_gnd” “inh_vdd”)

hnlCellOtherTermNames

The list of the names of all the bits of the terminals in `hnlCurrentCell` that are neither input nor output terminals. They are flattened and stored in the same order as `hnlCellOtherTerms`.

hnlTopCell

The `dbCellViewId` of the top cellview of the design being netlisted. This variable is defined throughout the netlisting process, but is invalidated after netlisting is completed because the top cell is automatically closed by HNL.

hnlTopCellNetFuncs

A variable defining the procedures to be called by `hnlDoNetBased` for netlisting the top cellview. The default value is

```
'( "hnlPrintTopCellHeader( )"
   "hnlPrintSignal( )"
   "hnlPrintTopCellFooter( )"
   )
```

hnlCurrentSignal

The signal in `hnlCurrentCell` that is being netlisted. Defined in `hnlPrintSignal()`. It becomes invalid after all signals in `hnlCurrentCell` have been netlisted.

hnlCurrentSignalTerms

The names of the terminals in `hnlCurrentCell` that are connected to the bits of the `hnlCurrentSignal`. Defined in `hnlPrintSignal()`.

hnlCurrentSignalName

Name of `hnlCurrentSignal`. Defined in `hnlPrintSignal()`.

hnlCurrentInst

The instance in `hnlCurrentCell` which has one of its pins connected to `hnlCurrentSignal`. This variable may refer to an iterated instance. The variable `hnlCurrentInst` should be used in conjunction with the variable `hnlCurrentIteration` to identify the instance being netlisted. Defined by `hnlPrintSignal()` after `hnlPrintSignalHeader()` has been called.

hnlCurrentInstName

The name of the iteration of the instance that is currently being netlisted as defined by `hnlCurrentInst` and `hnlCurrentIteration`. Defined in `hnlPrintSignal()`.

hnlCurrentIteration

The iteration number of the instance of `hnlCurrentInst` which is being netlisted. Defined in `hnlPrintSignal()`.

hnlCurrentType

The cell name of the master of `hnlCurrentInst`. Defined in `hnlPrintSignal()`.

hnlCurrentMaster

The `dbCellViewId` of the view switched master of `hnlCurrentInst`. Defined by `hnlPrintSignal()`. It is valid only during evaluation of the `hnlPrintInst()` function.

hnlCurrentInstPort

The instance terminal of `hnlCurrentInst` which is connected to `hnlCurrentSignal`. This variable may refer to a multiple bit terminal in order to access the terminal bit that is connected. `hnlCurrentInstPortIndex` should be used when needed. Defined by `hnlPrintSignal()`. Valid only during evaluation of the `hnlPrintInst()` function.

hnlCurrentPortName

The name of `hnlCurrentInstPort`. Defined by `hnlPrintSignal()`. To get the name of the terminal bit that is connected to `hnlCurrentSignal`, you must use `dbGetMemName(hnlCurrentInstPortName hnlCurrentInstPortIndex)`.

hnlCurrentInstPortIndex

The index of the bit of `hnlCurrentInstPort` that is connected to the current signal. Defined by `hnlPrintSignal()`.

hnlInstMasterPort

The terminal on `hnlCurrentMaster` that is connected to the current signal. Defined by `hnlPrintSignal()`.

hnlMacroCellNetFuncs

A variable defining the procedure to be called by `hnlDoNetBased` for netlisting all cells that are not the top cell. The default value is

```
'( "hnlPrintDeviceHeader( )"
   "hnlPrintPorts"
   "hnlPrintSignal( )"
   "hnlPrintDeviceFooter( )"
   )
```

hnlProcessAliasSignalWithSourceDirection

A variable that specifies that the netlister uses aliasing between more than two signals. By default, it is set to `nil`. It requires adding the `direction` property of the net, where the property value must be set to the source.

The following example shows aliasing between the signals `a`, `z<0>`, and `z<1>`.

```
     a       z<1>             a       z<0>
---[src]~~>[dst]-------  ---[src]~~>[dst]-------
```

Here, if the source net is `a`, then the `direction` property must be set on net a, where the property value is set to source.

HNL Procedures for Net-Based Netlisting

Various HNL SKILL functions are available for use by the formatter for net-based netlisting. For details, see <u>*OSS Functions*</u> in the *Digital Design Netlisting and Simulation SKILL Reference*.

Other Variables and Procedures

In addition to the variables and procedures previously described, other variables and procedures are available, as categorized below:

Controlling the Format of the Netlist File

If you call `hnlPrintString()` to output the netlist, you should set the following variables.

```
hnlMaxLineLength
hnlCommentStr
hnlDriverWillPrint
hnlLinePostfix
hnlLinePrefix
```

Variable and Name Mapping Functions

The same set of name mapping procedures are supported for net-based netlisting as for instance based netlisting. All the related variables should be set by the formatter accordingly. Following are the variables you should set.

```
hnlMapIfFirstChar
hnlMapIfInName
hnlMapInstFirstChar
hnlMapInstInName
hnlMapModelFirstChar
hnlMapModelInName
hnlMapNetFirstChar
hnlMapInNetName
hnlMaxNameLength
hnlNamePrefix
hnlMapTermFirstChar
hnlMapTermInName
simNetNamePrefix
simInstNamePrefix
simModeNamePrefix
```

Following are the available name mapping functions.

```
hnlMapInstName
hnlMapModelName
hnlMapNetName
hnlMapName
```

Other Procedures and Functions

All procedures and functions supported for instance-based netlisting are also supported for net-based netlisting, unless redefined in the “HNL Procedures for Net-Based Netlisting” section. For details on these procedures and functions, refer to the “Customizing the Hierarchical Netlister” chapter in this manual.

Other Variables

All variables supported in instance-based HNL are also supported for net-based netlisting, unless redefined in the “HNL Variables for Net-Based Netlisting” section.

Procedures the Formatter Must Define

The following procedures must be included when you define the formatter.

hnlPrintTopCellHeader( )

The procedure called before calling `hnlPrintSignal()` for netlisting the top-level cell. All the variables set up by `hnlDoNetBased()` are available.

hnlPrintTopCellFooter( )

The procedure called after `hnlPrintSignal()` is completed for the top cell. This is the last procedure called for netlisting the top cell.

hnlPrintDeviceHeader( )

The procedure called before calling `hnlPrintSignal()` for all but the top-level cellview. All the variables set up by `hnlDoNetBased()` are available. This procedure calls for netlisting information that is not managed/netlisted by `hnlPrintSignal()`.

hnlPrintDeviceFooter( )

The procedure called after `hnlPrintSignal()` is completed for all but the top-level cellview. This procedure is called to generate a netlist after all signals’ connectivities that are not handled by `hnlPrintSignal()`. Default is `nil` procedure.

hnlPrintPorts( )

The procedure to netlist the port definition for each cellview.

hnlPrintSignalHeader( )

The procedure to netlist the signal definition before it is netlisted. Called by `hnlPrintSignal` before `hnlPrintInst()`, which is before all instances connected to `hnlCurrentSignal` have been netlisted.

hnlPrintSignalFooter( )

The procedure to netlist the signal definition after it is netlisted. Called by `hnlPrintSignal` after `hnlPrintInst()`, which is after all instances connected to `hnlCurrentSignal` have been netlisted.

hnlPrintInst( )

The procedure to netlist an instance (defined by `hnlCurrentInst` and `hnlCurrentIteration`) and its connectivity to signal `hnlCurrentSignal`. All variables described in the previous section are available and assigned the proper values when this procedure is called. This procedure MUST be defined by all formatters.

Designing an HNL Net-Based Formatter

Designing a formatter for net-based syntax is similar to designing a formatter for instance-based syntax. Most of the required data is available, and the environment is ready to be used. Simply follow the procedure below to complete designing and coding of your formatter.

Determine Name Mapping and Netlist Syntax Needed

Determine the netlist and name mapping requirements needed to generate the correct netlist.

Initialize Variables

You must set `hnlInstBased` to `nil`. Do not forget to set the hnl variables that control name mapping and output formatting to the desired values. If you want to set values for other HNL variables, refer to the “Customizing the Hierarchical Netlister” chapter in this manual.

Code the Needed Procedures

Design the procedures for your formatter by following the procedure in the "Controlling the Format of the Netlist File" section. Then, code the following routines accordingly: `hnlPrintInst()`, `hnlPrintTopCellHeader()`, `hnlPrintTopCellFooter()`, `hnlPrintSignalHeader()`, `hnlPrintSignalFooter()`, `hnlPrintPorts()`, `hnlPrintDeviceHeader()`, and `hnlPrintDeviceFooter()`. You may need to code some additional procedures to help handle some specific formatter needs.

Net-Based Netlister Design Example

This section includes a simple net-based netlister design to demonstrate the fundamentals of designing a similar netlist formatter using the variables and procedures previously discussed.

For this example, the syntax of the target netlist complies with the following form:

```
Model modelName;
Ports
   [PortNames [=signalName] ;]*
endPorts
Object
   [InstanceName InstanceType ;]*
endObjects
Connection
   [signalName [InstanceName.pinName]* ;]*
endConnection
endModel [modelName].
```

where [ ] specifies optional field, [ ]+ specifies a field that must appear at least once, and [ ]* specifies a field that may occur any number of times. This syntax applies to all modules of a design, except that the keyword `Design` is to be used in place of `Model` for the top-level modules.

Designing the Netlister

For this example, the legal syntax for a name is a string that starts with an alphabet and does not contain any of the following: "." "," "+" "-" "*" "@" "%" "#" and "!". All comment lines are preceded by "#". It is also assumed that "." is used by the target simulator as hierarchy delimiter.

All module names are mapped using `hnlMapModelName()`, all signal (net) names are mapped using `hnlMapNetName()`, all port names are mapped using `hnlMapped Name()`, and all instance names are mapped by `hnlMapInstName()`. The port names are derived directly from the terminal names. Pin names of instances are the port names of its master cellview.

The Formatter

The netlist formatter is designed as shown below.

```
;
; This example shows how to use HNL to design a net-based netlist formatter.
;
; First, define the need variable for net-based netlisting.
hnlInstBased = nil
hnlFormatFuncsLoaded = t
hnlNetlistFileName = "netlist"
hnlSetDef( 'hnlMaxNameLength 12 )
hnlSetDef( 'hnlHierarchyDelimeter "." )
hnlSetDef( 'hnlMaxLineLength 72 )
hnlSetDef( 'hnlMapTermFirstChar list( "0" "1" "2" "3" "4" "5" "6" "7" "8" "9"
      "." "," "+" "-" "*" "%" "@" "#" "!"
)
   )
hnlSetDef( 'hnlMapTermInName list( "." "," "+" "-" "*" "#" "!" "@" "<" ">" )
    )
hnlSetDef( 'hnlMapNetFirstChar list( "0" "1" "2" "3" "4" "5" "6" "7" "8" "9"
      "." "," "+" "-" "*" "%" "@" "#" "!"
)
   )
hnlSetDef( 'hnlMapNetInName list( "." "," "+" "-" "*" "#" "!" "@" "<" ">" )
    )
hnlSetDef( 'hnlMapInstFirstChar list( "0" "1" "2" "3" "4" "5" "6" "7" "8" "9"
      "." "," "+" "-" "*" "%" "@" "#" "!"
)
   )
hnlSetDef( 'hnlMapInstInName list( "." "," "+" "-" "*" "#" "!" "@" "<" ">" )
 )
hnlSetDef( 'hnlMapModelFirstChar list( "0" "1" "2" "3" "4" "5" "6" "7" "8" "9"
      "." "," "+" "-" "*" "%" "@" "#" "!"
)
   )
hnlSetDef( 'hnlMapModelInName list( "." "," "+" "-" "*" "#" "!" "@" "<" ">" )
   )

hnlSetDef( 'hnlFormaterUnbindFuncs '( hnlSetOutPutVars
   hnlPrintNetlistHeader                   hnlPrintNetlistFooter
   hnlPrintTopCellHeader          hnlPrintTopCellFooter
   hnlPrintDeviceHeader           hnlPrintDeviceFooter
   hnlPrintPorts                  hnlPrintInst
   hnlPrintSignalHeader        hnlPrintSignalFooter
   hnlPrintMyPort              hnlListMyObject
      )
   )
;
;  Set the variables which control name mapping and output.
;
hnlIfNoProcedure( hnlSetOutputVars()
  prog( ()
     hnlNamePrefix = "hnl"
     simNetNamePrefix = "N"
     simModelNamePrefix = "M"
     simTermNamePrefix  = "T"
     simInstNamePrefix  = "I"
     simLinePrefix  = "+"
     simCommentStr = "#"
```

```
     hnlDriverWillPrint = t
     return( t )
     )
)
;
;  Print one line of comment at the begining of netlist.
;
hnlIfNoProcedure( hnlPrintNetlistHeader()
 let( ( buffer )
       sprintf( buffer "#\n#Netlist for design %s.\n#\n" hnlTopCell ~> cellName )
        hnlPrintString( buffer )
    )
)
;
;  Print one line of comment at the end of netlist.
;
hnlIfNoProcedure( hnlPrintNetlistFooter()
    hnlPrintString( "#\n#End of Netlist.\n#\n" )
)
;
;  Because the IO port definition is taken care off elsewhere, do
;    nothing.
;
hnlIfNoProcedure( hnlPrintPorts()
  t
)
;
;  Print the top cell definition for the generated netlist.
;
hnlIfNoProcedure( hnlPrintTopCellHeader()
 let( ( buffer )
    sprintf( buffer "#\n# Netist for top level Cell.\n#"
                hnlPrintString( buffer )
    if( hnlMultipleCells( hnlCurrentCell ~> cellName ) then
         sprintf( buffer "Design %s ;\n" hnlMapModelName( hnlCurrentCell ~>
                        fileName ))
    else
        sprintf( buffer "Design %s ;\n" hnlMapModelName( hnlCurrentCell ~>
                                                             cellName ))
    )
    hnlPrintString( buffer )
    hnlPrintMyPorts()
    hnlMyListObjects()
  )
)
;
;  Print the cell definition closure for the top cell.
;
hnlIfNoProcedure( hnlPrintTopCellFooter()
  let( ( buffer )
      if( hnlMultipleCells( hnlCurrentCell ~> cellName ) then
        sprintf( buffer "endDesign %s.\n" hnlMapModelName( hnlCurrentCell ~>
                        fileName ))
      else
        sprintf( buffer "endDesign %s.\n" hnlMapModelName( hnlCurrentCell ~>
                              cellName ))
      )
      hnlPrintString( buffer )
  )
```

```
)
;
;  Print the netlisting definition for a macro cell.
;
hnlIfNoProcedure( hnlPrintDeviceHeader()
  let( ( buffer )
      sprintf( buffer "#\n# Netist for macro Cell.\n#" )
      hnlPrintString( buffer )
      if( hnlMultipleCells( hnlCurrentCell ~> cellName ) then
        sprintf( buffer "Design %s ;\n" hnlMapModelName
            ( hnlCurrentCell ~>
                        fileName ))
      else
        sprintf( buffer "Design %s ;\n" hnlMapModelName
            ( hnlCurrentCell ~>
                        cellName ))
      )
      hnlPrintString( buffer )
      hnlPrintMyPorts()
      hnlMyListObjects()
   )
)

;
;  Print the netlist definiton closeure for a macro cell.
;

hnlIfNoProcedure( hnlPrintDeviceFooter()
  let( ( buffer )
     if( hnlMultipleCells( hnlCurrentCell ~> cellName ) then
        sprintf( buffer "endModel %s.\n" hnlMapModelName
            ( hnlCurrentCell ~>
                        fileName ))
     else
        sprintf( buffer "endModel %s.\n" hnlMapModelName
            ( hnlCurrentCell ~>
                        cellName ))
     )
     hnlPrintString( buffer )
   )
)
;
;  Print IO port definition.
;
hnlIfNoProcedure( hnlPrintMyPorts()
  let( ( term termNames thisName count bit buffer signal )
     hnlPrintString( "Ports\n" )
     termNames = hnlAllTermNames
     foreach( term hnlAllTerms
        count = term ~> width - 1
        for( bit 0 count
             thisName = car( termNames )
             termNames = cdr( termNames )
             hnlPrintString( hnlMapTermName( thisName ) )
             if( ( signal = dbGetMemNetSigName( term ~>
                  net bit ) ) == nil
               then  hnlPrintString( "= NC ;\n" )
             else
               sprintf( buffer "= %s ;\n" hnlMapNetName( signal ) )
```

```
              hnlPrintString( buffer )
           )
        )
     )
     hnlPrintString( "endPorts\n" )
  )
)
;
;  Print the instance type definition.
;
hnlIfNoProcedure( hnlMyListObjects()
  let( ( buffr inst allInst allMaster master temp count num cellName )
     hnlPrintString( "Objects\n" )
     allInst = hnlFindAllInstInCell( hnlCurrentCell )
     allMaster = hnlGetMasterCells( allInst )
     temp = allMaster
     foreach( inst allInst
        master = car( temp )
        temp = cdr( temp )
        if( hnlMultipleCells( master ~> cellName ) then
          cellName = hnlMapModelName( master ~> fileName )
        else
          cellName = hnlMapModelName( master ~> cellName )
        )
        count = inst ~> numInst - 1
        for( num 0 count
          sprintf( buffer "%s %s ;\n" hnlMapInstName(
                       dbGetMemName( inst ~> name num ) ) cellName )
          hnlPrintString( buffer )
           )
      )
      hnlPrintString( "endObjects\n" )
   )
)
;
;  Print the signal definition.
;
hnlIfNoProcedure( hnlPrintSignalHeader()
  let( ( buffer )
      sprintf( buffer "# Signal %s.\n" hnlCurrentSignalName )
      hnlPrintString( buffer )
      hnlPrintString( hnlMapNetName( hnlCurrentSignalName ) )
  )
)
;
;  Print the definition closure for a signal.
;
hnlIfNoProcedure( hnlPrintSignalFooter()
   hnlPrintString( ";\n" )
)
;
;  For each inst-terminal connected to hnlCurrentSig, print the
;  the connectivity in the desired format.
;
hnlIfNoProcedure( hnlPrintInst()
  let( ( buffer )
     sprintf( buffer " %s.%s" hnlMapInstName( hnlCurrentInstName )
                          hnlMapTermName( hnlCurrentInstPortName ) )
     hnlPrintString( buffer )
   )
)
```

Sample Output from Formatter Design

The following is sample output from the formatter design example shown in the previous section.

```
#
#Netlist for design design.
#
#
# Netist for macro Cell.
#Design test1 ;
Ports
I0= I0 ;
I1= I1 ;
T0= N1 ;
T2= N3 ;
O= O ;
endPorts
Objects
I1 or2 ;
I0 or2 ;
I2 xor2 ;
endObjects
# Signal I1.Y.
N4 I1.Y I2.B;
# Signal D<0>.
N1 I1.A;
# Signal I0.Y.
N5 I0.Y I2.A;
# Signal I1.
I1 I0.B;
# Signal I0.
I0 I0.A;
# Signal D<1>.
N3 I1.B;
# Signal O.
O I2.Y;
endModel test1.
#
# Netist for macro Cell.
#Design test2 ;
Ports
T6= N7 ;
T8= N9 ;
T0= N1 ;
T2= N3 ;
OUT= OUT ;
endPorts
Objects
I0 test1 ;
endObjects
# Signal D<0>.
N1 I0.T0;
# Signal I<0>.
N7 I0.I0;
# Signal D<1>.
N3 I0.T2;
# Signal OUT.
OUT I0.O;
# Signal I<1>.
```

```
N9 I0.I1;
endModel test2.
#
# Netist for macro Cell.
#Design test ;
Ports
O1= O1 ;
I1= I1 ;
I2= I2 ;
I3= I3 ;
I0= I0 ;
endPorts
Objects
I4 and2 ;
I5 and2 ;
I0 xor2 ;
endObjects
# Signal I4.Y.
N10 I4.Y I0.A;
# Signal O1.
O1 I0.Y;
# Signal I3.
I3 I5.B;
# Signal I2.
I2 I5.A;
# Signal I1.
I1 I4.B;
# Signal I0.
I0 I4.A;
# Signal I5.Y.
N11 I5.Y I0.B;
endModel test.
#
# Netist for top level Cell.
#Design design ;
Ports
T12= N13 ;
T14= N15 ;
T= T ;
N= N ;
P= P ;
D= D ;
M= M ;
J= J ;
JUNK= JUNK ;
T16= N17 ;
T18= N19 ;
T20= N21 ;
T22= N23 ;
T24= N25 ;
T26= N27 ;
T28= N29 ;
T30= N31 ;
endPorts
Objects
I2 or2 ;
R10 res ;
I11 buffer ;
I12 inv ;
I15 cmos ;
N19 nmos ;
```

```
P20 pmos ;
I3 and2 ;
I0 test ;
I32 test2 ;
I1 test1 ;
endObjects
# Signal A<3>.
N23 I1.I0 I0.I3;
# Signal D.
D I15.D N19.D;
# Signal R10.Y.
N32 R10.Y I12.A;
# Signal A<1>.
N19 I1.T0 I0.I1;
# Signal N19.S.
N33 P20.D N19.S;
# Signal DATA<2>.
N29 I32.T0;
# Signal I0.O1.
N34 I0.O1 I3.A I2.A;
# Signal IN<3>.
N25 I32.T6;
# Signal R<1>.
N15 I3.Y;
# Signal I1.O.
N35 I1.O I3.B I2.B;
# Signal A<2>.
N21 I1.I1 I0.I2;
# Signal T.
T I12.Y;
# Signal I11.Y.
N36 I11.Y R10.A;
# Signal JUNK.
JUNK I32.OUT;
# Signal A<0>.
N17 I1.T2 I0.I0;
# Signal P.
P I15.GP;
# Signal N.
N I15.GN N19.G;
# Signal DATA<3>.
N31 I32.T2;
# Signal M.
M P20.S;
# Signal J.
J P20.G I15.S I11.A;
# Signal IN<4>.
N27 I32.T8;
# Signal R<2>.
N13 I2.Y;
endDesign design.
#
#End of Netlist.
```

7

Customizing the Flat Netlister (FNL)

Most simulators and design analysis tools require a textual description of the design to be analyzed as input. Some of these tools require this description to be completely flat. This type of a network description is called a “flat netlist.” A flat netlist contains a complete description of the devices used in the design, their delays, and the connectivity between these devices. In addition, the flat netlist may contain model descriptions, which are ways of setting the parameters of a simulator primitive view for a device (for example, the saturation current *IS* and ohmic resistance *RS* of a diode in SPICE). A flat netlist must not contain any hierarchy. This requirement means that descriptions for a cell used in the design and references to the cell later in the netlist are not output. Instead, the contents of each cell are output every time the cell is referenced.

The Flat Netlister (FNL) is a Cadence tool to produce flat netlists. FNL flattens the design and presents the data in a simplified form for use by output formatting instructions. You can write these instructions to produce a new netlist syntax suitable as input to your simulator. Output formatting can be performed using the Cadence standard language, SKILL, the netlister's own substitution expressions, or a combination of both. FNL substitution expressions are a compact and simple language which can be used to format the netlister output without any understanding of the Cadence database structure. It can be used for most of your formatting needs. For more complex formats, it may be necessary to use SKILL instructions to produce some of the output. The full capability of SKILL is available for use in writing your format functions. In addition, several functions and variables specific to netlisting have been added. These functions give you complete access to the design database, as well as to the internal structure of the netlister. To use some of these functions effectively, you need to understand the basic structure of the Cadence database as described in the Virtuoso Design Environment SKILL Reference.

This chapter describes how FNL works, how the design is flattened, how and why names are generated, and how the output is formatted. Included in the explanation of output formatting is a description of the substitution expression syntax and the SKILL functions specific to the netlister which you can use to simplify customizing the FNL output. After netlister functionality is explained, the steps you must perform to modify the netlister output for a new syntax are explained. There is also a description of recommended library structure pertaining to netlisting and the modifications that need to be made to the Simulation Environment (SE) if you are netlisting for a new simulator. Examples of library elements from the SPICE library and the formatting instructions used to output the connectivity for these devices to the netlist file are at the end of this chapter.

Note: FNL always outputs unit connectivity. Hence, the data presented to the formatter is usually on a signal as opposed to a net basis. Some FNL variable names still contain the string `Net` within them. Read the descriptions of these variables carefully. As the descriptions state, they probably return data about signals and not about nets.

How FNL Works

FNL consists of two parts: the first part contains the output formatting instructions that you must provide; the second part contains the database traversal routines provided by Cadence.

Formatting Instructions

When netlisting begins, a cell, usually called `nlpglobals`, is read, and the output formatter is initialized. Along with preparing internal data structures, the primary formatting properties that can be defined only in this cellview are searched for, and if found, they are precompiled for later formatting of each device and net. Which formatting properties are found determines whether the netlist contains certain types of data. The primary formatting properties are shown here:

```
NLPcompleteElementString
NLPcreateModelString
NLPcreateNetString
NLPnetlistHeader
NLPnetlistFooter
NLPLineLength
NLPLinePrefix
NLPLinePostfix
NLPsingleLineCommentString
```

These are the only predetermined property names that the netlister searches for to format the netlist, and the global cellview is the only place searched for their values. (Refer to the "Global Cellview (`nlpglobals`) Contents" section in this chapter for a description of these properties.) If any other property value is placed in the netlist, one of these properties must specify the necessary search to find that property value and the way it should be formatted in the netlist.

Database Traversal Routines

Next, the design hierarchy is read into memory, and an internal view of the design is built that can be traversed as if the design had been flattened. Flattening is the process of replacing each instance, or reference to a device, with the contents of that device. This process is explained in detail in the “FNL Flattening Process” section in this chapter.

As the design is flattened, the names of all of the instances and nets must be expanded to ensure that each device is still uniquely identifiable by its name within the design. This is done by preceding the name of each instance (or net) with the name of the instance that contains it and separating the names with a slash (“/”). For example, if you place two inverters in your design, one called “inv1” and the other called “inv2,” and each contains a transistor called “trans1,” the flattened design contains two instances of transistor “trans1,” one with the name “/inv1/trans1,” and the other with the name “/inv2/trans1.” In a large hierarchical design, these names can quickly become too large for most simulators to accept. Therefore, as the design is flattened, each instance and net is assigned another name acceptable to the target simulator. This name is created by taking a prefix, which you can specify, and suffixing it with a unique number. This name relation is stored in a name map which the Cadence system can read and use to translate between the two names. Your users never need to see the name that was created for the restrictive name space of a simulator.

During this process certain consistency checks (such as terminal mismatches between levels of hierarchy) are performed. If there are no errors in this process, the output formatting begins. The flattened design data, which is now in core, is traversed again. As each device and net is encountered, one of the properties which must be defined in the global cellview is evaluated. (Refer to the “Formatting Instructions” section in this chapter.) This property must be either a substitution expression () or a SKILL function (`ilExpr`). If the property is a substitution expression, it specifies for the netlister how to output the connectivity for the current device or net. If the property is a SKILL function, it is evaluated, and the SKILL command either outputs any needed information to the netlist file directly or returns a value which the netlister outputs to the netlist file. The use and syntax of substitution expressions is explained in the “Formatting Substitution Expressions” section in this chapter. Netlist formatting using SKILL, including the extra functions and variables that have been added to simplify this process, are explained in the “SKILL Formatting” section in this chapter.

FNL Naming Conventions

Properties with predetermined names that the netlister searches for are prefixed with the letters *NLP* and are searched for only in the global cellview. (Refer to the “Global Cellview (`nlpglobals`) Contents” section in this chapter.) To avoid any confusion concerning which functions refer to these properties, as well as any property name conflicts, do not prefix your property names with the letters *NLP*. If you are using an existing simulation interface supported by Cadence, then netlisting format properties can also be found on the simulation primitive for each device.

All of the SKILL functions and variables defined in the netlister begin with the letters `fnl`. The name is lower case, except for the first letter of each word, which is upper case. To avoid variable or function name conflicts, do not prefix your function or variable names with the letters `fnl`. Predefined netlister functions and variables are explained in the “SKILL Formatting” section in this chapter.

FNL Flattening Process

The Cadence netlister “flattens” a design hierarchy and produces an expanded description of the design chosen for simulation. The way the design hierarchy is “expanded” to produce the netlist as well as the syntax of the netlist depend on the simulator used. For example, when generating a network description for input to the SILOS simulator, you may want the description to be at the logical gate level, since SILOS is capable of simulating such primitives as AND gates and AOIs. If you want to run a SPICE simulation on the same design, the network description must be expanded down to the transistor level, since SPICE has no understanding of logic gates.

Schematics usually consist of instances of symbols connected by nets. When expanding a schematic for input to a simulator, the symbol instances must be associated with their corresponding schematics or netlisting views so that the target simulator can understand the components in the design.

Locating an Instance of a Device

First, each instance (occurrence) in the schematic (represented by the symbol instance) must be located. The `cds.lib` file contains the path list that your design library, which contains the top-level schematic of the design, and reference libraries are located in. Each instance of the master (symbol) could be located in either the design library or the reference library. If the instance of the master is from one of the reference libraries, the instance maintains the referenced library's name. Therefore, it is important to put the same library path and library names in the `cds.lib` file. This maintains consistency if you run your simulation from within the Cadence-provided environment. When the netlister finds an instance of a device in a schematic, it tells the design manager to locate the library containing the device. The library is specified in the `cds.lib` file. Once the netlister locates the device in the specified library, it continues the design hierarchy expansion process. If the netlister does not locate the device, it generates an error message because it cannot create a netlist.

Switching Views

Once the device referred to in the schematic has been located, the symbol instance for the device needs to be associated with its corresponding schematic or simulator primitive. This process is called *switching* views. The list of valid views to be used for *switching* is defined by the SE `simViewList` variable. Each view in this list, along with the device's name, comprise an identification that is searched for in the specified library. If no identification (cell's name and view's name) in the simViewList is found in the specified library, an error message is generated, and no netlist can be produced.

Stopping Views

A view that is the most detailed description desired for simulation is called a *stopping* view. The list of valid stopping views is defined by the SE `simStopList` variable. If the view located in the previous switching views step corresponds to one specified in the `simStopList`, the expansion process for this instance is stopped, and the connectivity information for this instance is printed to the netlist file. If the view does not correspond to a stopping point, the expansion process continues, and you return to the step of locating all of the instances contained in this cellview.

The following are the SE variables that control the expansion process:

simLibName

The library name containing the top schematic of the design.

simViewList

The "viewing switch list." Each of the views in this list is searched for in the order it is listed in the library. The first view found is switched in place of the symbol.

simStopList

The "stopping point view list." The view found in the `simViewList` is checked to see if it is in this list. If it is, expansion stops. If it is not, the expansion process is applied to the new view.

Note: `simViewList` and `simStopList` are the variables used by SE and the netlister internally. Users cannot set the variable directly. The `simViewList` and `simStopList` variables can be set only on a per-simulator basis. To override the default view list or stop list, users set the corresponding list for the simulator they are using in their `.simrc` file, and the appropriate internal list is set to the same value by each simulation interface. The "Modify the Simulation Environment (SE)" section in this chapter explains how to set these lists.

The following are examples of the variables controlling expansion:

```
simViewList = '("spice" "schematic")
simStopList = '("spice")
```

Stop on a cell with a view named "spice." Otherwise, expand to the cell with view name "schematic."

```
simViewList = '("spice" "cmos.sch" "schematic")
simStopList = '("spice")
```

Stop if the cell's view name is “spice”. Otherwise, expand the view “cmos.sch” or “schematic”.

The concise algorithm for turning a design hierarchy into a netlist is explained next.

Opening a Design

Open the design (`simCellName simViewName`) in the `simLibName` library.

Locating a Cellview

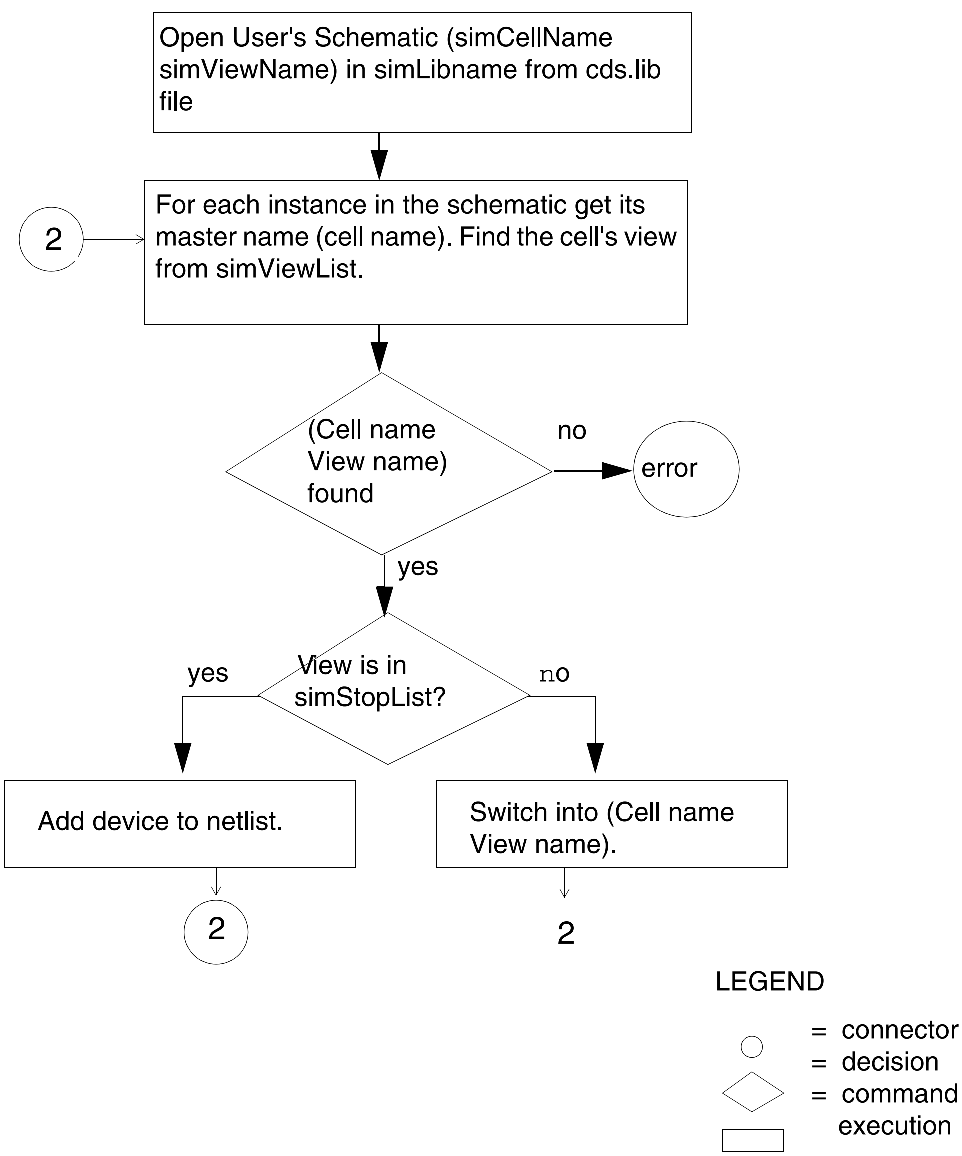
For each instance in the current schematic, get the cell name (master name) of the instance, then take each of the names in turn from the list of views (`simViewList`), and look for the specified cellview in the available reference libraries.

Expand or Format

If you find a cellview with a view name from the `simViewList`, check to see if the view name is also in the `simStopList`. If the view name is a stopping point, write out the instance in the netlist file. If you do not find a view name from the `simViewList`, generate an error message. If the view name does not appear in the `simStopList`, repeat the *Locating Cell View* step to expand the cellview.

Figure 7-1 on page 250 is a diagram showing the algorithm used to expand the simulation master into a netlist.

Fig 7-1Netlist Algorithm To Expand the Simulation Master



Global Nets

During the flattening process, if two nets in the same or different cellviews of the design hierarchy are called `netName` and are marked as global, then both nets are assigned the same name by the netlister. Because simulators make connections by name, both nets are perceived as electrically equivalent by the simulator.

Support of Multiplicity Factors for Flat Netlisters

Multiplicity factor means that for a given instance on which this property is specified, the instance will be treated as an array of instances with the upper limit being the *number* specified by this property. This is also referred to as *m-factor* and is denoted by an integer value.

This feature was introduced for customers to avoid keeping multiple copies of the same design for simulation and verification purposes since FNL till now didn't support this property. The support of this feature will enable users to specify the property on the instance with the SKILL variable `useMfactorToIterateInstances` set to `t` and the instances will be connected in parallel. The user can specify the following variable in the `.simrc` file.

SKILL Variable: `useMfactorToIterateInstances`

Type: `Boolean`

Valid Value: `t/nil`

This variable when set to `true (t)` will direct OSS to handle the instances which have the *m-factor* set to some integer value as iterated instances and thus the instance will appear in the flattened netlist, number of times as specified by the value of the *m-factor*. If this variable is set to `false (nil)`, OSS will continue its default behavior and ignore the *m-factor* properties set on the instances.

For simulation purposes, there would not be any change needed because this variable will not impact the default behavior of the hierarchical netlisters.

FNL Name Map

As explained in the “Database Traversal Routines” section in this chapter, the netlister automatically generates new names for each instance and net to avoid problems with names not being acceptable in a simulator input syntax. These names are generated by suffixing a prefix with a unique number. You can specify the netlister prefixes by setting variables in SE as explained in the “Modify the Simulation Environment (SE)” section in this chapter. The netlister automatically stores the cross-references between these two sets of names in a “map” file. You do not need to add the names to this map. If you need to know either name for a signal or instance, the functions and variables to retrieve this information are listed in the “Formatting Substitution Expressions” and “SKILL Formatting” sections in this chapter.

FNL Output Formatting

FNL automatically flattens the design and produces a name map, but it does not output any connectivity until it has been instructed on how to format the netlist. There is no default netlist syntax. The netlist syntax is determined by the property values with fixed names stored in a database cellview called the global cellview, or frequently called `nlpglobals`. To produce a netlist with the syntax required by your simulator, you must first create this global cellview.

In this global cellview, you add several properties to instruct the netlister how to format various aspects of the netlist. These instructions usually refer to other properties defined on library elements (the primitive views for your simulator) or SKILL procedures that format the output for each device type. You can also add global default format instructions to the global cellview property list. The format instructions on primitive library elements can then refer to, and share, these instructions.

The following sections explain the global cellview, the predefined properties you must create, substitution expression syntax, and SKILL formatting instructions.

Global Cellview (nlpglobals) Contents

The global cellview is basically a schematic. It must contain one instance whose master is (dummy dummy). (This is explained in the “Create Global Cellview” section in this chapter.) It must also contain the netlist formatting instructions. In addition, netlist formatting properties applicable to the entire design can also be stored in the global cellview. This global cellview is a way of storing common format instructions in a single place. For example, there is a property in the global cellview that specifies how each instance should be formatted. This property usually refers to a property on the primitive for each element found in the library; the property contains details on how this device type should be formatted. Formatting instructions that are common to many types of gates, such as formatting the delay characteristics of a logic gate, can also be stored in the global cellview and then referred to by all primitives for logic gates found in your library.

Creating Predetermined Properties

The following list shows all the predetermined property names the netlister searches for to determine the netlist format.

```
NLPcompleteElementString
NLPcreateModelString
NLPcreateNetString
NLPnetlistHeader
NLPnetlistFooter
NLPLineLength
NLPLinePrefix
```

```
NLPLinePostfix
NLPsingleLineCommentString
```

Note: These properties can be placed only in the global cellview, and these are the only properties the netlister looks for to determine the netlist syntax. If you want to place any other properties in the netlist, the value of these properties must specify how to locate, as well as how to format the property in the netlist. An example is included in the description of each property. Where pertinent, the examples are taken from the global cellview for SPICE. The example syntax is the same as that used for adding a property to a cellview using the Symbol/ Simulation Library Generator (S/SLG). The detailed syntax of the substitution expressions (`nlpExpr`), and the SKILL expressions (`ilExpr`) are explained in the two following sections.

NLPcompleteElement-String

This property is the most important one in the global cellview because its value instructs the netlister how to format each instance in the design being netlisted. Since FNL is instance-based (as opposed to net-based), this format instruction outputs all of the connectivity for the design. There is no default for this property. If you do not define its value in the global cellview, no connectivity information is placed in the netlist. The property type must be either `ilExpr` or `nlpExpr`.

This property value usually refers only to another property placed in the simulation primitive for each device in the library. In the following examples, the property value instructs the netlister to search for another property, which is called `NLPElementPostamble` in this example (although it can be any string) and uses the value to format the reference to this element. In the Cadence library, this property is located in each primitive.

When this property is evaluated, your formatting instruction has access to two types of property searches: you can specify either a local property search or a search throughout the design hierarchy. The method for specifying the type of property search is explained in the "Formatting Substitution Expressions" and "SKILL Formatting" sections in this chapter. A local property search looks for a property of the specified name only on the primitive instance currently being output to the netlist. A property search throughout the design hierarchy first looks for the property on the primitive instance currently being output to the netlist, then looks on the instance that contains it, and so on up the instance hierarchy. If it does not find the property on any of the instances, it looks for the property on the view switched master of the stopping instance. The stopping instance is the simulator primitive for the device found in the library. If it still does not find the property, it searches the global cellview for a property of this name.

The property search throughout the design hierarchy allows you flexibility in setting defaults, as well as the possibility of overriding them. You can place the default value of a property in the cellview for each library element type and make it specific for that device type, or you can place a default in the global cellview and make it a default for the entire design. You can override the default for all devices below a particular level of hierarchy or on only one instance.

Note: This property search mechanism is different than that used by the hierarchical netlister. Because most simulators cannot inherit properties through the design hierarchy entered using a hierarchical netlist, properties in the hierarchical netlister can be accessed only on the instance of the primitive and its master. This condition can produce different simulation results when the same design is simulated using a hierarchical and a flat netlist.

In addition to properties stored in the design, certain netlister-defined properties are also accessible; these properties are defined in the “Formatting Substitution Expressions” and “SKILL Formatting” sections in this chapter.

Example:

```
NLPcompleteElementString = nlpExpr("[@ NLPElementPostamble]\\n")
```

or:

```
NLPcompleteElementString = nlpExpr("[@ NLPElementPostamble:%: **No element format
property found for element [@ InstPathName]]\\n")
```

or:

```
NLPcompleteElementString = ilExpr("SpiceFormatElement()")
```

NLPcreateModelString

The value of this property instructs the netlister how to format the model description for each device type in the design being netlisted.

This format instruction is evaluated only the first time each device type is encountered. For example, if your design includes two diodes and you defined this property in your global cellview, the property is evaluated only for the first diode encountered. The `NLPcompleteElementString` property is evaluated for each diode instance.

There is no default for this property, and you do not need to define it if your simulator does not require model descriptions. Model descriptions are required by circuit simulators such as SPICE, but not by most logic simulators. If you define this property in your global cell, its type must be either `ilExpr` or `nlpExpr`.

The property value usually only refers to another property placed in the simulation primitive for each library device. In the following example, the property value instructs the netlister to search for the `NLPModelPreamble` property and uses the value to format the model description for this element type. In the Cadence library, this property is located in each primitive.

When this property is evaluated, your formatting instructions have access to properties stored in the simulation primitive corresponding to the device being output to the netlist and to properties stored in the global cellview. Since only one model statement is output irrespective of how many instances refer to it, it cannot output information stored on any of the device instances. In addition to properties stored in the design, certain netlister-defined properties are also accessible. These properties are defined in the “Formatting Substitution Expressions” and “SKILL Formatting” sections in this chapter.

Example:

```
NLPcreateModelString = nlpExpr("[@ NLPModelPreamble:%\\n]")
```

or:

```
NLPcreateModelString = ilExpr("SpiceFormatModel()")
```

NLPcreateNetString

The value of this property instructs the netlister how to format each signal description in the design being netlisted. Signal descriptions are not required by most simulators that use an instance-based netlist as input (that is, HILO or SPICE). However, signal descriptions can be used as comments in the netlist file to put a cross-reference between user-assigned node names and netlister-assigned node names. The second example is an example of this use.

There is no default for this property, and you do not need to define it if your simulator does not require signal descriptions. If you do define this property in your global cellview, its type must be either `ilExpr` or `nlpExpr`.

Example:

```
NLPcreateNetString = nlpExpr("[@ NLPNetFormat:%\\n]")
```

or:

```
NLPcreateNetString = nlpExpr("$  [@ NodeNumber] = [@
   NetPathName]\\n")
```

NLPnetlistHeader

The value of this property instructs the netlister how to format any header information to be placed in the netlist. This property is evaluated before any other information is printed to the netlist file.

There is no default for this property, and you do not need to define it if your simulator does not require any netlist header information. If you do define this property in your global cellview, its type must be either `ilExpr` or `nlpExpr`.

Example:

```
NLPnetlistHeader = ilExpr("MysimFormatHeader()")
```

NLPnetlistFooter

The value of this property instructs the netlister how to format any footer information to be placed in the netlist. This property is evaluated last after all other information is printed to the netlist file and can be used to place any closing information needed in the netlist file. You can also use it to clean up after netlisting. For example, you can call a SKILL function from this property to close any files you opened during the netlist process.

There is no default for this property, and you do not need to define it if your simulator does not require any netlist footer information. If you do define this property in your global cellview, its type must be either `ilExpr` or `nlpExpr`.

Example:

```
NLPnetlistFooter = ilExpr("MysimFormatFooter()")
```

NLPLineLength

FNL automatically folds (or wraps) lines when the line length in the netlist exceeds a specified maximum. The default maximum length is 72 characters. If the line length exceeds this maximum, the netlister searches backwards in the text to be output and inserts a new line at the first blank it encounters. The remaining text is put on the next line (or subsequent lines if the text is still too long to fit on the next line).

To change the default maximum line length, set the `NLPLineLength` property in the global cellview you are creating to an integer value.

Example:

```
NLPLineLength = 65
```

NLPLinePrefix

The netlister can automatically continue lines that are too long. If your simulator requires it, the netlister can add any line continuation character at the beginning of the next line. To put a continuation character at the beginning of the next line being continued, set the `NLPLinePrefix` property in your global cellview to any string value. This property has no default value and is not required by the netlister.

Example:

```
NLPLinePrefix = "+"
```

NLPLinePostfix

The netlister can automatically continue lines that are too long. If your simulator requires it, the netlister can add any line continuation character at the end of the current line. To put a continuation character at the end of the line requiring continuation, set the `NLPLinePostfix` property in your global cellview to any string value. This property has no default value and is not required by the netlister.

Example:

```
NLPLinePostfix = "+"
```

NLPsingleLine-CommentString

In addition to automatically post fixing and prefixing lines that the netlister folds, the netlister can distinguish between comments and other netlist lines. Some simulators do not allow continuation of comments in the netlist. When you put a comment string in the netlist, the netlister folds the line as usual, but instead of inserting a continuation character at the beginning of the new line, it inserts a comment character, converting a single comment which exceeds the simulator internal limit to two or more single-line comments. The `NLPsinglLineCommentString` property in the global cellview specifies the comment character used by your simulator. The netlister uses this character both to detect if the current output line is a comment and to insert the character at the beginning of a new line if a comment line needs to be broken into more than one line.

There is no default for this property, and you do not need to define it if your simulator allows continuation characters to be used in comments. If you do define this property in your global cellview, it must be a string value.

Example:

```
NLPsingleLineCommentString = "*"
```

Formatting Substitution Expressions

There are two ways to format the FNL output: You can use either the Cadence standard language, SKILL, or the netlister substitution language. These two languages are completely compatible and can be intermixed. Since netlist formatting is done by evaluating properties, one property could be a substitution expression and could refer to another property which is a call to a SKILL function. This section describes only the netlister substitution language. For details on SKILL formatting, refer to the “SKILL Formatting” section in this chapter.

The property type signals how the netlister should interpret it. If the property type is `nlpExpr`, the netlister interprets it as a substitution expression. All text in a substitution expression is copied verbatim to the netlist unless it is contained within square brackets ([ ]). The opening square bracket ([ ) signals that the following string must be interpreted and substituted or replaced with another value. The character following the opening square bracket is a command to the netlister. The following is a complete list of the command characters that the netlister understands:

- vertical bar (|)
- “at” sign (`@`)
- period (`.`)
- asterisk (`*`)
- dollar sign (`$`)
- pound sign (`#`)
- backquote (`` ` ``)
- up arrow (↑)
- ampersand (`&`)

Refer to the following explanations for details on the use of each character.

The netlister interprets the remainder of the string within the square brackets, as specified by the command character. The netlister then replaces the entire bracketed expression, including the square brackets and command character, with the evaluation of the expression.

The following sections explain the various commands available in a substitution expression, enhanced substitution expression syntax, and special netlister-defined properties that you can access by means of substitution expressions.

Terminal Name Substitution

A vertical bar (|) following an opening square bracket signals the netlister that the following string is a terminal name. Thus, the netlister replaces the entire bracketed expression with the netlister-assigned signal name attached to the named terminal of the current instance being output to the netlist. For example, the following format string defines a two-input NAND gate, in the syntax required by the SILOS simulator, with an output pin called `Q` and inputs called `A` and `B`.

```
[ |Q ] .NAND [ |A]  [ |B]
```

When the netlister processes this format string, the substitution expressions are evaluated and a line like the following is added to the netlist:

```
N23   .NAND   N12   N14
```

The names substituted are the letter `N` followed by a number. The netlister generates names by adding a unique number to a user-defined prefix. For more information on how to set these prefixes, refer to the “Modify the Simulation Environment (SE)” section in this chapter.

Property Substitution

An “at” sign (@) or a period (.) following an opening square bracket signals the netlister that the following string is the property name. Thus, the netlister replaces the entire bracketed expression with the property value if it is found. If the property is not found, nothing is output to the netlist. The difference between the “at” sign (@) and the period (.) is where the netlister searches for the property value. If the period (.) is used, the netlister searches only for the property on the instance currently being output to the netlist. If the “at” sign (@) is used, the netlister searches throughout the design hierarchy for the property.

First, the netlister searches on the instance currently being output to the netlist, then it searches on the instance of the cell that contains this instance, and so on up the instance hierarchy. Next, it searches for the property on the current instance master. The master is usually the simulation primitive for this device in the library (that is, the `nand2` `silos` cellview). The final place it searches is the global cellview for your simulator (that is, the `nlpglobals` `silos` view). The property search is halted and the value output to the netlist when a property of the specified name is found.

Note: This property search differs from that used by HNL. The scope of property searches in HNL is limited to the current instance and its master.

The sample format string is expanded by adding a substitution expression for the gate strength and its rise and fall delay times:

```
[ |Q ]  .NAND[.strg]  [ @ tr ]  [@ tf ]  [ |A]  [ |B ]
```

Next, assume that the `strg` property with value “`/C`” exists on the instance of the NAND gate and that the `tr` and `tf` properties have values 5 and 8, respectively. When the netlister processes this format string, the substitution expressions are evaluated and a line like the following is added to the netlist:

```
N23   .NAND/C   5   8   N12   N14
```

You can define the `strg` property only on the instance of the NAND gate, whereas the delay properties can be defined either there or on the cell instance that contained the gate. If you did not add any of these properties in your design, the following line would have been added to the netlist:

```
N23   .NAND   N12   N14
```

Using these property searches, you can parameterize the output of the netlister and allow users to specify parameters and delays to be output to the netlist from within their design.

Netlister-Defined Properties

To simplify netlist formatting, the netlister defines “properties” to provide information needed in most netlists. These properties have reserved names and you cannot set them. If a property of the same name exists in the design, it is ignored, and the netlister-defined value is returned. These properties do not exist in the design hierarchy, and they can be accessed only by querying the netlister. You can output the value of any of these properties in the same manner that user-defined properties are output. For example, to output the netlister-assigned instance name for the device currently being output to the netlist, your substitution expression format string can contain the following expression:

```
[@ElementNumber ]
```

The following is the complete list of netlister-defined properties:

ElementNumber

Represents the netlister-assigned name for the current iteration of the current instance being output to the netlist. It is defined only during processing of the `NLPcompleteElementString` property.

InstPathName

Represents the full user-assigned instance path name of the instance currently being output to the netlist. It is defined only during processing of the `NLPcompleteElementString` property.

NodeNumber

Represents the netlister-assigned name for the signal currently being output to the netlist. It is defined only during processing of the `NLPcreateNetString` property.

NetPathName

Represents the full user-assigned signal pathname of the signal currently being output to the netlist. It is defined only during processing of the `NLPcreateNetString` property. If a net has more than one bit, then the netlister outputs each member of the net separately.

BlockName

Represents the master cellname of the instance currently being output to the netlist. It is defined only during processing of the `NLPcompleteElementString` and `NLPcreateModelString` properties.

ModelNumber

Represents the netlister-assigned model name of the model referred to either by the instance currently being output to the netlist or by the model description currently being output. It is defined only during processing of the `NLPcompleteElementString` and `NLPcreateModelString` properties.

Net and Instance Name Substitution

The pound sign (#) and dollar sign ($) following an opening square bracket ([ ) are used as substitutions for netlister-assigned signal and instance names of entities other than the current one being output to the netlist. This situation is required, for example, in processing a SPICE mutual inductor, where the mutual inductance between two instances is expressed. The format string to generate the mutual inductance between the current instance and an instance called `inductor2` is shown here:

```
K [@ ElementNumber]  L [@ ElementNumber]  L [$inductor2]  [@ value]
```

This string is defined as a property on the current instance, along with the property `value`. [$inductor2] means substitute the netlister-generated instance name for the instance with the user-assigned name `inductor2`.

Suppose you have a three-terminal symbol for an *NMOS* enhancement transistor with its bulk node connected to ground. The beginning of the format string for this symbol might look like:

```
M [@ ElementNumber] [ |D] [ |G] [ |S] [ #gnd! ]
```

where [`#gnd!`] means to substitute the netlister-generated signal name of the global signal `gnd!`.

The scope of the “#” and “$” substitutions is restricted to the cellview that the netlister was traversing when it encountered the current instance or signal being output to the netlist.

Format Specification

String substitutions can optionally contain a format specification that tells the netlister how to print the value of a property. The format follows the property name to be substituted and is preceded by a colon (:). A percent sign (%) in the format string indicates where to insert the property value. All other text in the format string is copied verbatim to the netlist file. The format is used only if the property exists. If the property is not found, no text is output to the netlist file.

This formatting specification allows you to generate keywords along with property values. For example, for a SPICE transistor with a channel length of 3 microns and width of 12 microns, the format string:

```
M [@ ElementNumber] [ |D ] [|G ] [ |S ] [ |B] [@ W:W=%] [@ L:L=%]
```

produces the following line in the netlist file if the properties *W* and *L* exist in the instance hierarchy:

```
M1  1  2  3  0  W=12  L=3
```

However, if the values of `W` and `L` are undefined, the result is:

```
M1  1  2  3  0
```

A format string can also contain a second optional format specification. The second format specification is also preceded by a colon (:) and can follow only the first format specification. The second format string is used if the property being searched for is not found. In this case, the meaning of

```
[@ property:format1:format2 ]
```

is the following:

if the property exists then

use format1

else

use format2

endif

This powerful tool can be used to provide default values and to indicate error conditions. This example provides defaults for property values:

```
[ |Y ]  .INV [@ tr:tr=%:tr=1] [@ tf:tf=%:tf=1] [ |A ]
```

If the property called `tr` exists, then the output shows the string `tr=` followed by its value. If the property does not exist, the output shows the string `tr=1`. If the property `tr` is defined on an inverter instance and has a value of 7 but the property `tf` is not defined, then the substitution expression example results in this entry in the netlist file:

```
N23  .INV  tr=7  tf=1  N32
```

You can also nest substitution expressions. You can specify that a property is searched for only if another property exists. Nesting of format strings occurs in two ways, direct and indirect.

In the direct case, a string substitution function appears inside another string substitution function, as shown in this example:

```
 [@ prop1: PROP1 = % [@ prop2] ]
```

This substitution expression instructs the netlister to search for a property called `prop1`. If it is found, the output shows the string `PROP1=` followed by the property value. Then, the netlister searches for a property called `prop2`, and if found, the output shows its value. If the `PROP1` property is not found, nothing is output to the netlist, and the netlister does not search for the `prop2` property.

In the indirect case, a format string contains a property substitution that has a format string as its value. Consider the following four properties and their values:

```
tr = 2
tf = 3
SilosDelayTimes = [@ tr]  [@ tf]
SilosFormatString =  [|Y]  .NAND [@ SilosDelayTimes] [ |A ] [ |B ]
```

This type of substitution expression is one way of producing a format string for a NAND gate in SILOS syntax. If the netlister evaluates this formatting instruction when it encounters a NAND gate in a schematic, the following entry is output in the netlist file:

```
N34  .NAND  2  3  N12  N5
```

This indirect substitution expression is basically how the libraries provided with a Cadence simulation interface are structured. The required netlister property `NLPcompleteElementString` is defined in the global cell SILOS view to refer to a property such as `SilosFormatString`. This can be done by setting the `NLPcompleteElementString` property type to `nlpExpr` and its value to "[@ SilosFormatString]." The `SilosFormatString` property is then defined in the SILOS view of the NAND2 gate found in the library to be of type `nlpExpr` and to have a value as in the example. The `SilosDelayTimes` property is then defined in the global cell SILOS view as well. By placing it in the global cell, it can be shared by many different library elements. In this example, there is little to be gained by doing so, but the real formatting instructions for each library element are more complex. Placing shared properties in a single location not only reduces the size of your library, but it also reduces the number of devices that need to be fixed if an error is detected in formatting delay values.

Note: When using nested formatting instructions, the search for each property begins on the current instance being output to the netlist file. Although the `SilosDelayTimes` property in the example was defined in the global cellview, when the property substitution expression value is interpreted and the netlister is instructed to search for the property *tr*, it begins at the instance currently being output to the netlist, then searches on the instance of the cell that contains the instance of the NAND gate, and so on. The property search using the "at" sign (@) command character is the same, irrespective of where the property is located.

Property Scaling

Scaling enables you to use property values from schematics or simulation views for target programs requiring values in different units. For example, TA (the Cadence Timing Analyzer program) and the SILOS simulator both use delay values, but they have different units of time. TA expects time values in seconds, whereas SILOS uses `SILOS time units`. (A `SILOS time unit` may be 1 nanosecond, 1 picosecond, or any unit which is convenient.)

Without a scaling capability, you have to enter `n` sets of time-valued properties, where `n` is the number of different time scales. This situation is inefficient and leads to inconsistencies between the data used by different target programs. Time-valued properties should be given values in seconds. Then, scaling can be applied in the SILOS netlister formatting strings so that values are converted to `SILOS time units` before they are written into a SILOS netlist. If all time values are in seconds, then no scaling is required for TA.

The netlister scales are stored in SE variables. Scaling is performed by dividing property values by the value of the appropriate variable. For time scaling, the variable is called `simTimeUnit`. This variable has a default value of 1E-9, giving a `SILOS time unit` of 1 nanosecond. (Refer to the “Customizing the Simulation Environment (SE)” chapter in this manual or the *Design Analysis User Guide* for more information on setting defaults in SE.)

A property to be output by the netlister can be scaled by placing a command character that specifies the type of scaling to be performed before the command character specifying the property search to be used. The following is a description of the scaling factors available and examples of their use. If these scaling factors are not sufficient for your needs, use SKILL formatting instructions to provide any scaling required by your simulator.

Asterisk

The asterisk scaling character (*) instructs the netlister to take the property value if found and divide its value by the value of the SE `simTimeUnit` variable. The default value of the `simTimeUnit` variable is 1E-9. The result of this division is then rounded to the nearest integer and output to the netlist. The following is an example of a substitution expression using this scaling:

```
[*@ tr:% ]
```

Backquote

The backquote scaling character (‘) instructs the netlister to take the property value if found, multiply it by the value of the SE `simCapUnit` variable, and then divide the result by the value of the SE `simTimeUnit` variable (that is, `mr*simCapUnit / simTimeUnit`). The default value of the `simTimeUnit` variable is 1E-9 and of the `simCapUnit` variable is 1E-15. The result of this calculation is a floating-point number rounded to the nearest tenth. The following is an example of a substitution expression using this scaling:

```
[‘@ mr:% ]
```

Up arrow

The up arrow scaling character (↑) instructs the netlister to take the property value if found and divide its value by the value of the SE `simTimeUnit` variable. The default value of the `simTimeUnit` variable is 1E-9. This division produces a fixed-point result. The following is an example of a substitution expression using this scaling:

```
[ ↑@ mr:% ]
```

Ampersand

The ampersand scaling character (&) instructs the netlister to take the property value if found and divide its value by the value of the SE `simCapUnit` variable. The default value of the `simCapUnit` variable is 1E-15. The result is then rounded to the nearest integer and output to the netlist. The following is an example of a substitution expression using this scaling:

```
[&@ c ]
```

SKILL Formatting

This section explains the SKILL formatting ability of the netlister. The property type signals how the netlister attempts to interpret it. If the type of a property is `ilExpr`, the netlister attempts to execute its value as a SKILL expression. Only simple arithmetic operations and SKILL procedures can be executed in an `ilExpr` formatting instruction. Write your SKILL formatting instructions as SKILL procedures, store them in a file with the same name as your simulator with either an `.il` or an `.ile` suffix added. If you prefer to encrypt the file, use the SKILL `encrypt` function available in SE. If you store this file in the `*install_dir*/tools/dfII/etc/skill/fnl` directory, the netlister automatically loads it, and the SKILL functions defined in it can be called from your format instructions.

SKILL formatting allows you greater flexibility than is available using only substitution expression formatting instructions. In addition to the full functionality of the SKILL language, your SKILL formatting instructions have all information stored in the design hierarchy available by means of the SKILL-Level Database Access and netlister-defined functions and global variables. With these database access functions and netlister functions and variables, you can write complex formatting instructions. For example, you can output properties only if the value of another property is within a specified range, perform calculations based on the values of any number of properties, instruct the netlister to output the result into the netlist, manually traverse the connectivity of the design, or query the netlister for information about the design using netlister-defined global variables and functions.

The SKILL procedures you write can output information to the netlist in several ways. There is a netlister function that you can call to output information to the netlist, as well as a SKILL port that can be used to write to the netlist file. You can use either of these to write to the netlist file, but you cannot use both in the same netlist run. In addition, the return value of your procedure is interpreted. The return value is both an error flag and a means of outputting information to the netlist. If the SKILL formatting property returns `t`, the netlister does not output any information to the netlist file. This is a way of signaling that the procedure executed successfully, and that the formatting instruction has already output any required information to the netlist. If the function returns or evaluates to a SKILL `string`, `fixnum`, or `flonum`, the netlister outputs this value to the netlist file. A return value of `nil` signals that an error has occurred, and the netlister uses this information to determine that the netlisting process failed.

The following two sections explain the various functions and variables defined by the netlister that you can use in your SKILL formatting instructions. Following these sections is a short example of how SKILL formatting instructions could be used to output the connectivity of a two-input gate into the netlist. A complete output formatting example using the SKILL language is provided at the end of this chapter.

Netlister-Defined Variables

The following global variables are defined by the netlister which you can access as any SKILL variable. Not all are set, depending on the output functions being used. The scope of these variables is the same as their counterparts in the netlister-defined properties used in substitution expressions.

Note: These variables are read only. Never try to set these variables directly.

fnlNetlistFile

This variable is a SKILL *port*, the netlist output file, and is defined throughout the execution of the netlist run. If you write to this port directly, do not use the `fnlPrint()` function. The `fnlPrint()` function buffers its output for speed so that it can automatically fold lines that are too long. If you intermix calls to the `fnlPrint` function and write to the `fnlNetlistFile` port, the internal buffer for the print function may not be flushed and the netlist becomes garbled.

fnlCurrentNetExtName

This variable is a SKILL `string` and corresponds to the netlister-defined `NodeNumber` property. It represents the netlister-assigned signal name for the current signal being expanded. The variable is valid only during evaluation of the `NLPcreateNetString` property.

fnlCurrentInstExtName

This variable is a SKILL `string` and corresponds to the netlister-defined `ElementNumber` property. It represents the netlister-assigned instance name for the current iteration of the current instance being expanded. The variable is valid only during evaluation of the `NLPcompleteElementString` property.

fnlLineLength

This variable is a SKILL `fixnum` and corresponds to the netlister-defined `NLPLineLength` property. It represents the maximum line length in the output netlist. The variable is valid throughout the execution of the netlist run.

fnlLinePrefix

This variable is a SKILL `string` and corresponds to the netlister-defined `NLPLinePrefix` property. It represents the continuation string to be output at the beginning of the next line if the current netlist line exceeds the maximum length. The variable is valid throughout the execution of the netlist run; however, it may be `nil` if you have not set the `NLPLinePrefix` property.

fnlLinePostfix

This variable is a SKILL `string` and corresponds to the netlister-defined `NLPLinePostfix` property. It represents the continuation string to be output at the end of the current line if the current netlist line exceeds the maximum length. The variable is valid throughout the execution of the netlist run; however, it may be `nil` if you have not set the `NLPLinePostfix` property.

Note: The FNL API SimRunNetlist returns a list (nlpUnresolvedInhExprList) of unresolved net expressions when nlpCollectUnresolvedInhExpr is set to t as shown in the following sample output netlist:

```
(("newdefault!" "[@new:%:newdefault!]" "[@VDD:%:newdefault!]")
("new!" "[@VDD:%:new!]") ("netvdd!" "[@VDD:%:netvdd!]"))
```

All SE variables

Any variable or function defined or set in SE can be accessed during netlist formatting by a SKILL format instruction. This feature enables you to write your netlister formatting functions as part of SE and refer to them from formatting properties in the netlister.

Note: Although you can read values and execute functions defined in SE, you cannot return values or define functions during netlisting for later use by SE after netlisting has completed.

Netlister-Defined Functions

The following functions are defined in the netlister and can be called by your output functions. These functions are designed to provide any data used by the netlister that may be needed to format the netlist. Some of the functions listed here may seem similar to the global variables. They appear as functions instead of variables to improve the speed of the netlisting process, since not all of the data may be required by a particular output format.

Some of the functions provide valid results only when used during evaluation of the specified property. For example, there is no "current signal" when expanding an instance, since many signals may connect to a particular instance. Many of the functions also have counterparts as netlister-defined properties. The same restrictions applying to those properties also apply to the corresponding functions. (Refer to the "Formatting Substitution Expressions" section in this chapter.) The major difference between these functions and their corresponding properties is that a substitution expression, such as "[@ tr]," prints the property value to the netlist file, whereas a function returns the property value but does not print the value to the netlist file. You must explicitly call a function to output information to the netlist.

fnlTopCell

Returns the top-level cellview being netlisted. The function takes no arguments and returns a cellview object identifier (*`d_cellviewId`*). The function is valid throughout the netlist process. It may be especially useful during evaluation of the `NLPnetlistHeader` and `NLPnetlistFooter` properties, where you can use it to output information about the top-level design for the header and footer of the netlist. There is no equivalent netlister-defined property.

fnlCurrentSigPathName

Returns the full pathname of the current signal being expanded. It corresponds to the netlister-defined `NetPathName` property. As the corresponding property, this function is valid only during evaluation of the `NLPcreateNetString` property. This function takes no argument and returns a SKILL string. The full signal pathname returned by this function differs from `signal ~> name` in that all the instance names down the hierarchy to the current signal are included in the name. For example, `signal ~>name` can be *"*`dataIn`*,"* while this function returns *"*`/chip1/ALU/dataIn`*"* to uniquely identify this signal in the flattened design.

fnlCurrentSig

Returns the current signal being expanded. The function takes no arguments and returns a signal object identifier (`d_sigId`).

fnlSigCdsNameExtName

Returns the netlister-assigned name for the signal name given as an argument. The function is equivalent to the substitution expression "[#name]." As with the matching expression, the signal names allowed as arguments are restricted to the signals in the schematic that the netlister was traversing when it encountered the current instance or to the signals in the top-level design if the function is called during evaluation of the `NLPnetlistHeader` or `NLPnetlistFooter` properties. The function takes a string argument and returns a SKILL string if the signal is found. If the signal is not found, `nil` is returned.

fnlInstCdsNameExtName

Returns the netlister-assigned name for the instance name given as an argument. This function is equivalent to the substitution expression "[$name]." As with the matching expression, the instance names allowed as arguments are restricted to the instances in the schematic the netlister was traversing when it encountered the current instance or to the instances in the top-level design if the function is called during evaluation of the `NLPnetlistHeader` or `NLPnetlistFooter` properties. The function takes a string argument and returns a SKILL string if the instance is found. If the instance is not found, `nil` is returned.

fnlCurrentInst

Returns the current instance being expanded. The function takes no arguments and returns a instance object identifier *(`d_instanceId`)*. It is valid only during expansion of the `NLPcompleteElementString` property. Do not use the master field of the resulting `instanceId`. If you use it, you get the symbol cellview rather than the stopping cellview corresponding to the symbol placed in the schematic. To get the instance master in a format instruction, use the `fnlCurrentCell` function.

fnlCurrentInstCdsName

Returns the name of the current instance being expanded. The function takes no arguments and returns a SKILL string that is the full instance pathname to the current instance being expanded. The function is valid only during expansion of the `NLPcompleteElementString` property. The full instance pathname returned by this function differs from `instance`

`~>name` in that all the instance names down the hierarchy to the current element are included in the name. For example, `instance ~> name` can be `count1`, while this function returns `/chip1/ALU/count` to uniquely identify this instance in the flattened design.

fnlCurrentCell

Returns the master of the current instance being expanded. The function takes no arguments and returns a cellview object identifier *(`d_cellviewId`)*. It is valid only during evaluation of the `NLPcompleteElementString` and `NLPcreateModelString` properties. The cellview returned by this function may differ from `instance ~>master`. Do not de-reference the master field of an instance directly in any format instruction evaluated by the netlister. The cellview returned by this function takes into account the view list used during netlisting. This function returns the stopping cellview used to format the netlist, whereas the `instance ~> master` normally returns the symbol cellview placed in the schematic.

fnlCurrentCellCdsName

Returns the master cell name of the current instance being expanded. The function takes no arguments, returns a SKILL string, and is valid only during evaluation of the `NLPcompleteElementString` and `NLPcreateModelString` properties. The function corresponds to the netlister-defined `BlockName` property.

fnlCurrentModelExtName

Returns the netlister-assigned model name of the current instance being expanded. The function takes no arguments, returns a SKILL string, and is valid only during evaluation of the `NLPcompleteElementString` and `NLPcreateModelString` properties. The function corresponds to the netlister-defined `ModelNumber` property.

fnlGetGlobalSigNames

Returns a list of strings that are the names for all of the global signals contained in the design hierarchy. If the netlister-assigned name is required, you can pass these names to the `fnlSigCdsNameExtName()` function to translate them during the header or footer evaluation. This function takes no arguments and can be called at any time during the netlisting process.

fnlAbortNetlist()

Aborts netlisting. When the formatter detects an error during netlisting, it calls this function to inform the netlister to abort netlisting.

fnlTermCdsNameExtName(name)

Returns each netlister-assigned signal name for the signal attached to the terminal whose name is given as an argument. The function is equivalent to the substitution expression [|name]. As with the matching expression, the terminal names allowed as arguments are restricted to the terminals attached to the current instance. The function takes a string argument and returns a SKILL string if the terminal is found. If the terminal is not found, `nil` is returned.

fnlTermExtName(terminal bit)

Returns the netlister-assigned name for the signal attached to the bit of the terminal given as an argument. The function is similar to the substitution expression [|name]. As with the matching expression, the terminals allowed as arguments are restricted to the formal terminals of the master of the current instance. The function takes two arguments: the first is a `FormalTerminal` (*`d_termId`*), the second is a SKILL `fixnum` that is the bit of the terminal whose net name you are requesting. The function returns a SKILL string if the requested bit of the terminal is found. If the bit is not found, `nil` is returned. This function is valid only during the evaluation of the `NLPcompleteElementString` property.

fnlSearchPropString(propName localSearch)

Returns the SKILL string value of the property whose name is given as an argument. The `localSearch` argument can be either `nil` or `t`. If `localSearch` is `nil`, the function call corresponds to the substitution expression \“[@ propName] \” .If the `localSearch` argument is `t`, it corresponds to the expression \“[propName].”

This function searches for the properties and evaluates their values as if the property name was used in a substitution expression. Any `ilExpr` or `nlpExpr` type properties are fully evaluated, and the result of the evaluation is returned.

If the property is not found, this function returns `nil`. If the property is found, but there is no output, `t` is returned. If an error occurs during either the property search or evaluation, a Lisp Error is generated. Errors may occur because of SKILL syntax errors, errors during evaluation of a SKILL expression, netlister substitution expression syntax errors, or by exceeding an internal buffer size. The maximum size of any property value returned by a SKILL expression is 4K characters.

This function is valid only during evaluation of the `NLPcreateModelString` and `NLPcompleteElementString` properties.

fnlPrint(string)

Prints its argument to the netlist file. The argument can be either `t` or `nil`, which do not add any text to the netlist file and are simply a convenience, or any of the SKILL `string`, `fixnum`, or `flonum` types. The file is the same as defined by the `fnlNetlistFile` variable. If the argument exceeds the maximum line length of the netlist file (as defined by the `NLPlineLength` property), the string is broken at white space intervals and new lines are inserted. If the `NLPsingleLineCommentString`, `NLPlinePrefix`, or `NLPlinePostfix` properties are defined, the property values are used to insert comment characters or pre/postfix continuation strings, as needed.

Note: If you use this function to output information to the netlist file, do *not* write to the netlist file directly. This function buffers its output for speed. If you call this function and write to the netlist file directly, the internal buffer for this function may not be flushed and the netlist becomes garbled.

This function is defined throughout the netlist process.

fnlPathList

Takes no arguments and returns a list representing the current instance path down the schematic hierarchy to the current instance or signal being expanded. The returned list is a list of lists, where each sublist is a (`inst cellview index`) triple (the `inst` member can also be a `signal`). You can test this by doing an `objType` and checking whether it is a signal or an instance. When called during `NLPcreateNetString` expansion, the second element is a `signal`; otherwise, it is an instance. The instance/signal pointer is a pointer to the instance or signal being expanded in the prior cellview. The first instance is essentially invalid since it points to the instance of the top-level schematic which the netlister has placed as an instance in the `nlpglobals` cellview. The cellview pointer is the master cellview for the instance. For a low-level device, it is the stopping cellview, not the symbol; at intermediate levels, it is the schematic, not the symbol. The last element is an index which specifies the current index of an iterated instance. If the `inst ~> objType` is a signal, then the index value will be `zero`.

Sample: ((inst cellview index) (inst cellview index) (signal cellview index))

fnlCurrentIteration

Takes no arguments and returns an index of the current iterated instance being expanded. Only an instance can be iterated and placed in the schematic. If the `inst ~> objType` is a signal, then the index value is `zero`.

SKILL Formatting Example

The following example is the type of syntax that can be used when formatting the netlist with `ilExpr` properties. The example shows how formatting different cellviews can be done using SKILL functions and provides a specific example of formatting a two-input AND gate and the needed support functions. Functions and variables not defined in this example are defined by the netlister. For documentation on these, refer to the "Netlister-Defined Functions" and "Netlister-Defined Variables" sections in this chapter.

This example assumes that in the `nlpglobals` cellview, the `NLPcompleteElementString` property is type `ilExpr` and its value is a call to the user-defined SKILL `FormatInstance` function. When `FormatInstance` is executed, it searches for the `FormatInstanceString` property on the stopping cellview of the instance. The `FormatInstanceString` property is also type `ilExpr`. For the two-input AND gate, its value is the user-defined SKILL `Format2InputGate("AND")` function call. The `Format2InputGate` function also calls the `FormatDelayTimes` function to format the delay values into the output string. These functions are then defined in the `*install_dir*/tools/dfII/etc/skill/fnl/*simulator_name*.ile` file.

```
simSetDef( 'fnlFormatterUnbindFuncs
'( FormatInstance
Format2InputGate FormatDelayTimes )
)
; This function locates the format property for each
; device and evaluates it.
procedure( FormatInstance()
fnlSearchPropString("FormatInstanceString" nil)
)
; This function outputs the connectivity
; for a two-input logic gate.
;
procedure( Format2InputGate( gateName )
let( (tmp)
; Print the netlister-assigned name for the net
; attached to terminal
; "Y" followed by the name of this gate.
sprintf( tmp "%s .%s" fnlTermCdsNameExtName("Y")
gateName )
fnlPrint(tmp)
; Print any delay times for this gate.
FormatDelayTimes()
; Print the netlister assigned names for the nets
; attached to the terminals "A" and "B".
sprintf( tmp "%s %s\n" fnlTermCdsNameExtName("A")
fnlTermCdsNameExtName("B"))
```

```
fnlPrint(tmp)
)
)
; This function outputs the delay times for a basic logic
; gate.
procedure( FormatDelayTimes()
; Print the value of the rise time property. Note we do not
; check the return value, because fnlPrint will not output
; anything if passed t or nil.
fnlPrint( fnlSearchPropString("tr" nil) )
; Print the value of the fall time property.
fnlPrint( " " )
fnlPrint( fnlSearchPropString("tf" nil) )
)
The following line is added to the netlist file when a two input AND gate is
encountered in the design.
N45   .AND   2   3   N63   N64
```

Customizing FNL Output

Customizing FNL output consists of these steps:

1. Read documentation on, and have a working knowledge of, required tools.
2. Create a global cellview.
3. Create library elements.
4. Write SKILL formatting procedures.
5. Modify SE to recognize your simulator.

The following sections explain the details of what is required in each step.

Learn Required Information

Before attempting to modify the FNL, read the following material:

- *Cadence SKILL Language User Guide*
- The “Database Access” chapter of the Virtuoso Design Environment SKILL Reference
- *Virtuoso Schematic Editor L User Guide*
- *Simulation Environment Help*

Create Global Cellview

Before you can modify the netlister output, you must create a global cellview. The global cellview is basically a schematic. The netlister uses it as the root of the design being netlisted and as a storage location for global netlist formatting instructions. The default name of this cell is `nlpglobals`. When netlisting begins, the netlister tells the design manager to locate a library containing the cell with this name. The library is specified in the `cds.lib` file. This same process is used to flatten the design hierarchy.

To create the global cellview, create a schematic with the cell name `nlpglobals` and with the view name the same as your simulator name. For example, if your simulator name is `spice`, create a schematic called *(*`nlpglobals spice`). Place this schematic in a library to be included in the library search path used by your designers. Inside of this schematic, place a single instance whose master is *(*`dummy dummy`). This device can be found in the Cadence-provided library *`install_dir`*`/tools/dfII/etc/cdslib/basic`. When the netlister flattens the design, this instance is replaced by the design being netlisted. This is the only device that must be placed in the global cellview.

It is possible to place other devices or nets in the global cellview. If any additional devices are placed in the global cellview, every time your formatting instructions for FNL are used, the connectivity for these devices is output to the netlist following the connectivity for the design being netlisted.

Once you have added this device, you must extract and save the schematic. Then, you must define some of the netlist formatting properties you want the netlister to search for in this cellview. To define these, you can use either the property list editor or S/SLG. If you use the property list editor, you must extract and save the schematic when you have finished adding your formatting properties. It is recommended that you use S/SLG to add your netlist formatting properties. With S/SLG you can create a text file specifying the properties you want added to the global cellview (along with any other library elements you create), and simply load the file into the S/SLG program whenever you modify the property values or add new properties.

Since FNL is instance-based, start by adding the `NLPcompleteElementString` property. This property is the main formatting function for the netlister since it outputs the connectivity for each device. To simplify this format string, it is customary to make its value a substitution expression, which is a search for another property. This second property is then placed on a device in the library with the cell name of the device it represents (that is, `inv` or `nmos`) and a view name which is the same as your simulator (that is, `spice`). The same property is placed on every primitive device in your library, but its value differs for each. The value of the property in each device is the netlister formatting instruction for how to output a reference to this device type in the netlist. For more information on creating these library elements, refer to the “Create Library” section in this chapter.

Once you have added this property to the global cellview, you may want to complete the steps listed in the “Customizing FNL Output” section in this chapter to be able to run the netlister and test your format instruction. You can define any number of the properties the netlister looks for (listed in the “Create Global Cellview” section in this chapter) in the global cellview. It is recommended that you first create a few library elements, modify SE to recognize your simulator, and create a small schematic to test your format instructions. After you have completed the steps required to run the netlister, and it outputs the correct connectivity for your test schematic in the syntax required by your simulator, you can return to this point and complete the global cellview creation.

Completing the global cellview consists of adding properties to format any needed netlist header, netlist footer, and model descriptions, if your simulator requires them.

Create Library Elements

Once you have created the global cellview, the next step is to create a library of simulation primitives. These primitives serve two purposes: the netlister uses them to detect when to stop the flattening process and to define how this specific element type is to be output to the netlist. For example, if the `NLPcompleteElementString` property in your global cellview was type `nlpExpr` and had the value “[@ SpiceElementFormat],” you would add a property with the name `SpiceElementFormat` to every element in your library. This element would then instruct the netlister how to output this type of gate to the netlist. For example, to output the connectivity for a capacitor in the syntax required by the SPICE simulator, the *(*`capacitor spice`) element in the library would contain a property whose name is `SpiceElementFormat`, whose type is `nlpExpr`, and whose value is:

```
"C[@ ElementNumber]  [ |PLUS]
[ |MINUS]   poly [@ c]  [@ ic:ic=%]
"
```

The properties you must place in the simulation primitives in your library are determined by how you write your formatting instructions defined in the global cellview. The netlister does not search for any format instructions on your simulation primitives unless instructed to do so by the properties you define in the global cellview. The only requirement the netlister imposes on these elements is that they contain the same terminals that exist in the corresponding symbol placed in the schematic. This terminal correspondence is guaranteed if you use S/SLG to create your simulation primitives.

Use S/SLG to create all of your simulation primitives. Not only does this ensure that the terminal correspondence between your symbols and primitives are correct, it also allows you to create a text file specifying all of the properties and their values that you want added to each primitive. Then, you can create your entire library simply by loading this file into S/SLG.

Create one or two of these primitives with the correct netlist formatting instructions for your simulator syntax. Then, create a small schematic containing only these elements and complete the steps required to run the netlister on this small design. Once you have netlisted this small design and are satisfied with the results, you can return to this step and complete your library for all of the elements required by your designers.

Write SKILL Formatting Procedures

If you use SKILL formatting instructions to format your netlist, these are the procedures you write and store in a file. Your `ilExpr` type format properties can refer to these functions directly.

Give the encrypted or non-encrypted version of this file the same name as your simulator suffixed with "`.ile`" and place it in the `etc/skill/fnl` directory. This directory is relative to the directory in which your Cadence software is installed. For example, if the `si` program is stored in the *`install_dir`*`/tools/dfII/bin` directory and your simulator is called *spice*, then store your functions in the *`install_dir`*`/tools/dfII/local/fnl/spice.ile` file. When you store them in this standard location, they are automatically loaded and available for use by the netlister.

Note: You must use the `fnlFormatterUnbindFuncs` variable to list all functions you define in SKILL to format the netlist. You must use the `fnlFormatterUnbindVars` variable to list all the variables you define in SKILL to format the netlist. The following sections describe these variables in detail.

fnlFormatterUnbindFuncs

Th*e* `fnlFormatterUnbindFuncs` variable specifies the functions you define in your simulator-specific `fnl` file that must be unbound when the designer switches simulators. You must specify any functions you define. If you do not include a function in this list, the correct function may not be used when the designer switches simulators.

Set this variable to a list containing the names of all of the functions you define. Define this variable in the `fnl` file for your netlist formatter outside of any function definition. The following is an example of how to set this variable for SILOS in the *`install_dir`*`/tools/dfII/local/fnl/silos.ile` file (because Cadence provides an interface to this simulator, the exact file location is *`install_dir`*`/tools/dfII/etc/skill/fnl/silos.ile` but your interface file should be placed in the location specified):

```
simSetDef( 'fnlFormatterUnbindFuncs,'(   silosFormatHeader
   silosFormatFooter
      )
)
```

fnlFormatterUnbindVars

The fnlFormatterUnbindVars variable specifies the variables you define in your simulator-specific fnl file that must be unbound when the designer switches simulators. You must include any variables you define. If you do not include a variable in this list, the correct value may not be used when the designer switches simulators.

Set this variable to a list containing the names of all of the variables you define. Define this variable in the fnl file for your netlist formatter outside of any function definition. The following is an example of how to set this variable for SILOS in the *install_dir*/tools/dfII/local/fnl/silos.ile file:

```
simSetDef( 'fnlFormatterUnbindVars,'(   silosPinList
      silosPropList )
)
```

Modify the Simulation Environment (SE)

FNL is run as part of SE. Before you can run the netlister, you must modify SE to recognize your simulator. You do this by creating a file with the name of your simulator suffixed with ".ile" and placing it in the *install_dir*/tools/dfII/local/si/caplib directory. This file must contain the default variable settings for variables used by the netlister to flatten a design. The file is written using the Cadence standard language, SKILL. The following example uses the spice simulator. When creating your file, replace the word spice with the name of your simulator and then you are ready to run the netlister.

1. Set the default switch and stop lists. The normal procedure is to have a simulator-specific switch and stop list, which the designer can override on a per-simulator basis. Then, use these lists to set the netlister variables. Add the following two lines to the file:

```
simSetDef( 'spiceSimViewList,'("spice" "schematic") )
simSetDef( 'spiceSimStopList,'("spice") )
```

   The first line sets the default view switching list and the second line sets the default stopping point list. Notice that instead of using the equal sign (=) to set the variables, you call the simSetDef function. This function only sets a variable if it has not already been set. This feature allows the designer to override the default value. This function, along with all other SE functions, is explained in the "Customizing the Simulation Environment (SE)" chapter in this manual. Set both of these variables to a SKILL list of strings.

2. Set the variables used by the netlister. Do this with the following two lines:

```
simViewList = spiceSimViewList
simStopList = spiceSimStopList
```

   Note: These variables are set with an equals sign (=). This is because the designer should not be allowed to set these variables directly. If the designer wants to change the default value, the simulator-specific version of the variables should be set.

3. Instruct the netlister which global cellview to use and in which library this global cellview is located. The SE `simNlpGlobalCellName` variable is the global cell name, and the default value is `nlpglobals`. The SE `simNplGlobalViewName` variable is the global cell's view name, and it should be the name of your simulator. The SE `simNlpGlobalLibName` variable is the library name that contains this global element (`simNlpGlobalCellName simNlpGlobalViewName`).

4. Because you are currently only generating a netlist and no name translation of simulator output is to be performed, you must inform SE by setting the `simSedFile` variable:

```
simSedFile = 'simNoSedFile
```

5. The SE `simActions` variable controls the steps that are performed when you run a simulation using SE. Setting this variable is not required at this time. However, setting it to verify only the existence of needed variables and run the netlister allows you either to select the netlist or simulate commands and have the system generate only a netlist. As you continue to develop your simulation interface, you may want to modify the default value of this variable to perform additional steps when running a simulation. At this point, you can copy the following to your file:

```
simSetDef( 'simActions,'( simCheckVariables()
netlist()
)
)
```

6. SE calls a function with the same name as the simulator to be executed as part of the initialization function. The initialization function is to set any needed variables. This function is usually set up to reset many of the variables you have already added to your file. This is because it is possible to run SE interactively. By doing so, the designer can change the defaults for the simulator-specific variables that have been discussed. To ensure that the netlister makes use of these new values, you must reset the previously listed variables. In addition, this function is expected to set the `simCommand` variable. This variable defines the command used to run the simulator. Since the simulator is not to be executed yet, this can be a command to return. At the end of this procedure is a call to the `simPrintEnvironment` function. This function writes an environment file that is later used by the Cadence graphics program.

   The following is an example of the minimum contents of the function you should define:

```
simIfNoProcedure( spice()
   simViewList = spiceSimViewList
   simStopList = spiceSimStopList
   ; global library name
   simSetDef( 'simNlpGlobalLibName "sample" )
      ; global cell's view name
   simSetDef( 'simNlpGlobalViewName "spice" )
   simCommand = 'prog( () return(t) )
   simPrintEnvironment()
)
```

7. You must specify all functions you define in your `caplib` file with the `simSimulatorUnbindFuncs` variable. Specify all variables you define with the `simSimulatorUnbindVars` variable. You can use the following settings for the sample `caplib` file:

```
simSetDef( 'simSimulatorUnbindVars '   (   spiceSimViewList
         spiceSimStopList
         simActions
   )
)
simSetDef( 'simSimulatorUnbindFuncs'(spice))
```

8. At this point you are ready to run the netlister. The following section has a complete example of the file you need to create.

Sample caplib File

The following is an example of the file you must create before SE can run the netlister.

1. Change every name that contains the simulator name `spice` to contain the name of your simulator. For example, change `spiceSimViewList` to `verilogSimViewList`.
2. You have the option of encrypting or not encrypting the file. Encrypt this file using the SKILL `encrypt` command available in SE.
3. Give the file the same name as your simulator suffixed with "`.ile`" and place it in the `install_dir/tools/dfII/local/si/caplib` directory.

You are now ready to run the netlister. It can be executed as if it were a standard Cadence product. Refer to the *Simulation Environment Help* for details on how to run a simulation.

```
; Set the default spice specific view switch list for netlisting.
simSetDef( 'spiceSimViewList,'("spice" "schematic") )
; Set the default spice specific stopping view list for netlisting.
simSetDef( 'spiceSimStopList,'("spice") )
; Set the view switch list used for netlisting.
simViewList = spiceSimViewList
; Set the stopping view list used for netlisting.
simStopList = spiceSimStopList
; Set the default name of the global cell and view used for
; netlisting.
simSetDef( 'simNlpGlobalLibName "sample")
simSetDef( 'simNlpGlobalViewName "spice")
; Signal that no sed input file for name translation exists.
simSedFile = 'simNoSedFile
; Set the default actions to be performed when a simulation is
; run. At this point, verify that the required variables have
; been set, and run the netlister.
simSetDef( 'simActions,'( simCheckVariables()
netlist()
)
)
; Set the list of functions and variables that must be unbound
```

```
    ; when environments (simulators) are switched.

    simSetDef( 'simSimulatorUnbindVars '   (   spiceSimViewList
          spiceSimStopList
          simActions)
    )
    simSetDef( 'simSimulatorUnbindFuncs '(spice))
    ;
    ; spice() -
    ; This function sets the variables "simViewList" and "simStopList"
    ; to the simulator specific values in "spiceSimViewList"
    ; and "spiceSimStopList" respectively.
    ; Then the variable "simCommand" is set to the commands required
    ; to run a SPICE simulation(at this point, no simulation is run).
    ; This function is called by SE as part of its initialization process.
    simIfNoProcedure( spice()
    ; Reset the global variables used by the netlister in case the user
    ; has modified the simulator specific versions by running SE
    ; interactively.
    simViewList = spiceSimViewList
    simStopList = spiceSimStopList
    simCommand = 'prog( ()
    return(t)
    )
    ; Write the environment file in case the user has modified
    ; the environment while running SE interactively.
    simPrintEnvironment()
)
```

Modify Netlister-Generated Names

The netlister generates new unique names for signals, instances, and models in the flattened design by appending a unique number to a name prefix which you can specify. If you do not specify a prefix, the netlister assigns numbers as new names. There are three variables within SE that allow you to set these name prefixes. Their default value is `nil`, which instructs the netlister that no name prefix is to be used. If they are set to a string value, that string is used as the prefix.

The three variables are `simNetNamePrefix`, `simInstNamePrefix`, and `simModelNamePrefix`.

simNetNamePrefix

Specifies the name prefix to be used when the netlister generates a unique signal name.

simInstNamePrefix

Specifies the name prefix to be used when the netlister generates a unique instance name.

simModelNamePrefix

Specifies the name prefix to be used when the netlister generates a unique model name.

Set Netlister Name Prefixes

The following is a sample of what you could add to the SE customization file that you placed in the `install_dir/tools/dfII/local/si/caplib` directory to set the net, instance, and model name prefixes to `N`, `I`, and `Model`, respectively.

```
; Set the default signal name prefix used for generating
; names when netlisting.
simSetDef('simNetNamePrefix "N" )
; Set the default instance name prefix used for generating
; names when netlisting.
simSetDef('simInstNamePrefix "I" )
; Set the default model name prefix used for generating
; names when netlisting.
simSetDef('simModelNamePrefix "Model" )
```

FNL Format Example

The following is an example of formatting instructions for the netlister. The example is a simplified version of a portion of the Cadence SPICE™ Simulation Interface. You can enter these examples into your system and run them using the SE sample `caplib` file given as an example in the "Modify the Simulation Environment (SE)" section in this chapter. As with that file, change all occurrences of the name `spice` to the name of your simulator before entering them. This is to ensure that you do not overwrite the more complete versions provided by a Spice Simulation Interface.

The following is an S/SLG input file. It can be used to add the netlister properties to the global cellview.

Note: You must first create the global cellview and add the `dummy` device. (Refer to the "Create Global Cell" section in this chapter for more details.)

```
replaceViewProp = t
lmDefViewProp( nlpglobals spice
NLPLineLength = 65
NLPLinePrefix = "+"
NLPsingleLineCommentString = "*"
NLPElementComment = nlpExpr("* [@BlockName] ( [@ElementNumber] ) = [@InstPathName]
")
NLPcreateNetString = nlpExpr("* [ net @NodeNumber ] = [@NetPathName ] \ n")
NLPcompleteElementString = nlpExpr("[@NLPElementPostamble:%:** No element format
property found for element [@InstPathName ] ] \n")
NLPcreateModelString = nlpExpr("[@NLPModelPreamble:%\n ] ")
NLPmosfetModelCard = nlpExpr(".MODEL Model [ @ModelNumber]
[ @modelType] [ @level:level=% ]  [@kp:kp=% @gamma:gamma=%]  [ @lambda:lambda=% ]
")
```

```
"graphicsEditorUnits per userUnit" = 160.0
userUnits = "inches"
)
```

The following is another S/SLG input file. It can be used to create the netlister library elements to output the connectivity for a `capacitor`, `nmos`, and `pmos`.

Note: This file expects a `symbol` view to exist for each of these elements. They are used as templates by S/SLG to ensure that the correct terminal information is added to the new primitive views.

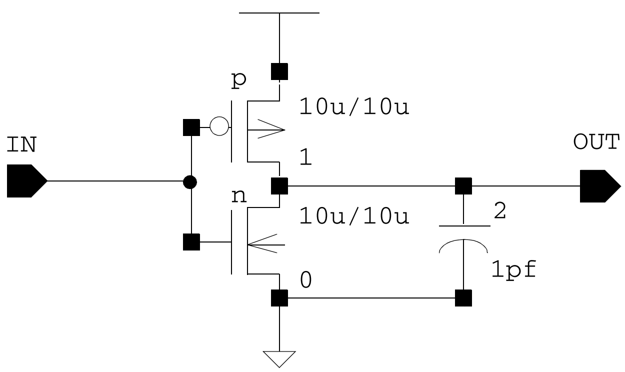
```
lmSimView( nmos symbol spice
NLPElementPostamble = nlpExpr(" [ @NLPElementComment:%\n ] M [@ElementNumber]  [
|D ]  [ |G ]  [  |S ]  [ #gnd! ] Model  [@ModelNumber]  [ @l: l=%]
[ @w: w=%]  [ @ic:ic=% ] ")
NLPModelPreamble = nlpExpr(" [ @NLPmosfetModelCard ] ")
gamma = 0.2
"graphicsEditorUnits per userUnit" = 160.0
lambda = 0.02
level = 2
modelType = "nmos"
tox = 6e-07
userUnits = "inches"
)
lmSimView( pmos symbol spice
NLPElementPostamble = nlpExpr(" [ @NLPElementComment:%\n ] M [@ElementNumber ]  [|D
]  [ |G ]  [ |S ]  [ #vdd! ] Model [@ModelNumber]  [ @l: l=% ]  [@w: w=%] [@ic:ic=%
]")
NLPModelPreamble = nlpExpr(" [ @NLPmosfetModelCard ] ")
gamma = 0.4
"graphicsEditorUnits per userUnit" = 160.0
lambda = 0.03
level = 2
modelType = "pmos"
tox = 6e-07
userUnits = "inches"
)
lmSimView( capacitor symbol spice
NLPElementPostamble = nlpExpr(" [ @NLPElementComment:%\n ] C [@ElementNumber]  [
|PLUS ]  [ |MINUS ]   poly [@c ]  [@ic:ic=% ] ")
"graphicsEditorUnits per userUnit" = 160.0
userUnits = "inches"
)
```

If you have used these files to create the corresponding library elements and have added the SE file as described, the following netlist is produced when you run the netlister on the schematic shown in Figure 7-1 on page 250.

```
* net 1 = vdd!
* net 0 = gnd!
* net 2 = /IN
* net 3 = /OUT
.MODEL Model1 pmos level=2 gamma=.4 lambda=.03
* pmos(0) = /1
M0 1 2 3 1 Model1 l=10u w=10u
* capacitor(1) = /2
C1 3 0 poly 1pf
```

```
.MODEL Model3 nmos level=2 gamma=.2 lambda=.02
* nmos(2) = /0
M2 3 2 0 0 Model3 l=10u w=10u
```

Figure 7-1 Inverter Schematic (*inv schematic*)



FNL SKILL Functions

For details on the FNL SKILL functions, see <u>*OSS Functions*</u> in the *Digital Design Netlisting and Simulation SKILL Reference*.

8

Customizing Post-Layout Simulation

Post-layout simulation is the process of running a simulation on a design with the parasitic parameters obtained from the corresponding physical layout. This is an important process in the design cycle because the results obtained from this process are much more precise compared to simulation on the schematic design alone.

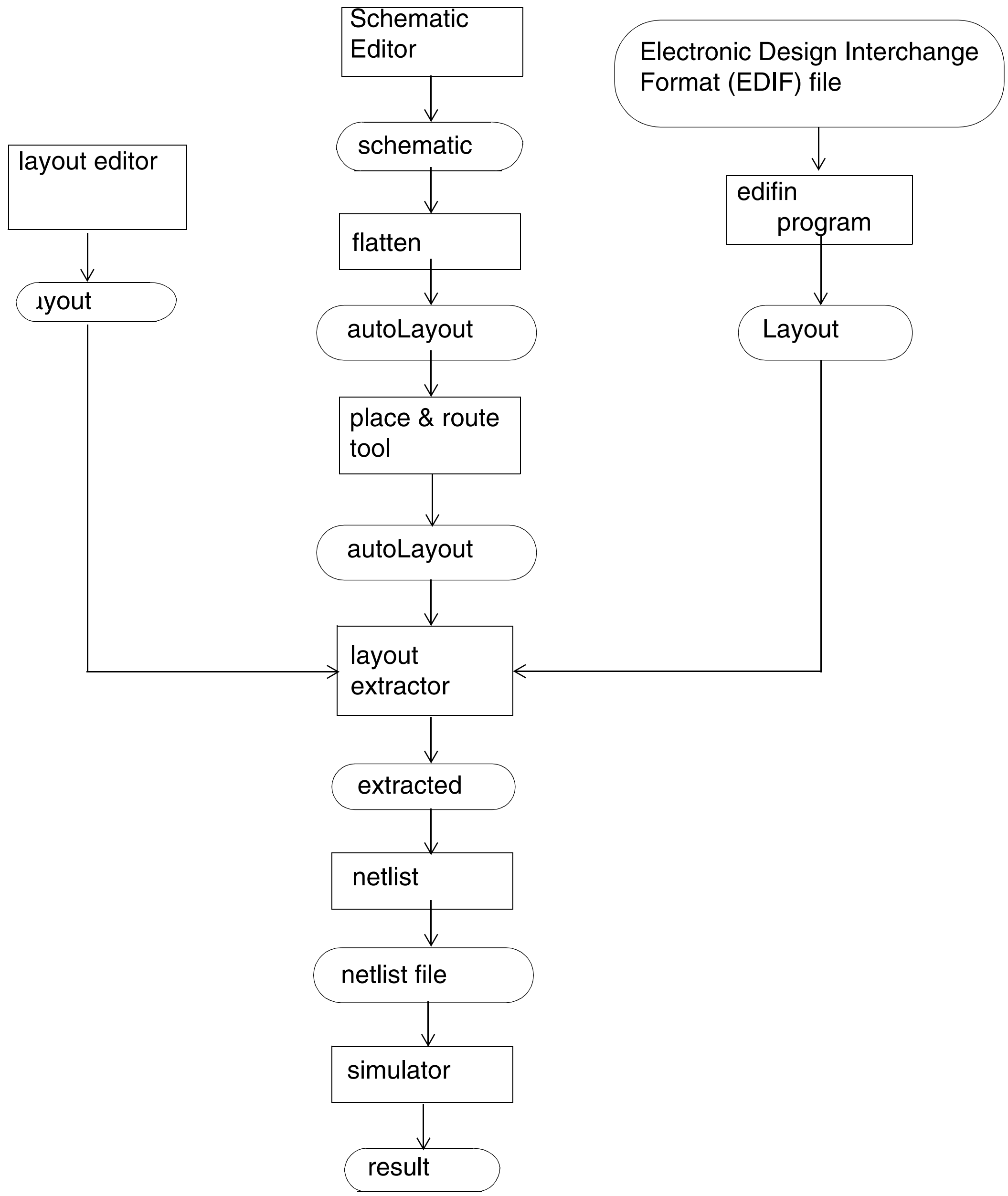
In the Cadence system, you can create physical layouts manually using the layout editor or automatically using the Place and Route tool, and you can translate physical layouts from the Electronic Design Interchange Format (EDIF) format using the *edifin* program. You obtain the parasitic parameters from the physical layout, using the layout parameter extractor.

Parasitics Extracted by the Layout Extractor

Figure 8-1 on page 252 shows the situation where the parasitic parameters are extracted using the layout extractor. The layouts created by the layout editor and the *edifin* program can be used as input to the layout extractor. The layout extractor generates an extracted layout view which can be netlisted and used later as input to the simulator. The parasitic parameters are part of the extracted layout view; thus, they can be netlisted directly by the appropriate substitution expression (`nlpExpr`) or by the SKILL command () that conforms to the syntax of your simulator. Refer to the “Customizing the Simulation Environment (SE)” chapter in this manual for more information on the netlist commands.

If this method is sufficient, then just refer to the “Customizing the Simulation Environment (SE)” chapter in this manual for instructions about producing a netlist from the extracted layout view and to the *Diva Interactive Verification User Guide* for more information on how to extract parasitic parameters from a layout.

Figure 8-1 Using the Layout Extractor To Extract Parasitics



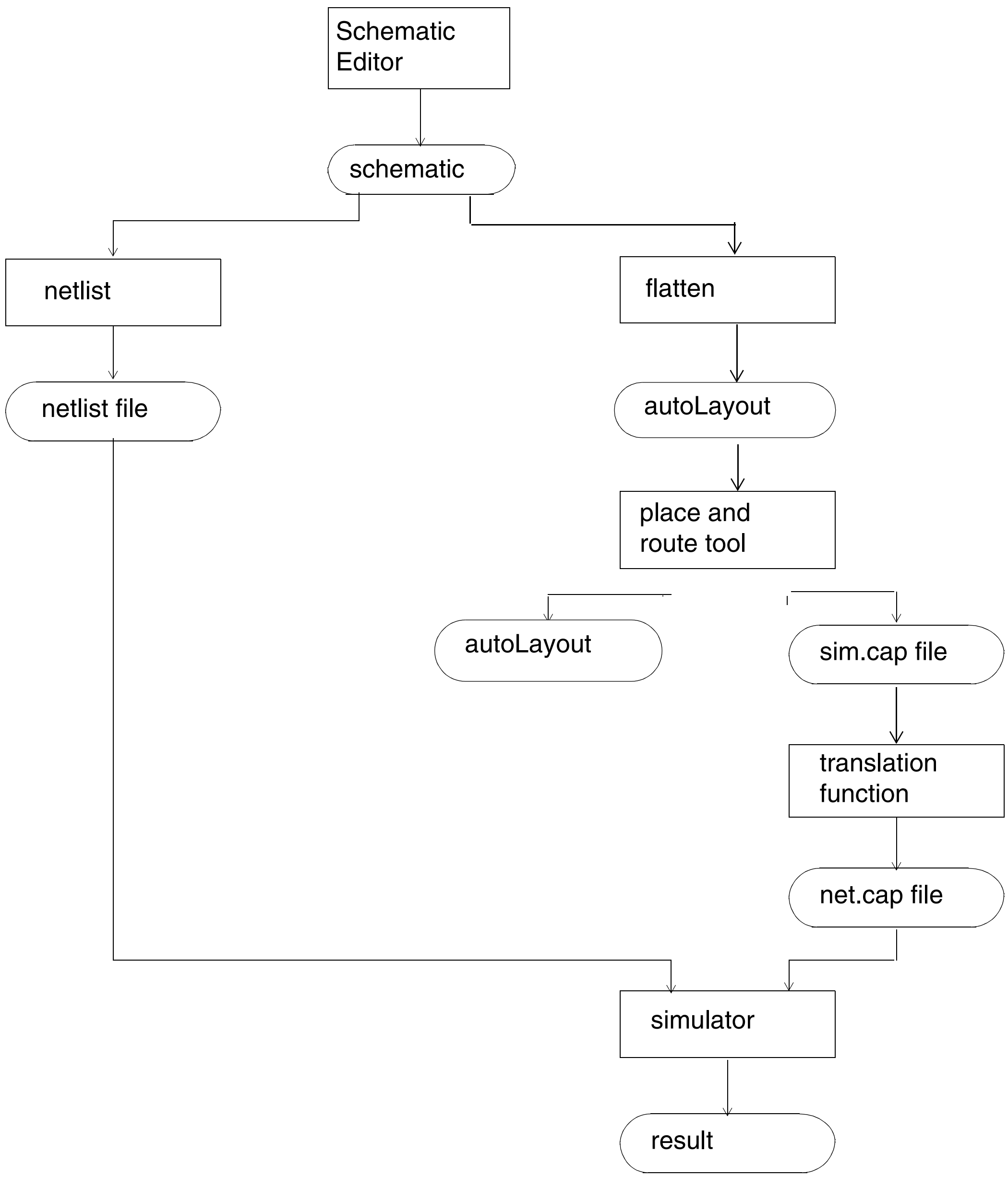
Parasitics Extracted by the Symbolic Layout Parameter Extraction Tool

Figure 8-2 on page 254 shows the situation where the Symbolic Layout Parameter Extraction (LPE) tool is used to calculate the parasitic parameters. In this case, the capacitance for all the nets in a routed view (generated by the Place and Route tool) are calculated and printed to a file called `sim.cap`. Refer to the *Simulation Environment Help* for more information on how to create the net capacitance file.

The information in the `sim.cap` file is translated into the format of your simulator and then merged with the netlist from the schematic as input to your simulator. The net capacitance in the `sim.cap` file can be displayed on the schematic regardless of which simulator is used. Refer to the *Simulation Environment Help* for more information on the display functions. Currently, this Place and Route extraction method (shown in Figure 8-2) is supported for the Cadence Timing Analyzer (TA), SILOS II®, and HILO3™. Furthermore, this same method supports both hierarchical and flat netlisting.

The remainder of this chapter describes how to customize this method for your simulator. Before continuing, your simulator should already be integrated into the Cadence system, so you should be familiar with the Simulation Environment (SE) and its requirements. Refer to the “Customizing the Simulation Environment (SE)” chapter in this manual for more information on how to integrate your simulator.

Figure 8-2 Using the Place and Route Tool To Extract Parasitics



Customization Steps

The customizing steps include translating the net capacitance file (`sim.cap`) to the format of your simulator and merging the translated result with the netlist description. You must write a new function to implement the translation step. This new function should be added to the file having the same name as your simulator plus the "`.ile`" suffix in the *`install_dir`*`/tools/dfII/local/si/caplib` directory. To avoid a naming conflict with the simulation interface, the name of the new function should follow the recommended SE naming conventions. That is, the first letter of each word comprising the name should be upper case and the rest of the name lower case, for example, `YourSimTranslation`.

The following sections describe the net capacitance file, customizing the translation process, customizing the control file to include the translated result, and all the variables and functions that are currently used for the supported simulators.

Net Capacitance File

The format for the net capacitance file is important because the new function needs to parse and format it for your simulator. The net capacitance file format is shown here:

```
;Net Capacitance File for Lib:testLib Cell:test1
  View:layout
"<net name>"   <capacitance>   ;<comment>
"<net name>"   <capacitance>
;<comment>
"<net name>"   <capacitance>
```

The first line serves as a header stamp to verify the integrity of the data, and it must be set as shown in the example. The rest of the first line can be any text. Comments must start with a semicolon (;), and they can be placed anywhere in the file. The unit of the capacitance is `farad`. If necessary, the `simCapUnit` variable should be used to convert the capacitance to the appropriate unit in your simulator. The `<net name>` is the full path name of the signal corresponding to the flattened design. For example, a signal name might look like this:

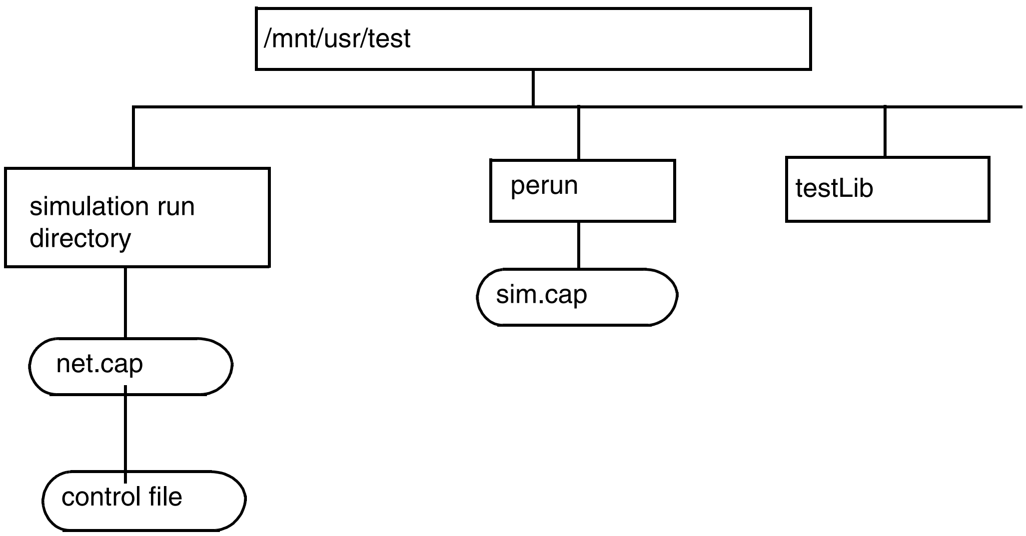
```
"/cpu/alu1/control"
```

The name of this file must be called `sim.cap`, and it should be located in the `perun` directory, contained in the directory that includes the library containing the cellview being simulated.

For example, if the `simLibName` variable is set to `testLib`, and `testLib` is found in the directory `/mnt/usr/test`, then, by default, the `sim.cap` file is located in `/mnt/usr/test/perun/sim.cap`.

If the `simCapFileDir` variable is not defined, then the system uses the library directory to construct the file system full pathname to locate the `sim.cap` file. Otherwise, the `simCapFileDir` variable must contain the full pathname of the `sim.cap` file. Figure 8-3 on page 256 shows an example of the default location of the `sim.cap` file. Refer to the "Variables" and "Functions" sections in this chapter for more information on the `simCapFileDir` variable.

Figure 8-3 Example of Directories With sim.cap and net.cap Files



Customizing the Translation Process

The primary task in customizing the translation process is to write the function that parses the `sim.cap` file and translates it into the format of your simulator. For ease of programming, the translated result is always stored in the `net.cap` file, and it is located in the current run directory. The `net.cap` file is then merged with the netlist from the schematic as input to the new simulator. Refer to Figure 8-2 on page 254 and Figure 8-3 on page 256 for the control flow and the location of the `net.cap` file, respectively.

Figure 8-4 on page 256 is a sample file of the function that implements the translation process. Of course, the exact format of your simulator should be inserted, and the function name `YourSimTranslation` should be replaced by a name that conforms to the SE naming conventions. This function should be added to the file with the same name as your simulator plus the `.ile` suffix in the *install_dir*`/tools/dfII/local/si/caplib` directory. The location of the function within the file is not important.

Figure 8-4 Sample Translation Process File

```
; YourSimTranslation
; Function to convert net capacitance file into the format of your
; simulator. The sim.cap file is translated into the file called
; net.cap in the current run directory.
;
; Global Variables
;    simDoPostLayout  - controls the post-layout process.
;
procedure( simDoSilosTranslation()
```

```
 prog( (capFile inf outf extName line message)
;
; Remove old capacitance file.
;
simDeleteRunDirFile( "net.cap" )
;
; return immediately if post-layout simulation
; option is not chosen
;
if( simDoPostLayout == nil return(t) )
;
; return if can not locate the full path to the sim.cap file
;
if( ( capFile = simCapFileDir ) == nil then
   return( t )
)
;
; setup for name mapping
;
if simPreNameConvert(simLibName simLibConfigName             simCellName
            simViewName simVersionName simRunDir) == nil then
   return( nil )
)
;
; "sim.cap" file does not exist, just return
;
if ( (inf = infile(capFile)) == nil return(t) )
;
; verify the header stamp
;
if( ! simCheckHeader(inf) then
   return( nil )
)
;
; Output file is called "net.cap"
;
if( (outf = simRunDirOutfile("net.cap")) == nil then
   close( inf )
   return( nil )
)
;
; Parse the sim.cap file
;
while( (line = lineread(inf)) != nil
     if( listp(line) then
           if( (! stringp(car(line))) || (! numberp
                (cadr(line))) then
     sprintf(message "ERROR: illegal file format, %s"
       "expecting \"<net name>\" <value> - ")
     simPrintError( message )
         simPrintErrorLine(line)
     sprintf(message "Line ignored.\n")
     simPrintError( message )
           else
     ;
     ; INSERT YOUR CODE HERE TO:
     ; translate the net name into the mapped name, scale
     ; the capacitance if necessary and print the
     ; information according to the format of your simulator
     ;
           )
```

```
        )
   )
   ;
   ; Clean up for name mapping
   ;
   simPostNameConvert()
   close( inf )
   close( outf )
   return( t )
            )
   )
```

The following is a step-by-step walk-through of the function.

1. First, the old translated file (`net.cap`) must be removed.

```
;
;Remove old capacitance file.
;
simDeleteRunDirFile( "net.cap" )
```

   This step prevents old capacitance information from merging with the netlist description.

2. Next, the `simDoPostLayout` variable is checked. This step is desirable because post-layout simulation is an optional step, and thus, you should be allowed to disable it (by setting the variable to `nil`). The default value is `t`. Refer to the “Variables” section for more information on the `simDoPostLayout` variable.

```
;
   ;Return immediately if post-layout simulation
;option is not chosen
;
if( simDoPostLayout == nil return(t) )
```

3. The full path name for the `sim.cap` file is constructed.

```
;
;Return if can not locate the full path to the
;sim.cap file
;
if( ( capFile = simCapFileDir ) == nil then
return( t )
)
```

4. Initialize the name mapping procedure.

```
;
;Set up for name mapping
;
if( simPreNameConvert(simLibName simLibConfigName
                      simCellName simViewName
                      simVersionName
                      simRunDir) == nil then
           return( nil )
)
```

5. The `sim.cap` file is opened for reading. If the `sim.cap` file does not exist, return immediately.

```
;
;"sim.cap" file does not exist, just return
;
if( (inf = infile(capFile)) == nil return(t) )
```

6. The header stamp of the `sim.cap` file is verified.

```
;
;Verify the header stamp
;
if( ! simCheckHeader(inf) then
               close( inf )
         return( nil )
)
```

The translation process is terminated if the header stamp is incorrect. Refer to the “Net Capacitance File” section for more information on the header stamp.

The resultant **net.cap** file is opened in the current run directory for writing.

```
;
   ;Output file is called "net.cap"
;
if( (outf = simRunDirOutfile("net.cap")) == nil then
        close( inf )
           return( nil )
)
```

7. The `sim.cap` file is parsed, formatted, and printed to the `net.cap` file.

```
;
   ;Parse the sim.cap file
;
while( (line = lineread(inf)) != nil
            if( listp(line) then
            if( (! stringp(car(line))) ||
                      (! numberp(cadr(line))) then
               sprintf(message "ERROR: illegal file
                  format, %s"
                 "expecting \"<net name>\" <value> - ")
                        simPrintError( message )
                        simPrintErrorLine(line)
                        sprintf(message "Line ignored.\n")
                        simPrintError( message )
           else
                 ;
                       ; INSERT YOUR CODE HERE TO:
                       ; translate the net name into the
                    ;  mapped name, scale the
                       ; capacitance if necessary and print
                 ; out the information
                       ; according to the format of your
                 ; simulator
                       ;
           )
       )
   )
```

The `lineread` function returns a line that looks like:

```
( "<net name>" <capacitance> )
```

and returns `t` for a comment line. Thus, the `listp` function filters out the comment lines.

The `<net name>` entry corresponds to the full net pathname of the schematic design. Mapping is needed from the `<net name>` to the name generated by the netlister. The `simNetCdsNameExtName` function performs the mapping, for example,

```
extName = simNetCdsNameExtName("/cpu/alu1/control")
```

The name generated by the netlister is stored in the `extName` variable. Refer to the "Customizing the Simulation Environment (SE)" chapter for more information on name mapping and the `simNetCdsNameExtName` function.

Insert the formatting and printing code after the `else` keyword. The code depends on the capability of your simulator. If your simulator can adjust the fanout loading, due to the capacitance on the net, the net capacitance (obtained from the `sim.cap` file) can be added to the net directly. Otherwise, you should add an additional grounded capacitor for each net in the `sim.cap` file. Furthermore, you should adjust the unit of the capacitance if your simulator does not use `farads`. Finally, you may round off the adjusted result depending on the precision of your simulator.

In the following example, the formatting code fragment for a simulator that can adjust the fanout loading on each net is shown:

```
extName = simNetCdsNameExtName(car(line))
fprintf(outf ".LOAD %s=%.1f\n" extName
   cadr(line)/simCapUnit)
```

The simulator operates in fanout units. Thus, the unit of the capacitance must be converted from `farad` to fanout units. You use the `simCapUnit` variable to do the conversion. Finally, the adjusted fanout unit is rounded off to the nearest tenths.

In another example, the following is the formatting code fragment for adding a grounded capacitor for each net capacitance in the `sim.cap` file:

```
extName = simNetCdsNameExtName(car(line))
capDeviceCount = 0
capVal = round(cadr(line)/1.e-15)
fprintf(outf "CAPACITOR(%d) IC%1d(%s);\n" capVal
   capDeviceCount++ extName)
```

To avoid a naming conflict with the existing instance names, these added capacitors are prefixed by `IC` and suffixed with a unique number (`capDeviceCount`). In this case, the grounded capacitor unit is `femtofarad`, and it must be an integer. Thus, conversion is by dividing the capacitance by 1.e-15 and rounding to the nearest integer.

8. Close the name mapping process.

```
;
; Clean up for name mapping
;
```

```
simPostNameConvert()
```

9. Finally, the `sim.cap` and `net.cap` files are closed and the function returns `t`.

```
close ( inf )
close ( outf )
return ( t )
```

The following are steps to include this new function.

1. Instruct SE to execute this function during the post-layout simulation process. You should modify the `simActions` variable to include the new function name. This variable is defined in the file which has the same name as your simulator plus the `.ile` suffix and is located in the *`install_dir`*`/tools/dfII/local/si/caplib` directory.

   The `simActions` variable might look like this:

```
;
;Set the default actions to be performed by your ;simulator
;
simSetDef( 'simActions,'(simCheckVariables()
   simInitRunDir()
   netlist()
   YourSimTranslation()
   simin()
   runSim()
  )
```

   The new function (`YourSimTranslation`) should come after the `netlist` function because your function needs the names generated by the netlister to perform the name mapping. Similarly, the new function should come before the `simin` function because the `simin` function merges the translated result (`net.cap`) with the netlist description. You may add more functions to this list; however, you must preserve the order of the functions `netlist()`, `YourSimTranslation()`, and `simin( )`. The “Customizing the Control File” section in this chapter describes the process of merging the `net.cap` file and the netlist file.

2. Add `YourSimTranslation` to the list of functions to be unbound when the designer switches simulators. Modify the `simSimulatorUnbindFuncs` variable to include the new function. This variable is in the file with the same name as your simulator plus the *”*`.ile`*”* suffix and is located in the *`install_dir`*`/tools/dfII/local/si/caplib` directory. You might want to include other functions in the `simSimulatorUnbindFuncs` list. Refer to the “Customizing the Simulation Environment (SE)” chapter in this manual for more information on this variable.

Customizing the Control File

Use the `control` file to merge the translated result (`net.cap`) with the netlist description. The `control` file contains information such as the netlist description, the input stimuli, the simulation time points, and any formatted net capacitance `(net.cap)`. Use the command character "?" to include the content of the `net.cap` file because this file may not exist. The command character "!" is also used to include a file. However, it generates an error message if the file does not exist. For example, the template `control` file for the `SILOS` simulator looks like this:

```
batch batch.out
input netlist
input .term
$ This is the silos control template file ”control.sil”
[?net.cap]
[!silos.inp]
!type errors
[!silos.sim]
!type errors
.end
exit
```

The location of the [`?net.cap`] line depends on your simulator. In most cases, the [`?net.cap`] line should come after the netlist inclusion because you use the contents of the `net.cap` file to modify or add to the netlist description.

You should create a template control file similar to the preceding sample file for your simulator so that future simulation runs include the post-layout simulation capability. In addition, all current control files associated with your simulator should be modified to include the [`?net.cap`] line in the appropriate locations. Refer to the "Template Control File" section of the "Customizing the Simulation Environment (SE)" chapter in this manual for more information related to the `control` file.

Variables

This section lists all variables used in the post-layout simulation process. You can set the values of these variables in the `.simrc` file in the *`install_dir`*`/tools/dfII/local` directory, the directory you start the software in, or your home directory. Refer to the “Customizing the Simulation Environment (SE)” chapter in this manual for more information.

simDoPostLayout

Enables/disables the post-layout simulation process. If the value is `nil`, then the process is disabled; otherwise, the process is enabled.

Defined in:

`etc/skill/si/simcap.ile`

Default: `nil`

simCapFileDir

Defines the full file system pathname of the net capacitance `(sim.cap)` file.

Defined in:

`NONE`

Example:

`"/mnt/usr/test/sim.cap"`

simCapUnit

Converts the capacitance unit to the unit used in the target simulator.

Defined in:

`etc/skill/si/simcap.ile`

Default: `1e-15`

Functions

The following section lists all functions used in the post-layout simulation process for the supported simulators. Refer to the “Customizing the Simulation Environment (SE)” chapter in this manual for more information.

simCheckHeader

Refer to chapter “Customizing the Simulation Environment (SE)” chapter in this manual for more information.

9

Generating a Library

Before you can generate a netlist for input to your simulator, you must create a complete library of simulation primitives for your simulator. You must have a *symbol* view for each simulation primitive in the library to use during schematic entry. You must also have a *simulation* view for each simulator that is being integrated into the simulation environment.

The simulation view contains formatting information for the Simulation Environment (SE) netlisters. The view may also contain property definitions for default simulation attributes, (such as delays for logic simulation and timing analysis, and transistor widths and lengths for circuit simulation). Depending on how you use the primitives in the Cadence system, you can create other views of them, for example, the *layout* view and the view.

System-supplied subcircuits are also included in the library. A *symbol* view for each subcircuit is usually used for schematic entry. There can be a *schematic* view associated with each subcircuit symbol in the subcircuit design. Whether a schematic view is required or not depends on the simulator. For example, if the simulator can handle subcircuit definitions explicitly (and system-supplied subcircuits can be predefined and saved), a schematic view is not necessary.

To assist with building your own library, Cadence provides the Symbol/Simulation Library Generator (S/SLG) utility program as part of the schematic entry package. S/SLG can generate symbols for schematic entry, as well as generate simulation views. S/SLG reduces the effort required for creating and maintaining the symbol and simulation library. Refer to the *Virtuoso Schematic Editor L User Guide* for details about S/SLG.

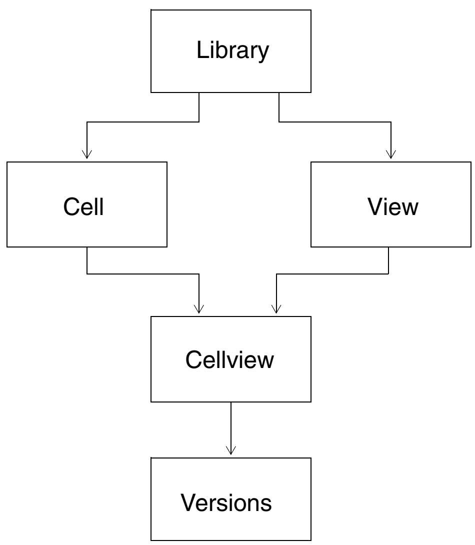
This chapter reviews the library structure used by the Cadence simulation environment and describes the procedure for creating and maintaining libraries for specific simulators.

Library Structure Overview

A library is a collection of design objects. There are two types of libraries: the reference library and the design library. Both libraries have the same internal structures and are accessed through the same mechanism. Design objects in the reference library are verified elements, and you can consider them as standard components. Reference libraries are usually shared among various designs. Although reference libraries can change, they are much more stable than the design objects you are developing. Conversely, design objects in the design library are dynamic, and subject to frequent change.

A design library is organized in four levels, as shown in Figure 9-1 on page 266.

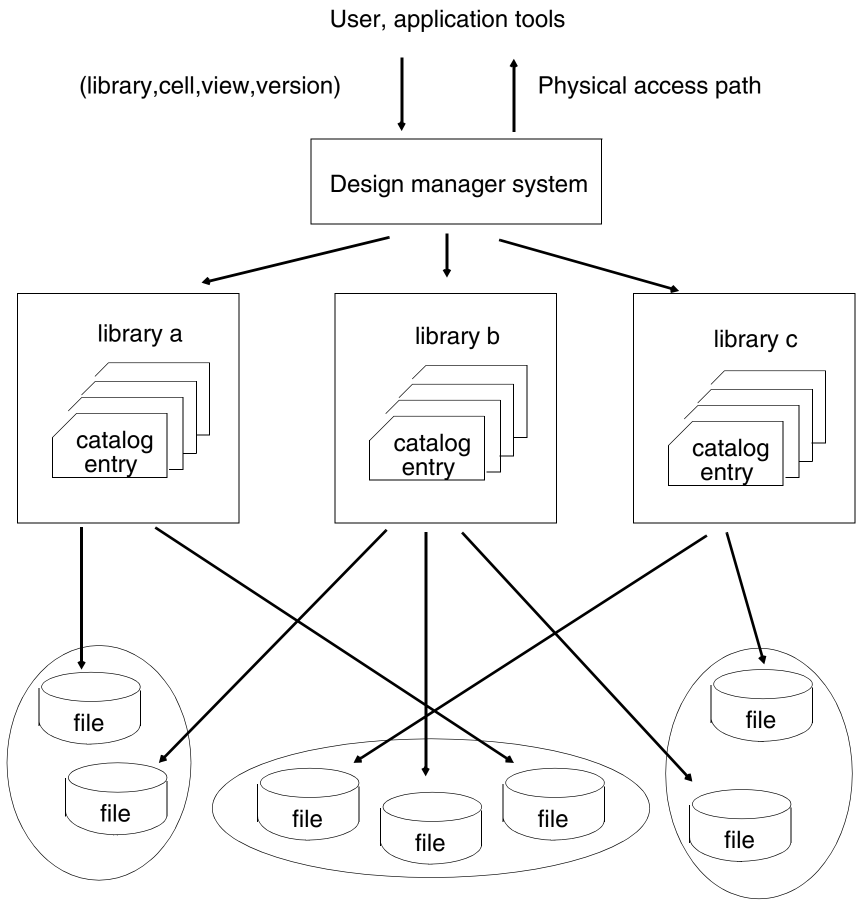
Figure 9-1 Structure of Cadence Model Library



- On the first level, a design library is identified by its location and its name.
- On the second level, a design library can contain a set of cells (NAND gate, NOR gate, etc.) and a set of views (schematic, layout, etc.).
- On the third level, a cellview is a data file that is created in association with a cell and a view.
- On the fourth level, version information is maintained about each cellview.

Design data is accessed through a library mapping scheme. Each library keeps a catalog of a cell, view, and version information, together with the access path of the data files. Data files can reside in different directories, or even on different nodes. To access a specific data file, you only need to refer to its logical name, which is defined by the items (library, cell, view, and version). Figure 9-2 on page 267 shows the interaction between you and the library.

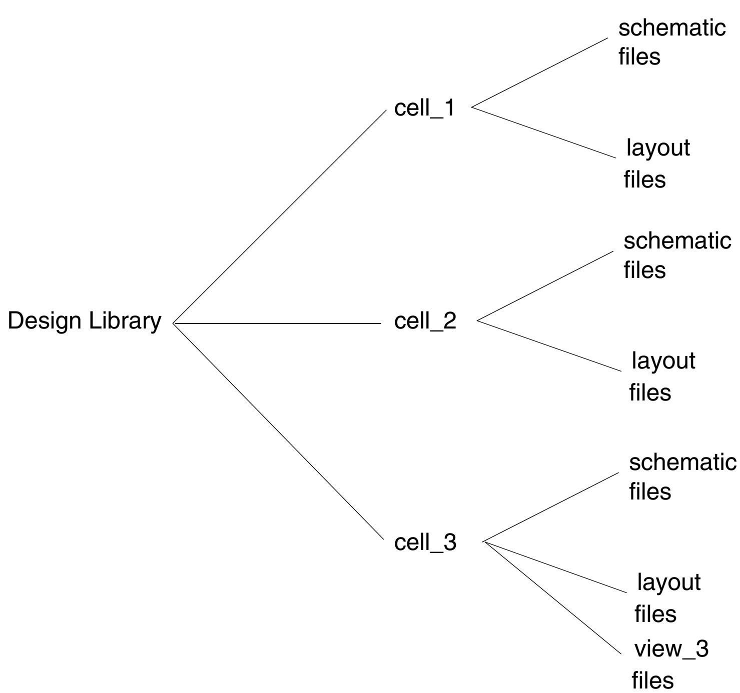
Figure 9-2 Interaction With the Library



Data Organization

Design library data is located in a system directory, where library files and design management information are kept. Design library provides a logical view of the library data files, and a central location for accessing design data. All design objects are referenced through logical names (e.g., 2-input-NAND). The system maintains a mapping from the logical name to its physical representation (i.e., the directory path. The directory structure of the design library has a default organization, as shown in Figure 9-3 on page 268.

Figure 9-3 Design Library Directory Structure



Library Directory Contents

The design library directory contains:

- one subdirectory for each cell in the library except those cells which are relocated to other locations.
- one library file, which keeps high-level summary information about the design library. Typical objects stored in the library file are cell catalog, version information of cells, cell status, and technology data.

- one directory for storing library attached files.
- audit trail files.

Logical Name to Physical Path Name Mapping

A logical name is used to identify a design object for both application programs and end users. The logical name of a design object consists of three components:

- cell name
- view type (e.g., schematic, layout...)
- version number (provided by you or the application programmer, or determined by the design management system).

With these three components and a predefined domain (library identifier), the design management system will automatically translate a logical name into a path name leading to the physical storage location of the data file. In addition to the default cellview mapping scheme, you are allowed to have your own customized name mapping function attached to the library through provided SKILL functions. This allows you to fully control the data file allocation scheme.

Cell

The library directory contains cells which are devices to be used in a design. The devices can be the primitives of your simulator. They can also be higher-level building cells, that is, subcircuits made up of primitives or other higher-level building cells. Whether a cell is a simulator primitive or a subcircuit depends on the simulator. Different views of the same cell (schematic and layout) are stored under the same directory so that information associated with a cell is grouped together. Library files containing high-level summary information of design data are stored in the library directory. Cells of a design library are directories one level below the design library. Views of all cells are stored in files in the cell directory.

Each cell is a subdirectory under the design library. The cell subdirectory name is the cellname. Under the cell subdirectory are various views of the device. The cellname should reflect the device name. During schematic editing, the system searches under the callname for symbols for device instances. For netlisting, the SE netlister searches for cells that are device instances in a schematic. Once a cell is located, the SI netlister retrieves netlist formatting information germane to the simulator.

Views

Each device view contains information on that device, and is a subdirectory under the device cell directory. The viewname (the view subdirectory) should reflect the information that the view contains. The Cadence system uses other views. For example, the layout editor uses the *layout* view. The simulation *views*, *spice*, *silos*, *hilo*, and *ta* are for netlisting to simulators that are integrated into the Cadence simulation environment. When you integrate a new simulator, you must create a simulation view for it for each simulation primitive supported by the simulator.

Version

Under each cellview are various versions of the view. Each version of the device is a file under the view directory. The version name is the name of the file. Each version may contain a different version of the device, and the latest version is called “current.”

Library Element Views

Certain views are required when you integrate a simulator to the Cadence environment. This section summarizes the views you must create to invoke a library for that simulator.

Symbol Views

You need to create a cell in the library for each simulation primitive of the simulator being integrated. To design circuits in terms of the simulation primitives, you must create a symbol view for each of them. During schematic editing, symbols of the simulation primitives can be instantiated for building the schematic.

Simulation Views

For each new simulator being integrated into the Cadence environment, you need to define a simulation view for each device that is a primitive of the simulator. For example, in the Cadence sample library, there is a `silos` view for each SILOS primitive of the SILOS simulator. The viewname should be the same as the simulator.

Each simulation view contains the appropriate netlist formatting properties for the element. These formatting properties enable the SE netlisters to produce netlists with the required syntax. Netlist formatting properties can be either an `nlpExpr` or an `ilExpr` property. For example, the *silos* view of the AND2 gate has the following netlisting properties:

```
Input_Pin_List  =  nlpExpr("[@lfA:%*][|A]
   [@lfB:%*][|B]") NLPElementPostamble  =
    nlpExpr("[@SILOS_AND_Image]") Pin_Net_Map  =
   nlpExpr("\\n$ 1 [|Y]=Y [|A]=A [|B]=B")
hnlSilosFormatInst  =  "hnlSilosPrintLogGate(\"AND\")"
```

The required netlist formatting properties and the syntax for format instructions to the SE Flat and Hierarchical netlisters are discussed in the "Customizing the Hierarchical Netlister (HNL)" and "Customizing the Flat Netlister (FNL)" chapters in this manual.

In the simulation view, you can also specify default values for netlist parameters used by the simulator, such as propagation delays and rise and fall delays.

Subcircuit Primitive Views

When you design with subcircuit primitives, in effect you are creating hierarchical designs. A particular simulator may or may not have facilities for handling subcircuit primitives. For simulators such as SILOS, you can define subcircuit primitives and save them in a file. Then you can invoke them in the netlist as if the subcircuits are primitives of the simulator.

For simulators that can handle subcircuit primitives, treat them like simulation primitives in the library. For each subcircuit primitive you create a simulation view of the cell. This view should contain netlist formatting properties for the subcircuit primitive invocation. The view can also have default delays similar to the simulation view for a simulation primitive.

If your simulator cannot handle subcircuit primitives explicitly, you can enter the subcircuit primitive definitions as a schematic view in the Cadence system. You should give this view a special name to designate it as a subcircuit primitive associated with the specific simulator. In particular, this name is required if the library is to be used by multiple simulators with different levels of abstraction in modeling their primitives.

In the Cadence-provided sample library, the *cmos.sch* views are subcircuits defined in terms of circuit-level primitives. For example, the AND2 cell contains an *and2/cmos.sch* view which is a transistor-level circuit of the AND2 gate. These views are used by simulators such as SPICE and HSPICE. There are views called *gate.sch*. For example, the *aoi21/gate.sch* view is a gate-level circuit for the complex AOI gate. These views are subcircuits defined in terms of logic-level primitives and are used by simulators such as HILO3™. Notice that AOI21 is a primitive for SILOS; therefore, there is also a *silos* view for the AOI21 gate. When the SE netlisters generate a netlist for SILOS, the aoi21 gate is output as a component. On the other hand, when a HILO3 netlist is generated, the AOI21 gate is output as a subcircuit.

The view switching mechanism in SE enables the SE netlisters to expand the design hierarchy for schematics containing high-level building cells. As discussed in the "Customizing the Flat Netlister (FNL)" chapter in this manual, the `simViewList` and `simStopList` SE variables control the hierarchy expansion process during netlisting.

Declare the *subcircuit* view a member of the `simViewList`. Then, the netlisters can switch into the subcircuit view during expansion, if required. The precise algorithm for view switching is discussed in the "Customizing the Flat Netlister (FNL)" chapter in this manual. For example, if the following declarations for the SE variables `simViewList` and `simStopList` are entered in the SPICE simulator,

```
simViewList  =  '("spice"  "cmos.sch"  "schematic")
simStopList = '("spice")
```

the netlister stops at cells that have a *spice* view. If there is no *spice* view, it expands a *cmos.sch* view. If neither a *spice* nor *cmos.sch* view exists, the *schematic* view is expanded.

Creating Your Own Library

Cadence supplies a standard sample library to support Cadence tools. This standard library is located in the Cadence hierarchy under the `etc/cdslib/samples` directory. One way to build a library for your target simulator is to augment the Cadence sample library with information germane to your simulator. This requires adding new cells and symbols for the simulator primitives that are not already in the Cadence sample library. It also requires creating simulation views for all simulation primitives of your simulator.

You can also create your own stand-alone library. You need to create a cell for each simulation primitive. You should create a symbol view and simulation view for each cell. If you use FNL to create a netlist, you should also create an `nlpglobals` cell, as outlined in the "Customizing the Flat Netlister (FNL)" chapter in this manual. Make sure that the full path names to your libraries are specified in the cell search path.

The S/SLG program supports library creation and maintenance. Use S/SLG for symbol and simulation view generation. Store S/SLG commands for symbol and simulation view generation in a library command file and load them into the S/SLG program for execution. You can run S/SLG either within the Cadence graphics environment or in the UNIX environment. To load an S/SLG command file inside the Cadence graphics environment, you can type in a single command (or a sequence of commands), press [RETURN], and that command (sequence of commands) is immediately executed (refer to the *Design Entry* manual for the complete procedure). When you prepare a library command file containing a set of S/SLG or IL commands, you can load this file into the S/SLG program. Each command in the file is sequentially executed. To load a file, type the command

```
lmLoadData( "myFile.lm" "myLib" "" "" "a" )
```

The `lmLoadData` command is used to load a given S/SLG command file when running the library management program in the Cadence environment; `myFile.lm` is the filename; `myLib` is the library name you are working on.

The `lmLoadData` function opens the specified library before loading the command file and closes the library when completed. It provides you with a quick way to execute all commands in the command file, however, you must use `lmOpenLib()` before invoking any S/SLG commands and `lmCloseLib()` when you are done.

The S/SLG program can be run as a stand-alone program in the UNIX environment. To run commands other than `lmDefCell`, run S/SLG in the UNIX environment to display output quickly. You can also switch between the stand-alone S/SLG program and the UNIX environment. Refer to the Virtuoso Schematic Editor L for more information.

Symbol Generation

You create a *symbol* view by drawing a graphic symbol of a device using the Cadence schematic editor. This process lets you create different shapes for device symbols. The procedure for creating symbols with the Cadence schematic editor is discussed in the Virtuoso Schematic Editor L guide.

Conversely, you can use S/SLG for symbol generation by using its `lmDefcell` command for drawing symbols. For example, to create a symbol view for an AND2 gate with inputs `in1` and `in2` and output `out`, use the command shown here:

```
lmDefcell( and2
input(in1 in2)
output(out)
)
```

You can also add other information to the symbol view. Refer to the Virtuoso Schematic Editor L for the syntax and applications of the `lmDefcell` command.

View Generation

Use the S/SLG `lmSimView` command to create simulation views. For example, to generate a SILOS view for an AND2 gate using the symbol view of the AND2 gate as a template, use the following command:

```
lmSimView(  AND2 symbol silos
Input_Pin_List  =  nlpExpr("[@lfA:%*][|A] [@lfB:%*][|B]")
NLPElementPostamble  =  nlpExpr("[@SILOS_AND_Image]")
Pin_Net_Map  =  nlpExpr("\\n$ 1 [|Y]=Y [|A]=A [|B]=B")
hnlSilosFormatInst  =  "hnlSilosPrintLogGate(\"AND\")"
)
```

In this example, the `Input_Pin_List`, `NLPElementPostamble`, and `Pin_Net_Map` properties are specified for the Flat Netlister (FNL), and the `hnlSilosFormatInst` property is specified for the Hierarchical Netlister (HNL). For the details on the use of `simView`, refer to the section *Library Management Commands* of *Appendix C* in Virtuoso Schematic Editor L.

The template view (the symbol view in the example), provides terminal information of the device for the simulation view. The terminal declarations must be identical between different views of a cell. The connections between hierarchy and view-switching in netlisting and probing are made by the use of terminal names.

Library Maintenance

Because S/SLG can load text command files, you can add and/or modify the properties specified in a view by editing the text command file, then reloading that file to S/SLG. You can maintain a library by maintaining the text file that created the library.

Furthermore, you can add new properties to a view. For example, to add the display scaling factor to the AND2 symbol view, use the S/SLG command `lmDefViewProp` as shown here:

```
lmDefViewProp(  and2 symbol
"DBU per UU"  =  160.0
userUnits  =  "inches"
)
```

We recommend that you save the S/SLG command files you used to create and modify your library for future library maintenance and update. If you drew your library component symbols manually, your command file should not contain any symbol creation commands. The command file should contain only commands for creating simulation views and for adding properties to views. Save the manually drawn symbols in a separate stand-alone library so that they can be copied for future library update.

Library Update Procedures

The procedure for upgrading the symbol and simulation library to a new release of Cadence software is explained below.

Updating a Modified Sample Library

Use the following procedure to upgrade the Cadence sample library with the modifications you made to the previous version:

1. Save your old sample library to a temporary location.
2. Install the new Cadence sample library.

3. If you drew symbols manually for devices you added to the Cadence sample library, copy them from the saved library into the new sample library.
4. If necessary, modify the S/SLG command file to comply with new Cadence software requirements.
5. Invoke S/SLG and load the command file to add simulation views to the newly-released Cadence sample library.

   If you drew your symbols manually, they are preserved in the library because your command file does not contain symbol creation commands. Conversely, if you let S/SLG draw the symbols for you, the symbol creation commands are part of the command file, and a set of additional symbol views are recreated.

Updating a Stand-Alone Library

If you created stand-alone libraries, the library upgrading procedure is as follows:

1. Update the S/SLG command file you used to create the stand-alone library to conform with any new Cadence software requirements.
2. Save a copy of this stand-alone library to a temporary location.
3. Invoke S/SLG and load the command file to recreate the library.

   Similar to updating a Cadence sample library, if you drew your symbols manually, they are preserved in the library because your command file does not contain any symbol creation commands. If you let S/SLG draw the symbols for you, the symbol creation commands are part of the command file, and a set of symbol views are recreated in your library.

Example of Updating a Cadence Sample Library

For an example of how to update a sample library, assume that you added new symbols for your own devices and also created new views for your own simulator. The original Cadence sample library is depicted in Figure 9-4 on page 276. The modified library is shown in Figure 9-5 on page 276, where you added the devices `mydev1`, `mydev2`, and the view `mysim` into the Cadence sample library. The manually drawn symbols for `mydev1` and `mydev2` were saved in the stand-alone library `mylib`. The contents of `mylib` are depicted in Figure 9-6 on page 277.

Figure 9-4 Example of the Cadence Sample Library

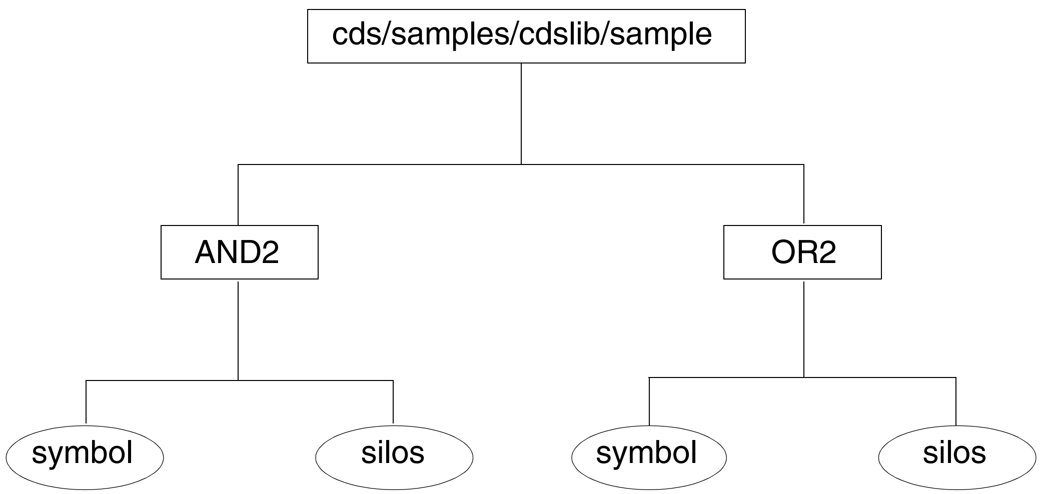


Figure 9-5 Example of the Modified Cadence Sample Library

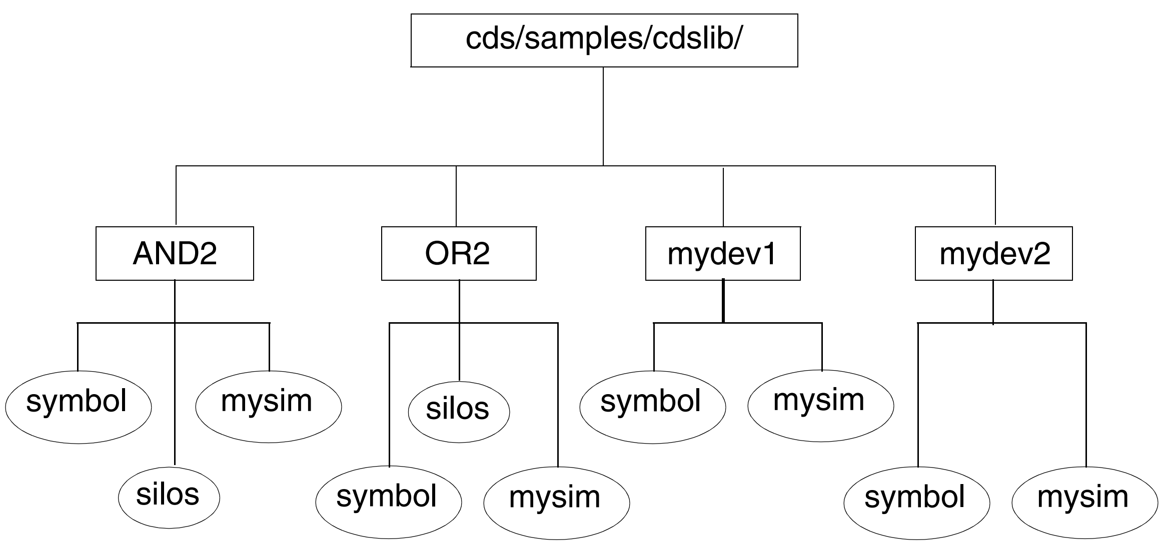
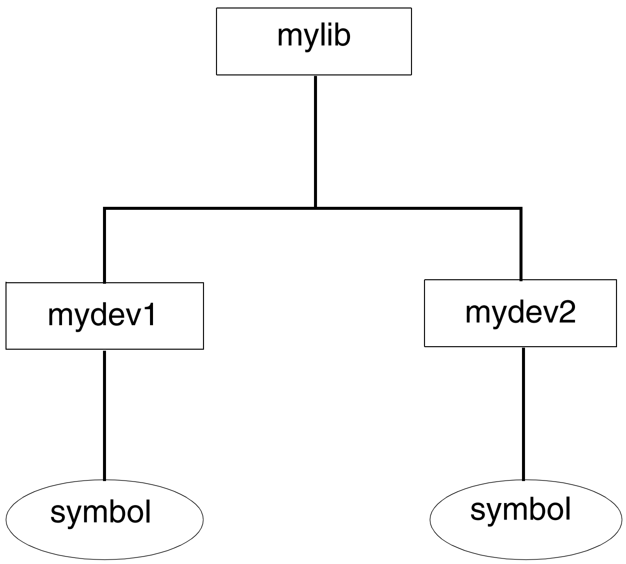


Figure 9-6 Example of a Saved Symbol Library



Use the steps outlined in the “Updating a Modified Sample Library” section in this chapter to update the new Cadence sample library for your simulator. For this example, follow these steps:

1. Save the current Cadence sample library in `cds/samples/cdslib/sample` to a temporary location.
2. Install the new release of the Cadence sample library into `cds/samples/cdslib/sample`. (Refer to Figure 9-4 on page 276.)
3. Change your directory to the location of `mylib` and merge the symbols in `mylib` with the newly released Cadence sample library. (Refer to Figure 9-6 on page 277.) You can use the `rdist` command to accomplish this, as follows:

   ```
   rdist  -c  mylib  cds7:/cds/samples/cdslib/sample
   ```

4. If required, modify the S/SLG command file that generates the *mysim* views for the `AND2`, `OR2`, `mydev1`, and `mydev2` devices to conform with new Cadence software requirements.
5. Change your directory to `cds/samples/cdslib/sample`. Invoke the `lm` program, and load the command file to add the *mysim* view to the sample library.

A

Support for HED

Hierarchy Editor (HED) lets you view many levels of a single design using table/tree views. It also helps you create and update configurations to traverse the design hierarchy; change global library, view, and stop lists; change cell and instance bindings and so on. This appendix chapter details the ways in which OSS supports HED.

For more information on these new features, see the *Cadence Hierarchy Editor User Guide*.

OSS had been supporting standard features of HED, such as cell/instance bindings and Bind to Open.

From 5.2.51, OSS also supports the following features of HED.

- Nested/Sub-Configurations – A nested configuration, also known as a sub-configuration is a configuration that is defined within another configuration. A sub-config can be nested at any level in a parent configuration.
- Occurrence Binding – Occurrence bindings are configuration rules that are defined at the occurrence level. An occurrence is an object that is defined by the full path from the top-level design to that object. In the hierarchy editor, setting any of the following attributes identifies the object as an occurrence:
    - Occurrence binding, that is, library, cell, and view binding
    - Occurrence stop point. See the subsection Occurrence Level under Stop Points.
    - Occurrence-Level Bind-to-Open. You can specify that an occurrence is unbound, that is, it is not bound to a specific library, cell or view, by setting a bind-to-open attribute on it. The bindings for the occurrence can be set later by other tools that use the configuration.
- Stop Points – A stop point on a design unit prevents the design unit from being expanded when the hierarchy is expanded. It can be applied at three levels:
    - Cell level – A stop point on a cell prevents the cell from being expanded when the hierarchy is expanded.

Note: A stop point on a cell applies to all occurrences of the cell.

- ❑ Instance (within a cell) level – You can specify a stop point on a single instance within a cell to prevent the instance from being expanded when the hierarchy is expanded.

  Note: A stop point on an instance can apply to multiple objects. If the cell that contains the instance is used in multiple places in the design, the stop point applies to the instance in all these places.

- ❑ Occurrence level – An occurrence stop point is a stop point on a specific path and applies only to one instance in the design. If an object has already been defined as an occurrence, when you add a stop point you are automatically adding it to the occurrence and not to the instance.

Cell and instance level stop points may be specified using the `nlAction` property on a cell and instance respectively. There is no other method to specify an occurrence stop point.

Note: The occurrence/instance stop point feature lets you add a stop point at one instance of a master while the other instance can continue to be a hierarchical cell. With the current implementation, OSS adds a cell to the stop cell list if it encounters a stop point on some occurrence/instance of the master. After this, if one of the occurrences/instances (of the same master) is encountered without a stop point, the cell is also added to the hierarchical cell list. Thus, the netlister determines whether to treat the cell as a stop cell or a hierarchical cell.

OSS supports these features when the SKILL flag `simSupportNewConfig` is set for netlisters.

APIs Modified to Support these Features

The following APIs have been modified to support these new HED features.

- `hnlMapCellModuleName`
- `hnlWriteBlockControlFile`
- `hnlMakeNetlistFileName`
- `hnlMapCellName`
- `hnlGetGlobalModelMappedName`
- `hnlCellInAllCells`
- `hnlMapModelName`

Bind to Open

OSS also supports Hierarchical Database “bind to open” (or in other words “bind to a NULL design unit”).

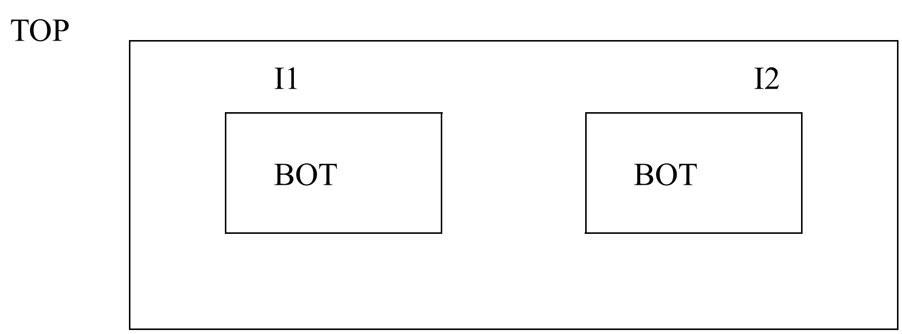
The binding specified through HED (Hierarchical Editor) can bind an instance to a NULL design unit or to a library/cell/view. The instances that are bound to a NULL design unit are termed as “bind to open”.

The following describes the changes in the respective packages to handle this feature.

- HNL: The Hierarchical Netlister (HNL) will not contain instance declaration for those instances which are bind to a NULL design unit, since OSS will ignore all such instances.
- FNL: The Flat Netlister (FNL) will not contain instance declaration for those instances which are bind to a NULL design unit, since OSS will ignore all such instances.
- MAP: The mapping functions for such instances will now generate errors since the instances which were bound to a NULL design unit were ignored by OSS.
- GENLD2: For the Genld APIs, there will be no change. Error message will be given for such instances.

Handling of views without DFII-DB data by OSS

Consider an example in which cell `TOP` has `2` instances, `I1` and `I2` of `BOT` cell. Both these reside in library `LIB`. The cell `BOT` has instances of stopping cells.



Example 1.

- The available view for TOP is `schematic.`
- The available views for BOT are `text, schematic and symbol.`

Consider the scenarios in which `text` view could be created.

Scenario 1: By Virtuoso import tools, it will have all files (`pc.db, master.tag, BOT.v` and `BOT.oa`), but `master.tag` file will have entry for BOT.v rather than BOT.oa. So, this OA data is sometimes called shadow OA.

Scenario 2A: By non-Virtuoso import tools (e.g: `ncvlog -use5x`), though the 5.X directory structure will be created (`lib/cell/view`) and supporting files in view (`pc.db, *.v` etc.), but OA data (`*.oa`) required by OSS (DFII-DB) will not be created.

Scenario 2B: By non-Virtuoso import tools, but the design is opened and saved. This will create OA data and also overwrite 'pc.db' file.

OA has a characteristic of explicitly binding instances to a `LIB:CELL`, which is an OSS characteristic too, only view can be switched either through config or without config.

The binding of instances to views can done in one the following ways:

Case A: Without using config i.e. simply specify `simViewList("text" "schematic" "symbol")`.

Case B: Use Hierarchy Editor to create a config for TOP and bind `I1` to 'text' view and `I2` to schematic (explicit binding).

Case C: Use Hierarchy Editor to create a config for TOP, and do not bind `I1` to any view. Instead, associate it with global view list (implicit binding) and `I2` to schematic (explicit binding).

Table A-1 : Behavior of OSS in all scenarios

	Scenario 1	**Scenario 2A**	**Scenario 2B**
Case A	OSS will first try to open the TOP text view. If for any reason, it is unable to open it, it will move to the next available view without giving any warning or error message.	As OA data is missing for text view, OSS will move to the next available view without giving any warning or error message.OSS cannot give any warning or error message, because this will be a costly operation without much value add.	OSS will first try to open the TOP text view.If for any reason, it is unable to open it, it will move to the next available view without giving any warning or error message.

Table A-1 : Behavior of OSS in all scenarios

	Scenario 1	Scenario 2A	Scenario 2B
Case B **Case C**	OSS will try to open the `text` view. If it is unable to open the view, then it will give an error message.	OSS will give an error message as there is explicit binding and no OA data is present.	OSS will try to open the `text` view. If it is unable to open the view, then it will give an error message.

Note: The views generated by the Hierarchy Editor are read differently by Diva and Assura. Although both read a part of the configuration and the environment variables, Assura overrides the device into a file, while Diva overrides or ignores the config file information.

B

Troubleshooting: Bus Direction

Determining a Bus Direction

A bus direction is determined on the basis of the ranges of ascending and descending spreads of a bus. OSS compares these ranges and the spread having the widest range determines the bus direction. For example, in case of two nets, `A<0:5>` and `A<6:0>`, bus direction is descending because the `A<6:0>` net has the widest range. If the ranges are same, OSS checks if any net matches the spread to determine a bus direction. For example, in case of these nets, `A<0:5>`, `A<5:0>`, `A<8:7>`, and `A<0:8>`, bus direction is ascending. Although the nets, `A<5:0>`, and `A<8:7>` when combined give the `<8:0>` spread, the `<0:8>` net is the exact match.

If OSS is unable to resolve a bus direction, a conflict arises, and a warning message is displayed on the screen. A conflict arises in the following cases:

- when both, ascending and descending spreads are same and no net matches the spread. For example, the following are the nets in a design:
    - `A<0:2>`
    - `A<4:6>`
    - `A<6:3>`
    - `A<1:0>`

  In this example, the nets having the same order are combined. This means `A<0:2>` and `A<4:6>` nets are combined to give the `A<0:6>` spread. Similar, `A<6:3>` and `A<1:0>` nets are combined to give the `A<6:0>` spread. The comparison between the spreads, `A<0:6>` and `A<6:0>` results in a conflict because both spreads have the same range and there is no matching net in a design.

- when both, ascending and descending spreads are same and two or more nets match the spread in reverse directions. For example, the following are the nets in a design:
    - `A<0:2>`
    - `A<6:0>`

- `A<4:6>`
- `A<6:3>`
- `A<0:6>`
- `A<1:0>`

In this example also the nets having the same order are combined similar to the previous example. This results in two spreads, `A<0:6>` and `A<6:0>`. This results in a conflict because both spreads have the same range. When matching nets are searched, two nets, `A<0:6>` and `A<6:0>` are found. This results in a conflict because these nets match the spread in reverse directions

Resolving a Conflict in Bus Direction

To resolve a conflict in bus direction, OSS allows you to specify bus direction using the *`hnlSetBusDirectionDescending`* variable. You can specify the value of the *`hnlSetBusDirectionDescending`* variable as t in the `.simrc` file. This indicates that the bus direction is descending. If the value of the variable is set to nil or it is not defined, the bus direction is determined as ascending, by default.