Saber® Managing Symbols and Models User Guide

Version Z-2007.03, March 2007

SYNOPSYS®

Printed in the U.S.A.

Saber® Managing Symbols and Models User Guide, Z-2007.03

Contents

1

Introduction to Managing Symbols and Models

If you have reason to create a custom model or symbol, or to maintain or create a custom library, or must use other symbols (from another vendor, for example), then this topic will be helpful.

The following topics describe the basics of managing symbols and models in a custom library:

- Important Definitions
- Typical Scenario for Symbol and Model Usage
- Why Do Symbols and Models Need to Be Managed?
- Using Supplied Vs. Other Symbols
- Why Keep a Custom Model Library?

Important Definitions

The following definitions are very useful for the discussions in this topic. It is helpful to read this small section even if you are familiar with these terms. Different manufacturers within the EDA industry define terms differently. For a more complete list of modeling terms refer to the Glossary. Terms used in a definition, that are also defined, appear in italics.

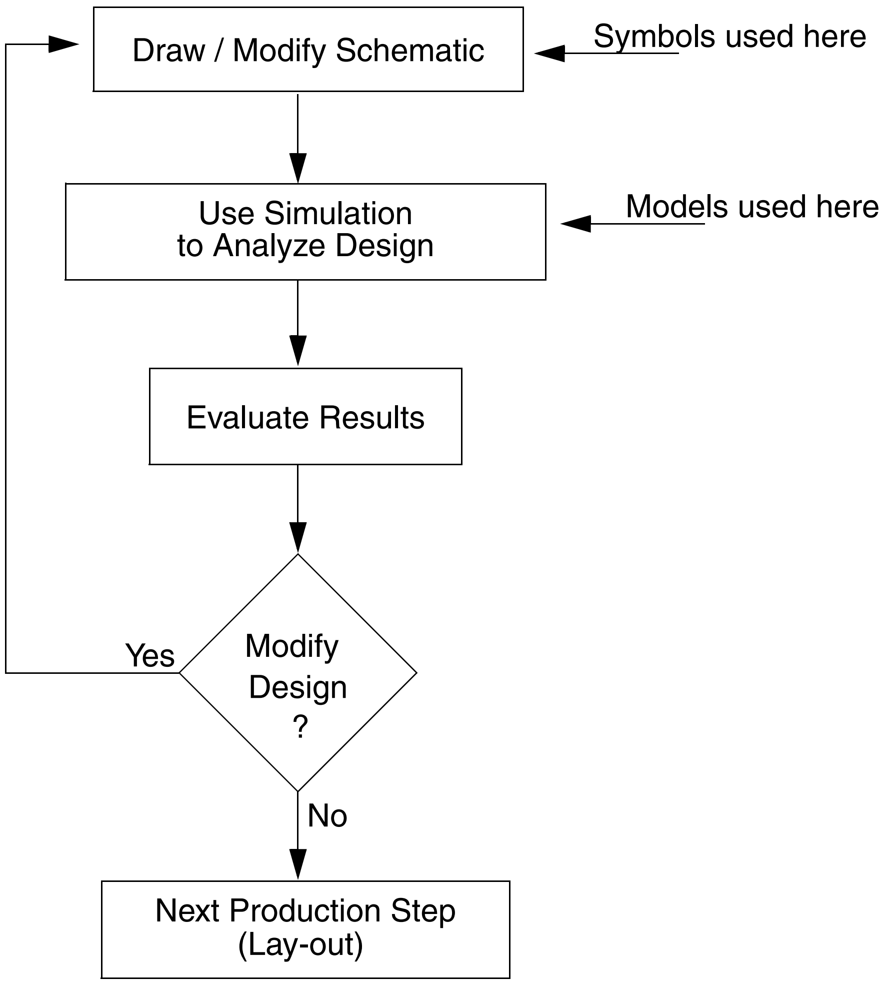
Template	A template is a text file that contains the model description, written in the MAST language, for use in simulation. A template models a general class of parts. Parameters must be supplied to model a specific part. See component.

Component	1) A component is a model that has been characterized to represent a more specific system element. A component usually corresponds to a commercially-available part. A component usually passes appropriate parameter values to a more general template. 2) A template instance in a netlist is also referred to as a component. See template.
Symbol	A symbol is the graphic object used in a schematic capture tool to represent a system element (part). It is used only to define the pin connections to the other system elements. It does not model the behavior of a part. However, a symbol can pass properties to a template. Sometimes referred to as base-symbol or instance-symbol, the former refers to the symbol as a file, the latter refers to the symbol as it is placed in a schematic. There can be many instances of the same base symbol in the same schematic.
Model	Model is a loosely defined term. It is predominantly used in the colloquial sense to mean some representation (mathematical equations) that approximates the behavior of a real or imagined system element. Somewhat interchangeable with template, model is the more general term. Model can also mean 1) either a template or a component; 2) simply a list of parameters that, when applied to a particular template, turns it into a component for a particular part (a parameter list is often the type of model that is supplied by some of the parts manufacturers).
Part	1) A part consists of all the information needed to describe a system element. This information includes the model (either template or component), any underlying parameters, and the symbol. 2) A part sometimes refers to the physical device.
Property	A property is a type of variable that is part of a symbol. Properties are used to characterize the symbol, often for programs outside of the graphic editor. Properties of symbols may be passed-on to the parameters of a model. See attribute.
Attribute	Attribute is generally interchangeable with property. Some schematic capture vendors refer to symbol variables as attributes, most vendors refer to them as properties. This document refers to them as properties. See property.

Parameter	A parameter is a variable that is part of a model (template) and is therefore needed for simulation. A parameter is generally (but not necessarily) a coefficient of a model equation. Parameter is often used interchangeably with argument. Parameters are not limited to numeric values.
Port	A port is an input or output connection point of a model or symbol, but, more often, it refers to symbol connections. See pin.
Pin	1) A pin is the name for a connection point of a template to a netlist. 2) A pin is a generic term for any connection point. See port.
Netlist	A netlist is a text file (.sin extension) that is an input to the simulator. A netlist is a description of a design that lists each element of the design, its values, and its interconnections with other design elements. A netlist can be created by hand but is more typically the output of a program called a netlister, which combines information from a schematic, the models, and a mapping file (if used).

Typical Scenario for Symbol and Model Usage

The following figure illustrates the basic steps for analyzing a design using the simulator.



Symbol and Model Usage

The schematic is typically the source that describes the interconnection between the design elements. (The interconnections can also be described by directly writing your own netlist. This method is cumbersome and, in most cases, is not necessary.) These design elements are represented in the schematic with graphical objects called symbols. These elements are also represented in the simulator by models (templates or components), which describe the behavior of the design elements. There must be a direct correspondence between a design element's symbol and its model.

A program called a netlister translates the schematic diagram (along with other information) into a file called a netlist which describes how all the design elements are connected. This netlist is the circuit description understood by the simulator.

As just described, a design element's symbol and its model must have a one-to-one correspondence. Managing this mapping between symbols and models is a main topic of this topic.

Why Do Symbols and Models Need to Be Managed?

In many cases symbols and models do not need to be managed. If you are using only supplied symbols and models, then you do not need to read this topic. We offer an extremely large and growing library of very high quality models (components and templates) and all of the symbols necessary for these models. These symbols are available for creating designs with the Saber Saber Sketch schematic editor as well as for each of the Frameway integrations into other manufacturers schematic editor tools. If you have reason to create a custom model or symbol, or to maintain or create a custom library, or must use other symbols (from another vendor, for example), then this topic will be helpful.

Reasons for Using Other Symbols

The following are some reasons for using other symbols:

- Convention.
  Your company prescribes that all symbols must conform to some internal standard. This is the most common reason for using other symbols. Usually this convention is in place to maintain compatibility with another EDA tool (or group of tools) such as layout.
- Compatibility.
  You have a large installed base using a particular type of symbol, and you need to maintain compatibility with existing symbols.
- Custom Part.
  You must create a custom symbol because there may be no adequate symbol available to correspond to a custom model you have created.

Reasons for Using Custom Models

The following are some reasons for using or creating custom models:

- New Part.
  You need to model a new (or un-characterized) part, one that does not exist in the supplied libraries.
- Hierarchy.
  You have used hierarchy to create a symbol for a sub-circuit schematic.

Using Supplied Versus Other Symbols

Large portions of this topic are devoted to the various techniques of properly associating symbols with their models (the mapping task). Using un-altered symbols is the easiest way to avoid the mapping task. If you use only supplied symbols, then mapping does not need to be considered. All mapping has already been done. If you must use other symbols, or wish to modify them, see Chapter 4: Choosing a Mapping Technique.

Availability of Supplied Symbols

Saber Sketch schematic provides flexibility, power, and superior ease of use. We also offer three integrations into other manufacturer's schematic capture tools. These Frameway integrations are available for Mentor Graphics, Cadence, and Viewlogic. Frameway integrations allow you to use another schematic capture tool while still making use of MAST models, the Saber Simulator, and other tools.

If you purchased Sketch, then you received a large library of symbols. All the symbols are compatible with the Saber Simulator. These are loaded under the $SABER_HOME directory when Saber is installed and should be directly accessible from the Parts Gallery tool in Saber Sketch

If you purchased one of the Frameway integrations, then you also received a complete set of symbols that are compatible with both the particular schematic capture tool (Mentor Graphics, Cadence, or Viewlogic) and the simulator. Using one of these schematic capture tools, you will have the choice of using symbols originally provided by the tool, symbols provided parts vendors, or using the symbols provided with the Frameway integration. Using supplied symbols is the easiest way to avoid the mapping task. These symbol libraries are loaded under the $SABER_HOME directory when your Frameway is installed.

Whether you are using Saber Sketch or one of the Frameway integrations, the supplied symbols should be directly accessible from your library search tool. The actual locations of these symbol libraries are as follows:

- Saber Sketch:
  `$SABER_HOME/symbol/sketch`
- Frameway integration into the Mentor Graphics environment:
  `$SABER_HOME/framework/falcon/symbols`
- Frameway integration into the Cadence environment:
  `$SABER_HOME/framework/artist/symbols`
- Frameway integration into the Viewlogic environment:
  `$SABER_HOME/framework/viewlogic/symbols`

Why Keep a Custom Model Library?

If you are working on a design that requires a very specialized part for which no model exists, then you will need to create one. You may wish to keep this model in a library just for that project or you may wish to add it to a larger company wide library. You may have inherited a custom library that contains known working models, and you would like to maintain them.

Information in this book will be helpful if you are creating, maintaining, or adding to a personal project library or your company library. This book discusses the criteria you may wish to consider before structuring your library, as well as the steps required to create both a model and the library. If you decide to create a custom library, please see the following; Structuring Your Custom Model Library. If you have already created your custom library and simply wish to add additional models, please see Chapter 3: Creating and Adding Models to Your Custom Library.

2 Structuring Your Custom Model Library

This topic discusses how to establish a location for your custom models. In many cases this will have already been done. If this is the case, then you only need to know the names of your custom model directories and you can add your new models to them. If these directories have not been set-up and you wish to add custom models, then you should read this topic. The basic steps for establishing a library are as follows:

- Create directory locations for your parts.
- Modify your search paths to include these directories.
- Add your model files to the library directories and make them available to your schematic capture tool.

Before you create these directories and modify your path, you should read this entire topic. It is a short topic and lists important considerations that will save time and frustration.

The remainder of this topic elaborates on each of the above three steps in the sections listed below. A short procedure for accomplishing each step is included at the end of each section.

Creating Your Part Directories
Modifying Your Search Paths
Adding Models to Your Library Directories

Creating Your Part Directories

The first step is to create the directories that will contain your new templates, components, and symbols. The following sections describe some considerations to keep in mind when structuring these new directories:

Consider Directory Names

When choosing names for your directory structures, you should use names that will aid in finding or distinguishing among them. Also, remember that your future needs may grow. You should choose names that will allow for expansion.

Consider Internal Conventions

Directory structure and naming conventions may already be in-place (formally or informally) in your company. Paying attention to these company protocols may aid in finding or distinguishing among these directories in the future.

Limit the Number of Directories

The number of directories you can add to your search path is limited only by your operating system and how it handles a combination of these search paths. However, breaking up your library into many smaller libraries (directories) makes your search path longer and complicates the management task. Also, storing your libraries in too few directories can make organization difficult to follow. We recommend a maximum of 10 directories. Here is an example of a simple and workable directory structure for your libraries.

```
/CustomLibs/Templates
/CustomLibs/TemplatesBeta
/CustomLibs/Symbols
/CustomLibs/SymbolsBeta
```

Procedure for Creating Your Part Directories

Use your operating system commands to create symbol and template directories that will store your new models. Enter the following lines at the command line prompt:

```
mkdir /CustomLibrary
cd /CustomLibrary
mkdir Symbols
mkdir SymbolsBeta
mkdir Templates
mkdir TemplatesBeta
```

These instructions make simple assumptions about the names and structure you have chosen, substitute your own model directory structure for that used in these instructions.

Modifying Your Search Paths

Once your library directories are in place, you must modify your search paths to allow the directories to be found.

The following sections describe how Saber applications find data files and shows how to modify the appropriate variables to include your newly created directories.

How the Applications Find Files
Procedure for Modifying Your SABER_DATA_PATH Variable

How the Applications Find Files

These applications look for files containing data they need in directories along the data search path, as listed in the following table in the order listed. For example, the first directory to be searched is the working directory.

	Saber Sketch	**Saber Simulator**	**Description**
1	.	.	Working directory where the application was started.
2	AI_SCH_PATH (Locates directories that contain custom symbols.)	SABER_DATA_PATH (Locates directories that contain custom templates and components)	Environment variable that you set to point to proper location(s).
3		SABER_HOME/config	Directory to hold configuration information specific to a site.
4	Directories and subdirectories in install_home specific to each application		

If there are multiple files with the same name in the data search path, Saber applications use the first one encountered. Your models will be found as long as they are in one of the locations listed above.

However, if you have created a library of custom models that you would like to be available for general use, the proper search path location for your directories is as part of the original SABER_DATA_PATH environment variable (or AI_SCH_PATH in the case of Saber Sketch finding symbols).

1. The working directory is the first location that is checked along the data search path. For quick-test purposes, it can be convenient to place library items in the current directory. You should not rely on this technique for long-term storage of your libraries, as the current directory may change depending on where the Saber application was invoked.
2. Templates and components are found by the Saber Simulator using the SABER_DATA_PATH environment variable. The SABER_DATA_PATH variable is a list of directories separated by a colon, a space, or both. Any custom libraries intended for use by others at your site should be stored in a directory that is part of SABER_DATA_PATH. If such a directory does not exist, you should create one and add its path to this variable.

Note:

Never point SABER_DATA_PATH to SABER_HOME.

3. Symbols are found by your schematic capture tool using whichever mechanism is provided with your particular tool (Saber Sketch, Design Architect, Artist, or ViewDraw).
4. Saber Sketch searches the value of the AI_SCH_PATH environment variable to search for directories containing symbols. The AI_SCH_PATH variable is a list of directories separated by a colon, a space, or both. Any custom symbols intended for use by others at your site should be stored in a directory that is part of AI_SCH_PATH. If such a directory does not exist, you should create one and add its path to AI_SCH_PATH. If AI_SCH_PATH does not exist, you should create it.
5. The $SABER_HOME/config directory holds configuration information specific to a site. Do not place any libraries in this directory.
6. The last place an application will search are the additional directories that are appended by the application. These are the homes for supplied data. For the Saber Simulator these directories are $SABER_HOME/bin, then $SABER_HOME/template/*, then $SABER_HOME/component/*/*.
7. Although technically possible, we do not recommend adding custom library directories to $SABER_HOME, or mixing custom templates with the MAST templates in any of the $SABER_HOME directories. Future upgrades may destroy the path to your custom libraries if they reside in a directory found by using $SABER_HOME.

Precompiled files (.sld files) created using the saber -p option are not found by using the search path shown in Table 2-1. They are found by using the list of directories contained in your PATH variable. Precompiled (also called preloaded) model files have priority over all other models.

Procedure for Modifying Your SABER_DATA_PATH Variable

Modify the SABER_DATA_PATH environment variable in your user start-up file to specify your new directory pathnames. If the variable is not present, then create it.

If your SABER_DATA_PATH environment variable includes directories that are provided with this software, you can remove these directories from the list. For example, directories containing template or component libraries provided with the Saber Simulator should not be included in the SABER_DATA_PATH environment variable.

Note that you must be careful when you use the wildcard (*) character to include directories in the SABER_DATA_PATH environment variable. If too many directories are included in the SABER_DATA_PATH environment variable, some files may not be found by the Saber Simulator or other applications.

The following sections provide specific instructions for users of UNIX and Windows NT:

UNIX Users
Windows NT Users

UNIX Users

In the following examples, substitute your own model directory structure for the place-holders (template_directory and dir).

Shell & File	**SABER_DATA_PATH Definition**
C .cshrc	If a SABER_DATA_PATH environment variable does not exist in your .cshrc file, enter the following line anywhere in the file: setenv SABER_DATA_PATH template_directory You may include more than one directory by specifying a colon separated list as follows: setenv SABER_DATA_PATH dir1:dir2:dir3
Bourne or Korn .profile	If a SABER_DATA_PATH environment variable does not exist in your .profile file, enter the following lines anywhere in the file: SABER_DATA_PATH= template_directory export SABER_DATA_PATH You may include more than one directory by specifying a colon separated list as follows: SABER_DATA_PATH=dir1:dir2:dir3 export SABER_DATA_PATH

To re-initialize your start-up file, log out and log in to your computer. (You do not need to reboot your system.)

If a SABER_DATA_PATH environment variable already exists in your .cshrc or .profile file, you can modify it to include the new directory.

Windows NT Users

The SABER_DATA_PATH environment variable can be defined as either a system or a user variable or both. As a system variable it will be available to all users. As a user variable it will only effect the environment of the single user, but it will take precedence over the system variable.

If SABER_DATA_PATH exists, modify it to include the paths to your new directories. If SABER_DATA_PATH does not exist, create it, using the paths to your new directories as the value.

To change or add a system SABER_DATA_PATH environment variable you must have system level permissions, contact your system administrator.

To change a user SABER_DATA_PATH environment variable, click on SABER_DATA_PATH in the User Environment Variables list. The Variable field displays SABER_DATA_PATH and the Value field displays a colon-separated list of paths to directories (the value of SABER_DATA_PATH). You can edit the Value field directly.

To add a user SABER_DATA_PATH environment variable, type SABER_DATA_PATH in the Variable field and type the colon separated list of paths to your new directories in the Value field, as follows:

Variable	SABER_DATA_PATH
Value	C:/CustomLibrary/Templates; C:/CustomLibrary/TemplatesBeta

When you have finished with your modifications, click on the SET button to save your changes, and the OK button to close the System window.

Adding Models to Your Library Directories

Once your directory structure is in place, you can add models to them as they are developed. Simply use your operating system commands to add model files to your library directories as you would any other file or directory. Naming of these files is the main consideration when adding models.

This section assumes you have a fully working, tested model (symbol and template), that you simply wish to add to your custom model library. For details on how to make your symbols and models work together, refer to Chapter 3: Creating and Adding Models to Your Custom Library.

The following sections provide details of the procedure:

Model Names
Procedure for Adding Models to Your New Library
Making Your New Model Available for Schematic Capture Tool

Model Names

Consider the name you are giving to your new symbol and template (i.e.: widget.sym and widget.sin). Does another model already exist with the same name (widget)? If more than one model exists with the same name, only the first one in the search path will be found. However, by relying on the search path to make this choice, any future changes to your directories or search structure could result in finding an unexpected model.

Procedure for Adding Models to Your New Library

Once your directory structure is in place, you can add files to them as you develop new models.

Change to the directory that currently contains your new model files by entering the following line at the command line prompt:

```
cd path_to_directory_containing_new_models
```

Move these new model files (the example uses symbol and template files called widget.sym and widget.sin, respectively) to the library directories you created earlier, by entering the following lines at the command line prompt:

```
mv widget.sym /CustomLibrary/Symbols/widget.sym
```

```
mv widget.sin /CustomLibrary/Templates/widget.sin
```

These instructions make simple assumptions about the names and structure you have chosen. Substitute your own model directory structure for that used in these instructions (/CustomLibrary/Symbols and /CustomLibrary/Templates).

Making Your New Model Available for Schematic Capture Tool

The previous sections discussed how to: create directory locations for your parts, modify your search paths to include these directories, and add your model files to the library directories. Modifying your SABER_DATA_PATH environment variable allowed the Saber Simulator to find your new templates (or components).

Now you must make modifications to allow your schematic capture tool to find your new symbols. Each schematic capture tool has a different mechanism for allowing symbols to show-up in its symbol browser. Refer to your schematic capture tool documentation (Saber Sketch, Design Architect, Artist, or ViewDraw) for details. If you are using the Saber Sketch design editor follow the instructions in Making Symbols Available in Saber Sketch.

Making Symbols Available in Saber Sketch

To make symbols available in Saber Sketch, two steps must be accomplished.

1. Modify AI_SCH_PATH to point to your new symbol directories.
2. Add the part description to the Parts Gallery.

Saber Sketch finds symbols in the same way the Saber Simulator finds templates, except that it uses a different environment variable. You modify AI_SCH_PATH in the same way you modified SABER_DATA_PATH in the previous section: Procedure for Modifying Your SABER_DATA_PATH Variable, by following the same instructions but substituting “AI_SCH_PATH” for “SABER_DATA_PATH”, “SaberSketch” for “Saber Simulator”, and “path to symbols” for “path to templates.”

To add a part, you open Saber Sketch and click on the Parts Gallery button (on the tool bar) to open the Parts Gallery window. From the Parts Gallery window, you select the Parts dropdown menu, then you select the Add menu item to open the Add Part to Category window. You can browse the Category, Symbol, and Template fields until you have your part set-up the way you want it, then click on the Add button.

3
Creating and Adding Models to Your Custom Library

This section outlines the basic steps for adding a new model to your library. If you are merely modifying a model (or symbol), then some of these steps may not be relevant. The third step, associating the symbol with the model, is helpful if you are using other symbols (from another schematic editor) and wish to map them to the supplied models. Regardless of the schematic editor you are using or what circuit element you are creating, if no model exists, then the following four steps must be accomplished to add a model to your library.

Modeling the System Elements
Selecting or Creating a Symbol
Associating the Symbol with the Model (Mapping)
Saving your New Model in a Retrievable Location

Modeling the System Elements

This guide is devoted to managing models. Other documentation is available for designing models. This section outlines the various techniques for developing a new model, lists the strengths and weaknesses of each relative to the others, and refers you to locations for further study:

Other Sources of Existing Models
Parameterize a General Model (Characterization)
Hierarchical (Macro) Modeling
Translate a SPICE Model to a MAST Model
Graphical Modeling
MAST Modeling

Other Sources of Existing Models

The easiest and fastest technique for fulfilling a modeling need is to find one that already exists. It may be exactly what you need, or it may only need slight modification. Either way, you can save time by simply asking. The following is a list of some of the common sources for models.

- Synopsys Systems. Our list of models is always growing. Even if you don’t find the model you need in the library you purchased, it may be available in some of the optional libraries, or it may have been created since your original purchase.
- Inside your company. Ask your colleagues who have worked in a similar area or ask your system administrator. The model may already exist on another file system, network, or site.
- User Groups. Your colleagues outside of your company may have already created the model you need. The ASSURE users group offers many of these models in the public domain. Those that were verifiable at the time of your release have been supplied on your distribution CD; you can find them in $SABER_HOME/user_grp/template. For the latest models (available online) or to submit a model, call your applications engineer.
- Manufacturer or Vendor of the Part. Many offer at least crude models (simply a parameter list, see definition of model), if not complete models of their parts. If they don’t offer one, ask when it will be available. Remember that we also provide a SPICE to MAST translator (spitos).
- University. Many universities are creating models for this simulator. They may have models from past or present projects.

Advantages. This is usually the simplest and quickest way to achieve a working model.

Disadvantages. It requires a little effort, and you still may not find a suitable model.

Parameterize a General Model (Characterization)

A simulation model for the symbol in a schematic can be either a component or a template. A component is a very specific model. A template is a more general model, which has parameters that can make it more specific (some or all of the parameters may have default values). If you use a template as your model, it may require that you assign values to some of its parameters. Parameter value assignment, in this case, is usually done on the symbol in the schematic.

Characterization is the process of converting a template (general model) into a component (specific model). A component usually corresponds to a specific physical part. Characterization is accomplished by giving values to the parameters of the template, usually including those with default values. Parameter value assignment, in this case, is done in a component file that calls the template and passes parameter values.

Some parts can be characterized more easily than others. For instance, some parameters can be found in a data book. Other parameters either cannot be found or are not reliable. Op-amps can usually be characterized from a data book, but transistors usually require empirical data (from a lab) to characterize the part accurately.

Advantages. This technique is quick and easy to accomplish if you have the correct parameter values. The existing template (such as an inductor with a series resistance) already includes all desired behavior. The resultant model is of high quality, and simulation runs quickly.

Disadvantages. This technique requires the proper existing template. There can be a large variation in the effort required to gain the proper parameter values.

Hierarchical (Macro) Modeling

You can group multiple existing models to create a new model. Your new model can have its own symbol, but for simulation purposes it is a group of sub-models. There are two ways this concept is used.

Within a single design, you may wish to group certain parts as a sub-schematic. Perhaps a group of parts is repeated many times in the design, or perhaps you just want to group some of the lower level details in order to clarify the higher level design. This kind of usage is referred to as hierarchical modeling.

You can also use this technique to model a part by using existing sub-models, even though those submodels are not part of the circuit. For example, you can model a transistor using resistors, capacitors, and dependent sources. This kind of usage is referred to as macro modeling. Macro modeling is also used (by definition) in SPICE-based simulation. Many of the macro models in MAST are the result of a conversion from a SPICE model.

Advantages. This technique is a quick and easy way to create a new model.

Disadvantages. This technique depends on the availability of underlying models. The resultant model may be inefficient or overly complicated, causing simulation to run slowly, as compared with an original behavioral model for the same part.

Translate a SPICE Model to a MAST Model

SPITOS translates from a SPICE model (netlist) to a MAST netlist. SPICE models exist only as netlists. They are constructed as combinations (macro models) of a handful of primitives (resistor, capacitor, sources, etc.). When translating from SPICE to MAST, you have the option of choosing primitives that exactly match the original SPICE primitives, or you can choose the improved (corrected) primitives (templates).

Advantages. If a SPICE model is the only one available, this is a quick and easy way to begin simulating.

Disadvantages. The resultant model may be inefficient or overly complicated, causing simulation to run slowly. It may also be less accurate than required. A behavioral MAST model for the same part can run faster and be more accurate.

Graphical Modeling

Graphical modeling allows you to build a model by adding or modifying equations on special symbols. This technique results in new models and is accomplished completely within your schematic editor.

Advantages. Creating the model is easy. There is no need to create a symbol. There is no need to learn a programming language. Simulation of models runs quickly.

Disadvantages. This technique is limited to higher abstraction models such as control systems. Models cannot be parameterized.

MAST Modeling

MAST is a very powerful modeling language. All of the supplied templates are written in MAST. All of the modeling techniques ultimately result in a MAST model of some form. This powerful language is provided with your purchase of the simulator. You can use MAST directly to create new behavioral models (templates).

Advantages. This is the most powerful of the modeling techniques. Any model that can be conceived can be created.

Disadvantages. Since you will be writing in a modeling language rather than using an automated modeling system (like Graphical Modeling), MAST modeling will generally (though not necessarily) take longer.

Selecting or Creating a Symbol

A symbol is a graphic object. If you are using a schematic capture tool to enter your design, then you need a symbol to represent your model on the schematic. You can choose from libraries of existing symbols or create any shape you wish to represent your part. The shape should be useful for quick and easy identification on the schematic, but otherwise has no meaning. To work correctly with the simulator, the symbol must also represent the correct number and type of connections and the correct properties to correlate with its template. The following sections discuss these procedures:

Using an Existing Symbol
Creating a Custom Symbol

Using an Existing Symbol

A symbol can exist in two places, in a symbol library as a base symbol or in a schematic as a symbol instance. There can be many instances of the same base symbol in a single schematic.

It is not necessary to create a new symbol for your part, even if your part uses a new template. You can choose from among the vast existing libraries of symbols for an appropriate graphic representation of your part. You can then copy that symbol to a new file and modify its properties to work with your new model. Refer to the section on Creating a Custom Symbol.

Using a Supplied Symbol

Supplied symbols come with Saber Sketch as well as with each of the three Frameway integrations (for Mentor Graphics, Cadence, and Viewlogic). Regardless of your schematic editing tool, these symbols can be found in the $SABER_HOME directories loaded from the distribution CD. For more information refer to Chapter 1: Availability of Supplied Symbols.

Supplied symbols are already mapped. Therefore, by using these symbols, the mapping task can usually be avoided.

Using a Symbol from Another Schematic Capture Tool

If you are using a schematic editor from Mentor Graphics, Cadence, or Viewlogic, then you have the symbols provided with those tools in addition to the symbols provided with your Frameway integration. For reasons to use these symbols, refer to Chapter 1: Reasons for Using Other Symbols. If you need to use other symbols, pay close attention to the section Associating the Symbol with the Model (Mapping).

Creating a Custom Symbol

If you cannot find an existing symbol to represent your new part, you may want to create a custom symbol. You use a symbol editor (either Saber Sketch or, if you are using a Frameway, one from Mentor Graphics, Cadence, or Viewlogic) to create the graphic. You must also make sure the symbol ports and properties are compatible with the template. For more information on making your symbol compatible, you should pay close attention to the section Associating the Symbol with the Model (Mapping). For instructions on how to create a symbol, refer to your schematic capture documentation or online help.

Associating the Symbol with the Model (Mapping)

The software includes a program called a netlister. The netlister is a translator that generates an input file called a netlist by interpreting the relationship between the properties of each schematic symbol and the parameters of its associated template. (Netlists are sometimes called .sin files. While all netlists are .sin files, many .sin files are not netlists. In all cases, .sin files conform to the MAST language rules.)To accomplish this netlist, there must be a correlation between the symbols used in the schematic and the corresponding models in the library. The table below lists the three mapping requirements, that is, the three areas that must correlate between a symbol and a model.

Symbol	Model
Name of the Symbol	Name of the Model
Properties of the Symbol	Parameters of the Model

Symbol	**Model**
Port Names of the Symbol	Pin Names of the Model

If you are using unmodified symbols, then these mappings have already been done for you. If you are using another vendor's symbols or custom made symbols, then you must cover these mapping requirements. There are three methods for accomplishing this mapping task. Each of these methods can be applied to any or all of the three mapping requirements. That is, you can mix-and-match the mapping methods between the mapping requirements, as long as all three of the mapping requirements are met. The three mapping methods are:

- Match the Names.
- Use Specially-Recognized Properties.
- Use a Mapping File.

For a brief explanation and comparison of these mapping methods, and useful criteria for choosing among them, see Chapter 4: Choosing a Mapping Technique.

For detailed information on using each of these techniques see:

Chapter 5: Using Name Matching to Map Symbols
Chapter 6: Using Specially-Recognized Properties for Mapping
Chapter 7: Specially-Recognized Properties Reference
Chapter 8: Using a Mapping File to Map Symbols
Chapter 9: Mapping File Reference

Saving your New Model in a Retrievable Location

Now that you have a new model and corresponding symbol, you must save them in a location where the tools can find them. Chapter 2: Structuring Your Custom Model Library discusses how to create these locations. Once you have a location (library directory or directories) use your operating system to move (or copy) your model files to the library directory or directories.

Consider the name you are giving to your new symbol and template (such as widget.sym and widget.sin). Does another model already exist with the same name (widget)? If more than one model exists with the same name, only the first one in the search path will be found. However, by relying on the search path to make this choice, any changes to your directories or search structure could result in finding an unexpected model.

4

Making User Templates Visible for UNIX

This topic describes the following:

How the Applications Find Files
Using Templates Written in MAST
Using Custom Models From Your Capture Tool
Using C or FORTRAN Routines Called by Templates (UNIX)

How the Applications Find Files

To make your own templates (or any other user files) available to the Saber Simulator or the other applications, you need to do one of the following:

- Place the files in a directory along the data search path where the applications will find them. (The data search path is described in this topic.)
- Use the appropriate environment variable to tell the applications where they are located as shown in the following table.

The applications look for files containing data they need in directories along the data search path, as listed in the following table in the order listed.

For example, the first directory to be searched is the working directory.

Data Search Path

Saber Sketch	Saber Simulator	Description
.	.	Working directory where the application was started.

Saber Sketch	Saber Simulator	Description
AI_SCH_PATH (Locates directories that contain custom symbols)	SABER_DATA_PATH (Locates directories that contain custom templates and components)	Environment variable that you set to point to proper location(s)
	install_home/config	Directory to hold configuration information specific to an installation.
Directories and subdirectories in install_home specific to each application		

If there are multiple files with the same name in the data search path, Saber applications use the first one encountered. Your models will be found as long as they are in one of the locations listed in the above table. However, if you have created a library of custom models that you would like to be available for general use, the proper search path location for your directories is as part of the original SABER_DATA_PATH environment variable (or AI_SCH_PATH in the case of Saber Sketch finding symbols).

1. The working directory is the first location that is checked along the data search path. For quick-test purposes, it can be convenient to place library items in the current directory. You should not rely on this technique for long-term storage of your libraries, as the current directory may change depending on where the Saber application was invoked.
2. Templates and components are found by the Saber Simulator using the SABER_DATA_PATH environment variable. The SABER_DATA_PATH variable is a colon-separated list of directories. Any custom libraries intended for use by others at your site should be stored in a directory that is part of SABER_DATA_PATH.

   If such a directory does not exist, you should create one and add its path to this variable.

   Never point SABER_DATA_PATH to install_home.

   Symbols are found by your schematic capture tool using whichever mechanism is provided with your particular tool (Saber Sketch, Design Architect, Artist, or ViewDraw).

   Saber Sketch searches the value of the AI_SCH_PATH environment variable to search for directories containing symbols. The AI_SCH_PATH variable is a colon-separated list of directories. Any custom symbols intended for use by others at your site should be stored in a directory that is part of AI_SCH_PATH. If such a directory does not exist, you should create one and add its path to AI_SCH_PATH. If AI_SCH_PATH does not exist, you should create it.

3. The install_home/config directory holds configuration information specific to an installation. Do not place any libraries in this directory.
4. The last place an application will search are the additional directories that are appended by the application. These are the homes for the software supplied data. For the Saber Simulator these directories are saber_home/bin, then saber_home/template/*, then saber_home/component/*/*.

Precompiled files (.sld files) created using the saber -p option are not found by using the search path shown in the table titled Data Search Path. They are found by using the list of directories contained in your path variable. For a procedure for modifying your path variable, refer to Step 3 in the topic titled Configuring for the UNIX Environment.

Precompiled (also called preloaded) model files have priority over all other models.

Using Templates Written in MAST

To use templates written in the MAST modeling language, you need to inform the software where they are located. The following methods can be used.

Method 1: Place the templates in a directory in the data search path. Once you have done this, the templates will be found by the applications when they are needed.

Method 2: Specify the directory containing the templates in an environment variable called SABER_DATA_PATH in your user start-up file. To add your own template library to the SABER_DATA_PATH environment variable, complete the following procedure.

1. Define or modify the SABER_DATA_PATH environment variable

Edit the appropriate file for your shell as shown in the following table:

Shell & File	SABER_DATA_PATH Definition

C
.cshrc
	If a SABER_DATA_PATH environment variable does not exist in your .cshrc file, enter the following line anywhere in the file:
	
	`setenv SABER_DATA_PATH "template_directory"`
	
	You may include more than one directory by specifying a colon separated list as follows:
	
	`setenv SABER_DATA_PATH "dir1:dir2:dir3"`

Bourne
.profile
	If a SABER_DATA_PATH environment variable does not exist in your .profile file, enter the following lines anywhere in the file:
	
	`SABER_DATA_PATH= "template_directory"`
	`export SABER_DATA_PATH`
	
	You may include more than one directory by specifying a colon separated list as follows:
	
	`SABER_DATA_PATH="dir1:dir2:dir3"`
	`export SABER_DATA_PATH`

In this table, template_directory is the full path name to the directory containing the templates or where dir1, dir2, and dir3 are full path names to three different directories.

If a SABER_DATA_PATH environment variable already exists in your .cshrc or .profile file, you can modify it to include the new directory.

If your SABER_DATA_PATH environment variable includes directories that are provided with the software, you should remove these directories from the list. For example, directories containing template or component libraries provided with the Saber Simulator should not be included in the SABER_DATA_PATH environment variable.

Use care when you use the wildcard (*) character to include directories in the SABER_DATA_PATH environment variable. If too many directories are included in the SABER_DATA_PATH environment variable, some files may not be found by the Saber Simulator or the other software applications.

2. Re-initialize your startup file

   To re-initialize your startup file, log out and log in to your computer. You do not need to reboot your system.

Using Custom Models From Your Capture Tool

You must make modifications to allow your schematic capture tool to find your new symbols. Each schematic capture tool has a different mechanism for allowing symbols to show-up in its symbol browser. Refer to your schematic capture tool documentation (Saber Sketch, Design Architect, Artist, or ViewDraw) for details. If you are using the Saber Sketch design editor use the following instructions.

Making Symbols Available in Saber Sketch

To make symbols available in Saber Sketch, two steps must be accomplished.

1. Modify AI_SCH_PATH to point to your new symbol directories.
2. Add the part description to the Parts Gallery.

Saber Sketch finds symbols in the same way the Saber Simulator finds templates, except that it uses a different environment variable. You modify AI_SCH_PATH in the same way you modified SABER_DATA_PATH.

To add a part, you open Saber Sketch and click on the Parts Gallery button (on the tool bar) to open the Parts Gallery window. From the Parts Gallery window, you select the Edit pulldown menu, then you select the New Part menu item to open the Create New Part window. You can browse the Category and Symbol fields until you have your part set-up the way you want it, then click on the Create button.

Using C or FORTRAN Routines Called by Templates

It is possible to create MAST templates that call routines written in FORTRAN or C. Such routines are called foreign routines. A procedure for incorporating such routines into a template is described in the Guide to Writing MAST Templates manual, topic titled Foreign Routines in MAST.

To make foreign routines available to the Saber Simulator, you complete the following procedure.

- Compile each foreign routine

  You must use one of the supported compilers listed in one of the tables titled Compatible SUN Compiler Versions, or Compatible HP-UX Operating System Compiler Versions, to avoid possible dynamic loading problems when trying to use a foreign routine.

To compile a FORTRAN routine, use the command for your system as shown in the following table.

Command to Compile a FORTRAN Foreign Routine

System	**Command**
Solaris	f77 -c -PIC -cg89 -dalign \ -ftrap=%none -xlibmil filename.f
HP-UX	f77 -c +Z filename.f

Replace filename with the name of the file you are compiling.

To compile a C routine, complete the following steps:

1. To find out if you need to add an underscore to the end of C routine names on your system, refer to the table titled Command to Compile a C Foreign Routine. If a trailing underscore is required, complete the following:

   In the file containing the C routine, add an underscore (_) to the end of the name of the routine in the header line of the routine.

   Do not add an underscore to the name of the file or to the name used in the MAST foreign command in your template to call the routine.

   For more detailed information on Foreign Routines, refer to the Saber MAST Language User Guide, Book 2.

2. Compile the C routine by using the command for your system shown in the following table.

   Command to Compile a C Foreign Routine

System	**Command**	**Trailing Underscore?**
Solaris	`cc -c -K PIC -cg89 \` `-dalign -ftrap=%none \` `-xlibmil filename.c`	yes
HP-UX	`cc -c +Z  filename.c`	no

   Replace filename with the name of the file you are compiling.

How to Make a Single Routine Available to the Saber Simulator

Once the subroutine has been created and compiled it must be made available to the Saber Simulator.

1. Make the compiled routine available to the Saber Simulator.

   Complete one of the following:

   - Place the compiled routine in a directory in the data search path. For more information on the data search path, refer to the topic titled How the Applications Find Files.
   - Use the procedure described in Step 1 and Step 2 to add the location of the compiled routine to your SABER_DATA_PATH environment variable.

2. Invoke the Saber Simulator

   Invoke the Saber Simulator by using the saber command and your usual command options (if any).

   In some cases, the Saber Simulator tries to automatically load subroutines into a simulation upon invocation. This can be the case when subroutines have been compiled but not linked to a library. If this is the case, the compiled subroutines will be in a file labeled filename.o, where filename indicates the original user-assigned subroutine file name. When started under these conditions, the Saber Simulator tries to dynamically link the filename.o files into the simulation by automatically issuing one of the following UNIX commands:

System	**Command**
Solaris	`ld -o filename.so -dy -G filename.o`
HP-UX	`ld -o filename.sl -b filename.o`

Multiple subroutine files are indicated by filename.o. Several different subroutines can be included in this list of file names. The single shared library file is indicated by filename.so (Sun) and filename.sl (HP).

How to Make a Library of Routines Available to the Saber Simulator

To make a library of routines available, perform these steps:

1. Compile the subroutines using the appropriate compiler.

   Refer to the table titled Command to Compile a FORTRAN Foreign Routine.

2. Link the compiled files together into a single shared library file.

   Once the subroutines have been compiled, they can be linked together into a single shared library file.

   To link multiple subroutines together, use one of the following UNIX commands:

System	**Command**
Solaris	`ld -o file.so -dy -G file1.o file2.o ...`
HP-UX	`ld -o file.sl -b file1.o file2.o ...`

   Multiple subroutine files are indicated by file1.o and file2.o ... Several different subroutines can be included in this list of file names. The single shared library file is indicated by file.so (Sun) and file.sl (HP).

3. Declare the shared library file as global

When several subroutines are combined to create a single shared library file, you will need to specify a SABER_GLOBAL variable at the operating system level. This variable needs to include the shared library file and make it available anytime the Saber Simulator is started. The Saber Simulator will then search the shared library file for any subroutines which are used but not found by other means.

Create the SABER_GLOBAL variable using the same method you used for creating the SABER_DATA_PATH variable, which is described in the table titled Data Search Path. You need to point the SABER_GLOBAL variable to the shared library file that was created in However, you must omit the .so file name extension. For example, if you created a file called my_lib_routines.so with the ld command, you need to set the SABER_GLOBAL variable to my_lib_routines.

4. Make the shared library file available to the Saber Simulator.

   Once you have created a shared library file and referenced it to the libai_saber.lib file, place the directory containing the shared library file in the SABER_DATA_PATH path variable, or place the shared library file in a directory contained in the SABER_DATA_PATH path variable.

5. Re-initialize your startup environment

   Reinitialize your start-up file by logging in to the machine (you may need to log out first).

```
login login_name
```

5

Making User Templates Visible for Windows

This topic describes the following:

How Applications Find Files
Making Symbols Available in Saber Sketch
Using Templates Written in MAST

Using C or FORTRAN Routines Called by Templates (Windows)

How the Applications Find Files

To make your own templates (or any other user files) available to the Saber Simulator or other applications, you need to do one of the following:

- Place the files in a directory along the data search path where applications will find them. (The data search path is described in this subsection.)
- Use the appropriate environment variable to tell the applications where they are located.

Applications look for files containing data they need in directories along the data search path, as listed in the following table in the order listed.

For example, the first directory to be searched is the working directory.

Data Search Path

	Saber Sketch	Saber Simulator	Description
1	.	.	Working directory of the design that the application is invoked on.

	Saber Sketch	Saber Simulator	Description
2	AI_SCH_PATH (Locates directories that contain custom symbols.)	SABER_DATA_PATH (Locates directories that contain custom templates and components)	Environment variable that you set to point to proper location(s).
3		saber_home\config	Directory to hold configuration information specific to a site.
4	Directories and subdirectories in saber_home specific to each application		

If there are multiple files with the same name in the data search path, Saber applications use the first one encountered. Your models will be found as long as they are in one of the locations listed in the Data Search Path table above.

However, if you have created a library of custom models that you would like to be available for general use, the proper search path location for your directories is as part of the SABER_DATA_PATH environment variable (or AI_SCH_PATH in the case of Saber Sketch finding symbols).

1. The working directory is the first location that is checked along the data search path. For quick-test purposes, it can be convenient to place library items in the current directory. You should not rely on this technique for long-term storage of your libraries, as the current directory changes depending on the location of the design that is being used by the application.
2. Templates and components are found by the Saber Simulator using the SABER_DATA_PATH environment variable. The SABER_DATA_PATH variable is a semicolon-separated list of directories. Any custom libraries intended for use by others at your site should be stored in a directory that is part of SABER_DATA_PATH. If such a directory does not exist, you should create one and add its path to this variable.

   The subsection titled Using Templates Written in MAST, describes how to define or modify a SABER_DATA_PATH environment variable. The AI_SCH_PATH environment variable can be modified in a similar way.

   Manually Creating Template Information Files describes how to update custom templates that do not have the proper permissions for a user. You must be a site manager with read and write permissions to use this feature.

Note:

Never point SABER_DATA_PATH to saber_home.

Saber Sketch searches the value of the AI_SCH_PATH environment variable to search for directories containing symbols. The AI_SCH_PATH variable is a semicolon-separated list of directories. Any custom symbols intended for use by others at your site should be stored in a directory that is part of AI_SCH_PATH. If such a directory does not exist, you should create one and add its path to AI_SCH_PATH. If AI_SCH_PATH does not exist, you should create it.

3. The saber_home\config directory holds configuration information specific to a site. Do not place any custom libraries in this directory.
4. The last place an application will search are the additional directories that are appended by the application. These are the homes for specific-supplied data. For the Saber Simulator these directories are

```
saber_home\bin, then saber_home\template\*, then
saber_home\component\*\*.
```

Do not place any custom libraries in this directory.

Precompiled files (.sld files) created using the saber -p option are found by using the list of directories contained in your Path variable. They are not found by using the search path shown in the Data Search Path.

Precompiled (also called preloaded) model files have priority over all other models. For more information on precompiled files, refer to the topic titled Predefined MAST Declarations.

To check the Path variable setting, do the following:

a. Navigate to, and start the System program:

   Start > Settings > Control Panel > System > Environment tab
b. Look at the System Environment Variable list for the Path variable.
c. Add the appropriate directories to the value.

Making Symbols Available in Saber Sketch

To make symbols available in Saber Sketch, two steps must be accomplished.

1. Modify AI_SCH_PATH to point to your new symbol directories.

2. Add the part description to the Parts Gallery.

Saber Sketch finds symbols in the same way the Saber Simulator finds templates, except that it uses a different environment variable. You modify AI_SCH_PATH in the same way you modified SABER_DATA_PATH.

To add a part, you open Saber Sketch and click on the Parts Gallery button (on the tool bar) to open the Parts Gallery window. From the Parts Gallery window, you select the Edit pulldown menu, then you select the New Part to open the Create New Part window. You can browse the Category, Symbol, and Template fields until you have your part set-up the way you want it, then click on the Create button.

Using Templates Written in MAST

To use templates written in the MAST modeling language, you need to inform the software where they are located. This description specifically refers to the SABER_DATA_PATH variable. The AI_SCH_PATH variable might also need to be set for custom symbols in Saber Sketch using the same procedure. The following methods can be used:

Method 1: Place the templates in a directory in the data search path. Once you have done this, the templates will be found by the applications when they are needed.

Method 2: Specify the directory containing the templates in an environment variable called SABER_DATA_PATH. To add your own template library to the SABER_DATA_PATH environment variable, complete the following procedure.

1. Define or modify the SABER_DATA_PATH environment variable

   In this example, dir1, dir2, and dir3 are full pathnames to three different directories.

```
SABER_DATA_PATH="dir1;dir2;dir3"
```

   To check the SABER_DATA_PATH variable setting, do the following:

   - Navigate to, and start the System program:

     Start > Settings > Control Panel > System > Environment tab

   - Look at the System Environment Variable list for the SABER_DATA_PATH variable.

- If it does not exist, create it and add the appropriate directories to the value.

If your SABER_DATA_PATH environment variable includes directories that are provided with the software, you can remove these directories from the list. For example, directories containing template or component libraries provided with the Saber Simulator should not be included in the SABER_DATA_PATH environment variable.

- Use care when you use the wildcard (*) character to include directories in the SABER_DATA_PATH environment variable. If too many directories are included in the SABER_DATA_PATH environment variable, some files may not be found by the Saber Simulator or other applications.

2. Re-initialize your startup environment

   To re-initialize your startup environment, log out and log in to your computer. You do not need to reboot your system.

Using C or FORTRAN Routines Called by Templates

It is possible to create MAST templates that call routines written in FORTRAN or C. Such routines are called foreign routines. A procedure for incorporating such routines into a template is described in the topic titled Foreign Routines in MAST which can be found in the Saber MAST Language User Guide.

To make foreign routines available to the Saber Simulator on a Windows system you must do the following:

- Insert the proper code in the header of each foreign routine
- Compile the routine on the Windows system
- Link multiple-compiled files into one file
- Set up environment variables so that the Saber Simulator can find the linked files

The C Language Header

If the C programming language is being used to create foreign routines for use with MAST and the Saber Simulator, the routine header must appear exactly as follows (substitute your foreign routine name for CROUTINE):

```
  __declspec(dllexport) void __stdcall   CROUTINE(double* 
inp,long*
 ninp,long* ifl,long* nifl,double* out,long* nout,long* 
ofl,
 long* nofl,double* aundef,long* ier)
 {
 }
```

The __declspec statement is important for Windows since it indicates that the routine is exported from the Dynamic Link Loader and can be found by the Saber Simulator. The __stdcall statement is used to indicate that this routine is called from FORTRAN with the FORTRAN calling conventions.

The CROUTINE string must be entered in upper-case characters.

The FORTRAN Language Header

If the FORTRAN programming language is being used to create foreign routines for use with MAST and the Saber Simulator, the routine header must appear exactly as follows (substitute your foreign routine name for FROUTINE):

```
subroutine 
FROUTINE(inp,ninp,ifl,nifl,out,nout,ofl,nofl,aundef,ier)
 !MS$ATTRIBUTES DLLEXPORT :: FROUTINE
 integer ninp,nifl,nout(2),nofl,ifl(*),ofl(*),ier
 real*8 inp(*),out(*),aundef
```

The ATTRIBUTES statement is important for Windows since it indicates that the routine is exported from the Dynamic Link Loader and can be found by the Saber Simulator.

The FROUTINE string must be entered in upper-case characters.

How to Make a Single Routine Available to the Saber Simulator

Once the subroutine has been created, it must be compiled to create the executable Dynamic Link/Load Library (DLL) file and then referenced to the Saber library. Both operations can be taken care of using the same command. The compiling and referencing operations are part of the C or FORTRAN language compilers and can be version-dependent.

One-Step Dynamic Library Linking

In some cases, the Saber Simulator tries to automatically load subroutines into a simulation upon invocation. This can be the case when subroutines have been compiled but not linked to a library. If this is the case, the compiled subroutines will be in a file labeled filename.obj, where filename indicates the original user-assigned subroutine file name. When started under these conditions, the Saber Simulator tries to dynamically link the filename.obj files into the simulation by automatically issuing the following command:

```
link /DLL /OUT:filename.dll filename.obj
saber_home\lib\libai_saber.lib
saber_home\lib\libai_analogy.lib
```

where saber_home is the software location. In a standard installation this is:

```
C:\<filename>\SaberDesigner5.2
```

This dynamic linking process, however, may not work if there are libraries which need to be included but are not part of libai_saber.lib or libai_analogy.lib. If this is the case, refer to the following sections titled One-Step C Language Compiling and Linking and One-Step FORTRAN Language Compiling and Linking depending on the programming language being used.

One-Step C Language Compiling and Linking

When the C programming language is used to create a subroutine, the following command must be used:

```
cl /LD filename.c saber_home\lib\libai_saber.lib

   saber_home\lib\libai_analogy.lib
```

Where saber_home is the software location. In a standard installation this is:

```
C:\<filename>\SaberDesigner5.2
```

where the name of the actual subroutine file, without extensions, is substituted for filename, and filename indicates the original user-assigned subroutine file name. The /LD command indicates a DLL file will be created. The resulting DLL file will be named filename.dll. For example, if the original C file was called adder.c, the resulting DLL file would be called adder.dll.

One-Step FORTRAN Language Compiling and Linking

When the FORTRAN programming language is used to create a subroutine, the following command must be used:

```
fl32 /LD filename.f

saber_home\lib\libai_saber.lib

saber_home\lib\libai_analogy.lib
```

where saber_home is the software location. In a standard installation this is:

```
C:\<filename>\SaberDesigner5.2
```

Where the name of the actual subroutine file, without extensions, is substituted for filename, and filename indicates the original user-assigned subroutine file name. The /LD command indicates a DLL file will be created. The resulting DLL file will be named filename.dll. For example, if the original FORTRAN file was called adder.f, the resulting DLL file would be called adder.dll. The %SABER_HOME% string is a path variable, set during the Saber software installation, which points to the location of the Saber program and its associated files.

How to Compile and Link Libraries of Routines

There may be situations where it is desirable to link several subroutines into a single DLL file, and then reference this file to a Saber library as shown in the following steps:

1. Compile the subroutines using the appropriate compiler.

   Compiling subroutines is a language-dependent operation.

   You must use one of the supported compilers listed in the topic titled Compatible Compiler Versions, to avoid possible dynamic loading problems when trying to use a foreign routine.

2. Link the compiled files together into a single DLL file.

Once the subroutines have been compiled, they can be linked together into a single DLL file. To link multiple subroutines together, use the following command:

```
link /DLL /OUT:dllname.dllfilename1.obj  filename2.obj
saber_home\lib\libai_saber.lib
  saber_home\lib\libai_analogy.lib
```

where saber_home is the software location. In a standard installation this is:

```
C:\<filename>\SaberDesigner5.2
```

The /OUT:dllname.dll command assigns a user-specified name to the resulting DLL file. Multiple subroutine files are indicated by filename1.obj and filename2.obj. Several different subroutines can be included in this list of file names.

3. Declare the DLL file as global.

   When several subroutines are combined to create a single DLL file, it is necessary to specify a SABER_GLOBA variable at the operating system level. This variable will point to the combined DLL file and make it available anytime the Saber Simulator is started. The Saber Simulator will then search the combined DLL file for any subroutines which are used but not found by other means.

   Set the SABER_GLOBAL variable as follows:

   - Navigate to, and start the System program:
     Start > Settings > Control Panel > System > Environment tab
   - Set the variable as follows:

```
Variable:   SABER_GLOBAL
  Value:    dllname
```

The Value entry field contains the name of the DLL file assigned in Step 2, but does not contain the .dll extension. More than one DLL file can be assigned to the Value by using a comma-separated list of file names. For example:

```
Variable:   SABER_GLOBAL
  Value:    dllname1, dllname2, dllname3
```

4. Make the combined DLL file available to the Saber Simulator.

   Once a DLL file has been created and referenced to the libai_saber.lib and libai_analogy.lib files, the directory containing the DLL file must be placed in the SABER_DATA_PATH path variable, or the DLL file must be placed in a directory contained in the SABER_DATA_PATH path variable. Use the following procedures to check and edit the SABER_DATA_PATH variable.

   Check or edit the SABER_DATA_PATH variable as follows:

   - Navigate to, and start the System program:
     Start > Settings > Control Panel > System > Environment tab
   - In either the System or User environment variable list box, an entry for SABER_DATA_PATH may appear. If it does not appear, create it. Enter the paths to the directories. If there is more than one path, list them and separate by colons.

5. Re-initialize your startup environment

   To re-initialize your startup environment, log out and log in to your computer. You do not need to reboot your system.

6
Choosing a Mapping Technique

This section outlines the criteria you should consider before choosing a mapping technique and compares the three techniques. It also contains six examples using each of the three techniques.

The following sections contain detailed instructions and reference material for using the three mapping techniques.

Chapter 5: Using Name Matching to Map Symbols
Chapter 6: Using Specially-Recognized Properties for Mapping
Chapter 7: Specially-Recognized Properties Reference
Chapter 8: Using a Mapping File to Map Symbols
Chapter 9: Mapping File Reference
Chapter 10: Using a Mapping File to Convert SPICE Symbols
Chapter 11: Using Cadence simInfo to Avoid Mapping Files

What is Mapping?

The software includes a program called a netlister. The netlister is a translator that generates an input file called a netlist by interpreting the relationship between the properties of each schematic symbol and the parameters of its associated template. To accomplish this netlist, there must be a correlation between the symbols used in the schematic and their corresponding models.

This correlation is called mapping. The table below lists the three mapping requirements, that is, the three areas that must correlate between a symbol and a model.

Symbol	Model
Name of the Symbol	Name of the Model
Properties of the Symbol	Parameters of the Model
Port Names of the Symbol	Pin Names of the Model

If you are using unmodified symbols, then these mappings have already been done for you. If you are using another vendor's symbols or custom made symbols, then you must cover these mapping requirements.

When is Mapping Necessary?

The need for mapping depends on the type of symbols you are using. If your symbols are required to have additional properties (non-simulation properties), then you will need to map -- you cannot use unmodified supplied symbols. Other EDA tools require that additional properties (and possibly other changes) be added to the symbols. These symbol changes make them incompatible for simulation, and therefore require mapping (to "map-out" these changes).

If your symbols only need to be compatible with another schematic capture tool (the Mentor Graphics Design Architect tool, the Cadence Artist tool, or the Viewlogic ViewDraw tool), then you can still avoid the mapping task by using symbols (they have already been mapped for this purpose). Four sets of symbols are provided. Each set is compatible with a different schematic capture tool (Saber Sketch, Design Architect, Artist, and ViewDraw). Each set contains a symbol for every model.

If your simulation needs are primarily in the mixed-technology area (electrical/mechanical/magnetic/hydraulic/etc.), then you are probably using Saber Sketch as your schematic capture tool with unmodified symbols and, therefore, do not need to consider mapping.

If your simulation needs are entirely in the electronic area (board or chip level), then your symbols will likely need to be compatible with other EDA tools (such as circuit layout and automatic test generation). These other tools usually require additional properties to be placed on the symbols. In this case, you will need to consider mapping techniques.

The following are some possible reasons for using other symbols:

- Convention
  Your company prescribes that all symbols must conform to some internal standard. This is the most common reason for using other symbols.
- Compatibility
  You have a large installed base using a particular type of symbol, and you need to maintain compatibility.
- Custom Part
  You must create a custom symbol because there may be no adequate symbol available to correspond to a custom model you have created.

Overview of Mapping Techniques

There are three methods for accomplishing the mapping task.

- Match the Names
  using symbol names, property names, and port names that exactly match the corresponding model names, parameter names, and pin names of your templates
- Use Specially-Recognized Properties
  whose names are recognized by the netlisters for the purpose of performing some or all of the mapping functions
- Use a Mapping File
  a separate text file, not part of the symbol or template, that is an input to the netlister for the purpose of performing some or all of the mapping functions

Each of these methods can be applied to any or all of the three mapping requirements. That is, you can mix-and-match the mapping methods between the mapping requirements, as long as all three of the mapping requirements are met.

These mapping methods are further explained in the following sections.

Overview of Name Matching
Overview of Specially-Recognized Properties
Overview of Mapping Files
Use Unaltered Symbols

Overview of Name Matching

If you are creating a new symbol or modifying an existing one, you should use symbol names, property names, and port names that exactly match the corresponding model names, parameter names, and pin names. In addition to using corresponding names, you must add a property called primitive. Name matching is the easiest way to accomplish the mapping task. In many cases name matching can cover all three mapping requirements. In some cases, however, using name matching alone cannot complete the mapping task.

If you cannot use name matching to fulfill all three mapping requirements, you can still use it to fulfill some of the mapping requirements. For instance, by renaming your symbol file to the name of your model file, you will fulfill the first mapping requirement. For example, if widget.sym is the present name of your symbol and gadget.sin is the name of the template to which you would like to map, then you rename (or copy) widget.sym to gadget.sym.

Advantages

Simple, easy, obvious (as to which symbols correspond with which models). Best all around, if it will suffice.

Disadvantages

Doesn’t work in all situations.

Detailed Use Instructions

Chapters 5: Using Name Matching to Map Symbols

Overview of Specially-Recognized Properties

Specially-recognized properties are typically used if you are maintaining an additional symbol library that allows you to modify or add properties to your symbols.

Advantages

From the symbol you can see what the simulator will use. (special properties override mapping file entries).

Disadvantages

Can clutter up the symbol.

Detailed Use Instructions

Chapter 6: Using Specially-Recognized Properties for Mapping
Chapter 7: Specially-Recognized Properties Reference

Overview of Mapping Files

A mapping file is a separate text file, not part of the symbol or template, that is an input to the netlister. Using a mapping file is the most cumbersome of the mapping techniques but allows you to fulfill all of the mapping requirements without modifying the symbol. A mapping file can also address other special concerns such as global nets.

Advantages

Transparent to the user
Most powerful

Disadvantages

Transparent to the user
Most cumbersome

Detailed Use Instructions

Chapter 8: Using a Mapping File to Map Symbols
Chapter 9: Mapping File Reference

Use Unaltered Symbols

This is not a mapping technique, but rather a method of avoiding the need for mapping. This is the easiest method for the purpose of simulation. Just use the symbols provided.

Advantages

Easiest.

Disadvantages

Doesn’t work in all situations.

Comparison of Mapping Techniques

The mapping methods can be compared using a number of criteria. Some of them are: ease of use, ability to modify symbol characteristics, and mapping precedence (when mapping techniques overlap). This section ranks and compares the mapping methods with respect to each of these criteria.

Ease of use

1. Unaltered Symbols
2. Name Matching
3. Special Properties
4. Mapping File

Ability to Modify Symbol Characteristics

1. Mapping File
2. Special Properties
3. Name Matching

Mapping Precedence

1. Special Properties
2. Mapping File
3. Name Matching

Other Factors that Can Determine the Mapping Method

After reading the previous sections, you might think that any mapping method will serve under any conditions and you simply need to make a choice. In many cases, any of the techniques will suffice. However, there are some special situations that can exclude some of the techniques. These factors are listed below.

You cannot use Name Matching if:

- You want to change the number of connection points
- You use Global connections
- You want to reduce properties
- Your template uses:

  ref connections
  struct
  enum
  union

You cannot use Specially-Recognized Properties if:

- Your template uses a ref connection.

Examples of Mapping Methods

The following examples demonstrate the use of the various mapping techniques to provide custom symbols in special circumstances.

Symbol with no-default properties in a structure
Symbol with default-valued properties in a structure
A template containing a ref connection point
A template containing an enumerated parameter
A user-created symbol for a digital part
A symbol for a hierarchical design

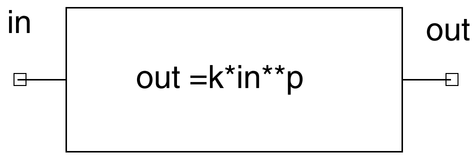
Symbol with no-default properties in a structure

The code below shows the user-created template nl_gain that defines the behavior of a gain-block used in the analysis of a control system. The following figure depicts the user-created symbol ctrl_1 that is associated with the template nl_gain. This template is the base from which the symbol properties are derived. It has an argument gain, which represents a structure (struc) containing the parameters k and p. Neither parameter has a default value assigned in the template.

If you want to make the parameters k and p properties of your symbol instead of the template argument gain, you must explicitly state the relationship between the gain argument and the properties k and p in a mapping file.

```
# file name: nl_gain.sin
#
template nl_gain in out =gain
ref nu  in
var nu  out
struc   {
number k
number p
}gain=()
{
equations   {
out=gain->k*(in**gain->p)
}
```

Symbol Name: ctrl_1



Symbol Properties
primitive=nl_gain
k=*value required*
p=*value required*

Label
Label Ttype = normalLabel
Label = “out=k*in**p”

Pin Properties
terminalName = in;
terminal Name = out;

Terminal Properties
direction = input
direction = output

If you map one argument of a template in a mapping file, you must map all arguments of that template.

The mapping file in this example is called ctrl_1.map. As shown in the next section of code, it contains only a saber definition section that consists of a generic entry and a specific symbol entry. Both entries are set off by braces. The name of the symbol appears in the first field of the specific symbol entry of the saber section. Because this symbol is to be targeted to the simulator netlist and simulated by the simulator, you map this symbol in the saber section of the mapping file.

```
#File name: ctrl_1.map
#
saber
{
#generic entry
: : : : : : : : : : : : : : : : : : : :
{
# specific symbol entries
ctrl_1: : :gain<-"(k=%{k},p=%{p})": : : : : : : : : : : : :
: : : ;
}
}
```

The fourth field of the specific symbol entry is used to construct the gain argument from the values of the symbol properties p and k with the following entry:

```
gain<-"(k=%{k},p=%{p})"
```

The netlister translates the symbol properties into netlist entries by interpreting this expression as follows. In the expression gain<-"(k=%{k},p=%{p})", the two special characters %{ indicate the beginning of the property name k (or p) and the special character } indicates the end of the property name k (or p). When a property name is used in this fashion, the value of the property is substituted in its place.

The characters (k=, p=, and) are literal values. Since the values of the property names k and p are 1.4 and 2, respectively (see the following figure), the expression is interpreted as follows:

```
gain<-"(k=1.4,p=2)"
```

Since the string (k=1.4,p=2) is placed within quotes on the right side of a left arrow, its literal value is assigned to the parameter gain as follows:

```
gain=gain(INST)<-(k=1.4,p=2)
```

where INST is the instance name

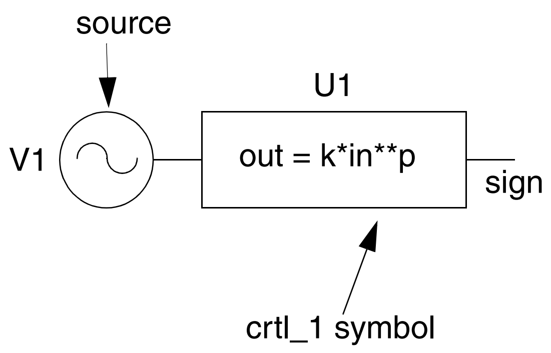
The symbol property primitive is assigned the value nl_gain, which is the template name. The values for symbol properties k and p are undetermined and have no default values. The entry *req* indicates that you must provide values for these properties when you place an instance of the symbol in a schematic because the template does not initialize these values.

The terminalNames are in and out. Since these pin names match the template connection points, they need no mapping file entry.

Creating, implementing, and testing the ctrl_1 symbol

The following figure shows a control-system circuit that incorporates the symbol ctrl_1. Note that the net (node) names in the circuit do not need to correspond to the names of the symbol connection points.

Schematic Name: demo_2



Property Values for Instance U1
primitive=nl_gain
name = U1
k = 1.4
p = 2

Property Values for Instance V1
primitive=src
name = V1
amplitude = 1.54
frequency = 12.6k
ac_mag = 1
ac_phase = 0

The following steps outline the procedure for creating, implementing, and testing the symbol.

1. Create the template nl_gain and place its file named nl_gain.sin in your project directory. (The netlister searches along the data search path for the template and mapping files. Your working directory is included on this data search path. You can add other directories of your own to the data search path by including them in the SABER_DATA_PATH environment variable.)
2. Using the Analog Artist symbol editor, create the symbol ctrl_1.

3. Using a text editor, create the mapping file ctrl_1.map as shown above and place the file in your working directory.
4. Using the Analog Artist schematic editor, create the schematic demo_2.
5. Add the instance names V1 and U1 to the symbols by selecting the Edit > Properties > Objects menu item and entering the appropriate value in the Instance Name field.
6. Use the Saber item on the menu bar to select the Saber Invocation Options form as follows:

   Saber > Saber > Simulator Startup > Options >
   Saber Invocation Options

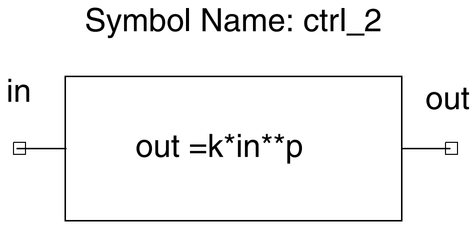
   Move the cursor to the Mapping Files field and enter the name of your mapping file (ctrl_1) and click on OK.
7. Click on OK in the Simulator Startup form to start the netlister. The netlist (demo_2.sin) that is created can be found in your working directory. It should appear similar to the netlist shown below.

```
# Instances found in the top level of design demo_2 #
src.V1 out:source = ac=(1,0),\
tran=(sin=(va=1.54,f=12.6k))
nl_gain.U1 out:sign in:source = gain=(k=1.4,p=2)
```

Symbol with default-valued properties in a structure

The code below shows the user-created template nl_gain2 that defines the behavior of a gain-block used in the analysis of a control-system. The following figure depicts the schematic symbol ctrl_2 that is associated with the template nl_gain2. The template nl_gain2 is the base from which the symbol properties are derived. It has an argument gain. This argument is represented by a structure (struc) containing the parameters k and p. Each parameter has a default value assigned in the template.

```
#file name: nl_gain2.sin
element template nl_gain2 in out =gain
ref nu  in
var nu  out
struc   {
  number k=1
  number p=1
  }gain=()
{
equations   {
  out=gain->k*(in**gain->p)
  }
}
```



Symbol Properties
primitive=nl_gain2
k=
p=

Label
Label Ttype = normalLabel
Label = “out=k*in**p”

Pin Properties
terminalName = in
terminal Name = out

Terminal Properties
direction = input
direction = output

If you want to make the parameters k and p the properties of your symbol instead of the template argument gain, you must explicitly state the relationship between the gain argument and the properties k and p in a mapping file.

The mapping file in this example is called ctrl_2.map. As shown in the code below, it contains only a saber definition section that consists of a generic entry and a specific symbol entry. Both entries are set off by braces. The name of the symbol appears in the first field of the specific symbol entry of the mapping file. Because this symbol is to be targeted to the simulator netlist and simulated by the simulator, you map this symbol in the saber section of the mapping file.

```
#File name: ctrl_2.map
saber
{
#generic entry
: : : : : : : : : : : : : : : : : : : : 
{
# specific symbol entries
ctrl_2: : : gain<-"(k=%{k},p=%{p})",gain<-"(k=%{k})",gain<-
"(p=%{p})": : : : : : : : : : : : : : : : : ;
}
}
```

The fourth field of the specific symbol entry is used to construct the gain argument from the values of the symbol properties k and p with the following entry:

```
gain<-"(k=%{k},p=%{p})",gain<-"(k=%{k})",gain<-"(p=%{p})"
```

The netlister that translates the symbol properties into netlist entries interprets this expression in the fourth field as follows. In the first part of the entry, the expression gain<-"(k=%{k},p=%{p})" contains the special characters %{ that indicate the beginning of the property name k (or p). It also contains the special character } that indicates the end of the property name k (or p). When a property name is used in this fashion, the value of the property is substituted in its place.

If the values of properties k and p were 1.732 and 1.414, respectively, the expression would be interpreted as gain<-"(k=1.732,p=1.414)".

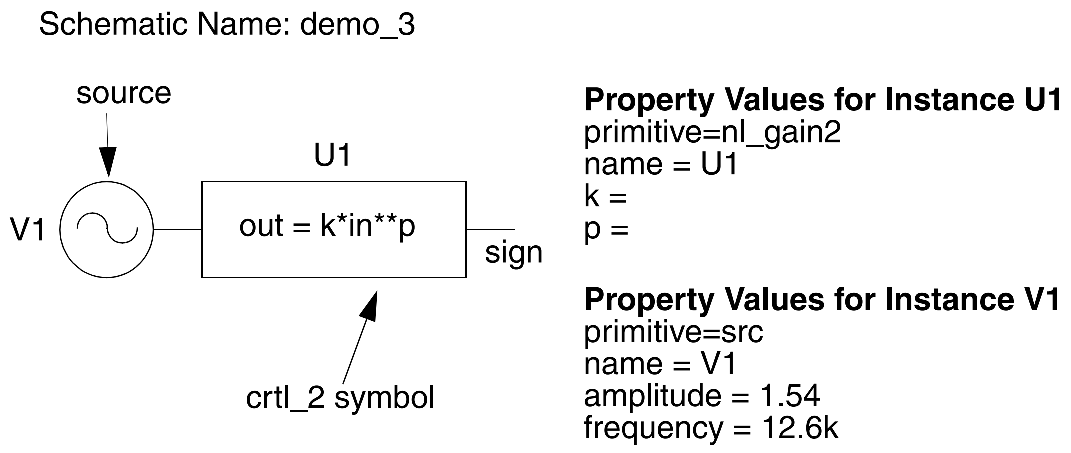
However, in the this example, no values for k and p are supplied by the user; therefore, the netlister uses the default values specified in the template as described below.

If the first item gain<-"(k=%{k},p=%{p})" in the mapping file expression fails evaluation because either or both parameter values are not specified, the next item gain<-"(k=%{k})" in the expression is evaluated. If the parameter k was assigned a value, it is used while the parameter p is assigned the default value. If the parameter k was not assigned a value, the parameter p in the third item gain<-"(p=%{p})" is evaluated. If no items are left to evaluate, no gain arguments are passed to the template and default values that are provided by the template are assigned in the template.

The symbol property primitive is assigned the value nl_gain2, which is the template name. The symbol connection points (terminalNames) are given the names in and out. These pin names match the template connection points and, therefore, need no mapping entry.

Creating, implementing, and testing the ctrl_2 symbol

The following figure shows a control-system circuit that incorporates the symbol ctrl_2. Note that the net (node) names in the circuit need not correspond to the names of the symbol connection points.



Property Values for Instance U1
primitive=nl_gain2
name = U1
k =
p =

Property Values for Instance V1
primitive=src
name = V1
amplitude = 1.54
frequency = 12.6k

The following steps outline the procedure for creating, implementing, and testing the symbol:

1. Create template nl_gain2 and place its file nl_gain2.sin in your working directory.
2. Using the Analog Artist symbol editor, create the symbol ctrl_2.
3. Using a text editor, create the mapping file ctrl_2.map and place the file in your working directory.
4. Using the Analog Artist schematic editor, create the schematic demo_3.
5. Add the instance names V1 and U1 to the instances by selecting the Edit > Properties > Objects menu item and entering the appropriate value in the Instance Name field.

6. Use the Saber item on the menu bar to select the Saber Invocation Options form as follows:

   Saber > Saber > Simulator Startup > Options > Saber Invocation Options

   Move the cursor to the Mapping Files field and enter the name of your mapping file (ctrl_2) and click on OK.
7. Click on OK in the Simulator Startup form to start the netlister. The netlist (demo_3.sin) that is created can be found in your working directory. It should appear similar to the netlist shown below.

```
# Instances found in the top level of design demo_3 #


nl_gain2.U1 out:sign in:source
src.V1 out:source = ac=(1,0),\
tran=(sin=(va=1.54,f=12.6K))
```

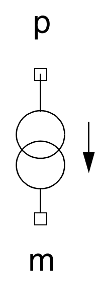
A template containing a ref connection point

This example uses the template (cccs) for a current-controlled current source to demonstrate the mapping of a ref connection point.

The cccs template is available from the simulator template library. It models a dependent current source for which the current-gain value is specified by the parameter k. The controlling input is a ref type connection point (ci) that refers in this example to the current flowing in a resistor. The following is the first line of this template which includes three connection points ci, p, and m: template cccs ci p m = k

The figure below shows a user-created symbol representing this template. It is called ctrl_current and has two connection points p and m. The third connection point ci, which is the ref connection point, does not appear on the symbol.

Symbol Name: ctrl_current



Symbol Properties
primitive=cccs
current_control_inst=*value required*
k=*value required*

Pin Properties
terminalName = p
terminalName = m

Terminal Properties
direction = inputOutput
direction = inputOutput

The ref type connection point requires the existence of a var (a type of system variable that can be accessed by another template) in the template of a resistor that provides the control current for the current source. The template for a standard resistor (r.sin) does not use a var, and therefore, cannot be used. However, you can use the special resistor template rz.sin which does use a var. The rz.sin template is available from the simulator template library.

The mapping file in this example is called ctrl_current.map. As shown in below, it contains only a saber definition section that consists of a generic entry and a specific symbol entry. Both entries are set off by braces. Because this symbol is to be targeted to the simulator netlist and simulated by the simulator, you map this symbol in the saber section of the mapping file.

```
#File name: ctrl_current.map
saber
{
#generic entry
: : : : : : : : : : : : : : : : :
current_control_inst: : :
{
# specific symbol entries
ctrl_current: : : : : : : : : : : : :ci<-
"i(%{current_control_inst})": : : : : : ;
}
}
```

The name of the symbol appears in the first field of the specific symbol entry. A ref connection requires that the property name, current_control_inst in this case, be placed in the 18th field of the generic entry of the saber section of the mapping file. This entry prevents the netlister from using the property current_control_inst as a simulator template parameter; therefore, this property does not appear in the netlist. The required entry in the 15th field of the specific symbol entry is ci<-"i(%{current_control_inst})".

In the expression ci<-"i(%{current_control_inst})", the special characters %{ indicate the beginning of the property name current_control_inst and the special character } indicates the end of that property name. If the value of property current_control_inst is set to rz.R1, the netlister interprets the previous expression as ci<-"i(rz.R1)".

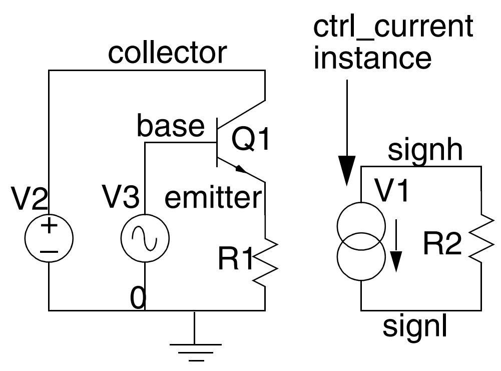
The netlister places the literal value i(rz.R1) in the netlist as follows indicating the literal value is connected to connection point ci: ci:i(rz.R1).

The symbol property primitive is assigned the value cccs, which is the template name. The values for symbol properties current_control_inst and k are undetermined and have no default values. Since their values must be provided on the instance by the user, these properties are assigned the value *value required*. The pin properties are given the names p and m. Since these pin names match the template connection points, they need no mapping file entry.

Creating, implementing, and testing the symbol

The figure below shows a simple analog circuit that incorporates the user-created symbol ctrl_current. Note that the net (node) names in the circuit need not correspond to the names of the symbol connection points.

Schematic Name: demo_4



Property Values for V1
primitive=cccs
name=V1
current_control_inst=RZ.R1
k=1

Property Values for V2
primitive=V
name=V2
dc_value=10

Property Values for V3
primitive=V
name-V3
frequency=12.6k
amplitude=1.54

Property Values for Q1
primitive=MPS6521
name=Q1

Property Values for R1
primitive=RZ
name=R1
rnom=1k

Property Values for R2
primitive=R
name=R2
rnom=100

The following steps outline the procedure for creating, implementing, and testing the symbol:

1. Using the Analog Artist symbol editor, create the symbol ctrl_current.
2. Using a text editor, create the mapping file ctrl_current.map and place the file in your working directory.
3. Using the Analog Artist schematic editor, create the schematic demo_4.
4. Add the instance names V1, V2, V3, Q1, R1, and R2 to the instances by selecting the Edit > Properties > Objects menu item and entering the appropriate value in the Instance Name field.
5. Use the Saber item on the menu bar to select the Saber Invocation Options form as follows:

Saber > Saber > Simulator Startup > Options > Saber Invocation Options

Move the cursor to the Mapping Files field and enter the name of your mapping file (ctrl_current) and click OK.

6. Click OK in the Simulator Startup form to start the netlister. The netlist (demo_4.sin) that is created can be found in your working directory. It should appear similar to the following netlist.

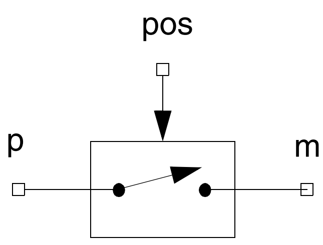
```
# Instances found in the top level of design demo_4


v.V2 p:collector m:0 = dc=10

v.V3 p:base m:0 = ac=(1,0), tran = (sin =  \
(va=1.54,f=12.6K))

mps6521.Q1 c:collector e:emitter b:base

rz.R1 p:emitter m:0 = rnom=1K

cccs.V1 p:signh m:signl ci:i(rz.R1) = k=1

r.R2 p:signh m:signl = rnom=100
```

A template containing an enumerated parameter

This example uses the template sw_1pno for a single-pole single-throw switch to demonstrate the mapping of an enumerated parameter.

The sw_1pno template is available from the simulator template library. It models a single-pole, normally-open switch. The figure below shows a user-created symbol (spst_switch) that represents the template. The first line of the template with the three connection points pos, p, and m is: template sw_1pno pos p m = ron, roff, tdbrk, tdmk, rfunc.

Symbol Name: spst_switch



Symbol Properties
primitive = sw_1pno
ron =
roff =
rdbrk =
tdmk =
rfunc =

Pin Properties
terminalName = p
terminalName = m
terminalName = pos

Terminal Properties
direction = inputOutput
direction = inputOutput
direction = inputOutput

The switch position is controlled by the logic state at the pos connection. The switch position determines the resistance value seen between the electrical connections p and m. The enumerated parameter rfunc determines whether the resistance is a continuous or a discontinuous function of time during the transition interval.

The mapping for this example is provided in the standard mapping file enum.map because the sw_1pno template is included in the simulator symbol libraries, which have the mapping provided. If you create your own template that includes an enumerated parameter, you must provide the mapping for that template. The enum entry required to map this template parameter (rfunc) is as follows:

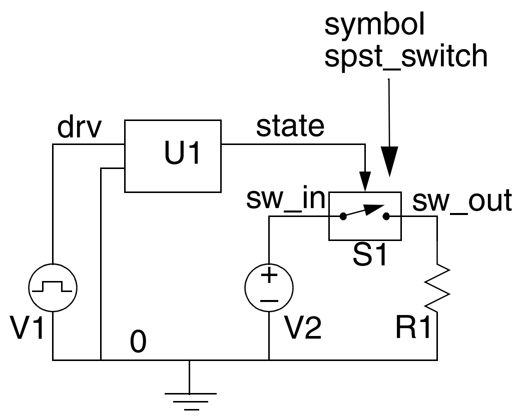
```
enum {cont,discont} rfunc {sw_1pno}
```

The netlister interprets this expression as follows. The expression {cont,discont} rfunc {sw_1pno} informs the netlister that the parameter rfunc of template sw_1pno can be assigned one of the two values cont or discont. The special characters { and } in the expression indicate the beginning and end of literal values, respectively.

Creating, implementing, and testing the symbol

The figure below shows a simple circuit that incorporates the user-created symbol spst_switch. Note that the net (node) names in the circuit need not correspond to the names of the connection points of the symbol.

Schematic Name: demo_5



Property Values for Instance S1
primitive = sw_1pno
name - S1
ron = 1
roff = 100k
rdbrk =
tdmk =
rfunc = discont

Property Values for Instance V1
primitive = v
name = V1
initial = 0
pulse = 5
delay = 0
tr = 1u
tf = 1u
width = 0.5
period = 1

Property Values for Instance V2
primitive = v
name = V2
dc_value = 10

Property Values for Instance U1
primitive = sdr_thr2
name = U1
vpull = 3
tdelay = 1m

Property Values for Instance R1
primitive = r
name = R1
rnom = 1k

The procedure for creating, implementing, and testing the symbol is outlined in the following steps:

1. Using the Analog Artist symbol editor, create the symbol spst_switch.
2. Again, you do not need to create a mapping file for this example because the standard mapping file enum.map provides the necessary mapping for the sw_1pno template. If, however, you create your own template that includes an enumerated parameter, you will need to specify the mapping for that template.
3. Using the Analog Artist schematic editor, create the schematic demo_5.

4. Add the instance names V1, V2, U1, and R1 to the instances by selecting the Edit > Properties > Objects menu item and entering the appropriate value in the Instance Name field.
5. Use the Saber item on the menu bar to select the Simulator Startup form as follows: Saber > Saber. Since you do not need to specify a mapping file, you do not need to use the Saber Invocation Options form.
6. Click OK in the Simulator Startup form to start the netlister. The netlist (demo_5.sin) that is created can be found in your working directory. It should appear similar to the netlist shown below.

```
# Instances found in the top level of design demo_5


v.V2 p:sw_in m:0 = dc=10

sw_1pno.S1 pos:@"state" p:sw_in m:sw_out = roff=100K,\
rfunc=discont, ron=1

sdr_thr2.U1 pos:@"state" drvm:0 drvp:drv = vpull=3,\
tdelay=1m

r.R1 p:sw_out m:0 = rnom=1K

v.V1 p:drv m:0 = tran =  \
(pulse=(v1=0,v2=5,tr=1u,tf=1u,td=0,pw=0.5,per=1))
```

A user-created symbol for a digital part

This example uses the digital template (inv_l4.sin) from the simulator template library to demonstrate the creation of the properties of a digital symbol.

To properly insert Hypermodel analog/digital interfaces, the netlister must have information on pin type and pin direction. Each of the pins of the inverter symbol must have the property port_type with the value digital to inform the netlister of the pin type.

This digital template models a logic-state inverter. The template senses the logic state at the in pin, inverts it, and, with the appropriate delay and initialization specifications applied, places the result at the out pin. The following figure shows the user-created symbol, called inverter, that is associated with this template.

Symbol Name: inverter

Symbol Properties
primitive = inv_l4
tplh =
tphl =
tihl =
tilh =
init =

Pin Properties
terminalName = in
terminalName = out

Terminal Properties
direction = input
port_type = digital
direction = output
port_type = digital

The first line of the template is:

```
template inv_l4 in out = tplh, tphl, tilh, tihl, init
```

If the value of a parameter is a single identifier (including the underscore character), the netlister usually considers it to be a nonliteral identifier. A nonliteral identifier can be a variable which gets evaluated. The property init, shown in the symbol property list in the figure above, typically takes enumerated type values _0, _1, _x, and _z that resemble nonliteral identifiers. To avoid misinterpretation by the netlister, you must specify the possible values of the property init in the enums section of the mapping file.

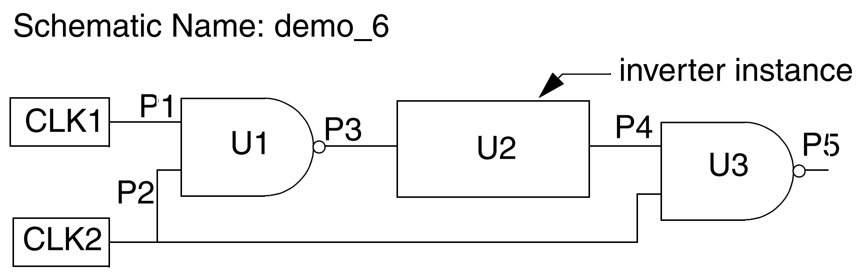
The mapping for this example is provided in the standard mapping file enum.map because the inv_l4 template is included in the simulator symbol libraries, which have the mapping provided. If you create your own template that includes an enumerated parameter, you must provide the mapping for that template. The enum entry required to map this template parameter (init) is: enum {_0,_1,_x,_z} init {inv_l4}.

The netlister that translates the symbol properties into netlist entries interprets this expression as follows. The expression enum {_0,_1,_x,_z} init {inv_l4} informs the netlister that the parameter init of template inv_l4 can be assigned the values _0, _1, _x, or _z.

The symbol property primitive is assigned the value inv_l4, which is the template name. The remaining symbol properties assume the default values that are provided by the template. Since these values can be changed by the user, the properties are not assigned a value. The terminalNames are in and out. Since these pin names match the template connection points, they need no mapping file entry.

Creating, implementing, and testing the inverter symbol

The figure below shows a digital circuit that incorporates the symbol inverter. Note that the net (node) names in the circuit do not need to correspond to the names of the connection points of the symbol.



Property Values for CLK1
primitive=clock_l4
name=CLK1
td=0
duty=0.5
freq=10k

Property Values for CLK2
primitive=clock_l4
name=CLK2
td=0
duty=0.25
freq=10k

Property Values for U1
primitive=nand2_l4
name=U1
tplh=
tphl=
tihl=
tilh=
init=

Property Values for U2
primitive=inv_l4
name=U2
tplh=25n
tphl=30n
tihl=20n
tilh=23n
init=_0

Property Values for U3
primitive=nand2_l4
name=U3
tplh=
tphl=
tihl=
tilh=
init=

The following steps outline the procedure for creating, implementing, and testing the inverter symbol:

1. Using the Analog Artist symbol editor, create the symbol inverter.

2. Again, you do not need to create a mapping file for this example because the standard mapping file enum.map provides the necessary mapping for the inv_l4 template. If, however, you create your own template that includes an enumerated parameter, you will need to specify the mapping for that template.
3. Using the Analog Artist schematic editor, create the schematic demo_6.
4. Add the instance names U1, U2, U3, CLK1, and CLK2 to the instances by selecting the Edit > Properties > Objects menu item and entering the appropriate value in the Instance Name field.
5. Use the Saber item on the menu bar to select the Simulator Startup form as follows: Saber > Saber .
6. Since you do not need to specify a mapping file, you do not need to use the Saber Invocation Options form.
7. Click on OK in the Simulator Startup form to start the netlister. The resulting netlist (demo_6.sin) is written in your working directory. It should be similar to the netlist shown below.

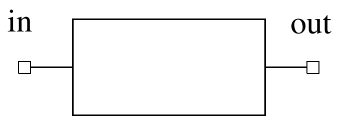
```
# Instances found in the top level of design demo_6


inv_l4.U2 out:p4 in:p3 = tilh=23n, tphl=30n, init=_0,\
tplh=25n, tihl=20n
clock_l4.CLK1 clock:p1 = td=0, freq=10K, duty=0.5
clock_l4.CLK2 clock:p2 = td=0, freq=10K, duty=.25
nand2_l4.U1 in1:p1 in2:p2 out:p3_U1_out
nand2_l4.U3 in1:p4 in2:p2 out:p5
```

A symbol for a hierarchical design

This example shows how to map the properties of a user-created symbol that represents both a template and a schematic model.

The symbol shown below is called inverter2. You can use it to represent the digital template inv_l4. You can also use it to represent the schematic model, which is called inverter2.

Symbol Name: inverter2



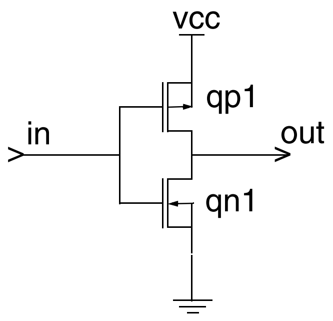
Symbol Properties
primitive=
tplh=
tphl=
tilh=
tihl=
init=

Pin Properties
terminalName = in
terminalName = out

Terminal Properties
direction = input
port_type = digital
direction = output
port_type = digital

The schematic model shown in the next figure consists of two MOS field-effect transistors.

Schematic Name: inverter2



Property Values for qp1
primitive = m+3p
name = qp1
saber_model = (type=_p)

Property Values for qn1
primitive = m_3p
name = qn1
saber_model = (type=_n)

If an instance of this inverter symbol has the property primitive with the value inv_l4, the template inv_l4 is used in the analyses, and the schematic model (inverter2)—if it exists for this symbol—is ignored. If the value of the property primitive is undefined, the schematic model is used in the analyses.

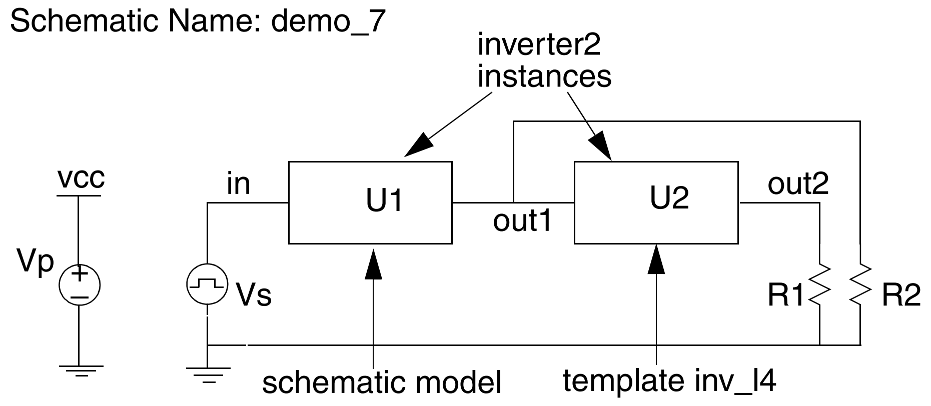
To use this feature for the inverter2 symbol, add the property primitive to field 16 of the generic entry of the undetermined section as shown below.

```
#File name= inverter2.map
tables {
global_power_pins[VCC]->unpow,[GND]->ungnd,[*]->null
}
undetermined {
#generic entry
: : : : : : : : : : : : : : :primitive: : : : :
}
saber {
#generic entry
: : : : : : :global_power_pins[<>] : : : : : : : : :primitive:
: : {
}
```

If the primitive property of an instance of the symbol inverter2 has a value specified, the netlister includes the template name as the value of the primitive property in the netlist. If the value of the primitive property is not specified, the netlister includes the schematic represented by the symbol in the netlist.

The global_power_pins[<>] entry in the eighth field of the saber section references the global_power_pins entry in the tables section. This entry indicates that when a part targeted to the simulator has a pin named VCC, the net connected to that pin is to be used as power for Hypermodel interfaces. Also, the pin is not to be written in the netlist because the simulation model for the part in question has no pin named VCC. Pins named GND are treated similarly except they designate ground connections for the Hypermodels. For more information regarding the use of this field, the description of field eight in the Generic Entries section of Chapter 9: Mapping File Reference.

The following figure shows the symbol inverter2 used twice in a mixed-signal circuit. A value has not been given to the primitive property of instance U1. Therefore, this version of the inverter2 symbol represents the schematic model. The primitive property of instance U2 has been given the value inv_l4. Therefore, this version of the inverter2 symbol represents the template inv_l4.



Property Values for U1
primitive=
name=U1
tplh=
tphl=
tihl=
tilh-
init=

[Since inv_l4 template is not used, no values are entered.]

Property Values for U2
primitive=inv_l4
name=U2
tplh=25n
tphl=30n
tihl=20n
tilh=23n
init=

Property Values for Vp
primitive=v
name=Vp
dc_value=5

Property Values for Vs
primitive=v
name=Vs
initial=0
pulse=5
tr=10n
tf=15n

Property Values for R1
primitive=r
name=R1
rnom=10k

Property Values for R2
primitive=r
name=R2
rnom=10k

Creating, implementing, and testing the inverter2 symbol

To create, implement, and test the inverter2 symbol, perform these steps:

1. Using the Analog Artist symbol editor, create the symbol inverter2.
2. Using the Analog Artist schematic editor, create the schematic inverter2.
3. Add the instance names qp1 and qn1 to the instances by selecting the Edit > Properties > Objects menu item and entering the appropriate value in the Instance Name field.

4. Using a text editor, create the mapping file inverter2.map and place the file in your working directory. This mapping file maps the schematic model of the inverter symbol.
5. Using the Analog Artist schematic editor, create the schematic demo_7.
6. Add the instance names R1, R2, Vp, and Vs to the instances by selecting the Edit > Properties > Objects menu item and entering the appropriate value in the Instance Name field.
7. Use the Saber item on the menu bar to select the Saber Invocation Options form as follows:

   Saber > Saber > Simulator Startup > Options > Saber Invocation Options

   Move the cursor to the Mapping Files field and enter the name of your mapping file (inverter2) and click on OK.

8. Click on OK in the Simulator Startup form to start the netlister. The netlist (demo_7.sin) that is created can be found in your working directory. It should appear similar to the netlist shown below.

```
# Intermediate template inverter2

template inverter2 out:out in:in gnd:0 vcc:vcc

{
m_3p.qp1 d:out g:in s:vcc = model=(type=_p)
m_3p.qn1 d:out g:in s:0 = model=(type=_n)
}

# Instances found in the top level of design demo_7

ide_d2an.U2_out a:out2 d:out2_U2_out m:0
ide_a2dn.U2_in a:out1 d:out1_U2_in m:0
inverter2.U1 out:out1 in:in gnd:0 vcc:vcc
v.Vp p:vcc m:0 = dc=5
inv_l4.U2 out:out2_U2_out in:out1_u2_in = tphl=30n,\
tilh=23n, tplh=25n, tihl=20n
v.Vs p:in m:0 = tran=(pulse=(v1=0,v2=5,tr=10n,tf=15n))
r.R1 p:out2 m:0 = rnom=10k
r.R2 p:out1 m:0 = rnom=10k
```

7
Using Name Matching to Map Symbols

This section describes creating symbol properties and giving them values corresponding to the names of the template, template parameters, and template pins. If you are creating your own symbol that you would like to associate with a template, you should follow the guidelines presented in this section. Topics discussed in this section include:

When Can Name Matching be Used?

Making the names associated with your symbol correlate with the names associated with the template is a technique is most appropriate if you are creating a new symbol. Although the symbol you want to create can be any graphical representation that suits you, the symbol properties must be presented in a prescribed format to enable the Frameway integration software to translate these properties into entries suitable for a netlist.

In most simple cases, applying the set of rules outlined later in this chapter ensures direct translation of symbols. However, some symbols possess properties that require the netlister to have supplemental information to perform the translation. This supplemental information can be provided by either of the other two mapping methods. Even if you have decided that additional mapping is necessary, following the guidelines in this chapter will minimize the need for special properties or mapping files.

The following special situations cannot be handled entirely by name matching:

- A symbol has several properties that need to be combined into a structure (struc) format for use by the template.
- A symbol represents a hierarchical circuit.
- A template contains special features, such as reference connections, enumerated parameters, string parameters, or does not contain connection points.

If you need to create a symbol and any of these situations apply, you will have to use specially-recognized properties or create your own mapping file. If the template to be used contains reference connections, then you will need to use a mapping file. If you create a symbol for a template that does not include these special situations, then you can map the symbol by using name matching.

Creating Symbols and Symbol Properties Corresponding to Template Features

You may want to create a new schematic symbol if you have created your own template and you cannot find a symbol that is appropriate, or you may want to improve on an existing symbol that is insufficient for a particular template or was just incorrectly constructed.

The symbol may be any graphical representation of the device, and you can often avoid the need to create a mapping file if you apply the following general guidelines for creating symbols (including properties). The Example of Name Matching shows how these four guidelines are applied to a user-created symbol.

General Guidelines for Symbol Creation

- If the symbol is used to represent more than one template, you should add a property called primitive to the symbol. You should give the primitive property the name of the template as its value. Note that even if you must use a mapping file, a symbol with the primitive property requires fewer mapping entries than one without it.
- When you place properties on the symbol that represent Saber simulation parameters, such as tplh and rnom, you should choose property names that match the template parameters.

- If you want to specify a reference designator for an instance of a symbol, use whichever mechanism is provided by your particular schematic editor (Saber Sketch, or one provided by Mentor Graphics, Cadence, or Viewlogic). The netlister will automatically pick up the name.
- If you are using a non-Saber template, you must put the port_type=digital property on digital pins and designate the port direction. If you are using a Saber template, this is not required.
- You should choose symbol pin names that match the template pin names. if this is not possible, use field 6 in a mapping file to match the symbol and template pin names.
- If you need to add properties that don't correspond to template parameters (such as for PCB layout) you can exclude them from the netlist by using a mapping file. For Saber templates this is done automatically, and no action is required.

For Viewlogic users only

Note that when you place attributes on symbols, if you do not define the case of the attributes (that is, upper, lower, or mixed case) using the Case_ attribute, the netlister uses the following defaults:

- If the target simulator is Verilog, the model name defaults to uppercase.
- If the target simulator is Saber, the model name defaults to lowercase.
- Pin names default to lowercase.
- Symbol and instance attributes default to lowercase.
- You cannot change schematic, net, or pin attributes; therefore, they are always uppercase.

For information regarding the Case_ attribute, refer to Chapter7: Specially-Recognized Properties Reference.

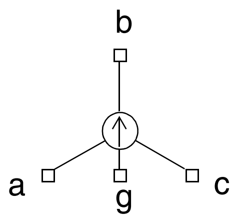
Example of Name Mapping

Creating a Symbol and a Template for a Three-Phase Current Source

The code below shows a template (written in MAST) called g3ph.sin that defines the behavior of a three-phase current source. The following figure shows the schematic symbol representing the three-phase current source template. Naming the symbol g3ph.sym would be the simplest way to map the symbol name; however, since the primitive property allows the symbol name to be different from the template name, for demonstration purposes the name i3ph is given to the symbol.

```
#file name: g3ph.sin
template g3ph a b c g = f,iph
electrical a, b, c, g
number f         #frequency
number iph        #peak amplitude of the generated current
{
val i i0,i120,i240
values  {
i0 = iph*sin(2*3.14*f*time)
i120 = iph*sin(2*3.14*f*time+2*3.14*1/3)
i240 = iph*sin(2*3.14*f*time+2*3.14*2/3)
step_size = 1/(f*20)    #maximum step size
}
equations{
i(a->g) +=i0
i(b->g) +=i120
i(c->g) +=i240
}
}
```

Symbol Name: i3ph



The symbol properties and identifiers are defined in partner software as follows:

Mentor Graphics

Symbol Identifier:	primitive = g3ph

Symbol Properties:	f = *req*
	iph = *req*

Pin Name:	PIN = a
	PIN = b
	PIN = c
	PIN = g

Cadence

Symbol Identifier:	primitive = g3ph

Symbol Properties:	f = *req*
	iph = *req*

Pin Names:	terminalName = a ;
	direction = inputOutput
	terminalName = b ;
	direction = inputOutput
	terminalName = c ;
	direction = inputOutput
	terminalName = g ;
	direction = inputOutput

Viewlogic

Symbol Identifier:	Block Type > module prefix = g3ph

Symbol Properties:	f = *req*
	iph = *req*

Pin Names:	a pintype = analog b pintype = analog c pintype = analog g pintype = analog

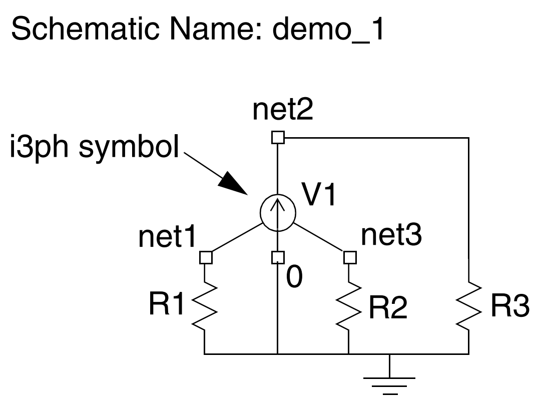
The g3ph template is the base from which the symbol properties of the current source are derived. This template contains the parameters f and iph, which represent the frequency and the peak current for each leg of the current source, respectively.

The following list outlines the i3ph symbol specifications:

- The symbol property primitive is assigned the value g3ph, which is the template name.
- Because the template does not give initial values to f and iph, they are specified as required by using the *req* expression.
- By assigning each pin label property the appropriate pin name (a, b, c, or g), the symbol connection points match the names of the template connection points.
- Viewlogic users only:
  Specifying the symbol Block Type to be module allows the symbol to be considered primitive. With the Block Type specified as module, the netlister can recognize the prefix attribute and, from its value, determine the template (g3ph) to use for the symbol.

Creating, Implementing, and Testing the i3ph Symbol

The figure below shows a schematic called demo_1. The following table lists the values of each of the properties for each symbol instance in demo_1. The schematic illustrates how an instance of the three-phase current source symbol can be applied in a network. Note that the net (node) names in the circuit need not match the names of the symbol connection points and that the simulator reference node must be named 0 (zero). If you use the ground symbol from the Saber simulator symbol library (or the aiSupport or Cadence analogLib library), the ground node is automatically named 0.



Mentor Graphics Properties

Instance V1:	Instance R1:	Instance R2:	Instance R3:
primitive=g3ph	primitive=r	primitive=r	primitive=r
inst=V1	inst=R1	inst=R2	inst=R3
f=60	rnom=1k	rnom=1k	rnom=1k
iph=120			

Cadence Properties

Instance V1:	Instance R1:	Instance R2:	Instance R3:
primitive=g3ph	primitive=r	primitive=r	primitive=r
name=V1	name=R1	name=R2	name=R3
f=60	rnom=1k	rnom=1k	rnom=1k
iph=120			

Viewlogic Attributes/Labels

Instance V1:	Instance R1:	Instance R2:	Instance R3:
prefix=g3ph	prefix=r	prefix=r	prefix=r
label=V1	label=R1	label=R2	label=R3
f=60	rnom=1k	rnom=1k	rnom=1k
iph=120			

The following steps outline the procedure for creating, implementing, and testing the symbol:

1. Create the template g3ph and place its file named g3ph.sin in your working directory.
2. Using your symbol editor (Design Architect, Analog Artist, or ViewDraw), create the symbol i3ph shown above.

   Viewlogic users only:
   Remember to set the Block Type to module by using the Change -> Block -> Type menu item from the ViewDraw symbol editor menu banner. Also note that the pintype defaults to analog, so, in this case, it is not necessary to specify it on the symbol pins.
3. Using the your schematic editor, create the schematic demo_1 as shown in the figure.
4. Add the instance names to the symbols.
5. Invoke the netlister.

   Mentor Graphics and Cadence: Click on the SABER icon. In the Saber Simulator Startup dialog box, set the Invoke Simulator field to no. Click on OK in the Saber Simulator Startup dialog box to start the netlister.

   Viewlogic: Select the Saber > Saber UI Window item on the ViewDraw window menu banner to open the Saber Simulation Window. Select the Saber > Extract Design item on the Saber Simulation Window menu banner. This selection causes the Extract Design dialog box to appear. Ensure that the name of the design (demo_1) appears in the Design Name field and click on Accept to start the netlister.

Since the creation of the symbol did not include exceptions to the general guidelines, the netlister can generate a netlist for the schematic demo_1 without a mapping file. The result should be similar to that shown below:

```
# Instances found in the top level of design demo_1 #


r.R3 p:net2 m:0 = rnom=1k
g3ph.V1 c:net3 g:0 a:net1 b:net2 = iph=120, f=60
r.R1 p:net1 m:0 = rnom=1k
r.R2 p:net3 m:0 = rnom=1k
```

The instance reference designators that you specified appear in the netlist following the template name for each symbol as r.R3, g3ph.V1, etc.

8

Using Specially-Recognized Properties for Mapping

This section describes how to use specially-recognized properties. These specially-recognized properties provide a means for specifying the properties of a symbol or instance in such a way that they can be translated into netlist entries without using a mapping file. If you would like to map other symbols, and you can add or modify properties on your symbols, you should follow the guidelines presented in this section. Topics discussed in this section include:

Overview of Specially-Recognized Properties

If you use the specially-recognized properties described here to make an existing symbol compatible with the Saber Simulator, you do not need to create a mapping file entry for that symbol provided the template to be used does not contain reference connections.

A property that is attached to a symbol on a schematic sheet may or may not require a user-specified value. That is, if the template associated with the symbol provides a default value for the property, you have the choice whether or not to provide a value for the property on the instance. If no default value is specified in the template, you must provide a value. You can scan the template to determine whether a default value for a particular parameter exists, or you can refer to the online template description.

When you create a symbol, you must give each property a value. You can follow the convention of this software or the convention of your other schematic capture tool (Mentor Graphics, Cadence, or Viewlogic).

The convention of this software requires that you indicate, for each property, whether a user-specified value is required when the symbol is placed in a schematic (instance of the symbol). If no default value is specified in the template associated with the symbol, you give the property on the base-symbol the value *req*. If a default value is specified in the template, you give the property on the base-symbol the value *opt*. The Saber Simulator convention is shown in the following example:

```
Saber_rnom=*opt*
```

```
Saber_dc=*req*
```

The convention of your other schematic capture tool (Mentor Graphics, Cadence, or Viewlogic) requires that each property contain a valid value, such as 1k or 200n, or is left blank. Giving a value indicates that a value is required. Leaving the field blank indicates that a value is optional.

List of Specially-Recognized Properties

This section lists the special properties with a brief description of each. For a complete description of any particular property, see Chapter 7: Specially-Recognized Properties Reference. The properties are grouped into the following functional areas.

Model and Simulator Specification
Parameter Specification
Pin Specification
Hypermodel Specification
File Inclusion
Partner Simulator (Verilog) Specification
Special Viewlogic-Only Properties

Most of the special properties, except those in the last two categories (Verilog and Viewlogic), apply to all of the netlisters. The properties in the first five categories can be used with any of the schematic capture tools. However, some of the functions from previous categories are repeated in the last category (Viewlogic). If you are using the ViewDraw editor you should use the properties from this group whenever possible. Do not use these properties if you are not using ViewDraw.

The Verilog properties are only valid in the Saber-Verilog simulation environment, that is, they are supported only by the DVETOS, CATOS, and VWLTOSV netlisters (supplied netlisters to translate from Mentor Graphics, Cadence or Viewlogic to Saber-Verilog). You can use these properties in

Design Architect, Analog Artist, or ViewDraw, or you can use a mapping file to relate them to other properties used in Design Architect, Analog Artist, or ViewDraw. Values defined in the mapping file for these properties must be in correct Verilog syntax. The properties are then incorporated into the Verilog netlist by the Verilog writer.

Model and Simulator Specification

- Target_Simulator
  specifies in which netlist (Saber or partner simulator) to place the symbol.
- Primitive
  specifies the name of the template that the symbol represents, but does not specify which simulator should be used.
- SaberTemplate
  specifies the name of the template that the symbol represents and specifies that the Saber Simulator should be used.

Parameter Specification

- Saber_ParameterName
  specifies a single template parameter.
- SaberParameters
  specifies one or more Saber Simulator parameters as the value of a single property.
- SaberString_ParameterName
  places quotes around a string value when the string value is passed to a template.
- SaberEnum_ParameterName
  specifies an enumerated parameter.

Pin Specification

- Port_Type
  specifies whether a port (pin) of a symbol is digital or analog. This property is not required for Saber templates.
- SaberPin_PinName
  specifies the connection of a pin to a net (wire).
- SaberPinOrder
  redefines symbol pin order for the Saber Simulator.

- PartnerPinOrder
  redefines symbol pin order for a partner simulator.
- SaberRemovePort
  prevents a port from appearing in the netlist.

Hypermodel Specification

In order for these properties to function properly, you must point the netlister to the Hypermodel libraries that the properties specify. If you are invoking the netlister from a command line, include the -h option. If you are invoking the netlister from the user interface, select the libraries in the dialog box controlling netlister settings.

- SaberTech
  specifies the technology of the Hypermodel interface for a specific pin.
- SaberModel
  defines the model parameter of Hypermodel interface for a specific pin.
- SaberModelName
  specifies the name of an existing Hypermodel interface in the Hypermodel library for a specific pin.
- SaberModelPowerPin
  specifies the power supply for a specific pin.
- SaberModelGroundPin
  specifies the ground for a specific pin.

File Inclusion

- SaberPrepend
  designates one or more names of files containing templates to be included at the beginning of a netlist.
- SaberInclude
  designates one or more names of files containing templates to be included at a specified level of hierarchy in a netlist.
- SaberAppend
  designates one or more names of files containing templates to be included at the end of a netlist.

Partner Simulator (Verilog) Specifications

- delay (instance property)
  specifies the time delay for the part.
- delay (net property)
  specifies the time delay of the net.
- drive (instance property)
  specifies the drive strength for the part.
- drive (net property)
  specifies the drive strength of the net.
- type (net property)
  causes the net to be declared in the netlist as the type given by the value of this property.
- charge (net property)
  specifies the charge strength for the net
- timescale (instance property)
  specifies the time scale for the Verilog simulator.

Special Viewlogic-Only Properties

- Case_
  specifies the case (upper or lower) of certain attributes.
- Mast_Model
  specifies the simulation model name for the Saber Simulator.
- Mast_Pinorder
  redefines pin order.
- Verilog_Model causes the netlister to use the value of the this attribute as the Verilog model name.
- Verilog_Pinorder
  redefines symbol pin order for the Verilog simulator.
- Prefix
  same as Primitive but for Viewlogic netlisters.

Property Value Limitations (Mentor Graphics Only)

This section applies to Mentor Graphics users only.

Due to a character limit on the value of Design Architect symbol properties, you should not use a property (such as Saber_model) on a symbol to specify the argument definition directly. If you exceed the property value character limit, the property value will not be included in the Saber Simulator netlist, giving an incorrect result. If this occurs, you may encounter a message similar to the following when you run a Saber simulation on a Design Architect schematic:

```
...: the device model has an undefined value.
/my_library/parts/m1_p
```

You can use two properties to avoid this situation: SaberPrepend and SaberInclude. The one difference between the two methods is that the SaberInclude property provides a way to include files within templates of the hierarchy of the design, while the SaberPrepend property includes files globally for the whole design. Also, you can locate the SaberPrepend property directly on the symbol that needs the value specified in the included file. For more details on these methods, see:

Using SaberPrepend to Avoid Property Value Limitations
Using the SaberInclude File to Avoid Property Value Limitations

Using SaberPrepend to Avoid Property Value Limitations

This section applies to Mentor Graphics users only.

You can use the SaberPrepend property on a symbol to provide the name of a file in which you specify the argument definition for the symbol. Then, you use the name of the argument definition that you declared in the file as the property value (for example, of Saber_model) on the symbol rather than the argument definition. Now, when you run a Saber simulation on the schematic, the Saber Simulator uses the named argument definition from the file, avoiding the property value character limit.

The following example shows how multiple argument definitions might appear in a file:

```
m..model nmos=(type=_n,level=2,kp=2.5e-5 \
tox=7.5e-8,vto=1,cbd=.1p,cbs=.1p,cgso=.1p,cgdo=.1p,\
cgbo=.1p,rd=1,rs=1)
m..model pmos=(type=_p,level=2,kp=1e-5,tox=7.5e-8,vto=1,\
cbd=.1p,cbs=.1p,cgso=.1p,cgdo=.1p,cgbo=.1p,rd=1,rs=1)
```

In this example, m..model indicates a structure named model in the m (MOSFET) template. The first argument definition is named nmos; the second argument definition is named pmos. The nmos argument definition, for example, uses the listed specifications from the model structure in the m template when you specify the nmos definition name. So, if you are using this file to define an n-type MOSFET, you would specify the name of the file as the value of the SaberPrepend property and nmos as the value of the Saber_model property on the instance.

To use the SaberPrepend and Saber_model properties, follow these steps:

1. Create the file name.sin, where name can be any name, and place the file in a location that is in the data search path. For more information about the data search path, refer to Chapter 2: Modifying Your Search Paths.
   Your current working directory is automatically included in the data search path. Also, note that the file name must include the .sin extension.
2. Enter the argument definition in the file name.sin.
3. In Design Architect, add the SaberPrepend property on any symbol in the design and specify its value to be the name of the file that you created (either name or name.sin).
4. Edit the Saber_model property on the symbol and specify its value to be the name of the argument definition (definition_name) that you entered in the file name.sin. Remember, the value of the Saber_model property is the name of the argument definition in the file, not the name of the file.

If you place multiple instances of the same symbol defined by using the SaberPrepend and Saber_model properties on a schematic, you need only place the SaberPrepend property on one instance, which can be at any level of the hierarchy.

Using the SaberInclude File to Avoid Property Value Limitations

This Section applies to Mentor Graphics users only.

An alternative method of avoiding the character limit on values of Design Architect symbol properties is to use the SaberInclude property rather than the SaberPrepend property.

To create and specify a Saber include file, follow these steps:

1. Create the file name.sin, where name can be any name, and place the file in a location that is in the data search path.

Your current working directory is automatically included in the data search path. Also, note that the file name must include the .sin extension.

2. Enter the argument definition in the file name.sin.
3. In Design Architect, select the Saber Include File item from the Parts palette. This causes the saber symbol to appear.
4. Place the saber symbol on the schematic at the top level of the hierarchy. The saber symbol includes, among others, the SaberInclude property.
5. Edit the SaberInclude property and specify its value to be the name of the file that you created (either name or name.sin).
6. Edit the property, on the appropriate schematic symbol, that will be used for the model parameter of the template. You can use any property, such as Saber_model. Specify the property value to be the name of the argument definition that you entered in the file name.sin. Remember, the value of this property is the name of the argument definition (definition_name) in the file, not the name of the file.

9

Specially-Recognized Properties Reference

This section describes the unique function of each specially-recognized property. These property descriptions are organized alphabetically for quick reference. For listings of special properties organized by functional groups, see Chapter 6: Using Specially-Recognized Properties for Mapping.

Specially-Recognized Properties Descriptions

Each property description indicates whether the special property is used on the body or on the pin of a symbol. Each description also indicates whether it can be used on the instance of that symbol, on the base symbol, or on a net (wire).

Some properties can be used only on the base symbol. When the property description indicates this, note that you must specify this property before you place an instance of the symbol on a schematic sheet. Note also that you cannot override the symbol property with a different value on the instance.

Each property description indicates whether the special property applies to the Saber Sketch design editor, the Mentor Graphics Design Architect tool, the Cadence Artist tool, or the Viewlogic ViewDraw tool (most properties apply to all). Each description also indicates whether the special property applies to the Saber or the Saber-Verilog simulation environment, or both. The descriptions also include examples that show you how to use these properties.

Although the names of the specially-recognized properties are described containing upper and lower case characters, the Frameway integration software does not distinguish between the two cases in property names (that is, they are not case sensitive).

This section uses terms common to Mentor Graphics, Cadence and Saber Sketch design capture tools. For Viewlogic viewers Instance Properties are

Instance Attributes, Symbol Properties are Unattached Attributes, and Pin Properties are Attached Attributes.

Specially Recognized Property NAME	**Used On**		**Applies To**	
	Body / Pin	**Instance, Net, Base Symbol**	**Schematic Editors**	**Simulation Environment**
Case_	Body	Instance or Base	Viewlogic Only	Saber or SV

This attribute specifies the case of certain attributes. The format is as follows:

Case_attribute_name

where attribute_name is the name of an attribute on the base symbol or instance of the symbol. The value of the Case_ attribute must be either upper or lower. For example, if you have the attribute Verilog_Model specified as follows,

VERILOG_MODEL=MYMODEL

you could specify the corresponding case_ attribute as follows:

CASE_VERILOG_MODEL=LOWER

The netlister would then define the Verilog model name in lowercase (mymodel). If you do not define attribute cases by using this Case_ attribute, the netlister uses the following defaults:

If the target simulator is Verilog, the model name defaults to uppercase.

If the target simulator is the Saber Simulator, the model name defaults to lowercase.

Pin names default to lowercase.

Symbol and instance attributes default to lowercase.

You cannot change schematic, net, or pin attributes; therefore, they are always uppercase.

charge	n/a	Net	All	Saber-Verilog

When you specify this property on a net (wire), it causes the net to be declared in the netlist with the value of the charge property specifying the charge strength for the net. For example, the property specification

charge small

results in a (small) entry in the Verilog netlist. The following netlist segment shows the resulting charge specification for net trir.

```
wire (strong1,pull0) #(10) strong;
trireg (small) trir;
buf (highz1,strong0) #(2:2:3,3:3:4)
buf1(trir,strong);
```

Specially Recognized Property NAME	**Used On**		**Applies To**	
	Body / Pin	**Instance, Net, Base Symbol**	**Schematic Editors**	**Simulation Environment**
delay	Body	Instance or Net	All	Saber-Verilog

instance property. When you specify this property on an instance of a Verilog part, the instance is included in the Verilog netlist with the value of the delay property specifying the delay for the part. For example, the property specification delay 2:2:3,3:3:4

results in the #(2:2:3,3:3:4) entry in the following Verilog netlist segment:

```
wire (strong1,pull0) #(10) strong;
trireg (small) trir;
buf (highz1,strong0) #(2:2:3,3:3:4)
buf1(trir,strong);
```

net property. When you specify this property on a net (wire), it causes the net to be declared in the netlist with the property value specifying the delay of the net. For example, the property specification

delay 10

results in the #(10) entry in the following Verilog netlist segment:

```
wire (strong1,pull0) #(10) strong;
trireg (small) trir;
buf (highz1,strong0) #(2:2:3,3:3:4)
   buf1(trir,strong);
```

Specially Recognized Property NAME	**Used On**		**Applies To**	
	Body / Pin	**Instance, Net, Base Symbol**	**Schematic Editors**	**Simulation Environment**
drive	Body	Instance or Net	All	Saber-Verilog

instance property. When you specify this property on an instance of a Verilog part, the instance is included in the Verilog netlist with the value of the drive property specifying the drive strength for the part. For example, the property specification

drive highz1,strong0

results in the (highz1,strong0) entry in the following Verilog netlist segment:

```
wire (strong1,pull0) #(10) strong;
trireg (small) trir;
buf (highz1,strong0) #(2:2:3,3:3:4)
   buf1(trir,strong);
```

net property. When you specify this property on a net (wire), it causes the net to be declared in the netlist with the property value specifying the drive strength of the net. For example, the property specification drive strong1,pull0 results in the (strong1,pull0) entry in the following Verilog netlist segment:

```
wire (strong1,pull0) #(10) strong;
trireg (small) trir;
buf (highz1,strong0) #(2:2:3,3:3:4)
   buf1(trir,strong);
```

For more information about Verilog language constructs, refer to the Verilog Reference Manual.

Mast_Model	Body	Instance or Base	Viewlogic Only	Saber or SV

This attribute specifies the simulation model name for the Saber Simulator. The value of this attribute is the name of the Saber Simulator model that corresponds to this symbol. If the Target_Simulator attribute exists with the value saber, then the netlister uses the value of the Mast_Model attribute as the Saber Simulator model name. Otherwise, this attribute is ignored. This property has precedence over the SaberTemplate and the Primitive property, but is only recognized in the Viewlogic environment. See the Primitive and SaberTemplate property descriptions.

Specially Recognized Property NAME	**Used On**		**Applies To**	
	Body / Pin	**Instance, Net, Base Symbol**	**Schematic Editors**	**Simulation Environment**
Mast_Pinorder	Body	Base Symbol	Viewlogic Only	Saber or SV

The value of this attribute has the same format as the Viewlogic pinorder attribute. You specify pin names with spaces between each name. For example: o1 i1 i2

The pins o1, i1, and i2 must exist on the symbol. This attribute does not redefine pin names, only pin order.

You use this attribute in conjunction with the Target_Simulator attribute. If the value of the Target_Simulator attribute is saber, the netlister searches for the Mast_Pinorder attribute to determine the Saber Simulator pin order. If the netlister finds the Target_Simulator attribute but not the Mast_Pinorder attribute, it will search for the corresponding MAST pin order file. This file is stored in the symbol directory and has the name symbol_name.mpo. The format of this file is the same as the Viewlogic pin order (.pin) file. For more information regarding pin order files, refer to the Viewlogic Schematic Design User's Guide.

If you specify the pin order for a symbol in the mapping file, that order overrides the order specified by the Mast_Pinorder attribute.

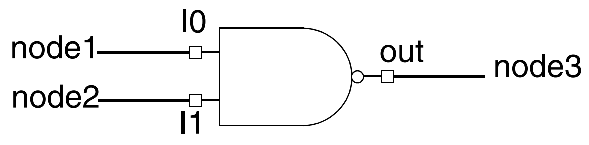
Specially Recognized Property NAME	Used On		Applies To	
	Body / Pin	**Instance, Net, Base Symbol**	**Schematic Editors**	**Simulation Environment**
PartnerPinOrder	Body	Symbol Only	All	Saber-Verilog

You can use the PartnerPinOrder property when you use a partner simulator to match the order of the symbol pins with the order of the connection points of the partner model. Place the symbol pins in the same order as they are defined in the model.

The following example shows the PartnerPinOrder property with a value specified for the Verilog partner simulator:

PartnerPinOrder=out,I0,I1

The following figure shows this example used on a symbol. The resulting Verilog netlist entry for the symbol is included.



Symbol Property
PartnerPinOrder=out,I0,I1

Instance Property
Target_Simulator=partner

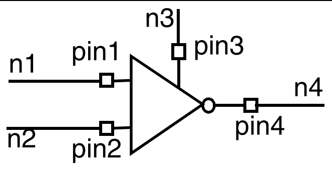
Netlist entry:
nand2
\I1 (node3 , node1 , node2);

Specially Recognized Property NAME	**Used On**		**Applies To**	
	Body / Pin	**Instance, Net, Base Symbol**	**Schematic Editors**	**Simulation Environment**
Port_Type	Pin	Instance or Base	All	Saber or SV

This property specifies whether a port (pin) of a symbol is a digital or an analog type port. For symbols targeted to the Saber Simulator, the default value is analog. For symbols targeted to a partner simulator, the default value is digital. However, both analog and digital are valid property values. The following shows an example for each port type:

Port_Type=digital   Port_Type=analog

As shown in the following figure, a symbol that represents a comparator has both analog and digital port types. The resulting netlist entry for the comparator symbol is included.



Pin Properties (pin1)
Port_Type=analog

Pin Properties (pin2)
Port_Type=analog

Pin Properties (pin3)
Port_Type=digital

Pin Properties (pin4)
Port_Type=digital

Symbol Properties
SaberPinOrder=pin1,pin2,pin3,pin4

Instance Properties
SaberTemplate=comp_l4
Saber_td=20n
Saber_p_offset=1m
Saber_m_offset=2m
Saber_hys=2m
SaberEnum_enable_init=_1

Netlist entry:
comp_l4.I5 n1 n2 n3 n4 = enable_init=_1,
m_offset=2m,hys=2m, p_offset=1m, td=20n

Prefix	Body	Instance or Base	Viewlogic Only	Saber or SV

The Prefix property is similar to the Primitive property except that it is only recognized by the Viewlogic netlisters. If you use the Prefix property (rather than SaberTemplate), then all properties not used for simulation (such as layout properties) must be excluded from the netlist using a mapping file. The prefix property specifies the name of the template that the symbol represents but does not specify which simulator should be used. This property designates the template as primitive (no hierarchical template below this level). This property has a lower precedence than the SaberTemplate property. See property descriptions for Primitive, Mast_Model, and SaberTemplate.

Specially Recognized Property NAME	**Used On**		**Applies To**	
	Body / Pin	**Instance, Net, Base Symbol**	**Schematic Editors**	**Simulation Environment**
Primitive	Body	Instance or Base	All, except Viewlogic	Saber or Saber-Verilog
The Primitive property is recognized by all netlisters except Viewlogic, but is primarily used with the Saber Sketch schematic designer (or DesignStar). If you use the Primitive property (rather than SaberTemplate), then all properties not used for simulation (such as layout properties) must be excluded from the netlist using a mapping file. The Primitive property specifies the name of the template that the symbol represents but does not specify which simulator should be used. This property designates the template as primitive (no hierarchical template below this level). This property has a lower precedence than the SaberTemplate property. See property descriptions for Prefix, Mast_Model, and SaberTemplate.				
SaberAppend	Body	Instance Only	All	Saber or SV
You can add the SaberAppend property to a symbol to designate one or more names of files containing templates to be included at the end of a netlist. You can use this property, for example, to add additional components to a netlist that do not appear in the schematic of the design, such as stimulus sources. The value assigned to a SaberAppend property is a comma-separated list of template file names (files with an extension .sin). You do not need to include the .sin extension in the property value.				
SaberEnum_ Parameter	Body	Instance or Base	All	Saber or SV
You can use this property to specify an enumerated parameter. In the following example, the property is used to specify the value of a logic state. This example demonstrates how you can use the SaberEnum_ParameterName property to set the initial logic state of a digital symbol pin. SaberEnum_enable_init=_1 The previous figure (for the Port_Type property) shows this example used on an instance of a symbol that represents a comparator. The resulting netlist entry for the symbol is included in the figure. A template argument with enumerated values can only be given a value from a list of values declared in the template. These values are typically character strings such as yes, no, _z, or _1. Enumerated values use the same syntax as parameters in the MAST language.				

Specially Recognized Property NAME	Used On		Applies To	
	Body / Pin	**Instance, Net, Base Symbol**	**Schematic Editors**	**Simulation Environment**
SaberInclude	Body	Instance or Base	All	Saber or SV

This property is used when the symbol represents a schematic or sub-schematic, but not a primitive. If placed on a primitive, this property will be ignored.

The value of this property is a comma separated list of files to be included in the Saber Simulator netlist. The listed files are included at the top of the template corresponding to the schematic represented by the symbol to which the property is attached.

If the SaberInclude property is attached to a symbol called Saber, the files are included at the top of the template corresponding to the schematic containing the Saber symbol, but the Saber symbol is not referenced in the netlist. Example:

SaberInclude=npnmodel, pnpmodel

SaberModel	Pin	Instance Only	All	Saber or SV

This is an instance pin property you use to specify a complete definition of the model parameter of a Saber Simulator hypermodel interface in MAST syntax for a specific pin. The following is an example value of the SaberModel property:

```
model=(type=_3,vcc=4.5,voh=2.5,ioh=0.4m,vol=0.35,\
   iol=8m,vcmax=5.5,io=224m,co=10p,reft=27,tr=5n,\
   tf=5n,vih=2.7,iih=20u,vil=0.4,iil=0.1m,ci=5p,\
   vxh=2.2,vxl=0.9,tdon=6n,tdoff=6n,amax=0.5,\
   amin=0.5,vsmax=5.5,vsmin=4.5,dmax=0.5,\
   dmin=-0.5,xmin=0.5)
```

SaberModelName	Pin	Instance Only	All	Saber or SV

This is an instance pin property that you use to specify the name of an existing Hypermodel interface in the hypermodel library for a specific pin. For example, the entry for a 74LS04 part in the ti2.shm Hypermodel file is as follows:

74LS04:adadadg dadadap::ti74ls_15

The Hypermodel interface name is ti74ls_15, which is the value you would specify for the SaberModelName property.

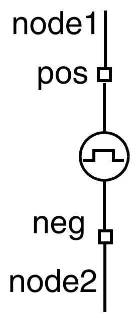
Specially Recognized Property NAME	Used On		Applies To	
	Body / Pin	Instance, Net, Base Symbol	Schematic Editors	Simulation Environment
SaberParameters	Body	Instance or Base	All	Saber or SV

This property specifies a set of parameters in the MAST syntax used in the template. You use this property when you want to specify one or more Saber Simulator parameters as the value of a single property, as shown in the following example:

SaberParameters=tran=(pulse=(0,5,1u,1n,2n,2u,8u))

The properties list can have only one SaberParameters property.

The following figure shows this example used on an instance of the symbol that represents a pulse voltage source. The resulting netlist entry for the symbol is included.



Symbol Properties
SaberPinOrder=pos,neg

Pin Properties (pos)
Port_Type=analog

Pin Properties (neg)
Port_Type=analog

Instance Properties
SaberTemplate=v
SaberParameters=tran=(pulse=(0,5,1u,1n,2n,2u,8u)),\
ac=(mag=1,phase=90)

Netlist entry:
v.I5 node1 node2 =tran=(pulse=(0,5,1u,1n, 2n, 2u, 8u)),\
ac=(mag=1, phase=90)

Specially Recognized Property NAME	**Used On**		**Applies To**	
	Body / Pin	**Instance, Net, Base Symbol**	**Schematic Editors**	**Simulation Environment**
Saber_ParameterName	Body	Instance or Base	All	Saber or SV

This property specifies a single template parameter in the MAST syntax used in the template. The suffix ParameterName can be the name of any single Saber Simulator template parameter that is not an enumerated type or a string. The following examples demonstrate how you can use this property to specify various template parameters.

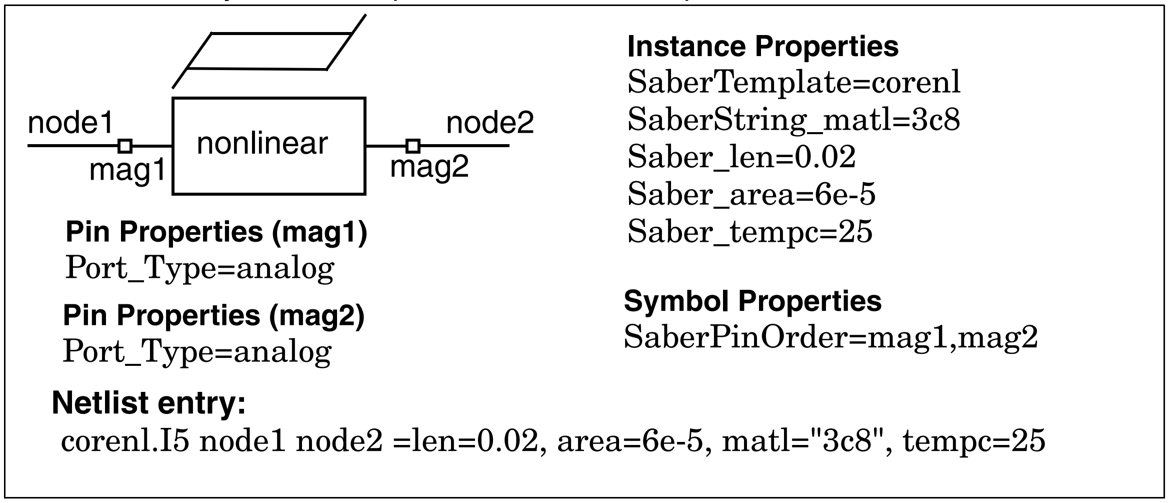
Saber_td=20n

Saber_p_offset=1m

Saber_len=0.02

Saber_area=6e-5

The first two examples are shown in the figure for the port_type property used on an instance of the symbol of the comp_l4 template. The following figure shows the last two examples used on an instance of the symbol that represent the corenl template.



Pin Properties (mag1)
Port_Type=analog

Pin Properties (mag2)
Port_Type=analog

Instance Properties
SaberTemplate=corenl
SaberString_matl=3c8
Saber_len=0.02
Saber_area=6e-5
Saber_tempc=25

Symbol Properties
SaberPinOrder=mag1,mag2

Netlist entry:
```
corenl.I5 node1 node2 =len=0.02, area=6e-5, matl="3c8", tempc=25
```

Specially Recognized Property NAME	**Used On**		**Applies To**	
	Body / Pin	**Instance, Net, Base Symbol**	**Schematic Editors**	**Simulation Environment**
SaberPinOrder	Body	Base Symbol	All	Saber or SV
This property is not needed for use with templates in the Template Database. You can use the SaberPinOrder property to match the order of the symbol pins with the order of the connection points of the template. Place the symbol pins in the same order as they are defined in the template. The following examples show the SaberPinOrder property with values specified: SaberPinOrder=pin1,pin2,pin3,pin4 SaberPinOrder=mag1,mag2 SaberPinOrder=pos,neg The previous three figures show each of these examples used on a symbol. The resulting netlist entry for the symbol is included in each figure.				
SaberPin_ PinName	Body	Instance or Base	All	Saber or SV
You can use this property to specify the connection of a pin (PinName) to a net (wire). The net is the specified value of the property. For example, an instance of a transistor may have a property SaberPin_s with a value Vee as follows: SaberPin_s=Vee The netlister will connect pin s to the net Vee.				
SaberPrepend	Body	Instance Only	All	Saber or SV
You can add the SaberPrepend property to an instance to designate one or more names of files containing templates to be included at the beginning of a netlist. When the file names are included at the beginning of the netlist, the templates contained in the file are available when referenced by intermediate (hierarchical) templates that appear later in the netlist. The value assigned to a SaberPrepend property is a comma-separated list of template file names (files with an extension .sin). You do not need to include the .sin extension in the property value.				

Specially Recognized Property NAME	Used On		Applies To	
	Body / Pin	**Instance, Net, Base Symbol**	**Schematic Editors**	**Simulation Environment**
SaberString_ ParameterName	Body	Instance or Base Symbol	All	Saber or Saber-Verilog

You can use this property to place quotes around a string value when the string value is passed to a template. For example, a symbol representing the template corenl may have a property SaberString_matl with a value 3c8 as follows:

SaberString_matl=3c8

Since the template corenl has the parameter matl (core material) which requires a value that is a string, the netlister will place the value 3c8 between quotes and pass it to the parameter matl of the template corenl as follows:

matl= "3c8"

The previous figure (for the Saber_ParameterName property) shows this example used in an instance of the symbol that represents the template corenl. The resulting netlist entry for this symbol is included in the figure.

SaberTech	Pin	Instance Only	All	Saber or SV

You use this property to specify the technology of the Hypermodel interface for a specific pin. The following is an example value:

ttl

SaberTemplate	Body	Instance or Base	All	Saber or SV

This property specifies the name of the template that the symbol represents and specifies that the Saber Simulator should be used. This property does not necessarily designate the symbol as a primitive. The SaberTemplate property takes precedence over the Primitive property.

The following examples specify the templates comp_l4, corenl, and v, and are also used in the previous three figures.

SaberTemplate=comp_l4

SaberTemplate=corenl

SaberTemplate=v

The resulting netlist entry for the symbol is included in each figure.

Specially Recognized Property NAME	**Used On**		**Applies To**	
	Body / Pin	**Instance, Net, Base Symbol**	**Schematic Editors**	**Simulation Environment**
Target_ Simulator	Body	Instance or Base	All	Saber-Verilog

This property specifies (to the netlister) in which netlist to place the symbol (the Saber Simulator netlist or the partner simulator netlist). Valid property values are saber and partner. You assign the value saber to this property when the symbol is to be simulated by the Saber Simulator; you assign the value partner when the symbol is to be simulated by a partner simulator. For example,

```
Target_Simulator=saber
Target_Simulator=partner
```

The use of this property is illustrated in the figure for the PartnerPinOrder property.

timescale	Body	Instance Only	All	Saber-Verilog

You can attach this property to the top cell of your schematic, with a value such as 1ns/1ns, to specify the time scale for the Verilog simulator. The verilog.ntf file supplied contains the following entry near the top of the file:

```
%(TIMESCALE_DECLARATION)
```

When the netlister finds the timescale property on the top cell of your schematic, it places an entry in the netlist defining the value of the timescale property. The entry follows the line containing the default value from the verilog.ntf file.

Because the last of these entries takes precedence, the Verilog simulator will use the time scale defined by the timescale property.

Specially Recognized Property NAME	**Used On**		**Applies To**	
	Body / Pin	**Instance, Net, Base Symbol**	**Schematic Editors**	**Simulation Environment**
type	n/a	Net Only	All	Saber-Verilog
When you specify this property on a net (wire), it causes the net to be declared in the netlist as the type given by the property value. For example, the property specification type trireg results in a trireg entry in the Verilog netlist. The following Verilog netlist segment shows the resulting type specification for net trir. wire (strong1,pull0) #(10) strong; trireg (small) trir; buf (highz1,strong0) #(2:2:3,3:3:4) buf1(trir,strong); For more information about Verilog language constructs, refer to the Verilog Reference Manual.				
Verilog_Model	Body	Instance or Base	Viewlogic Only	Saber-Verilog
The value of this attribute is the name of the Verilog model that corresponds to this symbol. If the Target_Simulator attribute exists with the value partner, then the netlister uses the value of the Verilog_Model attribute as the Verilog model name. Otherwise, this attribute is ignored.				
Verilog_ Pinorder	Body	Base Symbol	Viewlogic Only	Saber-Verilog
The value of this attribute has the same format as the Viewlogic pinorder attribute. You specify pin names with spaces between each name. For example o1 i1 i2 The pins o1, i1, and i2 must exist on the symbol. This attribute does not redefine pin names, only pin order. You use this attribute in conjunction with the Target_Simulator attribute. If the value of the Target_Simulator attribute is partner, the netlister searches for the Verilog_Pinorder attribute to determine the Verilog pin order. If the netlister finds the Target_Simulator attribute but not the Verilog_Pinorder attribute, it will search for the corresponding Verilog pin order file. This file is stored in the symbol directory and has the name symbol_name.vpo. The format of this file is the same as the Viewlogic pin order (.pin) file. For more information regarding pin order files, refer to the Viewlogic Schematic Design User's Guide. If you specify the pin order for a symbol in the mapping file, that order overrides the order specified by the Verilog_Pinorder attribute.				

10

Reserved Properties on Symbols and Ports

Symbols and ports used in the drawing tools Saber Sketch and Saber Bundle have reserved properties, meaning properties that have specific uses defined by the application. You should use reserved properties only as they are defined in the following sections:

- Saber Sketch Symbols and Ports
- Saber Bundle Symbols and Ports
- Parts Databases

Saber Sketch Symbols and Ports

The following properties are reserved for Saber Sketch Symbols and Ports.

Hierarchical Block Symbol — Reserved Properties

The hierarchical block symbol defines a sub-schematic as an instance in another schematic. The convention is that the name of the symbol and the name of the sub-schematic are the same. The instance name comes from the ref property.

Symbol

The properties that are reserved and used for hierarchical symbols are:

Name	Value	Type	Value Editor	View	Protected	Qualifier	Description
schematic	name	string	text	value	yes	none	The name of the schematic which implements the behavior/structure of the symbol.
ref	*opt*	string	text	value	no	none	The reference designator or instance name of the symbol.

Other properties may be defined and passed into the sub-schematic based on the model as desired for modeling.

Port

There are no reserved properties for ports on hierarchical block symbols.

HDL Symbol — Reserved Properties

An HDL (Hardware Description Language) symbol is used to represent a template written in an HDL such as MAST or VHDL-AMS.

The HDL symbol defines the interface of the HDL model as seen in a schematic. This includes the ports or connections to the model and the properties that are arguments (generics in VHDL terminology) used with the model. The reserved properties define the relationship between the symbol and model to that of the reference designator for the model. The instance name of the symbol is primitive.ref.

Symbol

Symbol The properties that are reserved and used for HDL symbols are:

Name	Value	Type	Value Editor	View	Protected	Qualifier	Description
primitive	name	string	text	valu e	yes	SABER	The name of the template for SABER.
ref	*opt*	string	text	valu e	no	none	The reference designator of the model. This is common through for all HDLs.

Other properties may be defined and passed into the HDL model based on the model arguments.

Port

The reserved properties for ports on HDL symbols define the type of port (analog digital) of the connection on the model and are used for Hypermodel insertion decisions. The reserved port properties for HDL symbols are:

Name	Value	Type	Value Editor	View	Protected	Qualifier	Description
pin_fault	*opt*	string	text	value	yes	none	This property is defined for use by Testify.

Hierarchical Connector Symbol — Reserved Properties

In order for a node in a schematic to be connected to a node in a parent schematic (besides using a global connection), the node must be attached to a hierarchical connector. For each hierarchical connector within a schematic there must be exactly one port of the same name on the symbol which is associated with this schematic. Saber Sketch will then make a nodal connection.

Symbol

The properties that are reserved and used for hierarchical connector symbols are:

Name	Value	Type	Value Editor	View	Protected	Qualifier	Description
connector	hierarchical	string	text	value	yes	none	Indicates that the connector is a hierarchical connector.
name	*req*	string	text	value	no	none	The name of the node that is connected.

Port

There are no reserved properties for ports on hierarchical connector symbols.

On-page Connector Symbol — Reserved Properties

An on-page connector is used to connect to an on-page connector on the same page. Even though the on-page connector does not affect the nodal connectivity (connection is made by node name on the wire), on-page connectors can make schematics easier to read.

Symbol

The properties that are reserved and used for on-page connector symbols are:

Name	Value	Type	Value Editor	View	Protected	Qualifier	Description
connector	onpage	string	text	value	yes	none	Indicates that the connector is an on-page connector.
name	*req*	string	text	value	no	none	The name of the node that is connected.

Port

There are no reserved properties for ports On-page connector symbols.

Off-page Connector Symbol — Reserved Properties

An off-page connector connects to an off-page connector on another sheet. Even though the off-page connector does not really affect the nodal connectivity (connection is made by node name on the wire), off-page connectors can make schematics easier to read.

Symbol

The properties that are reserved and used for off-page connector symbols are:

Name	Value	Type	Value Editor	View	Protected	Qualifier	Description
connector	offpage	string	text	value	yes	none	Indicates that the connector is an off-page connector.
name	*req*	string	text	value	no	none	The name of the node that is connected.

Port

There are no reserved properties for ports on off-page connector symbols.

Global Connector Symbol — Reserved Properties

A global connector is used to connect to a node that exists in a another schematic. This makes the connection to all nodes of the name in all parent schematics without regard to the ports on the intervening symbols. This is typically used for power and ground connections.

Symbol

The properties that are reserved and used for global connector symbols are:

Name	Value	Type	Value Editor	View	Protected	Qualifier	Description
connector	global	string	text	value	yes	none	Indicates that the connector is a global connector.
name	*req*	string	text	value	no	none	The name of the node that is connected.

Port

There are no reserved properties for ports on global connector symbols.

Border Annotation Drawing — Reserved Properties

A border annotation drawing is used to add annotations to the border drawing frame. The properties on the border annotation drawing fall into two categories. The first define the type of the drawing and information used by the border annotation tool. The properties in the second category are defined by the creator of the symbol and contain whatever information is desired.

Drawing

The properties that are reserved and used for global connector symbols are:

Name	Value	Type	Value Editor	View	Protected	Qualifier	Description
symboltype	border_annotation	string	text	value	yes	none	Indicates that the drawing is a border annotation.

Name	**Value**	**Type**	**Value Editor**	**View**	**Protected**	**Qualifier**	**Description**
annotation_ subregion	header, line, or data	string	text	value	yes	none	The type of the border annotation. The header type is used with annotations that have multiple lines and contains only text. The line type is used as the data bearing containers with the header. The data type contains data and is used by itself (such as for a title block).
annotation_ type	title, revision, notes, included, associate d	string	text	value	yes	none	Defines the type of the annotation. The creator can define other types. The names specified must be used so tools can find the correct annotation type.
annotation_ position	*opt*	string	text	value	no	none	Uses by annotation tool.
annotation_ direction	*opt*	string	text	value	no	none	Uses by annotation tool.
annotation_ order	*opt*	string	text	value	no	none	Uses by annotation tool.

Name	Value	Type	Value Editor	View	Protected	Qualifier	Description
annotation_region_name	*opt*	string	text	value	no	none	Uses by annotation tool.
Issue	*opt*	string	text	value	no	none	This is the only user-defined data that is currently reserved. It is the issue number of the sheet and must be present in the title block for the release report of CVS to include the issue number.

Graphics Definition — Reserved Properties

A container for properties common to all graphics items is a symbol named _graphics_def_.ai_sym. The purpose of having properties on the graphics definition is to allow the selection and filter mechanisms to select graphics items.

Properties on graphics items allow you to be able to select graphics items using the Filter. There are no reserved properties on graphics items in Saber Sketch (variant only applies to Saber iQBus).

Symbol

The properties that are reserved and used for hierarchical symbols are:

Name	Value	Type	Value Editor	View	Protected	Qualifier	Description
variant	*opt*	list	list	value	no	none	The variant the graphics item is part of.

Other properties may be defined as desired.

Port

There are no reserved properties for ports on the graphics definition symbol.

Other Reserved Properties in Saber Sketch

The other properties reserved for either internal use or future expansion are:

Name	**Value**	**Type**	**Value Editor**	**View**	**Protected**	**Qualifier**	**Description**
connector	reference	string	text	value	yes	none	A nodal reference connector.
connector	lsplice	string	text	value	yes	none	A logical splice (solder dot used internally).
connector	*opt*	string	text	value	yes	none	Managed internally. Indicates sheet:instanc e.port of the referenced port.
conn_sheet	*opt*	string	text	value	yes	none	Managed internally. Indicates the sheet of the symbol with the referenced port.
conn_name	*opt*	string	text	value	yes	none	Managed internally. Indicates the instance name of the symbol with the referenced port.
conn_pin	*opt*	string	text	value	yes	none	Managed internally. Indicates the name of the referenced port.

Name	Value	Type	Value Editor	View	Protected	Qualifier	Description
conn_namepin	*opt*	string	text	value	yes	none	Managed internally. Indicates the name.pin of the referenced port.
ruler	*opt*	string	text	value	yes	none	Indicates the location in ruler coordinates of the referenced port.

Saber iQBus Symbols and Ports

Saber iQBus elements use the same properties as Saber Sketch elements of the same type, plus the variant property.

Component — Reserved Properties

A component is a symbol used in Saber iQBus that may be either a hierarchical or an HDL (leaf) symbol. It is different from a normal symbol in that it has one or more connector shells defined for it. The inline component is similar to a component except that it is built into a harness, has two connections, and is treated like a splice in Saber Bundle. It is used for in-line fusible links and other such devices. A harness component is a component that built into a harness and treated as a set of shells in Saber Bundle.

Symbol

The properties that are reserved and used for components are:

Name	Value	Type	Value Editor	View	Protected	Qualifier	Description
ref	*opt*	string	ref	value	no	none	Unique Name of component

Name	Value	Type	Value Editor	View	Protected	Qualifier	Description
symboltype	component inline_component	string	text	value	yes	none	Indicates the type of component.
variant	*opt*	list	list	value	no	none	List of variants the component is part of.
harness	*opt*	string	list	value	no	none	The harness the component is included in. This is not normally used. It is only used for harness components or inline components.
part_no	*opt*	string	text	value	no	HARNESS	The part number of the component.
part_type	*opt*	string	text	value	no	HARNESS	The type of the part for use in aerospace reports.
description	See right	string	ref	ref	no	HARNESS	Description of the component. The symbol should have a default description defined. The description will be replaced by the description on the ref list value attribute if it exists.

Name	Value	Type	Value Editor	View	Protected	Qualifier	Description
bundle_symbol	*opt*	string	text	value	no	BUNDLE	The symbol to use in Saber Bundle for this component if it is a “harness component”.
bundle_view	*opt*	string	text	value	no	BUNDLE	The view to use in Saber Bundle for this component if it is a “harness component”.
cost	*opt*	string	text	value	no	BUNDLE	Cost of the component.
weight	*opt*	string	text	value	no	BUNDLE	Weight of the component.
catia_pn	*opt*	string	text	value	no	CATIA 3D	Part number for Catia.
proe_pn	*opt*	string	text	value	no	PROE 3D	Part number for Pro/ENGIN EER.
ideas_pn	*opt*	string	text	value	no	IDEAS 3D	Part number for SDRC Ideas.
ug_pn	*opt*	string	text	value	no	UG 3D	Part number of Unigraphics Harness.

Name	Value	Type	Value Editor	View	Protected	Qualifier	Description
shell_*	*opt*	shell	shell	shell	no	HARNESS	This property contains information about the definition of the shell on the component and the mating shell selected. The property value editor supports the shell selected from the shell parts database in the symbol editor. In the harness drawing the view displays the shell part name and does not allow editing of the property.
passives	*opt*	passive	passives	passive	no	BUNDLE	Contains information about which passives are defined fro this inline component. It is not support for the component or the harness component.
primitive	name	string	text	value	yes	SABER	The name of the template for SABER.

Other properties may be defined and passed into the HDL model based on the model arguments.

Port

The reserved properties for ports on components fall into two groups. The first defines the relationship between the port and the connector shell. The second, for components that have HDL models, defines the type of port (analog, digital) of the connection on the model and is used for Hypermodel insertion. The reserved port properties for component are:

Name	Value	Type	Value Editor	View	Protected	Qualifier	Description
shell	*opt*	string	text	value	no	none	The shell that the port is associated with.
cavity	*opt*	string	text	value	no	none	The cavity in the shell that the port is associated with.

Shell Definition — Reserved Properties

The shell definition is a symbol used to define the shell database instance for the shells on the component connector and the inline connector. These shells are never edited directly. Their properties can be edited through the Connector Manager. In addition, the properties also are used to populate the properties of the bundle shell symbol.

Symbol

The properties that are reserved and used for shell definitions are:

Name	Value	Type	Value Editor	View	Protected	Qualifier	Description
ref	*opt*	string	ref	value	no	none	Unique Name of shell.
variant	*opt*	list	list	value	no	none	List of variants the shell is part of.

Name	Value	Type	Value Editor	View	Protected	Qualifier	Description
harness	*opt*	string	list	value	no	none	The harness the shell on the component or the shell of the inline is included in.
mating_harness	*opt*	string	text	value	no	HARNESS	Used on shell for inline connector only.
mating_shells	*opt*	list	list	value	no	HARNESS	The list of shells that can mate to this shell.
part_no	*opt*	string	text	value	no	HARNESS	The part number of the shell.
description	See right	string	ref	ref	no	HARNESS	Description of the shell. The symbol should have a default description defined. The description will be replaced by the description on the ref list value attribute if it exists or the description from the parts database.
bundle_symbol	*opt*	string	text	value	no	BUNDLE	The symbol to use in Saber Bundle for this shell.
bundle_view	*opt*	string	text	value	no	BUNDLE	The view to use in Saber Bundle for this shell.
cost	*opt*	string	text	value	no	BUNDLE	Cost of the shell.

Name	Value	Type	Value Editor	View	Protected	Qualifier	Description
weight	*opt*	string	text	value	no	BUNDLE	Weight of the shell.
catia_pn	*opt*	string	text	value	no	CATIA 3D	Part number for Catia.
proe_pn	*opt*	string	text	value	no	PROE 3D	Part number for Pro/ENGINEER.
ideas_pn	*opt*	string	text	value	no	IDEAS 3D	Part number for SDRC Ideas.
ug_pn	*opt*	string	text	value	no	UG 3D	Part number for Unigraphics Harness.
gender	*opt*	string	text	value	no	HARNESS	Gender of this shell.
color	*opt*	string	text	value	no	HARNESS	Color of this shell.
cavity_occupancy	*opt*	string	occupancy	occupancy	no	BUNDLE	Contains information about how many cavities are occupied.
passives	*opt*	passive	passive	passive	no	BUNDLE	Contains information about which passives are part of this shell.

Port

The reserved properties for ports on shell definitions define the order in which the cavities are to be filled. The reserved port properties for shell definitions are:

Name	Value	Type	Value Editor	View	Protected	Qualifier	Description
order	*opt*	string	text	value	no	none	The order of the cavity for display in the Connector Manager.
passives	*opt*	passive	passives	passives	no	BUNDLE	Contains information about which passives are part of this cavity.
terminal_ orientation	*opt*	string	text	value	no	HARNESS	Defines the orientation of the terminal with respect to the shell. It is used for inline connector symbols. It may also be used for component shells in the future. It is created and managed by the Connector Manager.

Free Terminal — Reserved Properties

The free terminal is a symbol used in Saber iQBus to represent the terminal on the end of a wire that is not part of a connector shell. Two examples are spade terminals and lug terminals.

Symbol

The properties that are reserved and used for free terminals are:

Name	Value	Type	Value Editor	View	Protected	Qualifier	Description
ref	*opt*	string	ref	value	no	none	Unique Name of free terminal
connector	terminal	string	text	value	yes	none	
connectortype	global	string	text	value	yes	none	If this free terminal makes a connection to a global node then the connector type must be global. Otherwise this property is not used.
variant	*opt*	list	list	value	no	none	List of variants the free terminal is part of.
harness	*opt*	string	list	value	no	none	The harness the free terminal is part of.
part_no	*opt*	string	text	value	no	HARNESS	The part number of the free terminal.

Name	Value	Type	Value Editor	View	Protected	Qualifier	Description
description	See right	string	ref	ref	no	HARNESS	Description of the free terminal. The symbol should have a default description defined. The description will be replaced by the description on the ref list value attribute if it exists.
bundle_symbol	*opt*	string	text	value	no	BUNDLE	The symbol to use in Saber Bundle for this free terminal.
bundle_view	*opt*	string	text	value	no	BUNDLE	The view to use in Saber Bundle for this free terminal.
cost	*opt*	string	text	value	no	BUNDLE	Cost of the free terminal.
weight	*opt*	string	text	value	no	BUNDLE	Weight of the free terminal.
passives	*opt*	passive	passives	passive	no	BUNDLE	Contains information about which passives are part of this free terminal.
catia_pn	*opt*	string	text	value	no	CATIA 3D	Part number for Catia.
proe_pn	*opt*	string	text	value	no	PROE 3D	Part number for Pro/ENGIN EER.
ideas_pn	*opt*	string	text	value	no	IDEAS 3D	Part number for SDRC Ideas.

Name	Value	Type	Value Editor	View	Protected	Qualifier	Description
ug_pn	*opt*	string	text	value	no	UG 3D	Part number for Unigraphics Harness.
name	*opt*	string	text	value	yes	none	Name of the node that the free terminal connects to. This will name the node of all wires connected. It is generally used for ground and power connections.
primitive	name	string	text	value	yes	SABER	The name of the template for SABER.

Other properties may be defined and passed into the HDL model based on the model arguments.

Port

There are no reserved properties for ports on the free terminal.

Physical Splice — Reserved Properties

The physical splice is a symbol used to represent the junctions of two or more physical wires. The symbol _splice_.ai_sym is used for the physical splice.

Symbol

The properties that are reserved and used for physical splices are:

Name	**Value**	**Type**	**Value Editor**	**View**	**Protected**	**Qualifier**	**Description**
ref	*opt*	string	ref	value	no	none	Unique Name of physical splice.
connector	splice	string	text	value	yes	none	
variant	*opt*	list	list	value	no	none	List of variants the splice is part of.
harness	*opt*	string	list	value	no	none	The harness the splice is part of.
part_no	*opt*	string	text	value	no	HARNESS	The part number of the splice.
description	See right	string	ref	ref	no	HARNESS	Description of the splice. The symbol should have a default description defined. The description will be replaced by the description on the ref list value attribute if it exists.
bundle_symbol	*opt*	string	text	value	no	BUNDLE	The symbol to use in Saber Bundle for this physical splice.
bundle_view	*opt*	string	text	value	no	BUNDLE	The view to use in Saber Bundle for this physical splice.
cost	*opt*	string	text	value	no	BUNDLE	Cost of the physical splice.

Name	Value	Type	Value Editor	View	Protected	Qualifier	Description
weight	*opt*	string	text	value	no	BUNDLE	Weight of the physical splice.
passives	*opt*	string	text	value	no	BUNDLE	Contains information about which passives are part of this splice.
primitive	*opt*	string	text	value	yes	BUNDLE	The name of the template for SABER.

Port

There are no reserved properties for ports on the physical splice.

Physical Wire — Reserved Properties

The physical wire is the cut wire that makes up the harness. It ends on terminals or splices. It is not a logical wire.

Wire

The properties that are reserved and used for physical wires are:

Name	Value	Type	Value Editor	View	Protected	Qualifier	Description
ref	*opt*	string	ref	value	no	none	Unique Name of physical wire. Same as wire name.
variant	*opt*	list	list	value	no	none	List of variants the physical wire is part of.

Name	**Value**	**Type**	**Value Editor**	**View**	**Protected**	**Qualifier**	**Description**
harness	*opt*	string	list	value	no	none	The harness the physical wire is part of.
part_no	*opt*	string	wire	value	no	HARNESS	The part number of the physical wire.
cost	*opt*	string	wire	value	no	BUNDLE	Cost per unit length of the physical wire.
weight	*opt*	string	wire	value	no	BUNDLE	Weight per unit length of the physical wire.
length	*opt*	string	text	value	no	none	Length of the wire.
color	*opt*	string	wire	value	no	BUNDLE	Color of the wire.
wire_type	*opt*	string	wire	value	no	HARNESS SABER	The type of the wire.
bend_radius	*opt*	string	wire	value	no	HARNESS	The bend radius of the wire.
gauge	*opt*	string	wire	value	no	HARNESS SABER	The gauge of the wire.
area	*opt*	string	wire	value	no	HARNESS SABER	The area of the conductor in the wire.
diameter	*opt*	string	wire	value	no	HARNESS SABER	The diameter of the wire.

Name	Value	Type	Value Editor	View	Protected	Qualifier	Description
primitive	wirep	string	text	value	no	SABER	The name of the template for SABER.

Other properties may be defined and passed into the HDL model based on the model arguments. If these are to come from the wire parts database then use a property value editor of wire.

Physical Cable — Reserved Properties

A cable is a combination of one or more physical wires which may or may not be surrounded by wrapping and/or shielding. It is not a logical or physical wire and does not terminate on symbols or components.

Cable

The properties that are reserved and used for cables are:

Name	Value	Type	Value Editor	View	Protected	Qualifier	Description
ref	*opt*	string	ref	value	no	none	Unique Name of cable. Same as cable name.
length	*req*	string	text	value	no	none	Length of the cable.
variant	*opt*	list	list	value	no	none	List of variants the cable is part of.
harness	*opt*	string	list	value	no	none	The harness the cable is part of.
part_no	*opt*	string	wire	value	no	HARNESS	The part number of the cable.

Name	Value	Type	Value Editor	View	Protected	Qualifier	Description
cost	*opt*	string	wire	value	no	BUNDLE	Cost per unit length of the cable.
weight	*opt*	string	wire	value	no	BUNDLE	Weight per unit length of the cable.
color	*opt*	string	cable	value	no	BUNDLE	Color of the cable.
wire_type	*opt*	string	cable	value	no	SABER HARNESS	The type of wire in the cable.
bend_radius	*opt*	string	cable	value	no	HARNESS	The bend radius of the cable.
primitive	wirep	string	text	value	no	SABER	The name of the template for SABER.
diameter	*opt*	string	wire	value	no	HARNESS SABER	The diameter of the wire in the cable.
area	*opt*	string	wire	value	no	HARNESS SABER	The area of the conductor of the wire in the cable.
gauge	*opt*	string	wire	value	no	HARNESS SABER	The gauge of the wire in the cable.

Other properties may be defined and passed into the HDL model based on the model arguments. If these are to come from the cable parts database then use a property value editor of cable.

Physical Cable Definition — Reserved Properties

A container for properties common to all physical cables is a symbol named _cable_def_.ai_sym. The supported properties on a physical cable definition are identical to those on the physical cable.

Cable

The properties that are reserved and used for physical cables are:

Name	Value	Type	Value Editor	View	Protected	Qualifier	Description
ref	*opt*	string	ref	value	no	none	Unique Name of cable. Same as cable name.
length	*req*	string	text	value	no	none	Length of the cable.
variant	*opt*	list	list	value	no	none	List of variants the cable is part of.
harness	*opt*	string	list	value	no	none	The harness the cable is part of.
part_no	*opt*	string	wire	value	no	HARNE SS	The part number of the cable.
cost	*opt*	string	wire	value	no	BUNDL E	Cost per unit length of the cable.
weight	*opt*	string	wire	value	no	BUNDL E	Weight per unit length of the cable.
color	*opt*	string	cable	value	no	BUNDL E	Color of the cable.
wire_type	*opt*	string	cable	value	no	SABER HARNE SS	The type of wire in the cable.

Name	Value	Type	Value Editor	View	Protected	Qualifier	Description
bend_radius	*opt*	string	cable	value	no	HARNESS	The bend radius of the cable.
primitive	wirep	string	text	value	no	SABER	The name of the template for SABER.
diameter	*opt*	string	wire	value	no	HARNESS SABER	The diameter of the wire in the cable.
area	*opt*	string	wire	value	no	HARNESS SABER	The area of the conductor of the wire in the cable.
gauge	*opt*	string	wire	value	no	HARNESS SABER	The gauge of the wire in the cable.

Other properties may be defined and passed into the HDL model based on the model arguments. If these are to come from the cable parts database then use a property value editor of cable.

Inline Connector Symbol — Reserved Properties

An inline connector is a symbol that associates a pair of cavities in two connector shell housings with the physical wires that are connected to terminals in these cavities. Because the inline connector is only used to represent this association the underlying shell database contains all of the information that defines the details related to the cavity, terminal, and passive definitions the only information present on the symbol is related to simulation.

Symbol

The properties that are reserved and used for in-line connectors are:

Name	Value	Type	Value Editor	View	Protected	Qualifier	Description
ref	*opt*	string	ref	value	no	none	Unique name of in-line connector.
primitive	name	string	text	value	no	SABER	The name of the template for SABER if there in-line connector is to be modeled.

Other properties may be defined and passed into the HDL model based on the model arguments. If these are to come from the wire parts database, use a property value editor of wire.

Port

The reserved properties for ports on in-line connectors define the connector shell housing that the port represents. The reserved port properties for in-line connectors are:

Name	Value	Type	Value Editor	View	Protected	Qualifier	Description
shell	name	string	text	value	no	HARNESS	The name of the shell the cavity is in.
cavity	name	string	text	value	no	HARNESS	The name of the cavity to which the port is associated.

Sheet Symbol — Reserved Properties

The sheet symbol is a mechanism to support the representation of a group of connections to be displayed as a symbol in one sheet and have the connections to ports and/or splices in another sheet. The port of a sheet symbol is connected to other ports and splices by wires. The sheet symbol port takes as properties the names of those other ports and splices.

Symbol

The properties that are reserved and used for sheet symbols are:

Name	Value	Type	Value Editor	View	Protected	Qualifier	Description
connector	sheet	string	text	value	yes	none	

Port

The reserved properties for ports on sheet symbols define the name of the object that is connected. The port type that is used defines whether the type of the port is a port reference or a splice reference. The reserved port properties for sheet symbols are:

Name	Value	Type	Value Editor	View	Protected	Qualifier	Description
connection	sheet:sym_i ns tance.port	string	text	value	no	HARNE SS	Reference to the port on the object referenced.

Reference Symbol — Reserved Properties

The reference symbol is used by the shared wire to allow for a physical wire to be distributed on one or two sheets without being visually connected. The reference symbol displays information about what the other end of the shared wire is connected to. The insertion and property setting for reference symbols is managed as part of the shared wire. The user can control the property visibility.

Symbol

The properties that are reserved and used for reference symbols are:

Name	Value	Type	Value Editor	View	Protected	Qualifier	Description
connector	reference	string	text	value	yes	none	
connection	sheet:sym_in s tance.port	string	text	value	yes	HARNE SS	The connected element.
ruler	*opt*	string	text	value	yes	HARNE SS	The grid location of the connected element.
conn_she et	*opt*	string	text	value	yes	HARNE SS	Managed internally. Indicates the sheet of the symbol with the referenced port.
conn_nam e	*opt*	string	text	value	yes	HARNE SS	Managed internally. Indicates the instance name of the symbol with the referenced port.
conn_pin	*opt*	string	text	value	yes	HARNE SS	Managed internally. Indicates the name of the referenced port.
conn_nam epin	*opt*	string	text	value	yes	HARNE SS	Managed internally. Indicates the name.pin of the referenced port.

Port

There are no reserved port properties for reference symbols.

Saber Bundle Symbols and Ports

The following properties are reserved for Saber Bundle Symbols and Ports.

Bundle Shell — Reserved Properties

The bundle shell is a symbol representing a connector shell in the bundle drawing. The properties on the shell either come from information in the shell database of the corresponding wiring diagram or can be entered manually. The default symbol used for the bundle shell is _bundle_shell_.ai_sym.

Symbol

The properties that are reserved and used for bundle shells are:

Name	Value	Type	Value Editor	View	Protected	Qualifier	Description
ref	*opt*	string	ref	value	no	none	Unique Name of shell.
symboltype	shell	string	text	value	yes	none	Identifies type of symbol.
variant	*opt*	list	list	value	no	none	List of variants the shell is part of.
mating_harness	*opt*	string	text	value	no	HARNESS	Used on shell for inline connector only.
part_no	*opt*	string	text	value	no	HARNESS	The part number of the shell.

Name	Value	Type	Value Editor	View	Protected	Qualifier	Description
description	See right	string	ref	ref	no	HARNESS	Description of the shell. The symbol should have a default description defined. The description will be replaced by the description on the ref list value attribute if it exists.
cost	*opt*	string	text	value	no	BUNDLE	Cost of the shell.
weight	*opt*	string	text	value	no	BUNDLE	Weight of the shell.
color	*opt*	string	text	value	no	HARNESS	Color of this shell.
cavity_occupancy	*opt*	string	occupancy	occupancy	no	BUNDLE	Contains information about how many cavities are occupied.
passives	*opt*	passive	passives	passive	no	BUNDLE	Contains information about which passives are part of this shell. The value can only be set in the Connector Manager.

Port

There are no reserved port properties for bundle shells.

Bundle Terminal — Reserved Properties

The bundle terminal is a symbol representing a free terminal in the bundle drawing. The properties on the shell either come from information on the free terminal of the corresponding wiring diagram or can be entered manually. The default symbol used for the bundle terminal is _bundle_terminal_.ai_sym.

Symbol

The properties that are reserved and used for free terminals are:

Name	Value	Type	Value Editor	View	Protected	Qualifier	Description
ref	*opt*	string	ref	value	no	none	Unique Name of free terminal.
symboltyp e	terminal	string	text	value	yes	none	Identifies type of symbol
variant	*opt*	list	list	value	no	none	List of variants the free terminal is part of.
part_no	*opt*	string	text	value	no	HARNE SS	The part number of the free terminal.
description	See right	string	ref	ref	no	HARNE SS	Description of the free terminal.
cost	*opt*	string	text	value	no	BUNDL E	Cost of the free terminal.
weight	*opt*	string	text	value	no	BUNDL E	Weight of the free terminal.

Name	Value	Type	Value Editor	View	Protected	Qualifier	Description
passives	*opt*	passive	passives	passive	no	BUNDLE	Contains information about which passives are part of this free terminal. The value can only be set in the Connector Manager.

Port

There are no reserved port properties for bundle terminals.

Bundle Splice — Reserved Properties

The bundle splice is a symbol representing a splice in the bundle drawing. The properties on the splice either come from information on the splice of the corresponding wiring diagram or can be entered manually. The default symbol used for the bundle splice is _bundle_splice_.ai_sym.

Symbol

The properties that are reserved and used for splices are:

Name	Value	Type	Value Editor	View	Protected	Qualifier	Description
ref	*opt*	string	ref	value	no	none	Unique Name of splice.
symboltype	splice	string	text	value	yes	none	Identifies type of symbol.
variant	*opt*	list	list	value	no	none	List of variants the splice is part of.

Name	Value	Type	Value Editor	View	Protected	Qualifier	Description
part_no	*opt*	string	text	value	no	HARNESS	The part number of the splice.
description	See right	string	ref	ref	no	HARNESS	Description of the splice. The symbol should have a default description defined. The description will be replaced by the description on the ref list value attribute if it exists.
cost	*opt*	string	text	value	no	BUNDLE	Cost of the splice.
weight	*opt*	string	text	value	no	BUNDLE	Weight of the splice.
passives	*opt*	passive	passives	passive	no	BUNDLE	Contains information about which passives are part of this splice. The value can only be set in the Connector Manager.

Port

There are no reserved port properties for splices.

Bundle Harness Component — Reserved Properties

The bundle harness component is a symbol representing a component built into a harness. The shells on the component are represented with a bundle shell. This symbol is provided so that there can be a graphical representation of the component in the bundle drawing. The properties on the harness component either come from information on the component of the corresponding wiring diagram or can be entered manually. Since all bundle segments are connected to the related shells, there are not ports on the bundle harness component. The default symbol used for the bundle harness component is _bundle_ component_.ai_sym.

Symbol

The properties that are reserved and used for harness components are:

Name	Value	Type	Value Editor	View	Protected	Qualifier	Description
ref	*opt*	string	ref	value	no	none	Unique Name of harness component.
symboltyp e	component	string	text	value	yes	none	Identifies type of symbol.
variant	*opt*	list	list	value	no	none	List of variants the harness component is part of.
part_no	*opt*	string	text	value	no	HARNE SS	The part number of the harness component.

Name	**Value**	**Type**	**Value Editor**	**View**	**Protected**	**Qualifier**	**Description**
description	See right	string	ref	ref	no	HARNESS	Description of the harness component. The symbol should have a default description defined. The description will be replaced by the description on the ref list value attribute if it exists.
cost	*opt*	string	text	value	no	BUNDLE	Cost of the harness component.
weight	*opt*	string	text	value	no	BUNDLE	Weight of the harness component
passives	*opt*	passive	passives	passive	no	BUNDLE	Contains information about which passives are part of this harness component. This value can only be set in the Connector Manager.

Port

There are no ports for harness components.

Bundle Inline Component — Reserved Properties

The bundle inline component is a symbol representing an inline component in the bundle drawing. The properties on the inline component either come from information on the inline component of the corresponding wiring diagram or can be entered manually. The default symbol used for the bundle inline component is _bundle_inline_component_.ai_sym.

Symbol

The properties that are reserved and used for inline components are:

Name	Value	Type	Value Editor	View	Protected	Qualifier	Description
ref	*opt*	string	ref	value	no	none	Unique Name of inline component.
symboltype	inline_component	string	text	value	yes	none	Identifies type of symbol.
variant	*opt*	list	list	value	no	none	List of variants the inline component is part of.
part_no	*opt*	string	text	value	no	HARNESS	The part number of the inline component.

Name	Value	Type	Value Editor	View	Protected	Qualifier	Description
description	See right	string	ref	ref	no	HARNESS	Description of the inline component. The symbol should have a default description defined. The description will be replaced by the description on the ref list value attribute if it exists.
cost	*opt*	string	text	value	no	BUNDLE	Cost of the inline component.
weight	*opt*	string	text	value	no	BUNDLE	Weight of the inline component.
passives	*opt*	passive	passives	passive	no	BUNDLE	Contains information about which passives are part of this inline component. This value can only be set in the Connector Manager.

Port

There are no reserved port properties for inline components.

Bundle Passive — Reserved Properties

The bundle passive is a symbol representing a passive in the bundle drawing. The properties either come from the passive elements in the Parts Gallery or can be entered manually. The default symbol used for the bundle passive is _bundle_passive_.ai_sym.

Symbol

The properties that are reserved and used for passives are:

Name	Value	Type	Value Editor	View	Protected	Qualifier	Description
symboltyp e	passive	string	text	value	yes	none	Identifies type of symbol.
variant	*opt*	list	list	value	no	none	List of variants the passive is part of.
part_no	*opt*	string	text	value	no	HARNE SS	The part number of the passive.
description	See right	string	ref	ref	no	HARNE SS	Description of the passive.
cost	*opt*	string	text	value	no	BUNDL E	Cost of the passive if no length specified otherwise it is the cost per unit length if the length is specified.
weight	*opt*	string	text	value	no	BUNDL E	Weight of the passive if no length specified otherwise it is the weight per unit length if the length is specified.

Name	Value	Type	Value Editor	View	Protected	Qualifier	Description
length	*opt*	string	text	value	no	BUNDLE	Length of the passive if the passive has extent. If this is not set or missing then the passive is a lumped item.
type	*opt*	string	text	value	no	BUNDLE	Type of the passive. This identifies the kind of passive which can then be used by other analysis programs.
passives	*opt*	string	text	value	no	BUNDLE	Contains information about which passives are part of this passive. This is used to identify the passive parts which are part of this passive but not displayed graphically.

Port

There are no reserved port properties for passives.

Bundle Segment — Reserved Properties

The bundle segment is a collection of wires. The bundle segment properties define characteristics about the bundle segment.

Bundle Segment

The properties that are reserved and used for bundle segments are:

Name	Value	Type	Value Editor	View	Protected	Qualifier	Description
length	*opt*	string	text	value	no	BUNDLE	The length of the bundle segment.
variant	*opt*	list	list	value	no	none	List of variants the bundle segment is part of.
wires	*opt*	string	text	value	yes	BUNDLE	The value of the wires property is defined by the system and is a list of the wires in the bundle segment.

Bundle Segment Definition — Reserved Properties

A container for properties common to all bundle segments is a symbol named _bundle_segment_def_.ai_sym. The supported properties on a bundle segment definition are identical to those on the bundle segment.

Bundle Segment Definition

Name	Value	Type	Value Editor	View	Protected	Qualifier	Description
length	*opt*	string	text	value	no	BUNDLE	The length of the bundle segment.
variant	*opt*	list	list	value	no	none	List of variants the bundle segment is part of.
wires	*opt*	string	text	value	yes	BUNDLE	The value of the wires property is defined by the system and is a list of the wires in the bundle segment.

The properties that are reserved and used for bundle segment definitions are:

Parts Databases

Parts database files allow you to use the parts browser to find custom parts when drawing a schematic.

Shell Parts Database — Reserved Properties

The shell parts database contains information used to define a specific shell. The properties contained in the parts database are applied to the properties in the Shell Definition symbol. The properties in the parts database contain only the values of the properties so that any display or editing attributes come from the properties on the symbol. As such the only fields of interest are the Name, Value, and Description.

Name	Value	Description
mating_shells	*opt*	The list of shells that can mate to this shell.

Name	Value	Description
part_no	*opt*	The part number of the shell.
description	See right	Description of the shell.
bundle_symbol	*opt*	The symbol to use in Saber Bundle for this shell.
bundle_view	*opt*	The view to use in Saber Bundle for this shell.
cost	*opt*	Cost of the shell.
weight	*opt*	Weight of the shell.
catia_pn	*opt*	Part number for Catia.
proe_pn	*opt*	Part number for Pro/ENGINEER.
ideas_pn	*opt*	Part number for SDRC Ideas.
ug_pn	*opt*	Part number for Unigraphics Harness.
gender	*opt*	Gender of this shell.
color	*opt*	Color of this shell.
passives	*opt*	Contains information about which passives are part of this shell.

Wire Parts Database — Reserved Properties

The wire parts database contains information used to define a wire. The properties contained in the parts database are applied to the properties on a physical wire that have a property value editor of wire. The properties in the parts database contain only the values of the properties so that any display or editing attributes come from the properties on the wire. As such the only fields of interest are the Name, Value, and Description.

Name	Value	Description
part_no	*opt*	The part number of the physical wire.

Name	**Value**	**Description**
cost	*opt*	Cost per unit length of the physical wire.
weight	*opt*	Weight per unit length of the physical wire.
color	*opt*	Color of the wire.
length	*opt*	Length of the wire.
wire_type	*opt*	The type of the wire.
bend_radius	*opt*	The bend radius of the wire.
gauge	*opt*	The gauge of the wire.
area	*opt*	The area of the conductor in the wire.
diameter	*opt*	The diameter of the wire.

Cable Parts Database — Reserved Properties

The cable parts database contains information used to define a cable. The properties contained in the parts database are applied to the properties on a physical cable that have a property value editor of cable. The properties in the parts database contain only the values of the properties so that any display or editing attributes come from the properties on the cable. As such the only fields of interest are the Name, Value, and Description.

Name	**Value**	**Description**
part_no	*req*	The part number of the physical cable.
wire_type	*opt	The type of wire in the cable.
cost	*opt	Cost per unit length of the physical cable.
weight	*opt	Weight per unit length of the physical cable.
length	*opt	Length of the cable.
color	*opt	Color of the cable.

Name	Value	Description
bend_radius	*opt	The bend radius of the cable.
diameter	*opt	The diameter of the wire in the cable.
area	*opt	The area of the conductor of the wire in the cable.
gauge	*opt	The gauge of the wire in the cable.

Other properties may be defined and passed into the database based on the model arguments.

Passive Parts Database — Reserved Properties

The passive parts database contains information used to define a passive. The properties contained in the parts database are applied to the properties on any element that uses a passive property value editor. The properties in the parts database contain only the values of the properties so that the display and/or editing attributes come from the properties on the element. As such the only fields of interest are the Name, Value, and Description.

Name	Value	Description
part_no	*opt	The part number of the passive.
type	*opt	Type of the passive.
weight	*opt	Weight per unit of the passive.
length	*opt	Length of the passive.
family	*opt	Passive family.

11

Using a Mapping File to Map Symbols

This section describes how to create a mapping file. A mapping file contains entries to define the relationship between the symbol properties of an individual symbol and the template name, template parameters, pin names, and other features of the corresponding template. If you do not want to modify your symbols, you should follow the guidelines presented in this section. Topics covered in this section include:

Overview of Mapping Files

Properties of some symbols have features for which the netlister needs supplemental information to perform the translation. You can provide this supplemental information by means of a configuration file called a mapping file.

Some symbols of the simulator libraries and all the symbols of the Mentor Graphics gen_lib and accuparts_lib, the Cadence sample, basic, and analogLib, and the Viewlogic viewspice (analog) and builtin libraries require specific support from standard mapping files provided with the Frameway integration software.

Symbols that are not obtained from the gen_lib, accuparts_lib, sample, basic, analogLib, viewspice, builtin or simulator libraries are not supported by the standard mapping files. In this case, you may need to create your own mapping file. This section shows how the mapping file was developed for a small subset of symbols. These procedures are intended as guidelines for you to create a similar mapping file for your library.

What Is a Mapping File?

A mapping file is a text file in which data is entered in a specified format. It typically contains several sections that are set off by braces. Each section contains specific information about symbol properties that the netlister uses to translate schematic symbols into a netlist. All sections are optional and may be left out if not needed. For more specific information about mapping files, including syntax and individual descriptions of each field, refer to Chapter 9: Mapping File Reference.

A mapping file is a configuration file that provides information necessary for the netlister to associate symbols with templates. A mapping file is a simple text file. Because it is composed of simple text, you can use any text editor that creates an unformatted text file to create or modify a mapping file.

Creating a Mapping File

Structure of a Mapping File

The mapping file can contain any of the following sections:

- A tables section
- An enums section
- One or more definition sections: saber, partner, and undetermined
- An include or exclude section

The tables section contains tables that can be referenced from any of the three definition sections.

You use the enums section to specify properties corresponding to template parameters that accept enumerated values.

You use the definition sections saber, partner, and undetermined to define the correspondence between schematic entities and simulator entities. They can be arranged in any order or omitted if not needed. Each of the definition sections can contain one or more generic entries. A generic entry can contain one or more specific symbol entries.

You use a generic entry to specify mapping information that pertains to an entire symbol library. You use a specific symbol entry to map features specific to a symbol. If both a generic and a specific symbol entry in a mapping file exist for the same feature, the specific symbol entry overrides the generic entry for that symbol.

The include section is no longer used by the netlister and may be left blank.

The exclude section is used to list symbols that are to be ignored by the netlister when it creates the netlist. For example, you can exclude one or more symbols from the netlist to increase the speed of a simulation. Or, you can exclude symbols from the netlist that have no underlying functional representation, such as vdd or vcc. The exclude section takes the following form where symbol_a, symbol_b, and symbol_c are the names of symbols to be excluded from the netlist.

```
exclude{symbol_a,symbol_b,symbol_c}
```

The most noticeable feature of a definition section in a mapping file is the use of fields. A single entry line in a definition section consists of 21 fields separated by colons. As the following example illustrates, each field assigns a specific function to the data it contains. If you do not use a field, leave it blank.

```
symbol name : simulation model names : simulator target :
simulator model parameters : parameter-list property : port
names : port types : port direction : port order : port
parameters : net names : net parameters : template directory :
reference designator or instance name : nodes on properties :
primitives : graphical modeling : exclude as parameters: : : ;
```

Special Characters Used in the Mapping File

The mapping file entries make use of special characters to control the translation process.

%{} To specify the substitution of a property value, you can enclose a property name between the special character pair %{ and the character }. The special character pair %{ indicates the beginning of the name of the enclosed property while the special character } indicates the end of the name of the enclosed property.

Example: %{freq}

" "	In certain cases, you can enclose a string between the special character pair " ". The quotes indicate that the enclosed string is to be interpreted literally, except for places where property values are to be substituted. If, for example, in the expression "gain=%{gain}", the value of the property gain enclosed in braces is equal to 5k, the netlister enters gain=5k into the netlist. In other words, %{gain} is replaced by the value of the property gain while the expression gain= is taken literally. Note that if the value of gain is " " or NULL, this mapping will fail.
<-	In certain cases, you can place a string within quotes on the right side of a left arrow <-. This configuration indicates that the value of the string will be assigned to the parameter name on the left side of the arrow. Example: gain<-"(h=1.41)" Otherwise, if there is nothing to the left of the arrow, the netlister places the value directly into the netlist without an assignment.
<>	The character pair <> indicates that the value to be substituted is the name assigned to the appropriate data base entity such as a pin rather than the value of a property. Example: some_pin_dirs[<>] This entry is specified in the tables section as follows: some_pin_dirs[dig_in]->in,[dig_out]-> out,[*]->null

Saving Your Mapping File in a Retrievable Location

The following sections describe how to save your mapping file depending on the symbol source: Cadence and Viewlogic symbols, or Mentor Graphics single and multiple symbols.

Cadence and Viewlogic

If you are using Cadence or Viewlogic symbols, you can place the contents of the mapping file in a file named user.map in the data search path where it will be read automatically by the netlister. The entries in this mapping file take precedence over the entries in all other automatically read mapping files at start-up. Alternatively, you can place the mapping file directly in the data search path. The standard for naming mapping files is to add the extension .map to the file name. If you add a different extension to the file name, a warning message appears when the file is processed.

Mentor Graphics, single symbols

This topic is covered in Chapter 9: Mapping File Reference.

Mentor Graphics, multiple symbols

This topic is covered in Chapter 9: Mapping File Reference.

Designating that the Netlister Use the Mapping File

This section describes how to designate that the netlister use the mapping file, depending on the symbol source: Mentor Graphics, Cadence, or Viewlogic.

Mentor Graphics

You specify that your mapping file is to be used by the netlister by entering the file name in the Symbol Library Mapping File Name(s) field of the Saber Options dialog box. The entries in this file take precedence over the mapping files that are read automatically at start-up (except single-symbol mapping files). You can access the Saber Options dialog box either from the Saber menu item on the DVE session window menu banner or from the SABER icon by selecting one of the following menu/palette sequences:

- Click on the SABER icon.
- In the Saber Simulator Startup dialog box, ensure that the Invoke Simulator field is set to yes.

  OR
- Click on the Options button to invoke the Saber Options dialog box.
- From the Saber item on the menu banner, select:

  Saber>Invoke Saber> Saber Simulator>Saber Simulator Startup>Options>Saber Options

In either case, if you set the Invoke Simulator field to no in the Saber Simulator Startup dialog box and then click on the Options button, the DVETOS Options dialog box appears.

Cadence

You specify that your mapping file is to be used by the netlister by entering the file name in the Mapping Files field of the Saber Invocation Options form. The entries in this file take precedence over the mapping files that are read automatically at start-up. You can access the Saber Invocation Options form by selecting the following menu sequence from the Saber item on the menu bar:

Saber>Saber>Simulator Startup>Options>Saber Invocation Options

Viewlogic

You specify that this mapping file is to be used by the netlister by adding the file name in the Mapping Files field of the Extract Design Options dialog box. To do so, follow these steps:

- Select the Saber>Saber UI Window item on the ViewDraw window menu banner to open the Saber Simulation Window.
- Select the Saber>Extract Options item on the Saber Simulation Window menu banner. This causes the Extract Design Options dialog box to appear.
- To specify the name of your mapping file, click on the Add Mapping File? button in the Mapping Files field. This causes the Add User Mapping File dialog box to appear.
- Move the cursor to the Mapping File field, enter the name of your mapping file and click on Accept.
- Click on Accept in the Extract Design Options dialog box to set the options you specified.

Mapping File Examples

As described previously, the mapping file consists of generic and specific symbol entries which each consist of 21 fields. Typically, however, you will not need to use all of the fields in a particular entry. In fact, quite often you will use only a few fields in a given generic or specific symbol entry. The steps in this section outline the process by which you would usually map most symbols. That is, the following steps do not discuss all fields of the mapping file; rather, they present several of the most commonly used fields. For a specific description of each field, refer to Chapter 9: Mapping File Reference.

The dvetos.map, catos.map, and vwltosv.map mapping files contain entries for many of the commonly used symbols, including those in the examples that follow. This section describes the method that was used to create some of these mapping entries. The description begins with the following blank mapping file. This mapping file contains all relevant sections but no entries.

```
tables{
}
enums{
}
saber{
#first generic entry
 : : : : : : : : : : : : : : : : : : : : {
#specific symbol entries
: : : : : : : : : : : : : : : : : : : : ;
}

#second generic entry
 : : : : : : : : : : : : : : : : : : : : {
#specific symbol entries
 : : : : : : : : : : : : : : : : : : : : ;
}
}
undetermined{
#generic entry
 : : : : : : : : : : : : : : : : : : : : {
}
}
```

Looking at one symbol at a time, the entries that map the properties of each symbol will be added to the blank mapping file. For each symbol, new entries are shown in bold text to distinguish them from previously-added entries.

The following examples are presented in three sections, one each for Mentor Graphics users, Cadence users, and Viewlogic users. Each section contains three examples: a voltage source symbol, a ground symbol, and an ASIC symbol.

Mentor Graphics Symbol Mapping Examples
Cadence Symbol Mapping Examples
Viewlogic Symbol Mapping Examples

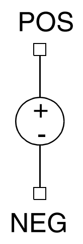
Mentor Graphics Symbol Mapping Examples

This section contains three examples (a voltage source symbol, a ground symbol, and an ASIC symbol) for use with the Frameway integration into the Mentor Graphics environment.

Mentor Graphics: Mapping the Voltage Source Symbols
Mentor Graphics: Mapping the Ground Symbol
Mentor Graphics: The Schematic for the ASIC Symbol

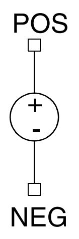
Mentor Graphics: Mapping the Voltage Source Symbols

The accuparts_lib symbol library contains the symbol named voltage_source. This symbol is used for all types of voltage sources. As a result, the symbols for a DC and a pulse voltage sources could look alike. The instance property values, however, will differ as illustrated in the figure below. The mapping entries for this symbol were created for the standard mapping file dvetos.map as shown in the following procedure. The completed entries are shown at the end of this procedure in bold text.



Instance Properties
inst=V1
instpar=5v

Pin Properties
PIN=NEG
PIN=POS



Instance Properties
inst=V2
instpar=pulse(0 5 10n 10n)

Pin Properties
PIN=NEG
PIN=POS

To map the voltage source symbols, perform these steps:

1. Specify the name of the library in the first field of the first saber generic entry.

   You specify the name of the symbol library in the first field of a saber generic entry.

The name accuparts_lib is placed in the first field of the first generic entry in the saber section.

The voltage source symbol is located in the accuparts_lib library, so accuparts_lib is placed in this generic entry. All symbols from this library are mapped by using specific symbol entries in this generic entry.

2. Specify the symbol name.

   You use the first field of the specific symbol entry to specify symbol names.

   The symbol name voltage_source is placed in the first field of the specific symbol entry.

3. Specify the Saber Simulator template to be used in the netlist.

   You use the second field of the specific symbol entry to specify the name of the Saber Simulator template to be used in the simulation. The file that contains the Saber Simulator template for a voltage source is v.sin.

   The name (v) of the template is placed in the second field of the specific symbol entry.

4. Specify the source of the simulator-model (template) parameters.

   You use the fourth field of the specific symbol entry to specify the source of the properties. If more than one source is listed, the sources must be separated by commas.

   The netlister must be informed that the property instpar of the voltage_source symbol provides the parameters for the v.sin template. The parameters are given in SPICE format, which is incompatible with the Saber Simulator format. Therefore, the mapping file function splist is needed to convert the list of the SPICE source parameters to a format appropriate for Saber Simulator template parameters.

   The following expression is placed in the fourth field of the specific symbol entry:

   ```
   <-splist(instpar)
   ```

   In this expression, the left arrow does not point to a name. Consequently, the result of the conversion is placed directly into the netlist. As an example, if the property instpar has the value 10V, the value is converted and placed in the Saber Simulator netlist as the numeric 10.

Note that your version of the dvetos.map file may include a mapping file function other than splist to map the instpar property of the voltage_source symbol. If so, either function is valid and you can use either one when you create your own mapping file.

5. Specify the correspondence between template and symbol connection points.

   The sixth field of the specific symbol entry specifies the source of a pin (port) name. In this example, the correspondence between the symbol pin names pos and neg and the connection point names p and m of the Saber Simulator templates is specified as a table entry in the tables section. This table entry has the unique name Analogy_spice_pins, which is referenced by the entry in the sixth field of the specific symbol entry.

   The following expression was placed in the sixth field of the specific symbol entry:

   ```
   Analogy_spice_pins[<>]
   ```

   The character pair <> indicates that the names to be used to look up the translation in the table entry are the names assigned to the pins (pos and neg).

   The following table entry is placed in the tables section:

   ```
   Analogy_spice_pins[POS]->p,[pos]->p,
   [NEG]->m,[neg]->m
   ```

   The Analogy_spice_pins table entry translates POS to p, pos to p, NEG to m, and neg to m.

6. Specify the property that defines the reference designator of the symbol.

   The 14th field of a specific symbol entry specifies the source of the value of the reference designator (or instance name). In this example, the value of the symbol property inst provides the reference designators for the Saber Simulator netlist for each of the sources V1 and V2.

   The property name inst is placed in the 14th field of the specific symbol entry.

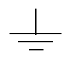
The entries that are needed to map the voltage_source symbol are added to the dvetos.map file as indicated in bold text in the following example:

```
tables{
Analogy_spice_pins[POS]->p,[pos]->p,[NEG]->m, [neg]-
>m
}
enums{
}
saber{
#first generic entry
accuparts_lib:::::::::::::::::::{
#specific symbol entries
voltage_source:v:: <-
splist(instpar)::Analogy_spice_pins[<>]:::::::
inst::::::;
}
#second generic entry
:::::::::::::::::::::{
#specific symbol entries
:::::::::::::::::::::;
}
}
undetermined{
#generic entry
::::::::::::::::::::: {
}
}
```

Mentor Graphics: Mapping the Ground Symbol

The symbol named ground is a CLASS G symbol from the accuparts_lib library. This symbol, shown below, is a device used to name global nets in Design Architect schematics and has no template associated with it. Net names and the symbol ground are mapped in the undetermined section, which is used for schematic entities and nets.

The completed entries in the mapping file for the ground symbol are shown in bold text at the end of this section.



Instance Properties
Global=ground
CLASS=g
PIN=ground

You can name a library in the first field of the generic entry. However, the first field of the generic entry of the undetermined section is left empty, which signifies that this generic entry provides the default generic mapping for the undetermined section.

Specify the net name of the ground symbol.

The following expression is placed in the 11th field of the generic entry of the undetermined section to reference an entry in the tables section:

```
Analogy_nets[global_net]
```

The following expression is placed in the tables section:

```
Analogy_nets[ground]->0,[GROUND]->0,[*]->*
```

The value of the property global_net (created from the global property of CLASS G symbols, such as the ground symbol) is converted to 0 (zero) by the Analogy_nets table entry. The expression [*]->* indicates that any name other than ground or GROUND for the global property of a CLASS G symbol remains the same.

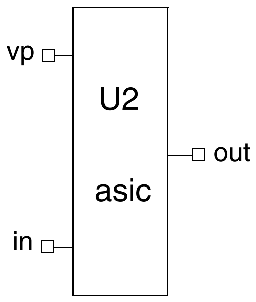
All entries required to map the ground symbol are added to the dvetos.map file as indicated in bold text in the following example:

```
tables{
Analogy_nets[ground]->0,[GROUND]->0,[*]->*
Analogy_spice_pins[POS]->p,[pos]->p,[NEG]->m, [neg]-
>m
}
enums{
}
saber{
#first generic entry
accuparts_lib::::::::::::::::::::{
#specific symbol entries
voltage_source:v:: <-splist(instpar)::
Analogy_spice_pins[<>]:::::::: inst:::::::;
}

#second generic entry
::::::::::::::::::::{
#specific symbol entries
::::::::::::::::::::;
}
}
undetermined{
#generic entry
::::::::::Analogy_nets[global_net]:::::::::: {
}
}
```

Mentor Graphics: The Schematic for the ASIC Symbol

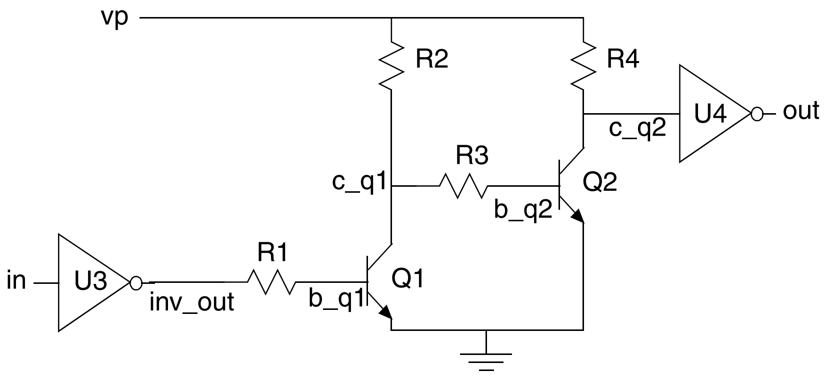
The symbol asic (instance U2), shown below, is a hierarchical user-created symbol. It represents the schematic, also named asic, shown in the next figure. The schematic is created with symbols obtained from the accuparts_lib and gen_lib symbol libraries. Mapping file entries for these symbols are developed for the standard mapping file dvetos.map as shown in the following sections.

The completed entries for each symbol are shown in bold text at the end of each section.



Instance Properties
inst=U2
res=1k
l=10u
w=5u

Pin Properties
PIN=vp
PIN=in
PIN=out



Using Properties to Pass Parameters Through Hierarchy The asic symbol represents the underlying schematic shown above. In addition to the inst property specifying the reference designator, the instance U2 of the asic symbol includes the properties res, l, and w. The res property is used in this case to specify the value (1k) for the resistors designated R1, R2, R3, and R4 on the underlying schematic. Each of the resistors includes the property instpar with the value res. When the netlister encounters this non-numeric instpar property value, it searches up through the hierarchy until it finds a property of the same name (res) and takes the associated numeric value and passes it to the res parameter that it creates for the asic template.

The l and w properties are used to specify the value of the area (length*width) of the transistors designated Q1 and Q2 on the underlying schematic. Each of the transistors includes the property area with the value {l}*{w}. When the netlister encounters this expression, it searches up through the hierarchy until it finds the properties named l and w. It takes the associated numeric values and passes them to l and w in the expression area={l}*{w} through the l and w parameters of the asic template.

For more information on hierarchical parameter passing, refer to the Analyzing Designs manual.

To map the asic symbol, you must add the inst property in the 14th field of the generic entry in the undetermined section of the mapping file as indicated in the following example by bold text:

```
tables{
Analogy_nets[ground]->0,[GROUND]->0,[*]->*
Analogy_spice_pins[POS]->p,[pos]->p,[NEG]->m,[neg]->m
}
enums{
}
saber{
#first generic entry
accuparts_lib::::::::::::::::::::{
#specific symbol entries
 voltage_source:v:: <-splist(instpar)::
Analogy_spice_pins[<>]:::::::: inst:::::::;
}
#second generic entry
::::::::::::::::::::{
#specific symbol entries
::::::::::::::::::::;
}
}
undetermined{
#generic entry
:::::::::Analogy_nets[global_net]:::inst::::::: {
}
}
```

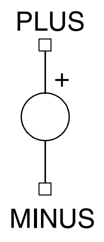
Cadence Symbol Mapping Examples

This section contains three examples (a voltage source symbol, a ground symbol, and an ASIC symbol) for use with the Frameway integration into the Cadence environment.

Cadence: Mapping the Voltage-Source Symbols
Cadence: Mapping the Ground Symbol
Cadence: The Schematic for the ASIC Symbol

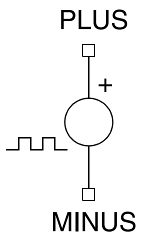
Cadence: Mapping the Voltage-Source Symbols

The analogLib symbol library contains the symbols named vdc and vpulse. These symbols are used for the voltage sources V1 and V2, respectively, in the ring oscillator circuit. These symbols are illustrated in Figure 7-5. The mapping entries for these symbols are created for the standard mapping file catos.map as shown in the following procedure. The completed entries are shown in bold text at the end of the procedure.



Instance Properties
name=V1
vdc=5v

Pin Properties
terminalName=MINUS
terminalName=PLUS



Instance Properties
name=V2
v1=0v
v2=5v
tdelay=2ns
tfall=1ns
trise=5ns
per=20n
pw=10n

Pin Properties
terminalName=MINUS
terminalName=PLUS

To map the voltage source symbols, perform these steps:

1. Specify the name of the library in the first field of the first saber generic entry.

   You specify the name of the symbol library in the first field of a saber generic entry.

   The name analogLib is placed in the first field of the first generic entry in the saber section.

The voltage source symbols are located in the analogLib library, so analogLib was placed in this generic entry. All symbols from this library are mapped using specific symbol entries in this generic entry.

2. Specify the symbol names for V1 and V2.

   You use the first field of the specific symbol entry to specify symbol names.

   The symbol name vdc is placed in the first field of the first specific symbol entry. The symbol name vpulse is placed in the first field of the second specific symbol entry.

3. Specify the simulator template to be used in the netlist.

   You use the second field of the specific symbol entry to specify the name of the simulator template to be used in the simulation. The file that contains the simulator template for a voltage source is v.sin.

   The name of the template (v) is placed in the second field of each of the specific symbol entries (vdc and vpulse).

4. Specify the source of the simulator-model (template) parameters.

   You use the fourth field of the specific symbol entries to specify the source of the properties. If more than one source is listed, the sources must be separated by commas.

   The following expression is placed in the fourth field of the vdc specific symbol entry:

   ```
   dc<-id(vdc),ac<-"(%{acm},%{acp})",ac<-"(%{acm},0)"
   ```

   The mapping function id does not change the value of the property. In the expression dc<-id(vdc), the left arrow means transfer the value of the vdc instance property to the dc template parameter. In the expression ac<-"(%{acm},%{acp})", the left arrow means transfer the value of the acm and acp properties (ac magnitude and phase, respectively) to the ac template parameter. In the expression ac<-"(%{acm},0)", the default value 0 for the acp property is given to the ac template parameter when the acp property is not specified.

The following expression is placed in the fourth field of the vpulse specific symbol entry:

```
tran<-"(pulse=(%{v1},%{v2},%{td},%{tr},%{tf},%{pw},
%{per}))",
tran<-"(pulse=(v1=%{v1},v2=%{v2},tr=%{tr},tf=%{tf},
td=%{td},pw=%{pw}))",
tran<-"(pulse=(v1=%{v1},v2=%{v2},tr=%{tr},tf=%{tf},
td=%{td},per=%{per}))",

tran<-"(pulse=(v1=%{v1},v2=%{v2},tr=%{tr},tf=%{tf},
td=%{td}))",
tran<-"(pulse=(v1=%{v1},v2=%{v2},tr=%{tr},tf=%{tf},
per=%{per},pw=%{pw}))",
tran<-"(pulse=(v1=%{v1},v2=%{v2},tr=%{tr},tf=%{tf},
pw=%{pw}))",
tran<-"(pulse=(v1=%{v1},v2=%{v2},tr=%{tr},tf=%{tf},
per=%{per}))",
ac<-"(%{acm},%{acp})",
ac<-"(%{acm},0)"
```

In this expression, the left arrows point to tran and ac, which means transfer the values of the specified properties to the tran and ac template parameters. The first item in this expression, where all of the %{ } entries have specified values, is used for the mapping.

5. Specify the correspondence between template and symbol connection points.

   The sixth field of the specific symbol entries specifies the source of a pin (port) name. In this example, the correspondence between the symbol pin names plus and minus and the connection point names p and m of the simulator templates is specified as a table entry in the tables section. This table entry has the unique name artist_pin_def, which is referenced by the entry in the sixth field of the specific symbol entry.

   The following expression is placed in the sixth field of each of the specific symbol entries:

   ```
   artist_pin_def[<>]
   ```

The character pair <> indicates that the names to be used to look up the translation in the table entry are the names assigned to the pins (PLUS and MINUS).

The following table entry is placed in the tables section:

```
artist_pin_def[PLUS]->p,[plus]->p,[MINUS]->m,[minus]->m
```

The artist_pin_def table entry translates PLUS to p and MINUS to m.

The entries that are needed to map the vdc and vpulse symbols are added to the catos.map file as indicated in bold text in the following example:

```
tables{
artist_pin_def[PLUS]->p,[plus]->p,[MINUS]->m,
[minus]->m
}
enums{
}
saber{
#first generic entry
analogLib:::::::::::::::::::{
#specific symbol entries
vdc:v:: dc<-id(vdc),ac<-"(%{acm},%{acp})",ac<-
"(%{acm},0)":: artist_pin_def[<>]:::::::::::::::;
vpulse:v::\
tran<-"(pulse=(%{v1},%{v2},%{td},%{tr},%{tf},
  %{pw}, %{per}))",\
tran<-"(pulse=(v1=%{v1},v2=%{v2},tr=%{tr},
  tf=%{tf},td=%{td},pw=%{pw}))",\
tran<-"(pulse=(v1=%{v1},v2=%{v2},tr=%{tr},
  tf=%{tf},td=%{td},per=%{per}))",\
tran<-"(pulse=(v1=%{v1},v2=%{v2},tr=%{tr},
  tf=%{tf}, td=%{td}))",\
tran<-"(pulse=(v1=%{v1},v2=%{v2},tr=%{tr},
  tf=%{tf},per=%{per},pw=%{pw}))",\
tran<-"(pulse=(v1=%{v1},v2=%{v2},tr=%{tr},
  tf=%{tf},pw=%{pw}))",\
tran<-"(pulse=(v1=%{v1},v2=%{v2},tr=%{tr},
  tf=%{tf},per=%{per}))",\
ac<-"(%{acm},%{acp})",\
ac<-"(%{acm},0)"::artist_pin_def[<>]:::::::::::;
}
#second generic entry
::::::::::::::::::::{
#specific symbol entries
::::::::::::::::::::;
}
}
undetermined{
#generic entry
 :::::::::::::::::::: {
}
}
```

Cadence: Mapping the Ground Symbol

The symbol named gnd is a symbol from the analogLib library. This symbol is a device used to name global nets in Analog Artist schematics and has no template associated with it. Net names and the symbol gnd are mapped in the undetermined section, which is used for schematic entities and nets.

The completed entries in the mapping file for the ground symbol are shown in bold text at the end of this section.

You can name a library in the first field of the generic entry. However, the first field of the generic entry of the undetermined section is left empty, which signifies that this generic entry provides the default generic mapping for the undetermined section.

Specify the net name of the ground symbol.

The following expression is placed in the 11th field of the generic entry of the undetermined section to reference an entry in the tables section:

```
artist_default_net_defs[<>]
```

The following expression is placed in the tables section:

```
artist_default_net_defs[gnd]->0,[*]->*
```

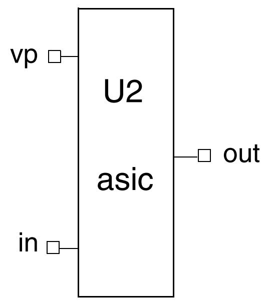
The artist_default_net_defs table entry renames the net gnd to 0 (zero). The expression [*]->* indicates that all other net names remain unchanged.

All entries required to map the gnd symbol are added to the catos.map file as indicated in bold text in the following example:

```
tables{
artist_default_net_defs[gnd]->0,[*]->*
artist_pin_def[PLUS]->p,[plus]->p,[MINUS]->m,[minus]->m
saber_logic_pin_defs[A]->in1,[B]->in2,[C]->in3,[D]->in4, [Y]-
>out
artist_digital_pins[*]->digital
}
enums{
enum {_0,_1,_x,_z} init {nand2_l4}
}
saber{
#first generic entry
analogLib::::::::::::::::::::{
#specific symbol entries
vdc:v:: \
dc<-id(vdc),\
ac<-"(%{acm},%{acp})",\
ac<-"(%{acm},0)":: \
artist_pin_def[<>]:::::::::::::::;
vpulse:v::\
tran<-"(pulse=(%{v1},%{v2},%{td},%{tr},%{tf},\
  %{pw},%{per}))",\
tran<-"(pulse=(v1=%{v1},v2=%{v2},tr=%{tr},tf=%{tf},\
  td=%{td},pw=%{pw}))",\
tran<-"(pulse=(v1=%{v1},v2=%{v2},tr=%{tr},tf=%{tf},\
  td=%{td},per=%{per}))",\
tran<-"(pulse=(v1=%{v1},v2=%{v2},tr=%{tr},tf=%{tf},\
  td=%{td}))",\
tran<-"(pulse=(v1=%{v1},v2=%{v2},tr=%{tr},tf=%{tf},\
  per=%{per},pw=%{pw}))",\
tran<-"(pulse=(v1=%{v1},v2=%{v2},tr=%{tr},tf=%{tf},\
  pw=%{pw}))",\
tran<-"(pulse=(v1=%{v1},v2=%{v2},tr=%{tr},tf=%{tf},\
  per=%{per}))",\
ac<-"(%{acm},%{acp})",\
ac<-"(%{acm},0)"::artist_pin_def[<>]:::::::::::::::;
}
#second generic entry
sample:::::::::::::::::::{
#specific symbol entries
nand2:nand2_l4::tplh, tphl, tilh, tihl,\
init::saber_logic_pin_defs[<>]: \
artist_digital_pins[<>]::::::::::::::;
}
}
undetermined{
#generic entry
::::::::::artist_default_net_defs[<>]:::::::::: {
}
}
```

Cadence: The Schematic for the ASIC Symbol

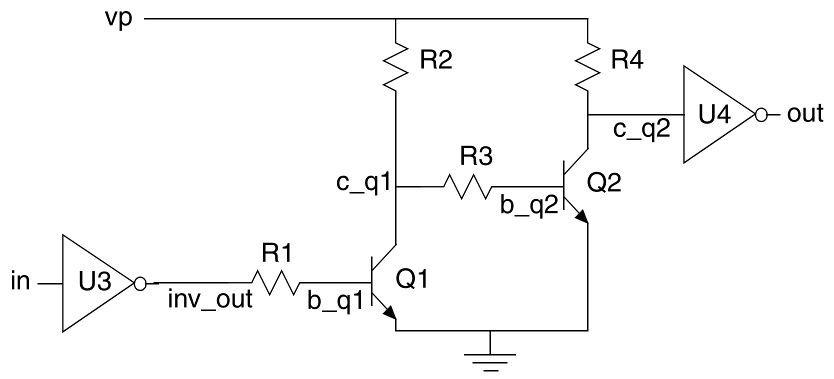
The symbol asic (instance U2) shown below is a hierarchical user-created symbol. It represents the schematic, also named asic, shown in the next figure. The schematic is created with symbols obtained from the analogLib and sample symbol libraries. Mapping file entries for these symbols are developed for the standard mapping file catos.map, as shown in the following sections. The completed entries for each symbol are shown in bold text at the end of each section.



Instance Properties
name=U2
res=1k
l=10u
w=5u

Terminal Properties
direction = input
direction = input
direction = output

Pin Properties
terminalName = vp ;
terminalName = in ;
terminalName = out ;



Using Properties to Pass Parameters Through Hierarchy The asic symbol represents the underlying schematic shown above. The instance U2 of the asic symbol includes the properties res, l, and w. The res property is used in this case to specify the value (1k) for the resistors designated R1, R2, R3, and R4 on the underlying schematic. Each of the resistors includes the property r with the value res. When the netlister encounters this non-numeric r property value, it searches up through the hierarchy until it finds a property of the same name (res) and takes the associated numeric value and passes it to the res parameter that it creates for the asic template.

The l and w properties are used to specify the value of the area (length*width) of the transistors designated Q1 and Q2 on the underlying schematic. Each of the transistors includes the property area with the value {l}*{w}. When the netlister encounters this expression, it searches up through the hierarchy until it finds the properties named l and w. It takes the associated numeric values and passes them to l and w in the expression area={l}*{w} through the l and w parameters of the asic template.

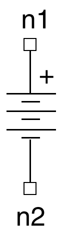
Viewlogic Symbol Mapping Examples

This section contains three examples (a voltage source symbol, a ground symbol, and an ASIC symbol) for use with the Frameway integration into the Viewlogic environment.

Viewlogic: Mapping the Voltage Source Symbols
Viewlogic: Mapping the Ground Symbol
Viewlogic: The Schematic for the ASIC Symbol

Viewlogic: Mapping the Voltage Source Symbols

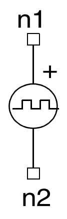
The viewspice (analog) symbol library contains the symbols named dc and pulse. These symbols are used for the voltage sources V1 and V2, respectively, in the ring oscillator circuit. These symbols are illustrated below. The mapping entries for these symbols are created for the standard mapping file vwltosv.map as shown in the following procedure. The completed entries are shown in bold text at the end of the procedure.



Instance Attribute/Label
label=V1
voltage=5v

Pin Labels
n1
n2

Attached Attributes
pintype=in
pintype=out



Instance Attributes/Label
label=V2
vinitial=0v
vpulsed=5v
tdelay=2ns
tfall=1ns
trise=5ns
tperiod=20ns
tpulwidth=10ns

Pin Labels
n1
n2

Attached Attributes
pintype=in
pintype=in

To map the voltage source symbols, perform these steps:

1. Specify the name of the library in the first field of the saber generic entry.

You use the first field of the generic entry to specify the name of the library; however, you can specify any name in this field or leave it empty. This generic entry contains specific symbol entries for both the viewspice (analog) and builtin symbol libraries, and the first field of the generic entry is left empty.

2. Specify the symbol names for V1 and V2.

   You use the first field of the specific symbol entry to specify symbol names.

   The symbol name dc is placed in the first field of the first specific symbol entry. The symbol name pulse is placed in the first field of the second specific symbol entry.

3. Specify the simulator template to be used in the netlist.

   You use the second field of the specific symbol entry to specify the name of the simulator template to be used in the simulation. The file that contains the simulator template for these voltage sources is spv.sin.

   The name of the template (spv) is placed in the second field of each of the specific symbol entries (dc and pulse).

4. Specify the source of the simulator-model (template) parameters.

   You use the fourth field of the specific symbol entries to specify the source of the attributes. If more than one source is listed, the sources must be separated by commas.

   The following expression is placed in the fourth field of the dc specific symbol entry:

   dc<-spconv(voltage)

   The attribute voltage provides the dc voltage value for the dc parameter in the spv.sin template in SPICE format. The mapping function spconv is an algorithm that converts a numeric SPICE parameter to a numeric simulator parameter. The spconv function also relates the attribute voltage to the dc parameter.

The following expression is placed in the fourth field of the pulse specific symbol entry:

```
tran<-"(pulse=(v1=%{spconv(vinitial)},
v2=%{spconv(vpulsed)},tr=%{spconv(trise)},
tf=%{spconv(tfall)},td=%{spconv(tdelay)},
pw=%{spconv(tpulwidth)},per=%{spconv(tperiod)}))",
tran<-"(pulse=(v1=%{spconv(vinitial)},
v2=%{spconv(vpulsed)},tr=%{spconv(trise)},
tf=%{spconv(tfall)},td=%{spconv(tdelay)},
pw=%{spconv(tpulwidth)}))",
tran<-"(pulse=(v1=%{spconv(vinitial)},
v2=%{spconv(vpulsed)},tr=%{spconv(trise)},
tf=%{spconv(tfall)},td=%{spconv(tdelay)},
per=%{spconv(tperiod)}))",
tran<-"(pulse=(v1=%{spconv(vinitial)},
v2=%{spconv(vpulsed)},tr=%{spconv(trise)},
tf=%{spconv(tfall)},td=%{spconv(tdelay)}))",

tran<-"(pulse=(v1=%{spconv(vinitial)},
v2=%{spconv(vpulsed)},tr=%{spconv(trise)},
tf=%{spconv(tfall)},pw=%{spconv(tpulwidth)},
per=%{spconv(tperiod)}))",
tran<-"(pulse=(v1=%{spconv(vinitial)},
v2=%{spconv(vpulsed)},tr=%{spconv(trise)},
tf=%{spconv(tfall)},pw=%{spconv(tpulwidth)}))",
tran<-"(pulse=(v1=%{spconv(vinitial)},
v2=%{spconv(vpulsed)},tr=%{spconv(trise)},
tf=%{spconv(tfall)}, per=%{spconv(tperiod)}))",
tran<-"(pulse=(v1=%{spconv(vinitial)},
v2=%{spconv(vpulsed)},tr=%{spconv(trise)},
tf=%{spconv(tfall)}))"::spr_pin[<>]:::::::::::::::;
```

In this expression, the left arrows point to tran, which means transfer the values of the specified attributes to the tran template parameter. The first item in this expression, where all of the %{ } entries have specified values, is used for the mapping.

5. Specify the correspondence between template and symbol connection points.

The sixth field of the specific symbol entries specifies the source of a pin (port) name. In this example, the correspondence between the symbol pin names n1 and n2 and the connection point names p and m of the simulator templates is specified as a table entry in the tables section. This table entry has the unique name spr_pin, which is referenced by the entry in the sixth field of the specific symbol entry.

The following expression is placed in the sixth field of each of the specific symbol entries:

```
spr_pin[<>]
```

The character pair <> indicates that the names to be used to look up the translation in the table entry are the names assigned to the pins (n1 and n2).

The following table entry is placed in the tables section:

```
spr_pin[n1]->p,[n2]->m,[*]->*
```

The spr_pin table entry translates n1 to p and n2 to m. The [*]->* entry indicates that all other pin names remain unchanged.

The entries that are needed to map the dc and pulse symbols are added to the vwltosv.map file, as indicated in bold text in the following example:

```
tables{
spr_pin[n1]->p,[n2]->m,[*]->*
}
enums{
}
saber{
# generic entry
::::::::::::::::::::{
# specific symbol entries
dc:spv:: dc<-
spconv(voltage)::spr_pin[<>]:::::::::::::::;
pulse:spv::tran<-"(pulse=(v1=%{spconv(vinitial)},
v2=%{spconv(vpulsed)},tr=%{spconv(trise)},
tf=%{spconv(tfall)},td=%{spconv(tdelay)},
pw=%{spconv(tpulwidth)},per=%{spconv(tperiod)}))",
tran<-"(pulse=(v1=%{spconv(vinitial)},
v2=%{spconv(vpulsed)},tr=%{spconv(trise)},
tf=%{spconv(tfall)},td=%{spconv(tdelay)},
pw=%{spconv(tpulwidth)}))",
tran<-"(pulse=(v1=%{spconv(vinitial)},
v2=%{spconv(vpulsed)},tr=%{spconv(trise)},
tf=%{spconv(tfall)},td=%{spconv(tdelay)},
per=%{spconv(tperiod)}))",
tran<-"(pulse=(v1=%{spconv(vinitial)},
v2=%{spconv(vpulsed)},tr=%{spconv(trise)},
tf=%{spconv(tfall)},td=%{spconv(tdelay)}))",
tran<-"(pulse=(v1=%{spconv(vinitial)},
v2=%{spconv(vpulsed)},tr=%{spconv(trise)},
tf=%{spconv(tfall)},pw=%{spconv(tpulwidth)},
per=%{spconv(tperiod)}))",
tran<-"(pulse=(v1=%{spconv(vinitial)},
v2=%{spconv(vpulsed)},tr=%{spconv(trise)},
tf=%{spconv(tfall)},pw=%{spconv(tpulwidth)}))",
tran<-"(pulse=(v1=%{spconv(vinitial)},
v2=%{spconv(vpulsed)},tr=%{spconv(trise)},
tf=%{spconv(tfall)}, per=%{spconv(tperiod)}))",
tran<-"(pulse=(v1=%{spconv(vinitial)},
v2=%{spconv(vpulsed)},tr=%{spconv(trise)},
tf=%{spconv(tfall)}))"::spr_pin[<>]:::::::::::::::;
}}
undetermined{
# generic entry
:::::::::::::::::::: {
}}
```

Viewlogic: Mapping the Ground Symbol

The symbol named gnd is a symbol from the viewspice (analog) library. This symbol is a device used to name global nets in ViewDraw schematics and has no template associated with it. Net names and the symbol gnd are mapped in the undetermined section, which is used for schematic entities and nets. The completed entries in the mapping file for the ground symbol are shown in bold text at the end of this section.

Specify the net name of the ground symbol.

The following expression is placed in the 11th field of the generic entry of the undetermined section to reference an entry in the tables section:

```
nets[<>]
```

The following expression is placed in the tables section:

```
nets[GND]->0,[gnd]->0,[*]->*
```

The nets table entry renames the net GND (or gnd) to 0 (zero). The expression [*]->* indicates that all other net names remain unchanged.

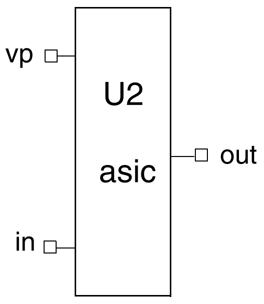
All entries required to map the gnd symbol are added to the vwltosv.map file, as indicated in bold text in the following example:

```
tables{
nets[GND]->0,[gnd]->0,[*]->*
spr_pin[n1]->p,[n2]->m,[*]->*
}
```

```
enums{
}
saber{
# generic entry
:::::::::::::::::::{
# specific symbol entries
dc:spv:: dc<-spconv(voltage)::spr_pin[<>]::::::::::::::;
pulse:spv::tran<-"(pulse=(v1=%{spconv(vinitial)},
v2=%{spconv(vpulsed)},tr=%{spconv(trise)},
tf=%{spconv(tfall)},td=%{spconv(tdelay)},
pw=%{spconv(tpulwidth)},per=%{spconv(tperiod)}))",
tran<-"(pulse=(v1=%{spconv(vinitial)},
v2=%{spconv(vpulsed)},tr=%{spconv(trise)},
tf=%{spconv(tfall)},td=%{spconv(tdelay)},
pw=%{spconv(tpulwidth)}))",
tran<-"(pulse=(v1=%{spconv(vinitial)},
v2=%{spconv(vpulsed)},tr=%{spconv(trise)},
tf=%{spconv(tfall)},td=%{spconv(tdelay)},
per=%{spconv(tperiod)}))",
tran<-"(pulse=(v1=%{spconv(vinitial)},
v2=%{spconv(vpulsed)},tr=%{spconv(trise)},
tf=%{spconv(tfall)},td=%{spconv(tdelay)}))",
tran<-"(pulse=(v1=%{spconv(vinitial)},
v2=%{spconv(vpulsed)},tr=%{spconv(trise)},
tf=%{spconv(tfall)},pw=%{spconv(tpulwidth)},
per=%{spconv(tperiod)}))",
tran<-"(pulse=(v1=%{spconv(vinitial)},
v2=%{spconv(vpulsed)},tr=%{spconv(trise)},
tf=%{spconv(tfall)},pw=%{spconv(tpulwidth)}))",
tran<-"(pulse=(v1=%{spconv(vinitial)},
v2=%{spconv(vpulsed)},tr=%{spconv(trise)},
tf=%{spconv(tfall)}, per=%{spconv(tperiod)}))",
tran<-"(pulse=(v1=%{spconv(vinitial)},
v2=%{spconv(vpulsed)},tr=%{spconv(trise)},
tf=%{spconv(tfall)}))"::spr_pin[<>]:::::::::::::::;
}}
undetermined{
# generic entry
::::::::::nets[<>]:::::::::: {
}}
```

Viewlogic: The Schematic for the ASIC Symbol

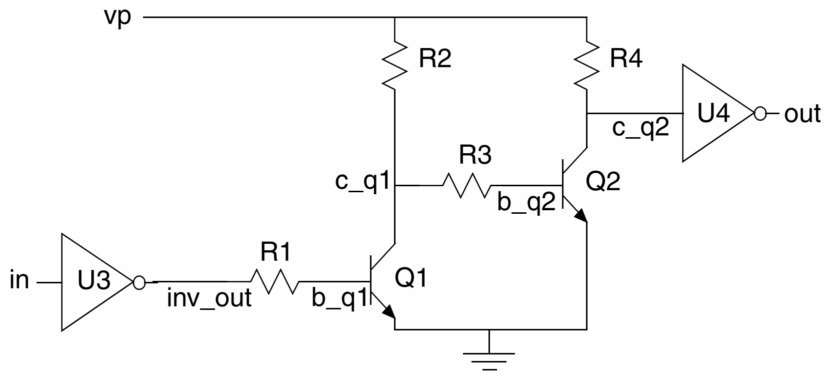
The symbol asic (instance U2) shown below is a hierarchical user-created symbol. It represents the schematic, also named asic, shown in the next figure. The schematic is created with symbols obtained from the viewspice (analog) and builtin symbol libraries. Mapping file entries for these symbols were developed for the standard mapping file vwltosv.map as shown in the following sections. The completed entries for each symbol are shown in bold text at the end of each section.



Instance Attribute/Label
label=U2
res=1k
l=10m
w=5m

Attached Attributes
pintype = in
pintype = in
pintype = out

Pin Labels
vp ;
in ;
out ;



Using Attributes to Pass Parameters Through Hierarchy The asic symbol represents the underlying schematic shown above. In addition to the label specifying the reference designator, the instance U2 of the asic symbol includes the attributes res, l, and w. The res attribute is used in this case to specify the value (1k) for the resistors designated R1, R2, R3, and R4 on the underlying schematic. Each of the resistors includes the attribute value with the value res. When the netlister encounters this non-numeric value attribute value, it searches up through the hierarchy until it finds an attribute of the same name (res) and takes the associated numeric value and passes it to the res parameter that it creates for the asic template.

The l and w attributes are used to specify the value of the area (length*width) of the transistors designated Q1 and Q2 on the underlying schematic. Each of the transistors includes the attribute area with the value {l}*{w}. When the netlister encounters this expression, it searches up through the hierarchy until it finds the attributes named l and w. It takes the associated numeric values and passes them to l and w in the expression area={l}*{w} through the l and w parameters of the asic template.

12

Mapping File Reference

A mapping file is a text file containing information used by the netlister to convert other symbols (Cadence, Viewlogic, Mentor Graphics) into netlist entries recognized by the Saber Simulator or a specific partner simulator. By using entries in a mapping file, you can associate a symbol with a differently named simulation model or convert the syntax of a symbol parameter to that required by the simulator.

Topics discussed in this section include:

Standard Mapping Files

You can create a mapping file yourself, or you can use a standard mapping file. The standard mapping files support only the standard symbol libraries. Standard mapping files for each of the Frameway integrations are provided; they are located in the $SABER_HOME/bin directory where the netlisters can find them automatically.

Mentor Graphics

The Frameway integration for Mentor Graphics provides the following standard mapping files:

- dvetos.map for use with the DVETOS netlister (Mentor symbols to simulator)
- dvetosv.map for use with the DVETOSV netlister (Mentor symbols to Saber/Verilog simulator)
- dvetos.map for use with the SNET netlister (Mentor symbols to the Saber Simulator)
- analogym.map for use with the Mentor Graphics netlisters

The dvetos.map, and dvetosv.map files support the Mentor Graphics accuparts_lib and gen_lib symbol libraries. The analogym.map file supports the supplied symbol libraries for use with the Mentor Graphics netlisters.

Cadence

The Frameway integration for Cadence provides the following standard mapping files:

- catos.map for use with the CATOS netlister (Cadence symbols to simulator)
- analogyc.map for use with the Cadence netlister

The catos.map file supports the Cadence analogLib, sample, and basic symbol libraries. The analogyc.map file supports the supplied symbol libraries for use with the Cadence netlister.

Viewlogic

The Frameway integration for Viewlogic provides the following standard mapping files:

- vwltos.map for use with the VWLTOSV netlister (Viewlogic symbols to simulator)
- vwltosv.map for use with the VWLTOSV netlister (Viewlogic symbols to Saber/Verilog simulator)
- analogyv.map for use with the Viewlogic netlister

The vwltos.map and vwltosv.map files support the Viewlogic viewspice (analog) and builtin symbol libraries. The analogyv.map file supports the supplied symbol libraries for use with the Viewlogic netlister.

User Mapping Files

In addition to the standard mapping file, you may also want to include a user mapping file in your system. The user mapping file allows you to map new symbols, or supplement or replace entries in the standard mapping file(s). If both the standard and the user mapping files are present, entries in the user mapping file take precedence over the entries in the standard mapping file(s).

You can specify a user mapping file by using the -m option of the netlister command on the command line or by using the appropriate simulator start-up form in your Frameway Integration. The mapping file must be in a directory in the data search path and have a filename with a .map extension. To assure that entries in your user mapping file override those in a standard mapping file, place the user mapping file in your local directory.

Structure of a Mapping File

The mapping file may contain any of the following optional sections:

- A tables section
- An enums section
- One or more definition sections: saber, partner, and undetermined
- An exclude section

Each section contains specific information about symbol properties that the netlister uses to translate schematic symbols into a netlist. Any of these sections may be left out if not needed.

A mapping file typically contains several sections that are set-off by a keyword followed by braces {} enclosing the section. The following example illustrates this syntax.

```
tables{
table entries
}
enums{
enum entries
}
saber{
first generic entry{
  first set of specific symbol entries
  }


second generic entry{
  second set of specific symbol entries
  }
}
partner{
generic entry{
  specific symbol entries
  }
}
undetermined{
generic entry{
  specific symbol entries
  }
}
include{
include entries
}
exclude{
exclude entries
}
```

The tables section contains tables that can be referenced from any of the three definition sections. You use such tables to translate property values to new values. As with all sections, the tables section and its entries are optional. However, the tables section must appear first if it is used.

The enums section is reserved to specify properties corresponding to template parameters that accept enumerated values. Mapping files used for templates in the Template Database no longer require this field. However, templates not in the Template Database require mapping files with this field properly configured.

You use the definition sections saber, partner, and undetermined to define the correspondence between schematic entities and simulator entities (symbols and models). If a definition section (saber, partner, or undetermined) is present, the two nested pairs of braces setting off the generic and specific symbol entries must be present, and a generic entry must be included. For examples, refer to the standard mapping files.

The include section is no longer used by the netlister and may be left blank. You can the exclude section to list symbols that are to be ignored by the netlister when it creates the netlist.

Each of these mapping file sections is discussed in detail in the following topics:

Special Characters Used in the Mapping File
Tables Section
Enums Section
Definitions Section
Include and Exclude Sections

Special Characters Used in the Mapping File

The mapping file entries make use of special characters to control the translation process.

%{}	To specify the substitution of a property value, you can enclose a property name between the special character pair %{ and the character }. The special character pair %{ indicates the beginning of the name of the enclosed property while the special character } indicates the end of the name of the enclosed property.
	
	Example: %{freq}
	
	You can also enclose a property mapped through a table.
	
	Example: %{decimal_of[percent]}
	
	Finally, you can enclose a property mapped through a mapping function.
	
	Example: %{spconv(res)}

" "	In certain cases, you can enclose a string between the special character pair " ". The quotes indicate that the enclosed string is to be interpreted literally, except for places where property values are to be substituted. If, for example, in the expression "gain=%{gain}", the value of the property gain enclosed in braces is equal to 5k, the netlister enters gain=5k into the netlist. In other words, %{gain} is replaced by the value of the property gain while the expression gain= is taken literally. Note that if the value of gain is " " or NULL, this mapping will fail.
<-	In certain cases, you can place a string within quotes on the right side of a left arrow <-. This configuration indicates that the value of the string will be assigned to the parameter name on the left side of the arrow. Example: gain<-"(h=1.41)" The result of this example appears in the netlist as gain=(h=1.41). Or, if there is nothing to the left of the arrow, the netlister places the value directly into the netlist without an assignment. Example: <-"rnom=1k,tnom=27" The result of this example appears in the netlist as rnom=1k, tnom=27.

<> The character pair <> indicates that the value to be substituted is the name assigned to the appropriate data base entity, such as a pin, rather than to the value of a property.

Example: some_pin_dirs[<>]

In this example, some_pin_dirs[<>] is specified in the tables section as

some_pin_dirs[dig_in]->in,[dig_out]->out,[*]->null

Tables Section

The tables section begins with the name tables followed by a list enclosed in braces {} that may be empty or may contain one or more tables. These tables can be referenced from any of the three definition sections (saber, partner, and undetermined). The following is an example of a table entry:

```
edit[*req*]->null, [*opt*]->null, [*]->*
```

Table entries and references to table entries are case sensitive. Legal entries between the [ ] are strings of length 0 or more that do not contain the characters '[', ']', ' ', '<NEWLINE>', and '<TAB>'. Legal entries after the -> are strings of length 0 or more that do not contain the characters ',', '{', '}', ' ', '<NEWLINE>', and '<TAB>'.

The above table, called edit, sets to null (or non-existence) a value equal to either of the strings *req* or *opt*. If the value is not equal to either of these strings, its value is unchanged ([*]->*).

The following table called adc_ports assigns port types to the ports of an analog-to-digital converter symbol:

```
adc_ports [in]->analog,[gate]->analog,\
  [gnd]->analog,[*]->digital
```

In this case, the in, gate, and gnd ports are identified as analog ports, and all other ports on the symbol are identified as digital ports.

You can extend a table entry to more than one line.

An entry in a tables section can map a zero-length string to a value. For example, you could use the following entry in the tables section of a mapping file:

add_value []->1k, [*]->*

This entry maps all zero-length strings to the value 1k. All other values remain unchanged.

Enums Section

Templates in the Template Database do not require this field in their mapping files.

You use the enums section to specify properties corresponding to template parameters that accept enumerated values. Enumerated values are syntactically equivalent to parameters in the MAST language. Therefore, special techniques must be adopted for the netlisters to recognize when an identifier, which is used for an enumerated value, is a literal enumerated value and when it is a parameter that was passed through hierarchy. For this purpose, the mapping file enum.map is available for the netlisters to automatically search for and use. This mapping file specifies for the netlister all the enumerated values in templates provided. With this mapping file in place, all enumerated values for these templates should be handled correctly in hierarchy.

The enums section can contain one or more enum declarations. Each enum declaration is given in MAST modeling language syntax and followed by a list of the templates that use that enum parameter. The series of values that the enum parameter can take and the list of templates using the enum parameter are both enclosed in braces.

```
enums {
enum {identifier1, identifier2, ...} \
enum_name {template1 , template2 , ...}
}
```

In this example, enum_name is the name of the enumerated parameter, identifier1 and identifier2 are values of the enumerated parameter, and template1 and template2 are templates that use this enumerated parameter.

If a template corresponding to a symbol placed on a schematic is included in the list of templates in an enum declaration and if a value is given to the enumerated parameter enum_name that is included in the list of values included in the enum declaration, that value will be passed directly to the instance of that template.

```
enums {
enum {_0,_1,_x,_z} init {nand2_l4}
}
```

For example, the symbol for a digital gate may have a property called init. If this property takes enumerated type values _0, _1, _x, and _z, then the definition of the property init in the enums section of the mapping file will specify the valid enumerated type values _0, _1, _x, and _z for the template parameter init. The entry for a nand2 gate will appear as follows:

The nand2_l4 entry specifies the simulator template that represents the nand2 symbol.

If a value string_xyz has been given to the enumerated parameter enum_name that does not exist in the list of values in the enum declaration, the schematic hierarchy is searched for a property named string_xyz. When it is found, its value is passed down through the hierarchy to the template containing the instance of that template.

Definitions Section

Definition sections include the saber, partner, and undetermined sections. You use these sections to map symbol information to the corresponding simulation model information. These sections can be arranged in any order or omitted if not needed. The saber definition section provides mapping for simulation models that are to be simulated by the simulator. The partner definition section provides mapping for predefined simulation models that are to be simulated by a partner simulator. You map symbols representing non-primitive cells in the undetermined definition section.

Each of the definition sections can contain one or more generic entries. A generic entry can contain one or more specific symbol entries. You use a generic entry to specify mapping information that pertains to an entire symbol library. You use a specific symbol entry to map features specific to one symbol. If both a generic and a specific symbol entry in a mapping file exist for the same feature, the specific symbol entry overrides the generic entry for that symbol.

The main feature of a definition section in a mapping file is the use of fields. A single entry line in the generic and specific symbol entries of a definition section consists of 21 fields separated by colons as follows:

```
(1):(2):(3):(4):(5):(6):(7):(8):(9):(10):(11):(12):
(13):(14):(15):(16):(17):(18):(19):(20):(21)
```

Each field in the list corresponds to an item that can be mapped. If you do not use a field, leave it blank. If an item is to be mapped, its position in the list must contain the mapping information for that item. For example, the second field in a generic entry is the simulation model name. If you want to specify where simulation model names are to come from when symbols are placed in a netlist, you must enter the source of the simulation model name in this field of the generic entry. This source might, for example, be a symbol property that is common to all symbols, and contains a symbol name. Each field of the generic and specific symbol entries ar described in more detail in Generic Entries and Specific Symbol Entries.

A source may be a property name, or it may be derived from a function, table, or other source or from a combination of sources. The possibilities are listed below:

```
property_name
table_name[property_name]
function_name(property_name)
function_name(table_name[property_name])
new_property_name<- function_name(property_name)
new_property_name<-
function_name(table_name[property_name])
new_property_name<- constructor
<- function_name(property_name)
<- function_name(table_name[property_name])
<- constructor
```

A property_name is the name of a symbol property in your Frameway Integration’s schematic capture tool (Analog Artist, ViewDraw, Design Architect). For example: nettype.

A table_name is the name of a table in the tables section of the mapping file. In the following example, the table called target maps values of a property called nettype. For example: target[nettype] .

In some cases, the symbol <> can be used in place of a property name. You can use this symbol to map port (or pin) names, port (or pin) types, port (or pin) directions, or net names. (See generic and specific symbol entry fields 6, 7, 8, and 11 in the sections on Generic Entries and specific Symbol Entries.) For example, if the source shown below appears in field 6 of a generic or specific symbol entry in a mapping file, the <> symbol indicates that the names to be used are the actual names assigned to the ports rather than names given as values of a property: target[<>] .

A function_name is the name of a function used to modify a property value of a source.

The following sections provide detailed instructions for the definitions section of a mapping file.

General Mapping Functions
SPICE to MAST Mapping Functions
Length and Width Mapping Functions
Generic Entries
Specific Symbol Entries
Multiple Generic Entries
Default Generic Mapping

General Mapping Functions

The following list describes the available mapping functions:

spconv	Converts a numeric SPICE parameter to a numeric simulator parameter. For example: spconv(td) In this example, if the value contained in the property td is 2.5ns, the netlister converts it to 2.5n.

id	Returns the value of a symbol property as is. This function is used with the <- character to copy the value of a property in the schematic to a parameter in the netlist. For example: dc<-id(vdc) In this example, the value of the symbol property called vdc is assigned to the simulator parameter named dc.
splist	Converts a list of SPICE parameters to the corresponding list of simulator parameters. For example: splist(spicepar) In this example, if the value of the property spicepar issin(0 220 60), the result of the function that appears in the simulator netlist is tran=(sin(0,220,60)).
string	Checks to see if a property value is enclosed in quotation marks(" "). If the property value is enclosed in quotation marks, the netlister places it in the netlist as is (including the quotation marks). If the property value is not enclosed in quotation marks, the netlister adds quotation marks when it includes the property value in the netlist. Note that the netlister automatically makes this mapping in releases including and subsequent to 5.0.
M	Used to extract the correct multiplier M versus m. Since the simulator converts all characters to lower case characters, it interprets the multiplier M of a value, which represents “mega,” as the multiplier m, which the simulator interprets as the multiplier “milli.” You can use the mapping function M to extract this multiplier correctly. All other multiplier values passed through this mapping function remain unchanged. For example, you could use the mapping function M in the fourth field of a mapping file entry as follows: rnom<-M(r) The character r in this example is the property of an instance in a schematic. If this property has the value 1.1M, the netlister places the entry rnom=1.1Meg in the parameter list of the instance of the template.

template	Allows the construction of legal template names in a constructor in the second field of a specific symbol mapping entry for a symbol. This function takes two arguments, a prefix and a component name. If the component name begins with a numeral, the prefix is placed in front of the component name. The arguments to the function can be a string between double quotation marks, a property name, or a mapping of a property through a table.

For example, the following could be an entry in a mapping file:

```
power_mos:"%{template("q",comp)}":::::::::::::::::;
```

The template q2n6755 is used when the comp property has the value 2n6755. On the other hand, when the comp property has the value irf130, the template irf130 is used.

Each invalid character in a component name is translated into an underscore character, unless it appears between a digit and a letter. In that case, it is deleted. For example, LT1004-2.5 is changed to LT1004_2_5, and HI-201HS is changed to HI201HS.

SPICE to MAST Mapping Functions

For an example of how you can use these SPICE to MAST mapping functions, refer to Chapter 10: Using a Mapping File to Convert SPICE Symbols.

spsource	Translates SPICE source parameters to MAST.
spvalues	Translates a general SPICE parameter list to MAST.
spresvals	Translates a SPICE resistor parameter list to MAST.
spstorage	Translates SPICE capacitor or inductor parameters to MAST.
sptran	Translates a SPICE transistor or diode parameter list to MAST.
spopttran	Translates the parameter list for a transistor or diode that does not have the model name in it.
sptranmod	Extracts the model name from a parameter list for a diode or a transistor.

spifpoly	If the SPICE parameter list for a capacitor or an inductor has poly parameters in it, it returns the second argument (typically the name of the template that implements the poly parameters for the device); otherwise, it returns the second argument (typically the name of the template that does not implement the poly parameters).

Length and Width Mapping Functions

You can use the following mapping functions listed below to map length and width specifications typically specified for MOSFETS and MOS gates, into the correct simulator format. For example, you may typically specify the length and width in the form w/l (in microns) as:

```
20/5
```

which has to be mapped to

```
w=20u,l=5u
```

Similarly, you may typically specify the length and width in the form pw/pl - nw/nl, where pw, pl, nw, and nl specify the width and length of the p-channel and n-channel transistors, respectively. For example (again, in microns):

```
20/5 - 10/5
```

which has to be mapped to

```
pw=20u,pl=5u,nw=10u,nl=5u
```

The following descriptions show how the mapping in the previous examples is derived. The first two functions (mwid and mlen) combine to produce the result of the first example (w=20u,l=5u). The remaining four functions combine to produce the result of the second example (pw=20u,pl=5u,nw=10u,nl=5u). Each of these mapping functions is of the form:

```
function_name (string, suffix).
```

mwid — Converts string of the form w/l to w and appends suffix.

For example:

w<-mwid ("20/5","u")

results in the following portion of the netlist entry:

w=20u

mlen — Converts string of the form w/l to l and appends suffix.

	For example:
	l<-mlen ("20/5","u")
	results in the following portion of the netlist entry:
	l=5u

gpwid	Converts string of the form pw/pl - nw/nl to pw and appends suffix. For example: pw<-gpwid ("20/5-10/5","u") results in the following portion of the netlist entry: pw=20u
gplen	Converts string of the form pw/pl - nw/nl to pl and appends suffix. For example: pl<-gplen ("20/5-10/5","u") results in the following portion of the netlist entry: pl=5u
gnwid	Converts string of the form pw/pl - nw/nl to nw and appends suffix. For example: nw<-gnwid ("20/5-10/5","u") results in the following portion of the netlist entry: nw=10u
gnlen	Converts string of the form pw/pl - nw/nl to nl and appends suffix. For example: nl<-gnlen ("20/5-10/5","u") results in the following portion of the netlist entry: nl=5u

The previous examples described the purpose of each of these mapping functions. There are also three basic ways you can use a mapping function as outlined in the following list. In each case, the value of the spicepar property is sin(0 220 60):

1. splist(spicepar) in the mapping file results in the following netlist entry:

   ```
   spicepar=tran=(sin(0,220,60))
   ```

   because without an assignment, the property being translated (spicepar) is used as the parameter in the netlist.

2. tran<-splist(spicepar) in the mapping file results in the following netlist entry:

   ```
   tran=tran=(sin(0,220,60))
   ```

3. <-splist(spicepar) in the mapping file results in the following netlist entry:

   ```
   tran=(sin(0,220,60))
   ```

A new_property_name is the template parameter name that corresponds to a symbol property. In list item 2, the tran parameter corresponds to the property spicepar. You can specify a new_property_name with any of the mapping functions described above or with the constructor described below.

If the source defined to the right of the arrow (<-) contains a template or module parameter or if it contains a list of parameters in the same order in which they are required by the template or module, the new_property_name is optional. When you omit the new_property_name, the netlister places the parameters directly into the appropriate netlist entry, as shown in list item 3 above.

You can use a constructor to map simulation model names, simulation model parameters, simulation model pin parameters, simulation model net parameters, and simulation model nodes defined in properties into a specific format required by a simulator. These various simulation model parameters defined in properties are mapped in the generic and specific symbol entry fields 2, 4, 10, 12, and 15, respectively. These items are described in Generic Entries and Specific Symbol Entries.

For example, a set of timing parameters for a symbol may need to appear in the following form in a netlist:

```
timing = <10NS>, <10NS>
```

The two parameter values for the timing parameter may be defined in two separate properties in the schematic capture system. For example, they might be defined in two properties called rise and fall with values of 10N. They can be combined into the correct format by using a constructor in the mapping file as follows:

```
timing <- "<%{rise}S>, <%{fall}S>"
```

In this example, timing is a new_property_name. The two properties (sources), rise and fall, are each enclosed in braces ({}) and preceded by a percent (%) sign. The strings "<", "S>, <" and "S>" are added in the appropriate places, and the entire sequence is enclosed in quotes. Any sequence of sources (each enclosed in braces and preceded by a % sign) and strings can be used in a constructor. The entire sequence must be enclosed in quotes.

A source used in a constructor can be any of the following:

```
property_name
table_name[property_name]
function_name(property_name)
function_name(table_name[property_name])
```

Generic Entries

Each of the definition sections can contain one or more generic entries. A generic entry can contain one or more specific symbol entries. You use a generic entry to specify mapping information that pertains to an entire symbol library. You use a specific symbol entry to map features specific to a symbol. If both a generic and a specific symbol entry in a mapping file exist for the same feature, the specific symbol entry overrides the generic entry for that symbol.

For example, in the saber section, you can specify a generic entry for symbol library A that provides mapping common to all symbols in that library. Following this generic entry, you can include specific symbol entries to provide mapping specific to each symbol in library A. Then, you can specify a second generic entry for symbol library B that provides mapping common to all symbols in that library. Following this generic entry, you can include specific symbol entries to provide mapping specific to each symbol in library B.

The three optional definition sections (saber, partner, and undetermined) all begin with the definition section name followed by a section enclosed in braces ({}) containing generic and specific symbol entries. Within a definition section, a generic entry is listed first followed by specific symbol entries enclosed in braces. When a definition section is included in a mapping file, the definition section name, at least one generic entry, and the nested pair of braces that set off the generic and specific symbol entries for that definition section must be included. Specific symbol entries are optional.

:

A generic entry in a definition section is made up of twenty-one different fields terminated by colons, as shown below.

```
(1):(2):(3):(4):(5):(6):(7):(8):(9):(10):(11):(12):
(13):(14):(15):(16):(17):(18):(19):(20):(21)
```

Each of the fields is optional. However, when you leave a field blank, its terminating colon must still be included. The following outlines what type of information is contained in each field in a generic entry:

```
symbol library name : simulation model names : simulator
target : simulator model parameters : parameter-list
property : port names : port types : port direction : port
order : port parameters : net names : net parameters :
template directory : reference designator or instance name
: nodes on properties : primitives : graphical modeling :
exclude as parameters: : :
```

A generic entry is terminated by the left brace indicating the beginning of the specific symbol entries. Each specific symbol entry is terminated by a semicolon.

You can specify most values for most fields as a comma-separated list of sources. If a field requires a single source and if a list of sources is given, the netlister checks each source in the list by starting at the left until it finds the information it needs. A few fields can be specified only by using a property name. When this restriction applies, it is noted in the description of that field.

Each field, in the following list, includes examples. These examples may be from all three Frameway integrations or from just one or two. However, unless otherwise stated, each field description applies to all Frameway integrations (all mapping files). The following list describes the fields that you can include in a generic entry:

1. The name of the symbol library.

   A symbol's library name must match the entry in this field. If you want to map several symbols having the same name but located in different libraries, leave this field blank.

   Each netlister has special requirements for the use of this field:

   - Sketch

Because symbols do not have libraries in Sketch, this field should always be empty.

- Cadence

  The library name in the mapping file is the same as the library containing the symbol. For example, all supplied symbols are in SaberLib.

  ```
  SaberLib::::::::::::::::
  ```

- Mentor Graphics

  The library name is the name of the leaf directory containing the symbol. For example, the leaf directory electric has the syntax

  ```
  electric::::::::::::::::
  ```

- Viewlogic

  Because the library name in Viewlogic is a part of the symbol name, for example SBR_ELECTRIC:r, the library name should always be empty.

2. List of possible sources of the simulation model names (template, model, module, primitive, etc.). When the generic entry is in the saber section, the netlister searches along the data search path for the templates specified by this field. The netlister uses the value of the first mapping source it finds as the name of a template.

   - Cadence Example

```
:palComponentName,sabermodel::::::::::::::::::
```

In this example, the netlister first searches for the palComponentName property and uses its value as the name of the simulation model for the corresponding symbol. If it does not find the palComponentName property, it then searches for the sabermodel property and uses its value as the name of the simulation model. If this entry is in the saber section, the netlister uses the first property value in this list that matches the name of a template. That is, if the palComponentName property value is not the same as any simulator templates, the netlister will then search for the sabermodel property to find a value that matches a simulator template. You can use a constructor in the definition of a source for this field.

3. The target simulator for a primitive symbol. This entry determines whether a primitive symbol will appear in a simulator netlist or a partner simulator netlist. If you specify this field in any one of the definition sections, you must specify it in all three (saber, partner, and undetermined).

   - Viewlogic Example

```
::cv_target[nettype]:::::::::::::::::
```

In this example, the netlister uses the attribute nettype to determine the target simulator from a table called cv_target. The entry in the tables section is as follows:

```
cv_target[analog]->saber,[*]->partner
```

If the attribute nettype contains the value analog, the target simulator for the symbol will be the simulator. Otherwise, the target simulator will be the partner simulator.

You can also use the Target_Simulator attribute (property) to designate the target simulator for a primitive cell. If you use the Target_Simulator attribute (property), a mapping file entry is not necessary. For information regarding specially-recognized properties to avoid mapping, refer to Chapter 6: Using Specially-Recognized Properties for Mapping.

4. The source of the properties of the symbol to be retained and passed on to the simulator.

   - Example

```
:::delay,drive:::::::::::::::
```

In this example from a partner definition section, the properties delay and drive are extracted as characteristics for a part in the partner simulator netlist. You can use a constructor in the definition of a source for this field. .

5. A list of properties containing composite parameter lists of primitive symbols for the simulator. The composite parameter list is a list of property names, separated by commas, to be passed through as parameters to the primitive symbols.

- Example:

```
::::parameters:::::::::::::::
```

In this example, if any symbol has the property named parameters attached, which can be a comma-separated list of parameters, the netlister uses its value to determine which parameters to extract for each symbol that has this property attached. You can use this method to specify which parameters to extract rather than listing the parameters directly in field 4.

The use of this field for the undetermined section and hierarchical parameter passing was made obsolete by the superior hierarchical parameter passing techniques provided in Release 3.1-1.4 and later of the Integration Toolkit from which the netlisters were constructed.

6. The source of a port (or pin) name.

   - Cadence Example:

```
::::artist_pin_def[<>]:::::::::::::::
```

In this example, the assigned port names (indicated by [<>]) are mapped to new names using a table called artist_pin_def. The entry in the tables section is shown below:

```
artist_pin_def[PLUS]->p,[MINUS]->m,[P+]->pp,
[P]->pm,[S+]->sp,[S-]->sm,[G]->g,[D]->d,[S]->s,
[B]->b,[NC-]->vm,[NC+]->vp,[IN+]->p1,[OUT+]->p2,
[IN-]->m1,[OUT-]->m2,[*]->*
```

In this example, ports named PLUS are renamed p, ports named MINUS are renamed m, and so forth. The item [*]->* at the end of the table entry maps all other port names to their current names.

You can use the <> symbol in this field to indicate that the port names to be used are the actual names assigned to the ports rather than given as values of a property. For more information about this symbol, see Special Characters Used in the Mapping File.

7. The source of the port (or pin) type of the symbol. The named source contains information about whether the ports of the symbol are analog or digital ports.

- Example 1:

```
:::::digital_ports[<>]::::::::::::::
```

In this example, the ports on all symbols in the library are digital when used in conjunction with the table digital_ports defined below:

```
digital_ports[*]->digital
```

This table indicates all ports are digital. Since all ports for partner simulators default to digital, this mapping would not be useful in the partner section. However, since all ports for the simulator default to analog, you can use this mapping in the saber section to specify that all ports in a digital library for the simulator are digital.

You can also use this field to specify the type of pin that a graphical model will have (electrical, mechanical, control, etc.).

- Example 2:

```
:::::pin_type::::::::::::::
```

The pin_type property on the pins of the graphical modeling symbols is used for this purpose. The following are some examples of values that you can specify for the pin_type property:

input nu	control system input connection point, no units (nu)
output nu	control system output connection point, no units (nu)
electrical	electrical connection point
state nu	event-driven analog connection point, no units (nu)

8. Used for the following three purposes:
   a. Provides the pin direction if not specified on the schematic:

The named source (property) contains information required by the partner simulator defining the direction of the port (or pin). In this example, the source of the port (or pin) direction is the value of the property pintype.

- Example

```
::::::pintype::::::::::::
```

b. Designates special pins such as power pins and how they are to be used in Hypermodel interface insertion:

   The power and ground connection points of the Hypermodel interfaces inserted at other pins of the symbol will be connected to the nets to which these pins are connected. The pin directions to designate power and ground pins corresponding to template or module connection points are pow, gnd, unpow, and ungnd (the netlister does not insert Hypermodel interfaces for pins designated as power or ground pins). Valid values for Hypermodel interface ports are in (input), out (output), bi (bidirectional), pow (power), gnd (ground), unpow (power pin to be excluded from netlist), ungnd (ground pin to be excluded from netlist), and ignore (pin to be excluded from the netlist).

   You can also use an entry in the pins section of a Hypermodel interface (.shm) file to designate power and ground pins.

c. Causes the netlister to ignore pins when they are not part of the simulation model

   This designation is done by using a table to assign the pin-direction ignore, unpow, or ungnd to pins that are to be excluded from the netlist. As you can see from the previous discussion, the unpow and ungnd direction designations serve a dual purpose. You can use this designation when pins occur on a symbol for which there are no connection points on a corresponding simulator template or Verilog module. When the pin direction of a pin is set to ignore, the netlister does not insert a Hypermodel interface at that pin, and the pin does not appear in the netlist.

9. The name of the property that contains the port (or pin) order. The value of the named property is a list of pin names, separated by commas, in the order required by the simulator. The source for this entry must be a property name. It cannot be derived indirectly, by using a table entry.

Example:

```
:::::::pinorder:::::::::::
```

In this example, the pinorder property value is a list specifying the port (or pin) order. For example, the value of this property may appear as follows:
Y,A,B

10. The sources of port properties to be retained and passed on to the simulator.
    - Example:

      None (not used by the simulator or Verilog simulator).
11. The source of the names of nets. This field is used only in the undetermined section. Typically, this field is mapped by using a “default” generic entry since the global net names changed are not normally specific to a library.
    - Mentor Graphics Example:

```
:::::::::Analogy_nets[global_net]::::::::::
```

In this example, the assigned net names are mapped to new names using a table called Analogy_nets. The entry in the tables section is as follows:

```
Analogy_nets[GND]->0,[gnd]->0,[ground]->0, [GROUND]-
>0,[*]->*
```

If a net is named GND, gnd, ground, or GROUND, the netlister changes its name to 0. Otherwise, the net name remains the same.

- Cadence Example:

```
:::::::::artist_net_defs[<>]::::::::::
```

In this example, the assigned net names (indicated by [<>]) are mapped to new names using a table called artist_net_defs. The entry in the tables section is as follows:

```
artist_net_defs[gnd!]->0,[*]->*
```

If a net is named gnd!, the netlister changes its name to 0. Otherwise, the net name remains the same.

- Viewlogic Example:

```
:::::::::nets[<>]::::::::::
```

In this example, the assigned net names (indicated by [<>]) are mapped to new names using a table called nets. The entry in the tables section is as follows:

```
nets[GND]->0,[gnd]->0,[*]->*
```

If a net is named GND or gnd, the netlister changes its name to 0. Otherwise, the net name remains the same.

12. The source of net properties to be retained and passed on to the simulator. This field is used only in the undetermined section.

    Example:

```
::::::::::type,charge,delay,drive::::::::
```

In this example, the net properties type, charge, delay, and drive are to be passed on to the simulator. Net properties exist only on non-primitive symbols. The simulator does not use net properties. You can use a constructor in the definition of a source for this field.

13. The path to the directory containing the template to be used.

    Example: None.

14. The source of the instance name. This can be a single property name or a constructor containing strings or properties to specify the source of a reference designator.

    - Example 1:

```
::::::::::::inst::::::
```

In this example, the netlister determines the instance name (or reference designator) for each symbol from the inst property.

- Example 2:

```
:::::::::::::"%{ref},%{des}",ref,des:::::::
```

  In this example, the netlister determines the instance name (or reference designator) for each symbol from one of the following combinations in the order given: the ref and des properties, the ref property, or the des property. Note that this field is not typically used by the CATOS netlister (Cadence).

15. The source of nodes defined in properties.
    - Example:

      None (see the example for field 15 in Specific Symbol Entries).
16. One or more properties that cause the netlister to treat the corresponding non-primitive symbol as primitive. You use this field only in the undetermined section because it is valid only for non-primitive symbols.
    - Example:

```
::::::::::::::primitive:::::
```

  In this example, the netlister treats all symbols that have the primitive property attached as primitive. That is, if an instance of a hierarchical symbol has the primitive property attached, the netlister maps it directly to a corresponding model in the simulator. The lower-level symbols in the hierarchy are ignored.

17. Names a property that the graphical modeling feature uses. You use this field only in the saber section.
    - Example:

```
::::::::::::::::model_generator::::
```

  In this example, graphical modeling determines what type of graphical model to create based on the value of the model_generator property.

18. A list of properties that the simulator is NOT to treat as parameters. In the saber section, this field is valid only for symbols that use the primitive property (prefix attribute, in Viewlogic) to specify the simulation model name, such as symbols in the supplied symbol libraries.

In these examples, the simulator recognizes that the current_control_instance and inductor_to_couple_1 properties are not to be treated as simulator template parameters. This field is NOT valid in the partner section.

In the undetermined section, this field should contain a comma-separated list of the parameter names you kept from being added to the hierarchical cell listed in field 1.

- Cadence and Mentor Graphics Example:

```
primitive:::::::::::: :::
current_control_instance,inductor_to_couple_1:::
```

- Viewlogic Example:

```
:prefix:::::::::::: :::
current_control_instance,inductor_to_couple_1:::
```

19. Reserved for future use.
20. Reserved for future use.
21. Reserved for future use.

For examples of mapping symbols from your Frameway integration symbol libraries, refer to Chapter 8: Using a Mapping File to Map Symbols.

Specific Symbol Entries

In addition to generic entries, a definition section (saber, partner, or undetermined) may contain one or more specific symbol entries within generic entries. Following the definition section name, generic and specific symbol entries are enclosed in braces {} . Within a definition section, a generic entry is listed first followed by specific symbol entries enclosed in braces.

You use specific symbol entries to override symbol properties that are not mapped (or mapped differently) by the corresponding generic entry. A specific symbol entry might be used to designate, for example, that the simulation model name is different from the symbol name, or to supply information needed in simulation or netlisting that is not on the symbol. If both a generic and a specific symbol entry in a mapping file refer to the same symbol property, the specific symbol entry overrides the generic entry.

A specific symbol entry in a definition section is made up of twenty-one fields terminated by colons. The entry itself is terminated by a semicolon. Each of the fields is optional. However, when you leave a field blank, its terminating colon must still be included. The following list outlines what type of information is contained in each field in the specific symbol entry:

```
symbol name : simulation model names : simulator target :
simulator model parameters : parameter-list property : port
names : port types : port direction : port order : port
parameters : net names : net parameters : template directory
: reference designator or instance name : nodes on
properties : primitives : graphical modeling : exclude as
parameters: : : ;
```

Note that a generic entry is terminated by the left brace indicating the beginning of the specific symbol entries. Each specific symbol entry is terminated by a semicolon.

Most of the fields can be specified as a comma-separated list of sources. If a field requires a single source and a list of sources is given, the netlister checks each source in the list starting at the left until it finds the information it needs. A few fields can be specified only by using a property name. A few other fields can be specified only by using the actual value of the item. When either of these restrictions applies, the restriction is noted in the description of that field.

The following list describes the fields that you can include in a specific symbol entry:

1. The name of the symbol to which the entry applies.
   - Example:

```
res::::::::::::::::::::;
```

In this example, res is the name of a symbol in the symbol library indicated by the first field of the generic entry. Since colons are used to delimit items, if you want to include the alias of the library that contains the symbol, you must use double quotes around the item.

   - Example:

```
"analog:res"::::::::::::::::::::;
```

It is important to use the library alias in the first field of each specific symbol entry if there is any chance that you may have symbols of the same name in different libraries. This ensures the netlister uses the correct mapping for each symbol.

2. A comma-separated list of possible simulation model names to use for this symbol. This field names the element (template, model, module, primitive, etc.) representing the symbol to include in the netlist.
   - Example:

```
res:my_r,r:::::::::::::::::::;
```

When the generic entry that contains this specific symbol entry is in the saber section, the netlister searches along the data search path for the templates specified by this field. The netlister uses the value of the first mapping source it finds as the name of a template. In this case, the netlister searches for the template my_r first. If found, the netlister uses it in the netlist for the symbol named res. If the netlister does not find the template my_r, it searches for the template r and uses it in the netlist for the symbol named res. You can use a constructor in the definition of a source for this field.

3. The target simulator for a primitive symbol. This entry determines whether a primitive symbol will appear in a simulator netlist or a partner simulator netlist. However, if you place the mapping for a symbol to be included in the partner netlist in the partner section, for example, you do not need to use this field; the symbol will automatically be targeted to the partner simulator. The same is true for the saber section.
   - Example:

```
and2:and2_l4:partner:::::::::::::::::
```

4. Sources of properties to be considered as individual properties of this symbol. If more than one source is listed, the sources must be separated by commas.
   - Example (Cadence):

```
res:r::rnom<-spconv(r),tnom<-id(tc1)::::::::: ::::::::;
```

In Analog Artist, the symbol res is a resistor with its value specified in SPICE format in a property called r. A second property called tc1 that gives the operating temperature for the resistor is also of interest to the simulator.

If this resistor is to be simulated by the simulator, the value of the property r of the resistor must be converted to a form the simulator recognizes. For example, a resistor with a value in SPICE format of 1kohm, must be given the value 1k in simulator format. In addition, the simulator expects to find the resistor value in a simulator parameter called rnom.

The mapping of the resistor value is done in two steps. First, the spconv function converts the SPICE value specified in the property r to a simulator value. Second, this value is assigned to the simulator template parameter rnom in the simulator netlist.

The tc1 property is mapped simply by assigning its value to the simulator template parameter tnom in the simulator netlist by using the id mapping function.

- Example (Mentor Graphics):

```
resistor:r::rnom<-
spconv(instpar):::::::::::::::::;
```

In Design Architect, the symbol resistor is a resistor with its value specified in SPICE format in a property called instpar.

If this resistor is to be simulated by the simulator, the value of the property instpar of the resistor must be converted to a form the simulator recognizes. For example, a resistor with a value in SPICE format of 1kohm, must be given the value 1k in simulator format. In addition, the simulator expects to find the resistor value in a simulator parameter called rnom.

The mapping of the resistor value is done in two steps. First, the spconv function converts the SPICE value specified in the property instpar to a simulator value. Second, this value is assigned to the simulator template parameter rnom in the simulator netlist.

- Example (Viewlogic):

```
"analog:res":spr::
rnom<-spconv(value):::::::::::::::::;
```

In ViewDraw, the symbol res is a resistor with its value specified in SPICE format in an attribute named value.

If this resistor is to be simulated by the simulator, the value of the attribute value of the resistor must be converted to a form the simulator recognizes. For example, a resistor with a value in SPICE format of 1kohm, must be given the value 1k in simulator format. In addition, the simulator expects to find the resistor value in a simulator parameter called rnom.

The mapping of the resistor value is done in two steps. First, the spconv function converts the SPICE value specified in the attribute value to a simulator value. Second, the netlister assigns this value to the simulator template parameter rnom in the simulator netlist.

5. A list of properties containing composite parameter lists of primitive symbols for the simulator. The composite parameter list is a list of property names, separated by commas, to be passed through as parameters to the primitive symbols. This entry overrides the same field of the generic entry for this symbol.

   Example:

```
saber {
::::parameters:::::::::::::::: {
generic_resistor::::resistance_parameters:::::::::\
  :::::::
generic_capacitor::::::::::::::::::::
}
}
```

In this example, if the generic_capacitor symbol has the property named parameters attached, which can be a comma-separated list of parameters, the netlister uses its value to determine which parameters to extract for that symbol. If the generic_resistor symbol has only the parameters property attached, the netlister uses its value to determine which parameters to extract for the generic_resistor symbol. However, if the generic_resistor symbol has only the resistance_parameters property or both the parameters and the resistance_parameters properties attached, the netlister uses the value of the resistance_parameters property to determine which parameters to extract for the generic_resistor symbol. You can use this method to specify which parameters to extract rather than listing the parameters directly in field 4.

The use of this field for the undetermined section and hierarchical parameter passing was made obsolete by the superior hierarchical parameter passing techniques provided in Release 3.1-1.4 (and later) of the Integration Toolkit from which the netlisters were constructed.

6. The source of a port (or pin) name.
   - Cadence and Viewlogic Example:

```
inv:inv_l4::::saber_inv_pin_defs[<>]:::::::::::::::;
```

In this example, the pin names for the symbol inv are obtained from a table entry called saber_inv_pin_defs shown below. The table entry converts each port (or pin) name on the symbol to a corresponding name for the simulator netlist.

```
saber_inv_pin_defs[A]->in,[Y]->out,[*]->*
```

If the pin is named A, the netlister changes its name to in, and if the pin is named Y, its name is changed to out. All other names are unchanged as indicated by [*]->*.

   - Mentor Graphics Example:

```
inv:inv_l4::::Analogy_logic_2pin_defs[<>]::::::::::::::
:;
```

In this example, the pin names for the symbol inv are obtained from a table entry called Analogy_logic_2pin_defs shown below. The table entry converts each port (or pin) name on the symbol to a corresponding name for the simulator netlist.

```
Analogy_logic_2pin_defs[i0]->in,[I0]->in,[EN]-
>enable,[en]->enable,[*]->*
```

If the pin is named io, the netlister changes its name to in, if the pin is named EN, its name is changed to enable, etc. All other names are unchanged as indicated by [*]->*.

   - Mentor Graphics, Cadence, and Viewlogic:

You can use the <> symbol in this field to indicate that the port names to be used are the actual names assigned to the ports rather than those names given as values of a property. .

7. The source of the port (or pin) type of a symbol. This entry indicates whether the ports of a symbol are analog or digital. This field is not required for supplied templates or templates in the Template Database.

   - Example:

```
saadc:::::adc_ports[<>]:::::::::::::;
```

An analog-to-digital converter contains both analog and digital ports. Therefore, it is not possible to designate a port type for the entire symbol. In this example, each port is individually designated analog or digital. The following table entry is referenced:

```
adc_ports[in]->analog,[gate]->analog, [gnd]-
>analog,[*]->digital
```

This table entry designates the ports in, gate, and gnd as analog signals; all other ports are specified as digital.

You can use the <> symbol in this field to indicate that the port names to be used are the actual names assigned to the ports rather than given as values of a property.

8. Used for the following three purposes:

   a. Provides the pin direction if not specified on the schematic.

   The named source (property) contains information required by the partner simulator defining the direction of the port (or pin). In this example, the source of the port (or pin) direction is the value of the property pintype.

   Example:

```
dff::::::pintype::::::::::::
```

   b. Designates special pins such as power pins and how they are to be used in Hypermodel interface insertion.

The power and ground connection points of the Hypermodel interfaces inserted at other pins of the symbol will be connected to the nets to which these pins are connected. The pin directions to designate power and ground pins corresponding to template or module connection points are pow, gnd, unpow, and ungnd (the netlister does not insert Hypermodel interfaces for pins designated as power or ground pins). Valid values for Hypermodel interface ports are in (input), out (output), bi (bidirectional), pow (power), gnd (ground), unpow (power pin to be excluded from netlist), ungnd (ground pin to be excluded from netlist), and ignore (pin to be excluded from the netlist).

You can also use an entry in the pins section of a Hypermodel interface (.shm) file to designate power and ground pins.

c. Causes the netlister to ignore pins when they are not part of the simulation model.

This is done by using a table to assign the pin-direction ignore, unpow, or ungnd to pins that are to be excluded from the netlist. As you can see from the previous discussion, the unpow and ungnd direction designations serve a dual purpose. You can use this designation when pins occur on a symbol for which there are no connection points on a corresponding simulator template or Verilog module. When the pin direction of a pin is set to ignore, the netlister does not insert a Hypermodel interface at that pin, and the pin does not appear in the netlist.

9. The source of the port (or pin) order. This source provides a list of ports by name, separated by commas. If you use a property rather than a comma-separated list, the value of the property must be a comma-separated list of ports.

- Mentor Graphics Example:

```
and2:and2_l4::::::out,i0,i1:::::::::::;
```

In this example, the order is given for the ports out, i0, and i1 of the and2 gate.

- Cadence Example:

```
and2:and2_l4::::::Y,A,B:::::::::::;
```

In this example, the order is given for the ports Y, A, and B of the and2 gate.

- Viewlogic Example:

```
"builtin:and2":and2_l4::::::Y,A0,A1:::::::::::;
```

  In this example, the order is given for the ports Y, A0, and A1 of the and2 gate.

10. The source of port properties to be retained and passed on to the simulator.
    - Example:

      None (not used by the simulator or Verilog simulator).
11. The source of the names of the nets contained in a non-primitive symbol. This field is used only in the undetermined section.
    - Example:

      None. There is typically no need to map a net name in a specific schematic.
12. The source of net properties to be retained and passed on to the simulator. This field is used only in the undetermined section.
    - Example:

```
my_sym::::::::::type,charge,delay,drive::::::::
```

  In this example, the net properties type, charge, delay, and drive are to be passed on to the simulator. Net properties exist only on non-primitive symbols; therefore, you must specify them in the undetermined definition section of the mapping file. The simulator does not use net properties.

13. The path to the template directory.
    - Example: None.
14. The source of the instance name. This field can be a single property name or a constructor containing strings or properties to specify the source of a reference designator.
    - Example 1:

```
and2:::::::::::::inst:::::::
```

In this example, the netlister determines the instance name (or reference designator) for the and2 symbol from the inst property.

- Example 2:

```
and2:::::::::::::"%{ref},%{des}",ref,des:::::::;
```

In this example, the netlister determines the instance name (or reference designator) for the and2 symbol from one of the following combinations in the order given: the ref and des attributes, the ref attribute, or the des attribute. This field is not typically used by the CATOS netlister (Cadence).

15. The source of nodes defined in properties. This field is used only in the saber section.

    - Example:

```
npn:q:::::::::::::s<-id(bn)::::::;
```

In this example, the bulk or substrate node for the npn transistor is found in a property called bn. The value of this property is connected to the connection point s in the simulator netlist.

16. Field 16 is not available as a Specific Symbol Entry. It must be left blank.
17. Names a property that the graphical modeling feature uses. You use this field only in the saber section.

    - Example:

```
my_other_part::::::::::::::::model_generator::::
```

In this example, graphical modeling determines what type of graphical model to create based on the value of the model_generator property.

18. See the description of Field 18 in Generic Entries.
19. Reserved for future use.
20. Reserved for future use.
21. Reserved for future use.

For examples of mapping symbols from your Frameway integration symbol libraries, refer to Chapter 8: Using a Mapping File to Map Symbols.

Multiple Generic Entries

You use a generic entry for each symbol library that you are mapping. This way, you can provide general mapping information pertaining to a specific symbol library in the appropriate generic entry and provide symbol-specific mapping information, as necessary, for each symbol in the library by using specific symbol entries.

The mapping files have the capability to distinguish symbols by symbol library. The netlisters keep track of the symbol library containing a given symbol and use it to locate the proper generic mapping and the specific symbol mapping within the generic mapping. For example, the catos.map file contains the following mappings for the cap symbol of the analogLib library and the and2 symbol of the sample library:

```
saber {
###### Generic entry.
analogLib:::::::::::::::::::: {
  ###### Specific symbol entries
  cap:c::c<-id(c)::artist_pin_def[<>]:::::::::::::::;
  }
###### Generic entry.
sample:::::::::::::::::::: {
  ###### Specific symbol entries
  and2:and2_l4::::saber_logic_pin_defs[<>]:
  artist_digital_pins[<>]::::::::::::::;
  }
}
```

The multiple-generic entry feature of the mapping file also allows you to specify different mappings for symbols of the same name that are contained in different symbol libraries.

Default Generic Mapping

If you have certain symbols that are not located in specific libraries, you can place the generic mappings for these symbols in a default mapping where the symbol library name (field 1) of the generic entry is left blank.

Include and Exclude Sections

The include section is no longer used by the netlister and may be left blank.

You use the exclude section to list symbols that are to be ignored by the netlister when it creates the netlist. For example, you can exclude one or more symbols from the netlist to increase the speed of a simulation. Or, you can exclude symbols from the netlist that have no underlying functional representation, such as vdd or vcc. The exclude section takes the following form where symbol_a, symbol_b, and symbol_c are the names of symbols to be excluded from the netlist.

```
exclude{
symbol_a,symbol_b,symbol_c
}
```

Interaction of Mapping, Special Properties, and Defaults

When more than one source of an item of information that is to be included in a netlist is available, the netlister chooses the source it uses, in order of precedence, from the sources listed below. The reference in parentheses ( ) indicates the number of the field in a generic or specific symbol mapping file entry that corresponds to the parameter. A source in a specific symbol entry overrides a source in a generic entry. Specially-recognized properties, such as the SaberTemplate property and the Target_Simulator property are described in Chapter 6: Using Specially-Recognized Properties for Mapping.

Simulation Model Name

The sources of the simulation model name in order of precedence are as follows:

1. The SaberTemplate property.
2. As designated in a mapping file entry (field 2).
3. The cell name

Target Simulator for Primitive Symbol

The sources of the target simulator for a primitive symbol in order of precedence are as follows:

1. If the primitive symbol has a SaberTemplate, SaberParameters, or Saber_ParameterName property, then the simulator.
2. As designated by the Target_Simulator property. If a specific symbol entry is in only one section of the mapping file (saber or partner), then the corresponding simulator

3. If a specific symbol entry is in both sections of the mapping file (saber and partner), then as designated in a mapping file entry (field 3).
4. If a template is found for the part, then the simulator.
5. If the primitive symbol has analog ports, then the simulator.
6. The partner simulator.

Note that the -ae command line option causes the netlister to assume that templates will be provided so it does not search for them. When you use this option, it places symbols not specifically assigned to the Verilog netlist, either by a mapping file or by the Target_Simulator property, into the simulator netlist rather than the Verilog simulator netlist.

You can use this command line option with the SaberPrepend property to allow the inclusion of templates at the beginning of the netlist so that they can be referenced later in the netlist.

Port or Pin Name

The sources of the port (or pin) name in order of precedence are:

1. As designated in a mapping file entry (field 6).
2. The existing symbol port name.

Port or Pin Type

The sources of the port (or pin) type in order of precedence are:

1. The template (for MAST models).
2. As designated by the Port_Type property.
3. As designated in a mapping file entry (field 7).If the target simulator is the simulator, then analog
4. If the target simulator is the partner simulator, then digital.
5. As implied by the location of the mapping entry (i.e. saber section versus partner section).

Port or Pin Direction

The sources of the port (or pin) direction for Hypermodel interface insertion versus pin declaration, in order of precedence:

1. As designated in a Hypermodel library file.
2. As designated in a mapping file entry (field 8).
3. As designated by the direction of the port symbol.

Port Ordering

The sources of port ordering, in order of precedence:

1. As designated by the SaberPinOrder or PartnerPinOrder properties.
2. As designated in a mapping file entry (field 9).
3. The existing port order.

Net Names

The sources of net names, in order of precedence:

1. As designated in a mapping file entry (field 11).
2. The existing net name.

Other Properties

If either of the following properties is present, it will be used in addition to the individual symbol properties designated in the mapping file (field 4):

- Parameters
- Saber_ParameterName
- These two properties are described in Chapter 6: Using Specially-Recognized Properties for Mapping.

Mapping File Considerations for Mentor Graphics Users

Mentor Graphics users can place mapping files directly into the component libraries; they are then read automatically by the netlister. Because Mentor Graphics also uses the concept of a mapping file in their Library Management System, the.map file extension can cause problems, so the.smp extension must be used. This section explains how this works.

If you have created your own component libraries with components having component interfaces named differently from the components themselves, and if you subsequently use symbols registered to these interfaces in a design, then the following discussion about.smp mapping files does not apply. To map such symbols, you must use your own mapping file such as user.map.

The following topics discuss these considerations for single and multiple component use:

1. Single Components in Mentor Graphics
2. Multiple Component Interface Support in Mentor Graphics

Single Components in Mentor Graphics

You can place the mapping file for an entire directory of Mentor Graphics components in that directory. The file must have the same name as the directory, and have the.smp extension. This file contains only a saber section, generic entry, and must have field one (the library name) left blank. For example, the directory $SABER_MGC8_SYMBOLS/sbr_electric_lib contains a file named sbr_electric_lib.smp with contents:

```
saber {
##### Generic entry
:
  :::::pin_type::::::
  inst,ref
  :::model_generator:
  current_control_instance,
  current_control_instance_1,
  current_control_instance_2,
  current_control_instance_3,
  current_control_instance_4,
  current_control_instance_5,
  inductor_to_couple_1,inductor_to_couple_2,inst,
  ref,model
  :::
}
```

This _generic_ mapping is applied to every component which comes from the $SABER_MGC8_SYMBOLS/sbr_electric_lib directory.

To get the equivalent of a _specific_ mapping entry for a single symbol in this library, place a mapping file into the Mentor Graphics component directory in which the symbol is located, where it will be read automatically by the netlister. This mapping file must have the same name as the component, with the extension.smp (not.map). Presence of this file requires that the directory-generic file described above also be present. For example, in previous releases, the component directory $SABER_MGC8_SYMBOLS/sbr_electric_lib/v_pulse had a file named v_pulse.smp with contents:

```
saber
{
  {
  :
  v::
  tran<-"(pulse=(v1=%{initial},v2=%{pulse},
    tr=%{tr},tf=%{tf},td=%{delay},pw=%{width},
    per=%{period}))",
  tran<-"(pulse=(v1=%{initial},v2=%{pulse},
    tr=%{tr},tf=%{tf},td=%{delay},pw=%{width}))",
  tran<-"(pulse=(v1=%{initial},v2=%{pulse},
    tr=%{tr},tf=%{tf},td=%{delay},per=%{period}))",
  tran<-"(pulse=(v1=%{initial},v2=%{pulse},
    tr=%{tr},tf=%{tf},td=%{delay}))",
  tran<-"(pulse=(v1=%{initial},v2=%{pulse},
    tr=%{tr},tf=%{tf},pw=%{width},per=%{period}))",
  tran<-"(pulse=(v1=%{initial},v2=%{pulse},
    tr=%{tr},tf=%{tf},pw=%{width}))",
  tran<-"(pulse=(v1=%{initial},v2=%{pulse},
    tr=%{tr},tf=%{tf},per=%{period}))",
  tran<-"(pulse=(v1=%{initial},v2=%{pulse},
    tr=%{tr},tf=%{tf}))",
  ac<-"(mag=%{ac_mag},phase=%{ac_phase})",
  ac<-"(mag=%{ac_mag},phase=0)",
  ns<-"(nv=%{white_noise},nf=%{flicker_noise})",
  ns<-"(nf=%{flicker_noise})",
  ns<-"(nv=%{white_noise})":::::::::::::::::;
  }
}
```

This file has no generic entry specified, but contains only an unnamed specific mapping file entry (meaning that field one is blank).

Note that, beginning with Release 5.0, this file is no longer shipped because use of the Template Database makes the file unnecessary.

Multiple Component Interface Support in Mentor Graphics

Starting with Release 4.3, multiple interfaces and symbols of components are supported. This has applications if mapping (via mapping files) is desired for components which have multiple interfaces. The resulting component_name+interface_name is matched against field one in a specific mapping file entry.

For example, the $MGC_APLIB/opamp_5pin contains 3 part interfaces: MG_STD, ANSI, and OLD. Prior to Release 4.3 the mapping file would have an entry as follows:

```
“opamp_5pin":::: etc.
```

Now the following three entries are required:

```
"opamp_5pin:MG_STD"::::: etc.
"opamp_5pin:ANSI"::::: etc.
"opamp_5pin:OLD"::::: etc.
```

Since the new component_name:interface_name contains an embedded ":" character, note the necessity of using "" characters around the field one value.

For each primitive instance, the component_name is used in field one (no interface name is needed). For example, all symbols in $SABER_MGC8_SYMBOLS libraries have interface names the same as the component names, so these components do not need the component_name:interface_name format. Whenever the interface name of the instance symbol is different from the component name, the component_name:interface_name format must be used in field one.

As another example, assume a component called /homes/my_home/my_designs/transistor contains the following symbol names, interface names, and port definitions:

Symbol Name	**Interface Name**	**Ports**
npn	BJT	C, B, E
pnp	BJT	C, B, E
nmos	MOS	D, G, S, B
pmos	MOS	D, G, S, B
njfet	JFET	D, G, S
pjfet	JFET	D, G, S

In this case, to map the symbol "pnp" would require a specific mapping file entry that appear as follows:

```
"transistor:BJT"::::: etc.
```

This mapping would also apply to the "npn" symbol, since it belongs to the same interface as symbol "pnp".

Note that it is the Interface Name, not the Symbol Name, that gets used in the mapping.

13

Using a Mapping File to Convert SPICE Symbols

The following sections contain a list of the mapping functions available for converting SPICE symbols, and two examples demonstrating the use of these mapping functions. These examples assume you are using the Frameway integration into the Mentor Graphics environment (therefore the references to the accuparts_lib). However, all of these mapping functions are also available with the other Frameway integrations and Sketch.

Mapping Functions Used to Convert SPICE Symbols

The netlisters include a set of mapping functions that you can use in a mapping file to convert schematic properties in SPICE syntax to simulation parameters in the syntax of the MAST analog hardware description language. These mapping functions are:

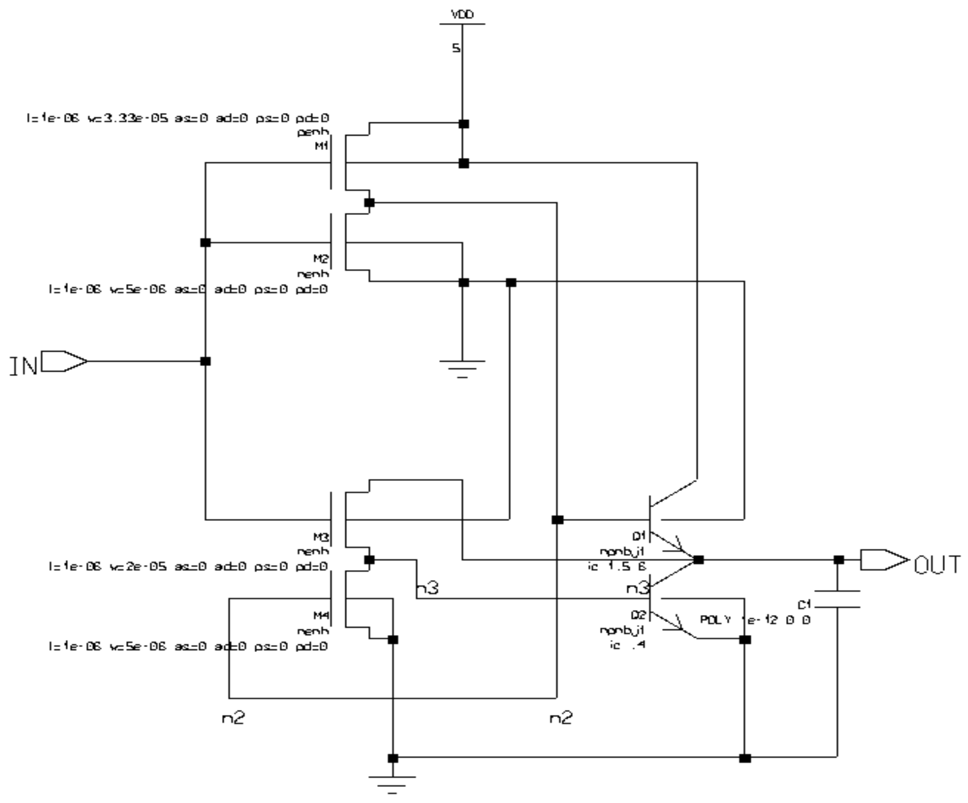
- spsource
  translates SPICE source parameters to MAST
- spvalues
  translates a general SPICE parameter list to MAST
- spresvals
  translates a SPICE resistor parameter list to MAST
- spstorage
  translates SPICE capacitor or inductor parameters to MAST

- sptran
  translates a SPICE transistor or diode parameter list to MAST
- spopttran
  translates the parameter list for a transistor or diode that does not have the model name in it
- sptranmod
  extracts the model name from a parameter list for a diode or a transistor
- spifpoly
  If the SPICE parameter list for a capacitor or an inductor has poly parameters in it, it returns the second argument (typically the name of the template that implements the poly parameters for the device). Otherwise, it returns the third argument (typically the name of the template that does not implement the poly parameters).

Inverter Example

This section uses an obsolete file, mentos8.map, as the basis for the example. The example is for illustration purposes only.

The first example demonstrates the use of spstorage, spifpoly, and spopttran in a transistor level model for the inverter that appears in the following figure. The mapping functions do not translate the model cards from a SPICE deck, as these are not stored in the schematic. You should use the SPITOS program to translate the SPICE model cards into MAST syntax.


VDD
5
l=1e-06 w=3.33e-05 as=0 ad=0 ps=0 pd=0
penh
M1
M2
nenh
l=1e-06 w=5e-06 as=0 ad=0 ps=0 pd=0
IN
M3
nenh
l=1e-06 w=2e-05 as=0 ad=0 ps=0 pd=0
n3
M4
nenh
l=1e-06 w=5e-06 as=0 ad=0 ps=0 pd=0
n2
n2
Q1
npnbjt
ic 1.5 6
n3
Q2
npnbjt
ic .4
POLY 1e-12 0 0
C1
OUT

In the following example, the model cards have been translated and placed in a file named ringmods.sin. This first example makes use of the following portion of mentos8.map to choose the proper templates and convert the simulation parameters for the npn_4t, capacitor, and mosfet symbols of accuparts_lib.

```
tables {
Analogy_spice_pins[POS]->p,[pos]->p,
  [NEG]->m,[neg]->m,
  [P1]->pp,[p1]->pp,
  [N1]->pm,[n1]->pm,
  [P2]->sp,[p2]->sp,
  [N2]->sm,[n2]->sm,
  [*]->*
Analogy_bjt_4pins[sub]->s,[SUB]->s,[*]->*
}
saber {
#Generic entry
accuparts_lib:comp:::::::::::inst::::::: {
  #Specific cell entries
  npn_4t:q::model,<-spopttran(instpar,"bjt")\
    ::Analogy_bjt_4pins[<>]:::::::::::::::;
  capacitor:"%{spifpoly(instpar,"spcp","spc")}",c\
    ::<-spstorage(instpar,"capacitor")\
    ::Analogy_spice_pins[<>]:::::::::::::::;
  mosfet:m::model,<-spopttran(instpar,"mosfet")
    :::::::::::::::::; #ports match
    }
  }
```

When you select the Report > Object menu item in the schematic editor with the capacitor symbol selected, the following abstracted list of properties is reported:

Instance	**Name**	**Location**	**Version**
I$15	$MGC_APLIB/capacitor/capacitor	(8.75, -2.50)	1

Property	**Name**	**Value**	**Type**	**Location**	**Switches**
T$530	INST	C1	string	(8.44,-2.13)	-Variable Initially on Symbol, Value Modified

Property	Name	Value	Type	Location	Switches
T$543	INSTPAR	POLY 1e-12 0 0	string	(8.44,-2.31)	-Variable Initially on Symbol, Value Modified

The template for this instance of the capacitor symbol is selected by the "%{spifpoly(instpar,"spcp","spc")}" entry in the second field of the specific symbol mapping entry. The spifpoly function determines if there are poly parameters on the instpar property of the symbol instance. Since there is, the spcp template is chosen, since its name was given as the second argument to the spifpoly mapping function. Furthermore, the <-spstorage(instpar,"capacitor") entry in the fourth field of the specific symbol mapping entry causes the instpar property value:

POLY 1e-12 0 0

to be translated to the parameter list:

pc=[1e-12,0,0]

in the netlist for the simulator. The second argument, "capacitor" of the spstorage mapping function informs the function as to what type of device the function is being used for, so that the property parameter name (pc) can be chosen. The two legal values for the second argument of the spstorage mapping function are capacitor and inductor.

The remaining devices in the schematic are transistors that are all mapped in a similar fashion. When you select Report > Object on some mosfet and npn_4t symbols in the schematic, the following abstracted list of properties is reported.

Instance	Name	Location	Version
I$6	$MGC_APLIB/mosfet/mosfet	(3.00,3.50)	1

Property	Name	Value	Type	Location	Switches
T$771	SaberPrepend	ringmode	string	(2.25,3.31)	-Not Visible

Property	Name	Value	Type	Location	Switches
T$201	INST	M1	string	(3.29,3.76)	-Variable Initially on Symbol, Value Modified, Attribute Modified
T$204	INSTPAR	l=1e-06 w=3.33e-05 as=0 ad=0 ps=0 pd=0	string	(3.29,4.14)	-Variable Initially on Symbol, Value Modified
T$205	MODEL	penh	string	(3.29,3.95)	-Variable Initially on Symbol, Value Modified, Attribute Modified

Instance	Name	Location	Version
I$4	$MGC_APLIB/mosfet/mosfet	(3.00,2.50)	1

Property	Name	Value	Type	Location	Switches
T$125	INST	M2	string	(3.29,2.24)	-Variable Initially on Symbol, Value Modified
T$128	INSTPAR	l=1e-06 w=5e-06 as=0 ad=0 ps=0 pd=0	string	(3.29,1.86)	-Variable Initially on Symbol, Value Modified
T$129	MODEL	nenh	string	(3.29,2.05)	-Variable Initially on Symbol, Value Modified

Instance	Name	Location	Version
I$1	$MGC_APLIB/npn_4t/npn_4t	(6.50,-1.00)	1

Property	Name	Value	Type	Location	Switches
T$63	MODEL	npnbjt	string	(6.69,-2.44)	-Variable
T$12	INST	Q1	string	(6.69,-2.15)	-Variable Initially on Symbol, Value Modified
T$30	INSTPAR	ic 1.5 6	string	(6.69,-1.63)	-Variable Initially on Symbol, Value Modified

The model entry in the fourth field of the specific symbol entries for the transistors specifies that model is a parameter for the template, and that the value of the model property is to be taken as the value of the model parameter. Since the value of the model parameter is an identifier, the netlister recognizes that it must get the actual model values elsewhere and make external declarations:

```
external q..model   npnbjt
external m..model   nenh, penh
```

for the models npnbjt, nenh, and penh set in the schematic. The SaberPrepend property with value ringmods causes the netlister to add the line:

<ringmods.sin

at the top of the netlist to include the model definitions:

```
q..model npnbjt = (type=_n,is=4.8e-18,bf=100,nf=1,
ikf=0.00096,br=8,nr=1,ikr=0.02,rb=200,re=15,rc=150,
cje=1.5e-14,mje=0.5,tf=2e-11,xtf=100,vtf=3,itf=0.04,
tr=4e-10,cjc=2e-14,mjc=0.5,cjs=8e-14,mjs=0.5)
```

```
m..model nenh = (type=_n,level=2,vto=0.626634,
kp=5.28837e-05,gamma=0.287822,phi=0.880849,
cgso=2.89e-10,cgdo=2.89e-10,rsh=68,cj=0.000345,
mj=0.916,cjsw=1.74e-10,mjsw=0.195,tox=2.25e-08,
nsub=9.31807e+16,nss=3e+10,nfs=0.01,tpg=1,xj=9e-07,
ld=0,uo=511.152,ucrit=200,uexp=0.0161578,vmax=57064.,
neff=0.261026,delta=3.02002)

m..model penh = (type=_p,level=2,vto=-0.617486,
kp=1.93032e-05,gamma=0.527415,phi=0.85,cgso=3.35e-10,
cgdo=3.35e-10,rsh=154,cj=0.000421,mj=0.3285,
cjsw=2.23e-10,mjsw=0.307,tox=2.25e-08,
nsub=8.55587e+14,nss=3e+10,nfs=1e+11,tpg=-1,
xj=5.03159e-09,ld=6e-08,uo=33.9632,ucrit=98755.5,
uexp=0.141605,vmax=6786.31,neff=12.182,delta=6.01138)
```

that were converted with SPITOS.

The <-spopttran(instpar,"bjt") entry in the fourth field of the specific symbol entry for the npn_4t transistor specifies that the optional parameters for a bjt are to be converted from the instpar property and placed at the end of the netlist entry for the transistor. Similarly, the same actions are performed for the <-spopttran(instpar,"mosfet") entry for the mosfet. The four legal values for the second argument of the spopttran mapping function are bjt, diode, jfet, and mosfet.

When a symbol for this inv schematic is made and instantiated eleven times, along with a vdd source, the following netlist results:

```
########################################################
#  Netlist for design ring
#  Created by the Netlister Toolkit 3.2-2.2
#  Created on Fri Jun 24 16:00:56 1994.
########################################################
<ringmods.sin
########################################################
#  Intermediate template inv
########################################################
template inv out:out in:in ground:0 vdd:vdd

{
external q..model   npnbjt
external m..model   nenh, penh

q.Q1 c:vdd e:out s:0 b:n2 = model=npnbjt, ic=[1.5,6]
q.Q2 c:out e:0 s:0 b:n3 = model=npnbjt, ic=[.4]
m.M2 d:n2 g:in s:0 b:0 = model=nenh, l=1e-06,w=5e-06,as=0,
ad=0,ps=0,pd=0
m.M3 d:out g:in s:n3 b:0 = model=nenh, l=1e-06,w=2e-05,
as=0,ad=0,ps=0,pd=0
m.M4 d:n3 g:n2 s:0 b:0 = model=nenh, l=1e-06,w=5e-06,as=0,
ad=0,ps=0,pd=0
m.M1 d:n2 g:in s:vdd b:vdd = model=penh, l=1e-06,
w=3.33e-05,as=0,ad=0,ps=0,pd=0
spcp.C1 p:out m:0 = pc=[1e-12,0,0]
}
```

```
########################################################
#  Instances found in the top level of design ring#
########################################################

inv.inv1 out:osc in:osc_inv ground:0 vdd:vdd
inv.inv2 out:n2 in:osc ground:0 vdd:vdd
inv.inv3 out:n3 in:n2 ground:0 vdd:vdd
inv.inv10 out:n10 in:n9 ground:0 vdd:vdd
inv.inv11 out:osc_inv in:n10 ground:0 vdd:vdd
inv.inv4 out:n4 in:n3 ground:0 vdd:vdd
inv.inv5 out:n5 in:n4 ground:0 vdd:vdd
v.vdd p:vdd m:0 = dc=6
inv.inv6 out:n6 in:n5 ground:0 vdd:vdd
inv.inv7 out:n7 in:n6 ground:0 vdd:vdd
inv.inv8 out:n8 in:n7 ground:0 vdd:vdd
inv.inv9 out:n9 in:n8 ground:0 vdd:vdd
```

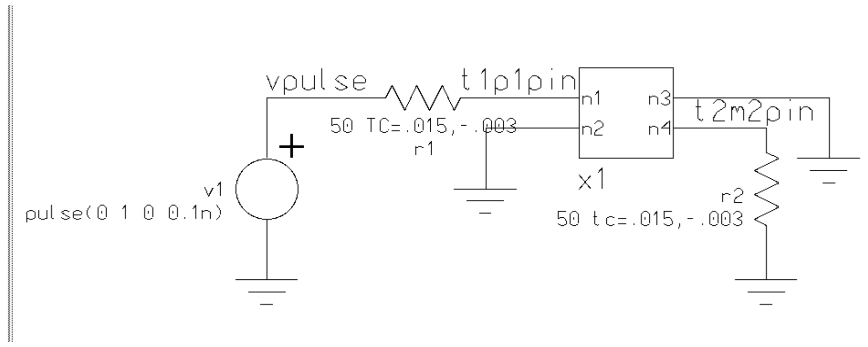
You can use the following simulator commands to simulate this ring oscillator.

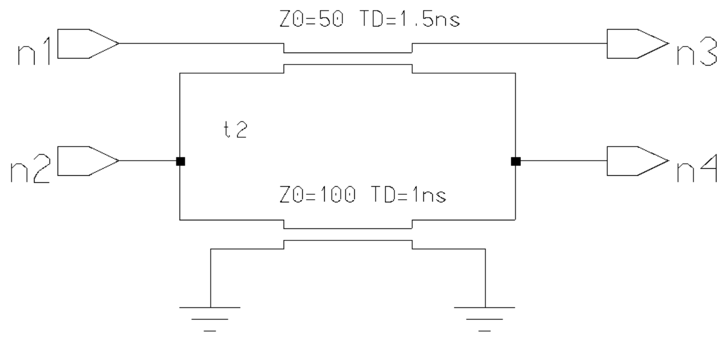
```
sigset vdd 6 osc_inv 3 (ip newip, defaultip zero)
tr(tstep 2n, tend 40n, trip newip, mon 10)
```

Transmission Line Example

This section uses an obsolete file, mentos8.map, as the basis for the example. The example is for illustration purposes only.

This example demonstrates the use of spresvals, spsource, and spvalues in a transmission line inverter that appears in the following two figures.


vpulse
t1p1pin
50 TC=.015,-.003
r1
n1
n2
n3
n4
t2m2pin
x1
v1
pulse(0 1 0 0.1n)
r2
50 tc=.015,-.003


Z0=50 TD=1.5ns
n1
n3
t2
n2
n4
Z0=100 TD=1ns

This example makes use of the following portion of mentos8.map to choose the proper templates and convert the simulation parameters for the voltage_source, resistor, and transmission_line symbols of accuparts_lib:

```
tables {
Analogy_spice_pins[POS]->p,[pos]->p,
  [NEG]->m,[neg]->m,
  [P1]->pp,[p1]->pp,
  [N1]->pm,[n1]->pm,
  [P2]->sp,[p2]->sp,
  [N2]->sm,[n2]->sm,
  [*]->*
Analogy_spt[N1]->p1,[n1]->p1,
  [N2]->p2,[n2]->p2,
  [N3]->m1,[n3]->m1,
  [N4]->m2,[n4]->m2
}
saber {
#Generic entry
accuparts_lib:comp:::::::::::inst::::::: {
  #Specific cell entries
  resistor:r::<-spresvals(instpar)::
    Analogy_spice_pins[<>]:::::::::::::::;
  transmission_line:spt::<-spvalues(instpar)
    ::Analogy_spt[<>]:::::::::::::::;
  voltage_source:v::<-spsource(instpar)
    ::Analogy_spice_pins[<>]:::::::::::::::;
  }
}
```

The value of the instpar property on the voltage_source symbol in the schematic above can be seen to have the value:

```
pulse(0 1 0 0.1n)
```

This value is translated into the proper MAST analog hardware description language syntax for the simulator by the mapping function spsource in the specific symbol entry for the voltage_source symbol in the mapping file. The <-spsource(instpar) entry in the fourth field of the specific symbol mapping causes the instpar property value: instpar

```
pulse(0 1 0 0.1n)
```

to be translated to the parameter list:

```
tran=(pulse=(0,1,0,0.1n))
```

in the netlist for the simulator.

The value of the instpar property on the resistor symbol in the schematic can be seen to have the value:

```
50 TC=.015,-.003
```

This value is translated into the proper MAST analog hardware description language syntax for the simulator by the mapping function spresvals in the specific symbol entry for the resistor symbol in the mapping file. The <-spresvals(instpar) entry in the fourth field of the specific symbol mapping causes the instpar property value:

```
50 TC=.015,-.003
```

to be translated to the parameter list:

```
50,TC=[.015,-.003]
```

in the netlist for the simulator.

The value of the instpar property on the transmission_line symbol in the schematic can be seen to have the value:

```
Z0=50 TD=1.5ns
```

This value is translated into the proper MAST analog hardware description language syntax for the simulator by the mapping function spvalues in the specific symbol entry for the transmission_line symbol in the mapping file. The <-spvalues(instpar) entry in the fourth field of the specific symbol mapping causes the instpar property value:

```
Z0=50 TD=1.5ns
```

to be translated to the parameter list:

```
Z0=50,TD=1.5n
```

in the netlist for the simulator.

When the schematics are netlisted with the mentos8 netlister, the following netlist results:

```
########################################################
#  Netlist for design transline
#  Created by the Netlister Toolkit  3.3c-2.6
#  Created on Thu Jun 30 13:54:50 1994.
########################################################
########################################################
#  Intermediate template tline
########################################################

template tline n1:n1 n2:n2 n3:n3 n4:n4 ground:0

{
spt.t1 p1:n1 p2:n2 m1:n3 m2:n4 = Z0=50,TD=1.5n
spt.t2 p1:n2 p2:0 m1:n4 m2:0 = Z0=100,TD=1n
}

########################################################
#  Instances found in the top level of design transline
########################################################

v.v1 p:vpulse m:0 = tran=(pulse=(0,1,0,0.1n))
r.r1 p:vpulse m:t1p1pin = 50,TC=[.015,-.003]
r.r2 p:t2m2pin m:0 = 50,tc=[.015,-.003]
tline.x1 n1:t1p1pin n2:0 n3:0 n4:t2m2pin ground:0
```

This transmission line inverter can be simulated with the following commands for the simulator:

```
dc
tr(tstep 0.1n, tend 40n
```

The sptranmod and sptran mapping functions are used together when the entire transistor parameter list is stored in one property such as the instpar property. Consider an instpar property on a mosfet with value:

```
mod1 l=10u w=5u ad=2p as=2p
```

In such a situation, the entry in the fourth field would be:

```
model<-sptranmod(instpar),  \
  <-sptran(instpar,"mosfet")
```

Because the model name is an identifier that will have to be declared, it is extracted separately with the first expression:

```
model<-sptranmod(instpar)
```

which is used to extract only the model name to produce the parameter:

```
model=mod1
```

and cause model mod1 to be declared:

```
external m..model mod1
```

when the instance is below the top level of the circuit hierarchy. The full parameter list is provided by the mapping:

```
<-sptran(instpar,"mosfet")
```

This produces the rest of the parameter list for the mosfet as follows:

```
model=mod1,l=10u,w=5u,ad=2p,as=2p
```

Notice that the model parameter gets listed twice. Although this may seem unusual, it does not affect processing by the simulator in any adverse way.

14

Using Cadence simInfo to Avoid Mapping Files

This section describes how to use the simInfo section of the CDF with the Frameway Integration for Cadence. Using simInfo can greatly reduce the need for Mapping Files. An example SKILL script is introduced, showing how to create cells that do not need to be mapped. A method is described for updating the standard Cadence library analogLib with a Saber sub-section. Information about the pre-defined simInfo properties is provided, listing both those that are recognized and those that are not recognized. The topics covered in detail include:

Definitions

CDF	In the Cadence paradigm, the Component Description Format (CDF) assigns attributes, properties, and parameters to components or libraries of components. When a CDF is attached to a library, all cells (components) in the library inherit the description. When a CDF is attached to a cell, the CDF stores information specific to that cell.
cell	A cell is a database object that forms a fundamental design unit. A component of a design. A collection of different aspects of the component's implementations, such as schematic, behavioral, or symbol representations.
simInfo	The simInfo (Simulation Information) is one section of the CDF. SimInfo allows for parameter and terminal specification for a given simulator. The simInfo section does not define the actual cell parameters and terminals but does define which cell parameters and terminals will be netlisted for the specified simulator.
SKILL	A programming language developed by Cadence to interface tools to the Design Framework II environment. SKILL provides database access, user interface functions, and graphics editing.

Mapping with simInfo

The catos family of netlisters (as of release 4.0-2.8) have been enhanced to make use of the simInfo section of the CDF. If a Saber sub-section of the simInfo section exists for a primitive-level component, then the netlister will read it. This feature reduces the need for mapping files.

Example SKILL Script

An example of how to use the simInfo section is supplied on-line at $SABER_HOME/user_grp/cds_sim_info. This directory contains an example SKILL script in the file SaberSimInfo.il. The script can be used to update the standard Cadence library analogLib with a Saber sub-section of the simInfo section of a CDF. This script allows the symbols in Cadence's analogLib to be mapped to MAST templates by using simInfo instead of mapping files. The script also serves as an example of how to create additional cells that do not need to be mapped. A small sample of this script follows:

```
libID = dmOpenLib( "analogLib" )

/******** diode -> d ********/
cdfID = cdfGetBaseCellCDF( dmFindCell( libID "diode" ))
cdfID->simInfo->Saber = '(nil
  componentName d
  termOrder (PLUS MINUS)
  instParameters (model area vd)
  propMapping (nil ic vd )
```

Updating the Library

To install the SimInfo mappings, you open the CIW (Cadence's Command Interpreter Window) with analogLib in your library search path. Then, in the CIW, you type

```
load “SaberSimInfo.il”
```

This command will create a user CDF with a simInfo section. The simInfo section contains all the information necessary to map the analogLib symbols to their corresponding MAST templates.

Recommendations

If you follow the instructions in the previous section, then only the Cadence symbols (analogLib) will be mapped by using simInfo. The supplied symbols will retain their mapping by using the standard mapping files provided (transparently and automatically). This is the mapping arrangement we recommend. If you wish to add additional symbols, we recommend using simInfo.

Templates Using the Union Data-Structure

Templates that make use of union data-structures cannot be mapped in the CDF. Some of the supplied templates use this data-structure. Therefore, we recommend using the standard mapping files to map between supplied symbols and templates.

If you wish to use simInfo to map all of your symbols, including supplied symbols, then you must replace any templates that use unions with templates that do not. Therefore, some new MAST templates are supplied to map the voltage and current sources and the resistor and capacitor. These templates can be found in $SABER_HOME/user_grp/cds_sim_info. You should make sure these templates are in your SABER_DATA_PATH. These templates can also serve as examples of how to write templates without unions.

Supplied symbols that use these new templates are included in the library:

```
$SABER_HOME/user_grp/cds_sim_info/NoMapSources
```

Pre-Defined Properties Recognized by Catos Netlisters

This section describes the pre-defined properties that are recognized by catos netlisters.

componentName

componentName is the type of component you are creating. The value of this property should correspond to the name of the MAST template that the symbol represents. The value of the MAST template property is used to set the template name for the symbol. If this property does not exist in the Saber sub-section of the CDF, then the template name will be determined as before. componentName partially replaces mapping file field number 2.

- Example: componentName r
- Example: componentName q

termOrder

termOrder is a list of terminals that define the net order. The nets are considered to be those nets connecting the terminals in the order given in this list. The value of this property should be a space-separated list of symbol terminal names in the same order as the connection point names on the corresponding template. If this property is used, the termMapping property should not be used. termOrder partially replaces mapping file field number 9.

Programmable nodes may also be used to define the net order. To do this, a property is defined in the CDF for the terminal and given a value of one of the following

- A terminal name on the same cell. The programmable terminal will be connected to the same net as the terminal that is named as the property value.
- The net name the terminal is to be connected to.

If the specified property does not exist on the cell, then the following error message will be displayed in the Simulation Window

```
Error, there is no property or terminal named
“property_name” on instance “instance_name” as specified
by the termOrder property “value_of_termOrder_property”
```

Reading of the schematic database continues so that any other errors may be found. No netlist is written.

- Example:

  termOrder (PLUS MINUS)
- Example (programmable node set to a net; bn=gnda!):

  termOrder (C B E progn(bn))
- Example (programmable node set to terminal C on the symbol; bn=C):

  termOrder (C B E progn(bn))

termMapping

termMapping is used when the MAST template has different names for its connection points than the symbol has for its terminals, and the termOrder property is not used. The value of this property is a space-separated list using the following format:

```
nil SymbolTermName1 MASTTemplateTermName1 SymbolTermName2
MASTTemplateTermName2 etc.
```

In this example, SymbolTermName# is a name of a terminal on the symbol, and MASTTemplateTermName# is a name of the corresponding connection point in the MAST template. If this property is used, the termOrder property should not be used. Do not use this property if programmable nodes are needed. termMapping partially replaces mapping file field number 6.

- Example: termMapping (nil PLUS p MINUS m)
- Example: termMapping (nil C c B b E e)

instParameters

instParameters is a list of parameters that are to be included with this cell in the netlist. The value of this property should be a space separated list of symbol parameters to be written into the netlist for the cell. If a symbol parameter name does not match the template parameter name, then the propMapping property must also be used (see below for an explanation of the propMapping property). instParameters partially replaces mapping file field number 4. Do not mix the use of instParameters and mapping.

- Example: instParameters (r)
- Example: instParameters (model length width)

propMapping

propMapping allows for entering different CDF parameter names in place of the names recognized by Analog Artist. The value of this property must use the following format:

```
nil MASTTemplateParamName1 CDFParamName1
MASTTemplateParamName2 CDFParamName2 etc.
```

In this example, MASTTemplateParamName# is a name of a parameter for the template, and CDFParamName# is the corresponding name of the CDF parameter on the symbol. propMapping partially replaces mapping file field number 4.

- Example: propMapping (nil rnom r)
- Example: propMapping (nil l length w width)

Pre-Defined Properties Not Recognized by Catos Netlisters

The following pre-defined properties are not recognized by catos netlisters:

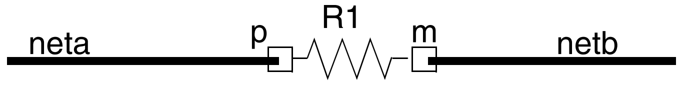
- otherParameters
- macroArguments
- deviceTerminals
- permuteRule
- namePrefix
- current
- netlistProcedure
- modelArguments

Specific Examples Using simInfo

The following examples show specific instances that use the simInfo->Saber properties.

Example 1: Simple resistor

Symbol: resistor



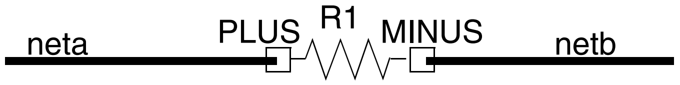
Instance Properties:
rnom = 10K

simInfo->Saber Properties
componentName = r
instParameters = (rnom)

Netlist Entry
r.R1 p:neta m:netb = rnom=10K

Example 2: Resistor, parameter mapping and term ordering

Symbol: resistor



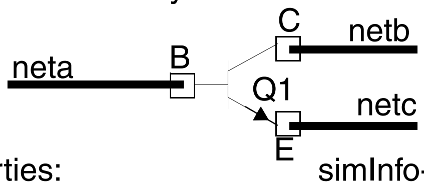
Instance Properties:
res = 10K

Netlist Entry
r.R1 neta netb = rnom=10K

simInfo->Saber Properties
componentName = r
termOrder = (PLUS MINUS)
instParameters = (res area)
propMapping = (nil rnom res)

Example 3: Simple transistor with terminal mapping

Symbol: transistor

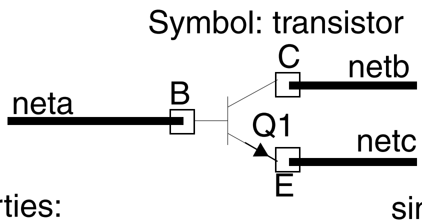


Instance Properties:
model = my_model1

Netlist Entry
q_3p.Q1 e:netc b:neta e:netb = model=my_model1

simInfo->Saber Properties
componentName = q_3p
termMapping = (nil e E b B c C)
instParameters = (model)

Example 4: Transistor with terminal ordering

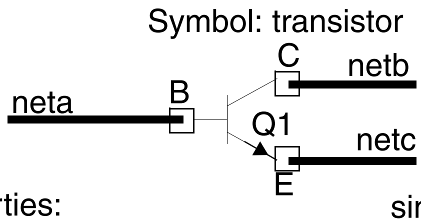


Instance Properties:
model = my_model1
area = 10m

simInfo->Saber Properties
componentName = q_3p
termOrder = (E B C)
instParameters = (model area)

Netlist Entry
q_3p.Q1 netc neta netb = model=my_model1, area=10m

Example 5: Transistor with programmable terminals and property mapping

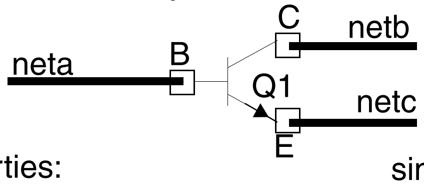


Instance Properties:
model = my_model1
a = 10m
bn = E

simInfo->Saber Properties
componentName = q
termOrder = (E B C prog(bn))
instParameters = (model a)
propMapping = (nil area a)

Netlist Entry
q.Q1 netc neta netb netc
= model=my_model1, area=10m

Example 6: Transistor with terminal/programmable terminals and property mapping



Symbol: transistor

Instance Properties:
model = my_model1
bn = netd
a = 10m

Netlist Entry
```
q.Q1 netc neta netb netd
      = model=my_model1, area=10m
```

simInfo->Saber Properties
componentName = q
termOrder = (E B C prog(bn))
instParameters = (model a)
propMapping = (nil area a)

15

The Template Information System

The Template Information System generates files containing information that can be derived from a template's source code without performing simulations. These files, called template information files, have an .ai_tdb extension and reside in the same directory as the template file. The Template Information System creates one template information file for each user-created MAST template file having an .sin extension.

The topics discussed in detail in this section are:

Template Information System Updates
Manually Creating Template Information Files
Editing Template Files with the Text Editor

Template Information System Updates

The Template Information System normally updates automatically, but in some cases the system must be updated manually. The following sections describe how the differences between automatic and manual updating.

Automatic Update

In most cases, the Template Information System automatically updates or creates template information files without requiring input from the user. These updates occur

- if the .ai_tdb file for a user-created template does not exist.
- if the .ai_tdb file exists but its time stamp is earlier than the time stamp of the associated user-created template file.
- whenever the tools making use of this information—such as the netlister, Saber Sketch, Monte Carlo, or Vary—need to access a template's .ai_tdb file.

If you move templates between directories, you can delete their associated .ai_tdb files in order to force the Template Information System to generate new .ai_tdb files to make sure that they are up-to-date.

Manual Update

The only cases when you need to manually employ the Template Information System to update .ai_tdb files are

- if you are the site manager with read and write permissions for the user-created template, and you have been requested, by a user without the proper permissions, to perform an update.
- when you would like to update more than one template information file at a time.
- when you would like to use the Template Information System as a MAST syntax checker on user-created templates prior to netlisting circuits using these templates.

Manually Creating Template Information Files

If you would like to manually create a template information file for a template you have the option of using a script from a command line or by using menu selections from a Saber application.

Using the command line script

A command-line script has been added to make it easy to update the template database files without invoking an interactive application. The usage of the script is:

```
ai_mk_tdb [-help] [-gui] [-sdp] file/dir...
```

For more information on this command invoke:

```
ai_mk_tdb -help
```

Using the menu selections

To use menu selections to create template information files, perform these steps:

1. From either Saber Sketch or Saber Guide, select the Edit>Update Template Information... menu choice in the pulldown menu bar to invoke the Generate Template Information dialog box.

   From the Frameway for Viewlogic, the Frameway for Mentor Graphics, or the Frameway for Cadence, select the Saber > Netlist > Update Template Information menu choice in the pulldown menu bar to invoke the Generate Template Information dialog box.
2. Double click on the + next to the directory listed in the Process listbox.

   Templates in the SABER_DATA_PATH are listed below their directory location.
3. Determine which templates information files you want to update.
   - Update template information files by leaving them in the Process listbox.

     For this operation to be successful, you must be able to write to the files.
   - To leave template information files unmodified, click the <-- button to place them into the Don’t Process listbox.
4. Click OK.

   While the template information files are being updated, the Generating Template Information dialog box gives status on the update.

   If the update succeeds, you are ready to proceed with your analysis.

If any templates have MAST syntax errors in them, a list of these templates will be included in the Generating Template Information dialog box. To edit these templates

a. Click on the template in the Errors listbox that you want to edit.
b. Click Fix to invoke the text editor.
c. Editing Template Files with the Text Editor gives instructions on editing these template files.

Editing Template Files with the Text Editor

When you attempt to generate a template information file, either from the menu pulldown bar or by invoking the Saber Sketch Property Editor on a template's symbol, Saber's MAST parser is run on the template in order to detect syntax errors.

If there are errors, you may edit the template file using the Template Information System text editor.

The text editor contains two main windows:

- The Transcript window displays a record of the transcript of the attempt to compile the template.
- The Editor window displays the editable text of the template.

Errors in the MAST syntax of a template file do not necessarily indicate that the template will cause errors if included in a netlist. For example, the MAST parser might return errors if the file does not contain a complete template, or if your SABER_DATA_PATH is set up incorrectly.

Glossary

AHDL
Analog Hardware Description Language. A term for a modeling language that is capable of representing both the structural and behavioral properties of a design. Structural refers to design connectivity (netlist information). Behavioral refers to the mathematical equations that underlie each model. An AHDL (such as MAST) commonly features hierarchical circuit descriptions and parameter passing. Advanced implementations include modeling mechanisms that process analog and digital signals, as well as nonelectrical phenomena.

Analog Model
An analog model is one in which all variables change in a piece-wise continuous manner.

Argument
An Argument is a coefficient of a model equation, usually provided as a template variable. Argument is often used interchangeably with parameter. Arguments are limited to numeric values. See parameter.

Attribute
Attribute is generally interchangeable with property. Some schematic capture vendors refer to symbol variables as attributes, but most vendors refer to them as properties. This document refers to them as properties. See property.

Characterization
Characterization is the process of converting a template (general model) into a component (specific model). A component usually corresponds to a specific physical part. Characterization is accomplished by giving values to the parameters of the template, usually including those with default values.

Circuit
A circuit is a group of models that connect to each other and function together to provide a known output for a given input. Somewhat synonymous with design, circuit generally (though not necessarily) refers more to those with exclusively electrical models. See design.

Component
1) A component is a model that has been characterized to represent a more specific system element. A component usually corresponds to a commercially-available part. A component usually passes appropriate parameter values to a more general template. 2) A template instance in a netlist is also referred to as a component. See template.

Component Library
The component library is a collection of component templates that may be purchased as a simulator option.

Connection Point
The connection point is a channel for dynamic communication between a model and its environment. In a MAST template, it is a pin, a state, a var, or a ref.

Design (noun)
A design is a group of models that connect to each other and function together to provide a known output for a given input. Somewhat synonymous with circuit, design is the more general term. See circuit.

Digital Model
A digital model is described by algorithms that execute asynchronously.

Digital Simulation
In digital simulation, the models respond to scheduled events at their inputs, and their outputs assume only a finite number of values from a predefined set of discrete states. These discrete states are usually referred to as logic levels.

Element (general)
Element is a very generic term for any symbol or model in a system, or for a block that requires a symbol or model.

Encrypted Template
An encrypted template is one in which some of the text has been transformed into random characters so that the contents are not human-readable. This transformation is done for one or more of the following reasons:

- To allow separate authorization of optional products
- To ensure library integrity
- To honor third-party agreements to protect proprietary information

External Declaration
An external declaration is the declaration of a reference to a variable or pin that has been declared in an instance closer to the root of the current system model hierarchy.

HDL
Hardware Description Language. See AHDL.

Hypermodel Template
A hypermodel template describes the interface between connection points of different kinds. For example, between a digital pin and an analog pin. Instances of Hypermodel templates do not represent physical parts in the design.

Include File
A file whose content replaces the calling instruction (include instruction) in the code of another file.

Input
Input is a synonym for a ref declared in the template header that is directly wired to a var.

Instance
See symbol instance or template instance.

Interface Template
A special kind of template that acts as a “translator” when connected between models that have different types of connection points (for example, electrical and mechanical). One interface template is required for each connection point (pin) tied to a connection point of another signal type. A hypermodel template is a special kind of interface template that is restricted to the electrical domain. See hypermodel.

Mapping
Mapping is the process of associating a symbol with a model (component or template). There are three mapping techniques: name matching, specially recognized properties, and mapping files.

Mapping File
A mapping file is a text file containing information that tells the netlister how to map (correlate) a symbol to its model. Using a mapping file is just one way (of three) to map a symbol.

MAST
MAST is the name of the modeling language. It is one implementation of an AHDL. MAST is capable of representing both the structural and behavioral properties of a design. Structural refers to design connectivity (netlist

information). Behavioral refers to the mathematical equations that underlie each model. MAST features hierarchical circuit descriptions, parameter passing, and modeling mechanisms that process analog and digital signals, as well as nonelectrical phenomena. See AHDL.

Model
Model is a loosely defined term. It is predominantly used in the colloquial sense to mean some representation (mathematical equations) that approximates the behavior of a real or imagined system element. Somewhat interchangeable with template, model is the more general term. Model can also mean 1) either a template or a component; 2) simply a list of parameters that, when applied to a particular template, turn it into a component for a particular part (a parameter list is often the type of model that is supplied by some of the parts manufacturers).

Model Definition
A model definition is a set of parameters, under one name, used to characterize a template so that it performs as a component.

Model File
A model file is a text file containing one or more model definitions.

Net
The point (in a schematic) where two or more symbols connect (by wires). Net and node are sometimes used interchangeably. See Node.

Netlist
A netlist is a text file (.sin extension) that is an input to the simulator. A netlist is a description of a design that lists each element of the design, its values, and its interconnections with other design elements. A netlist can be created by hand but is more typically the output of a program called a netlister, which combines information from a schematic, the models, and a mapping file (if used).

Netlist Entry (netlist statement)
See template instance.

Node
The point (in a netlist) where two or more models connect. Node and net are sometimes used interchangeably. See Net.

Parameter
A parameter is a variable that is part of a model (template) and, therefore, needed for simulation. A parameter is generally (but not necessarily) a coefficient of a model equation. Parameter is often used interchangeably with argument. Parameters are not limited to numeric values. See argument.

Part
1) A part consists of all the information needed to describe a system element. This information includes the symbol, the model (either template or component), and any underlying parameters. 2) A part sometimes refers to the physical device.

Pin
1) A pin is the name for an interface of a template to a netlist. 2) A generic term for any connection point. See port.

Port
An input or output connection point of a model or symbol, but more often used to refer to symbol connections. See pin.

Property
A property is a type of variable that is part of a symbol. Properties are used to characterize the symbol, often for programs outside of the graphic editor. Properties of symbols may be passed-on to the parameters of a model. See attribute.

ref
A ref (also called reference connection) is a system variable that is a type of connection point between templates. The connection point (ref) in one template reads the value of a var in another template without the need of a wire connection (as in a dependent source).

Reference Connection
See ref.

Symbol
A symbol is the graphic object used in a schematic capture tool to represent a system element (part). It is used only to define the connections to the other system elements. It does not model the behavior of a part. However, a symbol can pass properties to a template. A symbol can exist as a stored file (sometimes called base symbol) or as one or more instances in a schematic. See symbol instance.

Symbol Instance
A symbol instance is the occurrence (instance) of a symbol in a schematic. The instance will include the name of the symbol as well as a reference designator (also called instance designator) that distinguishes it from other instances of the same symbol in that schematic. See symbol.

Template
A template is a text file that contains the model description, written in the MAST language, for use in simulation. A template models a general class of parts. Parameters must be supplied to model a specific part. A template can exist as a stored file (sometimes called base template) or as one or more instances in a netlist. See template instance.

Template Instance
A template instance is a reference to a template in a netlist. Each instance has the form template_name.reference_designator connections = arguments, where template_name is the name of the template, reference_designator is a name that distinguishes this instance from all other instances of the same template in the netlist, connections is a list of pins, states, and system variables at the connection points of the template, and arguments is a list of the arguments for this instance. See template.

var
A var is a system variable that must have a template equation describing how the simulator is to solve for the var. See ref.

Index