MAST Language Reference Manual

Release 2003.06, June 2003

Comments?
E-mail your comments about Synopsys documentation to doc@synopsys.com

SYNOPSYS®

Copyright Notice and Proprietary Information

Right to Copy Documentation

Destination Control Statement

Disclaimer

Registered Trademarks, Trademarks, and Service Marks of Synopsys, Inc.

Registered Trademarks (®)

Trademarks (™)

MAST Language Reference Manual

chapter 1

Overview

Introduction

This manual describes the MAST modeling language. This language lets you create a model of any analog system or element that can be defined in terms of nonlinear “lumped” algebraic or differential equations. Some extensions are also provided by the use of ideal delay and scheduling.

After completing a model, you can use it as input to the Saber simulator. You can also use the MAST language to create digital models that take on discrete values at discrete times. With the MAST language and the Saber simulator, you can model and simulate most physical systems: electronic, mechanical, optical, hydraulic, etc. (or any combination of them).

The MAST language is a unique concept in simulator input—the same language describes models of elements, of subsystems, and of full systems. A system model can be as simple as a netlist that describes the interconnections of existing system components (using pre-defined models from a MAST library) or as complex as the full system description (using no pre-defined models).

At minimum, the Saber simulator requires (for each system to be simulated) an input file containing a netlist that completely describes the system in the MAST language.

NOTE

The simulator also accepts files containing SPICE input, after they have been converted to MAST format with the `spitos` *conversion utility. However, the Saber simulator is inherently compatible with the MAST language, so using MAST models provides both greater modeling flexibility and better simulation speed.*

The MAST language enables you to do the following:

- Add new models as needed
- Combine technologies (electrical, optical, mechanical, etc.) without the need for translation into electrical
- Alter models with ease
- Describe functions inside models
- Determine output responses as functions of model parameters and simulated variables
- Define systems hierarchically
- Describe complicated relationships between components
- Use models within other models
- Pass information from one model to another
- Model analog systems at any of three levels: behavioral, functional, and primitive

The principal unit of modeling used by the Saber simulator is called a template—the MAST description of the model. Depending on the model, MAST templates can vary in appearance, length, and complexity; however, they do share common features. These are covered in the remaining sections of this chapter.

The rest of this manual provides reference information on templates and on the characteristics of the MAST language itself. The *Guides To Writing MAST Templates, Book 1 and Book 2*, serve as companion documents, illustrating basic MAST functionality by way of writing templates to model common electrical devices (resistors, BJTs, voltage sources, AND gates, etc.).

Deprecated MAST Features

As part of formalizing the definition of the MAST language several language features that were undocumented for a long time have now been marked as deprecated. These features include:

1. the states section

   The states section has been replaced by the much more flexible apparatus of When statements

2. external declarations in the template body

External declarations should be in the template header

3. val and state declarations in functions

   Such declarations should be replaced by appropriate variable declarations of type number

The simulator will issue a warning for each use of a deprecated language feature if it is started with the -d deprec8 option. Deprecated language features will not be supported in the VeriasHDL Simulator but remain to be available in the Saber Simulator.

MAST Modeling Language

File for C language declarations included

The file `saberApi.h` in the *install_home*/`include` directory contains the declarations for the published C language interface of the Saber simulator. (15323)

A compound statement in a template cannot have an empty body

A compound statement in a template cannot have an empty body. For example, the following is not allowed:

```
values {
# comment
}
```

There is no workaround for this problem. (8980)

A foreign routine in a template that does not return a value can cause errors

If you include a foreign routine in a template and it does not return a value, it can cause excessive simulation time or other unpredictable results with no error reported.

The workaround is to make sure any foreign routine returns a value even if it is a “dummy” value. (10118)

You cannot use an enumerated type parameter (`enum`) as an external variable in a template

The workaround is to use an argdef (..) operator to pass the `enum` in from another template. (10272)

Templates and Hierarchy

A complete MAST description of an element, subsystem, or entire system is contained in a file and is called a template.

The MAST language supports designs, which means that templates can contain references to other templates. If one template (say template A) contains a reference to another (template B), this indicates that, in the model, the system represented by template B is a subsystem of that represented by template A. You can create a template that defines a subsystem, and then refer to it in the system template wherever the subsystem is used. When you take full advantage of hierarchical modeling, the most natural structure of the model is the structure of the system it models.

A reference in one template to another template is a netlist entry. Using netlist entries to define a system hierarchically can significantly increase the speed of simulation.

The MAST language places no restrictions on the depth of the template hierarchy. Moreover, any level may contain any number of references to lower levels, and called templates can be defined either inside or outside of the calling template.

Because we supply substantial libraries of standard component and element templates, you can usually simplify your model-writing effort by including references, whenever possible, to these templates. In many cases, you can define an entire system using only netlist entries that call library templates. Such a system is called a netlist.

A netlist entry is equivalent to the full definition of the referenced template. For example, a netlist reference to a predefined resistor template is equivalent to a full definition of that resistor. Moreover, the netlist reference would specify the value of the resistor (in ohms), and that value would be passed to the resistor template.

If a template consists only of a full model description instead of referencing other templates, it is called flat. Flat descriptions have two disadvantages:

- In most cases, simulation requires more time.
- If there are multiple occurrences of a subsystem, its description must be written in full for each occurrence, reducing simulation efficiency and increasing the size of the input file.

Naming the Template File

You can use your system's editor to create a template file in the MAST language. Give it a name of this form:

templatename.sin

where *templatename* is the name of your template, which can model a system, subsystem, or component. The number of characters allowed in *templatename* depends upon your operating system. The .sin extension is required for use by the Saber simulator. The *templatename* must start with a letter. The other characters in the name may be letters, numbers, or underscores. If you have an input file previously created in the SPICE format, you can convert it to Saber format with the Spice-to-MAST Translation tool nspitos. For a description of the nspitos tool, refer to nspitos in the *SaberBook Online Help System*.

The top-level template in a hierarchical system model contains other templates or references to them, and is not referred to by other templates. When invoking the Saber simulator, you call the top-level template directly, as follows, where brackets (*[]*) denote optional items:

saber *[options]* *templatename[*.sin*]*

Template Organization

This section presents an overview of template organization.

Templates have a general form consisting of several different sections. You use some or all of the possible sections depending on the requirements of your model and whether you include previously defined templates in your model.

Within your template, at any point, you can specify other files, the contents of which are read into the template. Included files are called "include files." Include files can contain part, or even all, of a template. The possible template sections are as follows:

```
  Unit definitions
  Connection point definitions
Template header
  Header declarations
{
  Local declarations
  Parameters section
  Netlist section
  When statements
  Values section
```

```
  Control section
  Equations section
}
```

In general, the more complicated the model, the more template sections you will probably use.

In a hierarchical system model, the top-level template models the whole system and can contain templates that define the subsystem and components. Templates within the top level template can either be explicitly defined within the template file, or can be defined in separate files and included by reference.

The top level template must not contain a template header, the header declarations section, or the braces (`{ }`) surrounding the body of the template. When you invoke the Saber simulator to analyze a system model, you use the name of the file containing the top level template in the Saber invocation line.

When writing templates, you should bear in mind two important rules that affect the positions of related items.

1. You must always define something before you use it. For example, if the template uses a variable, that variable must be declared prior to its first occurrence. The exception is the use of implicitly declared `pins` and some forward references to component's `vars`. Simvars and intrinsic functions are also implicitly declared.
2. Where you define something determines whether it is defined locally or globally. In particular, if you include or define a template before a template header, it is accessible by all templates below that one in the template hierarchy. This is generally true for other declarations. However, if you define it after the header declarations section (for example, in the local declarations section) then it is visible only in the local scope. "Local scope means that the particular definition is accessible only to that template and possibly to its descendants. (Other templates can include or define the same thing independently.)

As long as you comply with these rules, template sections do not have to be in a particular order.

Unit and Connection Point Definitions

The~unit definition specifies the units for certain variables such as through variables, across variables, `vars`, `refs`, `states`, and `vals`. Unit definitions set the identifier used to indicate the units of a number that will result from a calculation.

The connection point definition specifies the connection points a template can use. If they are pin-type, it implies the names of the through and across variables, two important variables associated with all pins.

The connection point definition is important because it tells the Saber simulator which variable to solve for and which must be equal to zero at a connection point. In addition, it allows the simulator to make sure that only compatible components are connected.

Because the connection point definition uses units, it must follow the unit definition in the template.

Standard unit and connection point definitions appear in `units.sin`, which is included in the file header.sin. If you invoke the Saber simulator with the `-la` option, header.sin is automatically loaded. If you wish, you may change units.sin and re-compile it using the `saber -p` option.

Header

The header for a template file defines the name of the template, the names of its connection points, and the names of the template's arguments used in a netlist entry. You must include the header in any template that you will call from another template; that is, any template except a top level template.

The header includes the name of the template, the names of the connection points, and any arguments associated with it. Whenever you define a particular element by referring to it in terms of the template that defines it (a netlist statement), the form and content of the header statement defines the form of the netlist statement.

Declarations

There are two sections that contain declarations: header declarations and local declarations. All names (identifiers) must be defined before they can be used. These definitions are called declarations. Keywords (names that are required part of MAST statements) are defined by the language and require no further declaration. A declaration tells the system the type to be associated with the name, thus defining how it is to be used. Some declarations can also include the assignment of an initial value.

The purpose of header declarations is to define to the system the names used in the header. The local declarations define the names used in the rest of the template—it contains declarations for all identifiers used inside the template.

The name of the template does not require a separate declaration; in fact, the header itself is the declaration of the template name. The other names use in the header (names of connection points and arguments) must be declared in the header declarations.

Declarations of connection points must define their type (`pin`, `ref`, `var`, or `state`). Declarations of arguments define the type of each argument. Argument types fall into one of three categories: simple, composite, or arrays of simple/composite.

Understanding declarations is a key to understanding how to use the MAST language and is explained in more detail in the chapter on Declarations and Data Structures.

The Parameters Section

The Parameters section is used to manipulate parameters. You can use it to add error checking to templates by testing the input values of arguments for validity and to model statistical distributions for Monte Carlo analyses.

Parameters and arguments are similar. They can have the same types of declarations, but parameters are declared locally in a template, while arguments are declared in the header declarations section and their values can be passed into a template by using a netlist statement. However, only arguments and initialized parameters can be changed by the `alter` command in the Saber simulator. Only parameters may be changed in the Parameters section.

The statements in the Parameters section are evaluated as follows:

- Once just after the input file is read
- Each time the Saber `alter` command is used (causing a temporary alteration of a template argument)
- For each run of a Monte Carlo simulation

Parameters may depend only upon other parameters, arguments and constants.

Assignment statements, expressions, and conditional statements (`if-else`) are allowed in this section. Mathematical expressions and intrinsic functions are allowed in this section with the exception of `d_by_dt(`*x*`)` (the derivative function) and the `delay` function. Calls to foreign subroutines are allowed.

The Netlist Section

The Netlist section consists of one ore more netlist statements that call other templates. These statements define elements of the system that are instances of the template(s) being called. The Netlist section is required only in templates that make reference to other templates; in fact, it is an implementation of another level of hierarchy.

The form of a netlist statement is as follows:

templatename`.`*refdes connection_pt_list* [`=` *argument_assignments*]

The *templatename* is the name of the template you are calling, as identified in its header.

The *refdes* is the reference designator, a unique name for that element of the system.

The *connection_pt_list* is a list of the nodes in the system to which the connection points of the template are connected. There is a direct correspondence between the number of connection points and the number of nodes. A template for an element such as a resistor, with two connection points, must be connected to two nodes of the system.

The *argument_assignment* is the assignment of values to the arguments of the template.

When Statements

When statements make it possible to construct state machines, which perform certain actions depending upon preceding system states, or upon the values of digital gates.

Conditional statements are allowed in this section, as well as mathematical expressions and intrinsic functions, except for the `d_by_dt(`*x*`)`, `delay`, and `random()` intrinsic functions. Foreign subroutines are also allowed. When statements are evaluated by the Saber simulator as needed.

When statements are used in discrete time simulation. You can use them in describing digital behavior, in testing for analog waveforms crossing a threshold, and in scheduling.

Values Section

The Values section of the template is used to define variables that are to be extracted during post-processing. It also can be used to transform variables into a form needed in the Equations section, including the use of foreign subroutines. The Values section is helpful in clarifying the template and making it more maintainable.

For example, you may wish to define a `val` that will allow you to extract the power (for example, `voltage•current`) of a resistor when you are analyzing the results of the simulation. To the Values section of your resistor template, you would add the following statement:

```
power_res = v_res * i_res
```

If `power_res` is not needed for solution of the system matrix, it is evaluated but not used during the simulation. The simulator places it in the `pfile` only if it is specified in the signal list (using the `siglist` variable for the analysis). After the simulation, you can use the simulator's `extract` command to add it to the `pfile`. This feature allows you to add any simulation values that will be useful during extraction without affecting the simulation speed.

Conditional statements are allowed in this section, as well as mathematical expressions and intrinsic functions, except for the `d_by_dt`, `delay` and `random()` intrinsic functions. Foreign subroutines are also allowed.

Control Section

The Control section declares specific information to the simulator that does not fit in other sections of the template—it is not required in all models. The information that the Control section provides is specific to the system being analyzed.

The Control section can contain the following types of statements:

- Conditional statements that can collapse two nodes into a single node, thereby speeding up the simulation. However, once nodes are collapsed during a simulation, you cannot undo it without exiting and re-entering the Saber simulator.
- Statements that declare groupings of nonlinear values for the purpose of piecewise linear evaluation, defining the independent variables for each group
- Statements that declare the sample points for each independent variable used in the piece-wise linear set
- Statements that declare Newton steps for the specified independent variables
- Statements that describe small-signal noise sources

Each of these statements is described in the *Templates* section.

Equations Section

The Equations section describes the analog characteristics at the terminals of the element the template is defining. In effect, this section defines the effect of the element on the rest of the system.

Statements in the Equations section indicate the relationship of the analog characteristics of special variables to variables in the rest of the system. These statements use the special operators `+=` (is added to) and `-=` (is subtracted from) to indicate this relationship. Mathematical expressions and intrinsic functions are allowed in this section. The intrinsic functions `d_by_dt` and `delay` are allowed as well, but may not be nested. The `random()` function is not allowed.

Miscellaneous

Include Files

There are two ways to “bring” the contents of other files into a template. One way is with the netlist statement, which refers to a template and defines the connections of its terminals to nodes of the system. The other way is with the include statement, which has the following form:

<*includefilename*

where, as shown, the character < must be in the first column. The *includefilename* (the name of the include file) may begin in any column.

When the simulator finds an include statement in a template, it replaces the statement with the complete contents of the file named in the statement. Files to be included can contain any information that is part of a template, up to and including multiple complete template definitions. The Saber simulator reads in the contents of include files directly, so you must locate the include statement precisely where the information is required.

When the Saber simulator encounters a file reference in a template—either in a netlist statement or in an include statement—it automatically searches for that file in a list of directories. This list is defined at the operating system level in the SABER_DATA_PATH environment variable, as described in the Saber installation instructions. Otherwise, you must specify the full path name of the include file. For example, you may wish to include the file consts.sin, which contains a number of constants useful for mathematical calculations, by putting the following statement in your template:

```
<consts.sin
```

Auto-Inclusion

Saber will automatically search for template references along the SABER_DATA_PATH in files whose names have the .sin extension. Therefore, templates referenced in netlist sections do not have to be formally included if they are named *templatename*.sin. Templates automatically included in this way (auto-inclusion) will be known globally (as if they were included in the top-level template), even if they’re referenced within the local section of a template.

Pre-compiled Templates

The Saber command defaults to a `-la` option, which loads analogy.sld at the top level of a system. This “saber load” file contains header.sin, which has standard unit and pin definitions included from units.sin, so these definitions are usually included automatically in a template.

Template Variables

Several different types of variables can be used in a template. Each type has its own function and uses. The types of variables are listed below. Understanding the concepts behind these variables is essential for anything but the most rudimentary template usage.

- System variables
  - through variables
  - across variables
  - `vars`
  - `refs`
- Simulator variables
- Parameters
- Arguments
- Values (`vals`)
- `states`

The task of a simulator is to describe the behavior of a mathematical model of a physical system. For the Saber simulator, the model is a system of simultaneous (linear or nonlinear) algebraic and ordinary differential equations, or both.

The purpose of the MAST language is to specify the model of the system to be simulated. The language must be able to do the following:

1. Allow arguments to be passed into templates.
2. Specify how the template is connected to the rest of the system. More specifically, this means specifying the interactions of elements in terms of the equations that arise from physical laws when entities are connected (at nodes) to form a system. In electrical systems, these laws are Kirchoff’s current and voltage laws (KCL and KVL). These two laws may be stated, respectively, as “the sum of the currents leaving a node is zero” and “the sum of voltage drops across elements in any loop is zero”.
3. Specify the equations that describe the analog behavior of the system. For example, the equations describing the analog behavior of a resistor are `v=i*r` and those for a capacitor are `i=c*dv/dt`, where, in both cases, `v` and `i` are the voltage across and the current through the element, respectively.

4. Specify the discrete behavior of the system. You can use When statements to tell the simulator how to schedule discrete events and time steps.
5. Allow flexibility in specifying system behavior.
6. Keep the simulation as fast as possible by separating variables into the following:
    a. Those that must be evaluated continuously
    b. Those that must be evaluated on an event-driven basis
    c. Those that can be evaluated only as needed or for post-processing
    d. Those that have to be evaluated only once for each simulation

To fulfill these various functions, the MAST language offers these different types of variables, each of which is described in the following section.

System Variables

System variables are the variables for which the simulator solves. They include across variables, `var`s, and `ref`s. You cannot assign values to these variables in a template. Only the simulator can assign their values.

To generalize KCL and KVL so they apply to both electrical and non-electrical systems, two associated quantities must be defined for each connection point: the *through* and the *across* variables. The rule for through variables is as follows:

> "The sum of all through variables leaving a node is zero."

The rule for across variables is as follows:

> "The sum of all across variable differences around any closed path is zero".

Across and through variables for various technologies are as follows:

	Through	**Across**
Electrical	current	voltage
Rotational	torque	angular velocity
Mechanical	force	position
Fluid	flow rate	pressure
Thermal	power	temperature

Modified Nodal Analysis

The supplied templates usually use a modified nodal analysis technique. In nodal analysis, through variables are added to and subtracted from the system matrix directly, and then the simulator solves for across variables. Therefore, through variables are dependent variables and across variables are independent variables. In many cases, however, nodal analysis must be modified. For example, modification is required where across variables are not functions of through variables, such as in an ideal voltage source, in which the current is whatever it needs to be to give the defined voltage. In such a case the through variable, current, is an independent variable, which is handled by adding an equation to the system of equations.

In general, most through variables are dependent variables and all across variables are independent variables. These variables are declared implicitly through the pin declarations. For example, when you define a connection point `x` to be electrical, the simulator automatically creates an independent variable of the form `v(x)` and the dependent variable `i(x)`.

An independent through variable, such as the current through a voltage source, must be declared as a `var` in the local declarations section. It must be associated with an equation in the Equations section of the form:

`var:` *expression* `=` *expression*

A `var` is not declared in the form `i(x)` (for example, as a current at a pin), because that form would denote a dependent variable. You can also use a `var` for situations that arise less frequently. In cases where you need to take multiple derivatives and delays (which cannot be nested), you can declare a `var`.

You can also define a `var` as an independent variable that can be passed another template, in which it is declared a `ref`.

In general, the dependent variable in a template is the through variable defined in relation to a connection point. The independent variables in a template are across variables defined in relation to a connection point, `var`s that are associated with an additional equation in the Equations section, and occasionally as `ref`s.

Dependent through variables and independent across variables are declared implicitly by the `pin` declarations. They are associated with equations in the Equations section using the following formats:

```
through(pin) += expression
through(pin) -= expression
through(pin1->pin2) += expression
```

The form `through(`*pin1*`->`*pin2*`) -=` *expression* is also possible, but is less readable and is not recommended.

A `var` must be declared in the local declarations section and must be associated with an equation in the Equations section, where the equation has the form `var: expression = expression`. A `ref` must be identified, along with other connection points, in the template header, and must be declared in the header declaration section. For examples, refer to the *Templates* section.

Simulator Variables

Simulator variables (simvars) are variables used by the simulation. They are not directly part of the system of equations that describe the model, but may be used in various ways throughout the template. They are used most frequently in conditional statements (`if-else`). Simvars include the following:

dc_domain	freq	next_time	time_domain
dc_done	freq_domain	statistical	time_init
dc_init	freq_mag	step_size	time_step_done
dc_start	freq_phase	time	tr_done
			tr_start

The simulator assigns values to most simvars according to its own rules and requirements. Exceptions are `next_time` and `step_size`, which give information to the simulator. A template need not provide values to these simvars. However, if a template does provide a value, the simulator uses the value to influence its choice of the next time step or time-step size in the simulation.

The simulator automatically declares simvars, so you need not declare them. However, if you declare a variable with the same name as a simvar (not recommended), then your declaration overrides the automatic one, so you cannot use that variable as a simvar.

You can use simvars in When statements and in the Values section, and in certain circumstances, in the Equations section. The statistical simvar can be used in the Parameters section and in the Netlist section.

Arguments

Arguments are named in the template header, and values for them are passed into a template via a netlist entry. You declare each argument in the header declarations section–either as one of the three simple types, as one of three composite types, or as an array of simple and/or composite types.

Arguments can receive their values in two ways:

1. Being initialized in the declaration
2. Having values passed in through the argument list in a netlist entry

A value passed in through a netlist entry supersedes any initialized value. You can change the value of an argument during a simulation run using the `alter` command of the simulator.

Parameters

Parameters are variables used in expressions that assign values to other types of variables. Parameters can be declared to be any of three simple or three composite types, or arrays or combinations thereof. Parameters and arguments have the same types of declarations. You declare parameters in the local declarations section.

A parameter can receive its value in two ways. You can either initialize it in the declaration or you can assign it a value in the Parameters section, by either an assignment statement or a foreign subroutine call. You can change this initial value while running the Saber simulator, using the `alter` command.

Within the Parameters section, only parameters can appear on the left hand side of an assignment statement, or be returned from a foreign subroutine call. You can use parameters to initialize states. Parameters can be used in all other sections in the body of a template: the Parameters section, Netlist section, When statements, the Values section, the Control section, and the Equations section.

Vals

`Vals` are variables that hold temporary information during simulation, and which can supply information not otherwise available during post-simulation processing.

`Vals` can receive their values only in the Values section of the template: either in an assignment statement (with the `val` on the left-hand side of the statement) or as the return value of a foreign subroutine call. You must declare `vals` in the local declarations area of the template. You can use them in the Values section, the Equations section, and in When statements.

States

`States` are variables that hold information pertinent to discrete time simulation. `State` variables may be “digital” (discrete in values and discrete in time), or “event-driven analog” (continuous in values and discrete in time).

`States` can receive their values in two ways:

1. Use a `state` as a connection point in the header of a template, then pass the value in from a netlist.
2. Assign the `state` a value in a When statement, where the state can be the left-hand term of an assignment statement, can be used in various scheduling statements, or can have its value returned by foreign subroutine calls.

If a `state` is passed in as a connection point, it must be declared in the header declaration section; otherwise, it must be declared in the local declaration section. You can use `states` in When statements and in the Values and Equations sections.

chapter 2

MAST Syntax Rules

The general syntax rules and reserved words for the MAST modeling language are divided into the following topics:

- Identifiers and Strings
- White Space Usage
- Netlist Statement
- Comments
- Line Continuation
- Section Keywords
- Expressing Real Numbers
- MAST Keywords
- MAST - Non-Reserved Keywords

Identifiers and Strings

Identifiers are names for variables, templates, nodes, etc. There are two valid forms for identifiers. One must start with an alphabetic character or an underscore (_), followed by alphabetic characters, digits, or underscores. The other form must start with the @ character and be followed by a string constant. There is no limit on the number of characters in an identifier, and all characters are significant. Because the MAST language is not case-sensitive, the following words all refer to the same identifier: `name`, `NAME`, `NaMe`.

Identifiers for connection point pins names can be unsigned integers, but only in netlist entries, and nowhere else in the template.

String constants consist of zero or more characters, without any double quotes or newlines, enclosed in double quotes. Empty strings can be specified as `""`.

String constants cannot span more than one line, although you can use concatenation as follows:

```
"This is a long string. Because it might not"//
"fit on one line, it is put together using"//
"concatenation."
```

There are no escape sequences (such as the C language backslash) available in string constants. By using a foreign subroutine, you may enter a double quote or a newline into a string (refer to Messages on page 5-10).

White Space Usage

Blanks, tabs, and comments (described later) are considered “white space”, and are ignored except to separate words when no other punctuation is required. For example, blanks (spaces) act as separators in a list of nodes on a template reference.

Netlist Statement

A netlist statement (or template reference) is a netlist entry in the netlist section of a template that “calls” (refers to) another template. It consists of the name of the template, followed by a period, followed by a reference designator as follows:

templatename.*refdes connection_points [= argument_assignments]*

The *templatename*.*refdes* is considered a complete unit, so no blanks are allowed within it. The reference designator (*refdes*) may consist of alphabetic characters, digits, and underscores, with no requirement that the first character be alphabetic.

The *connection_points* consists of the names of the nodes to which the model’s connection points are joined. Entries in this list must be separated by white space. The node names must either be in the same order and quantity as the connection points specified in the called template, or they must be specified with reference to the actual connection point names, in the form:

```
connection_point_name:node_name
```

For example, if the diode (with the reference designator dx) is connected to nodes named node1 and node2, you can specify it either as:

```
d.dx node1 node2
```

or as either of the following:

```
d.dx p:node1 n:node2
d.dx n:node2 p:node1
```

Also, node names cannot have a number as the first character unless all remaining characters are numbers:

Correct:	5, v16, vcc94b, 6431
Incorrect:	5v, +5V, 15v1

Non-alphanumeric characters (such as + or -) are not allowed.

The *argument_assignments* are required only for templates that have uninitialized arguments. Formats for this argument list depend on the data structure of the arguments. Multiple entries in an argument list must be separated by commas.

Comments

The MAST language ignores blank lines and comments. A comment must begin with a pound sign (#) and is recognized as running to the end of the line. A comment can start anywhere within a line, which is useful for temporarily removing a line or part of a line.

Line Continuation

Normally, a carriage return terminates a line. A backslash (\) at the end of a line with no comment indicates continuation of the line and is called a continuation character. It has no meaning in an input file except to indicate continuation, and if it is the last non-comment character on a line, it is discarded.

In addition, each of the following indicates that the line is to be continued if it is in context (part of the line) and it is the last non-comment character on the line:

```
+   -   *   /
&   |   <   >
(   {   [   ~
,   .   :   =
```

The semicolon (`;`) is allowed as an explicit line terminator, but it is not required. Typically a semicolon is used to allow more than one statement on a single line. One use for the semicolon is in data structure definitions, following a list of variables, to allow the closing brace (`}`) to be on the same line rather than the following line.

The line parser does not count the parentheses in a line, and so cannot determine whether a closing grouping symbol (`)`, `}`, `]`) is the final one in a statement. Therefore, in the absence of the line continuation character, a closing symbol indicates the end of a line.

Section Keywords

The left brace (`{`) that follows a section keyword must always be on the same line, right after the keyword. These keywords are `control_section`, `equations`, `parameters`, and `values`. The left brace must also directly follow `when`.

Expressing Real Numbers

All real numbers can be expressed either in the usual scientific notation, with the letters e or d expressing powers of 10, or with the following suffixes:

a	atto	10^{-18}
f	femto	10^{-15}
p	pico	10^{-12}
n	nano	10^{-9}
u (or mu)	micro	10^{-6}
m	milli	10^{-3}
k	kilo	10^{3}
meg (or me)	mega	10^{6}
g	giga	10^{9}
t	tera	10^{12}

You can express a number as a constant immediately followed by an appropriate abbreviation (do not include units).

No alphanumeric character can appear immediately after a suffix; a space or punctuation is required.

Note that `m` means 10^{-3}, and `me` and `meg` mean 10^{6}.

For example, the following are equivalent:

```
x=3p     x=3d-12     x=3e-12
```

The following are illegal specifications for numbers:

```
x = 3 p
```
(space not allowed between number and abbreviation)

```
x = 1mA
```
(units not allowed)

MAST Keywords

The following keywords are reserved in the MAST language and cannot be used as variable names in a template:

component	control_section*	element	else
enum	equations*	external	foreign
group	if	inf	number
parameters*	pin	ref	return
simvar	state	states	string
struc	template	undef	union
unit	val	values	var

*Section keywords that must be followed by a left brace, {, on the same line.

MAST - Non-Reserved Keywords

The following groups of keywords are not reserved and can be used as variable names, although it is good practice to treat them as if they were reserved:

Control Section Words

Simulator Variables

Intrinsic Functions

Predefined Numbers in header.sin File

If any of these keywords are declared as a variable name, it loses its special meaning.

Control Section Words

The following words are used in a control section:

collapse	noise_source	sample_points	newton_step
pl_set			

Simulator Variables

The following words are simulator variables (simvars):

dc_domain	dc_done	dc_init	dc_start
freq	freq_domain	freq_mag	freq_phase
next_time	statistical	time	step_size
time_domain	time_init	time_step_done	tr_done
tr_start			

Intrinsic Functions

The following are the available intrinsic functions:

abs	acos	acosh	asin
asinh	atan	atanh	cos
cosh	d_by_dt	delay	deschedule
error	event_on	expc	
instance	len	limexp	ln
log	message	random	schedule_event
schedule_next_time	sin	sinh	sqrt
tan	tanh	threshold	union_type
warning			

Predefined Numbers in header.sin File

The following are predefined numbers in the `header.sin` file:

Number Definition	Description
`temp = 27`	Temperature value
`mos_scale = 1.0`	Scale factor for mosfet physical dimensions (used by `m.sin` and `spm.sin`)
`mos_scalm = 1.0`	Process scale factor for mosfet physical dimensions (used by `m.sin` only)
`r_tol     = 0`	Resistor value tolerance (for example, a value of 0.05 indicates a 5% tolerance resistor)
`c_tol     = 0`	Capacitor value tolerance (for example, a value of 0.05 indicates a 5% tolerance capacitor)
`l_tol     = 0`	Inductor value tolerance (for example, a value of 0.1 indicates a 10% tolerance inductor)
`r_pdmax   = undef`	Maximum power dissipation for resistor (for example, 1/4 W resistors)
`c_vmax    = undef`	Forward voltage rating for capacitors
`c_vrmax   = c_vmax`	Reverse voltage rating for capacitors (different from `c_vmax` for electrolytics)
`include_stress = 1`	Allows removal of stress analysis in netlisted templates. Default value = 1 will run stress.
`use_2g6 = 0`	Allows invocation of SPICE2G.6 compatible MOS models through `m.sin` and `spm.sin`
`acc_fac = 1`	Accuracy factor
Global values for hydraulics library	
`rho       = 1k`	Global value of rho (kg/m**3)
`mu        = 14.3m`	Global value of mu (N-s/m**2)
`bulk      = 689.5meg`	Global value of bulk modulus (N/m**2)

Number Definition	Description
`patm        = -101325`	Global value of patm (N/m**2)
`pcav        = -95k`	Global value of pcav (N/m**2)
`valid_pres = inf`	Stress rating parameter

Logarithms are expressed in MAST as follows:

```
base e = ln
base 10 = log
```

This differs from how other programming languages (such as FORTRAN, RATFOR, and C) express logarithms:

```
base e = log
base 10 = log10
```

chapter 3

Declarations and Data Structures

Introduction

This chapter lists and describes the various “types” that variables can assume in a MAST template. It gives more information on pin and unit definitions, explains declarations, and shows how to refer to data structures in netlist entries and in the rest of a template.

- A *unit definition* specifies the unit types (such as voltage, current, or time) of the system’s variables. You can use units in declarations of `var`s, `ref`s, `state`s, and `val`s.

- A *pin definition* defines a pin type, which applies to any through and across variables for that pin type. Pins are a specific type of connection point that you must declare; these declarations refer to the pin definition. Connection points can be declared `pin`s, `var`s, `ref`s, and `state`s.

- *Connection point declarations* identify the types of the connection points named either in a template header or internally in a template.

- *Argument declarations* describe the types of values that can be passed into the template arguments from a netlist statement. It determines the syntax used to pass arguments into a template, and the syntax of references to arguments within the template. Argument types include `number`, `enum`, `string`, `struc`, `union`, and `argdef`, all of which can be parts of arrays.

- *Parameter declarations* describe the types of the values that parameters can take on. Parameters are used in expressions that assign values to other types of variables. A parameter declaration determines the syntax of the references to that parameter within the template. Parameters can be of the same types as arguments.

- *System variables* are the independent variables in the mathematical description of the system (across variables, `var`s, and `ref`s). The pin declaration causes an implicit declaration of the associated through and across variables. You must declare `vars` and `refs` and specify their units.

- *Val declarations* declare variables and specify their unit types. The simulator assigns values to `vals` only when needed, which is not necessarily at each time or frequency step.

  `Vals` can act as intermediate variables that receive values in the Values section and then are used to carry those values into equations in the Equations section or a when statement.

  Any expression of any type may be assigned to a `val`. The `extract` command uses the `dfile` and the information in the Values section to assign values to `vals`.

- *State declarations* identify variables to be used to model discrete time simulation and specify their units. States can be initialized and can be assigned values in when statements and in the values and equations sections of the template.

- *Simulation variable* (simvar) *declarations* are optional. You do not have to declare simvars because their definition is part of the simulator. Simvars are pre-defined variables that pass information from the simulator to the template or from the template to the simulator. You may use only the names pre-defined in the MAST language as the names of simvars.

- *External declarations* identify parameters, arguments, and pins brought from a higher level to a lower level template.

- *Foreign declarations* identify foreign functions and foreign states. There are two kinds of foreign functions: those that return a single number and those not restricted to returning a single number. Foreign states are used for mixed-simulator applications.

- *Group declarations* are a way to group variables together for extraction and for other purposes.

- *Template declarations* describe a template. This term refers to the entire template format.

- *Implicit declarations* are those that occur as a product of other definitions and declarations, and do not require explicit declaration. Variables that are declared implicitly include through and across variables (declared implicitly by pin definitions), simvars, connection point assignments used in netlist statements, external templates, node names, and net names.

Variable names fall into two categories: units and pins, and all other variables. In a template, the names of pins and units must all be different, but they need not be different from other variable names. For example, there may exist both a unit `v` and a variable `v`.

The following sections describe the variable types in more detail.

Unit and Pin Definitions

The MAST language lets you define units for use in describing systems. Units appear in:

- Definitions of pin-type connection points
- Declarations of `var`, `ref`, `state`, and `val` connection points

Unit definitions must precede any declarations that use them.

Standard unit and pin definitions are in the file `units.sin`. This file is included in the file `header.sin`. Normally, this file is included automatically by use of pre-loaded templates, so if you wish to use the standard definitions, you do not need to add unit and pin definitions to your template.

NOTE

If you enter the saber command with no contradictory options, it defaults to the "-la" option. This loads the "saber load" file analogy.sld at the top level of a system. This is a pre-compiled file made from analogy.sin using the "-p" option. The standard analogy.sld file contains header.sin, which includes units.sin. The spice.sld file also contains header.sin.

To define new units or pins, or to change definitions, there are several options.

- Change `units.sin` and run `saber -p` to pre-compile it for inclusion with the `-la` option.
- Change `units.sin` and include `header.sin` at the top level of the system hierarchy, and do not use the `-la` option. (The file can be included by writing `<header.sin` where the < sign is in the first column)

- Write new definitions of units and/or pins in the template before they are used in declarations. Pins and units cannot be redefined, so you cannot change pin and unit definitions in this way.

You may place unit and pin definitions above the header, in the header declarations section, or in the local declarations section. Regardless of where you place them, they are global to the entire system description.

Unit Definitions

Unit definitions can take one of two forms—analog or digital. The analog form of a `unit` definition is as follows:

`unit { "`*symbol*`", "`*unit*`", "`*definition*`" }` *identifier*

where *identifier* is the name being defined, and the three strings give the unit abbreviation, the full unit name, and the unit description, respectively.

Examples of `unit` declarations are:

```
unit {"V","Volt","Voltage"} v
unit {"A","Ampere","Current"} i
unit {"rpm","Revolutions/minute","Angular velocity"} w
unit {"kg.m","kilogram meter","Torque"} t
```

Logic states use the following form `unit` definition:

```
unit state {MASTname,"Boolean value","printmap","plotmap",
          MASTname,"Boolean value","printmap","plotmap"}
           name = MASTname
```

There must be as many lines in the unit definition as there are states in the unit state. Two discrete logic families are provided, *logic_4* and *logic_3*. The logic_4 unit definition describes four-state logic (0, 1, X, Z); therefore, it has four lines. The logic_3 unit definition (0, 1, Z) has three lines. By convention, the name of the unit is `logic_`*number*, where *number* refers to the number of logic states.

Each line has four entries that provide all the information needed for one of the states.

The first entry in the definition is the *MASTname* assigned to the state. This is the name used in the MAST language for the value of the state. For example, one of the *MASTnames* for the logic_4 family is `l4_0`, which corresponds to the logic state of 0. A `state` variable declared as a logic_4 type can be assigned to be `l4_0`.

The second entry is the *Boolean value* of the state, which is used by the waveform calculator in Scope. The calculator only accepts the values of 0, 1, and X for digital signals, so all states must be assigned one of these three values.

The third entry is the *printmap*. The *printmap* can be an arbitrary string. It is used when a digital state is printed to represent the value of the digital state.

The fourth entry is the *plotmap*. The *plotmap* is a string that is dictated by Scope. Its syntax is *symbol*`.`*style*. Symbol can be `low`, `middle`, `high`, or `unknown`. The graphic meaning is shown in the following table.

Symbol	Graphic Display
low	low line
middle	mid-level line
high	high line
unknown	low and high lines

Styles can be numbers 1-6 as shown in the table below. This field was used in prior releases. It is currently ignored by Scope.

Style	Mono Graphic Display	Color Graphic Display
1	solid line	black
2	long-dash line	dark blue
3	dash line	red
4	2dot-2dash line	purple
5	dot-dash line	dark green
6	dot line	brown

At the end of the definition is an initializer (*name*=*MASTname*), which must be one of the MASTnames described above. This provides the default value for states declared to be of a particular logic family when they are not initialized in a template.

An example of a unit state is the logic_4 family defined as follows:

```
unit state {l4_0,"0","0","low.1",
        l4_1,"1","1","high.1",
        l4_x,"x","x","middle.1",
        l4_z,"x","z","middle.1"} logic_4=l4_x
```

Pin Definitions

Pin-type connection points need to be defined in terms of the units they will use.

NOTE
It is generally unnecessary to insert this section in a template, because standard pin definitions are specified in the units.sin file. This file is automatically included when you load the Saber simulator.

The two general forms of a pin definition are:

pin *identifier* **across** *unit1* **through** *unit2*

pin *identifier* through *unit1* across *unit2*

In this example, *identifier* is the name of the pin type being defined, and *unit1* and *unit2* are the through and across unit types that are to be associated with that pin type. Examples of pin types contained in `units.sin` are:

```
pin electrical through i across v
pin rotational through w across t
```

Once you have defined these pins (for example, in `units.sin`), you can use the definition to declare the types of pins in a template.

Across variables are those whose values are equalized when two or more pins are tied together at a node (as, for example, voltage in electrical systems). Across variables follow a generalized KVL law: the sum of across variables around a closed loop is zero.

Through variables are those (like current in electrical systems) whose values follow a generalized KCL law: the sum of through variables flowing out of a node is zero.

Connection Point Declarations

Naming a variable and specifying its type is called a *declaration*. There are four kinds of connection points that you can declare in a template:

- pin-type
- state
- var
- ref

Pins are analog connection points that use through and across variables (for electrical circuits, these are used to form KCL and KVL equations). When you specify them in the template header and declare them in the header declarations section, pins are available for external connection. When you declare them in the local declarations section, pins can be internal nodes.

A pin declaration identifies the type of node to which a pin may be connected. The general format of pin declarations is the following:

`pintype` *id[,id...]*

where the *pintype* is a word already specified in a pin definition, and the *id*s are the names of the pins being declared.

An example of a pin declaration is the following:

```
electrical c,b,e,s
```

This declares that the four pins `c`, `b`, `e`, and `s` are `electrical` pin-type connection points. An automatic side effect of this declaration is that the simulator implicitly makes available `v(c)`, `v(b)`, `v(e)`, and `v(s)` (with respect to ground) as across system variables; it makes `i(c)`, `i(b)`, `i(e)`, and `i(s)` as through system variables.

Although it is recommended that pins be declared, they do not have to be declared if you use them in a netlist statement within the same template.

Pin names can be integers as long as they are used only in a netlist. If you use non-integer pin names, you can then use them in other places within the template. This implicit declaration is valid only after the netlist statement occurring within the template. This means, for example, that the pin names could be used in a Values section following the netlist statement, but not in a When statement preceding the netlist statement. The following example shows a valid use of the implicit declaration:

```
  template templatename a b
  {
    val v v
    templatename2.1 a b = 1k
    values
    {
      v = v(a)-v(b)
    }
  ...
}
```

Parameter and Argument Declarations

Parameters are template variables that are used to assign values to other types of variables. They are declared in the local declarations sections. The way they are declared determines the syntax used to refer to them later in the template. Parameters may be declared with initial values assigned. If they are not assigned initial values, then they are given an initial value of undef, which is described in the chapter about *Intrinsic Functions and Values*.

Arguments are template parameters that are listed as part of the template header—their values can be passed in from a netlist entry. They appear in the netlist entry to the right of the first equals sign (=):

`template.`*refdes connection_points [=argument_list]*

Arguments are declared in the header declarations section of the template. Because they are parameters, the way they are declared determines the syntax used to pass their values. If arguments in a template are not assigned initial values in their declarations, then they must be assigned values when the template is referenced in a netlist entry. If they are assigned initial values in the template and are not given specified values in a netlist entry, the initial values become defaults.

Within the template, parameters and arguments are essentially the same--they may be of the same types and they can both be initialized and referred to using the same mechanisms. Thus,

NOTE

Unless otherwise indicated or required, the term parameter is used for both parameters and arguments.

Parameter Types

A parameter may be declared as either a simple type or composite type. Each type has different variations as listed in the following table.

Simple	Composite
number	structure
enumerated	union
string	

Simple Parameter Types

There are three simple types of parameter:

- number
- enum (enumerated)

- string

These parameters use an equals sign (=) to assign one appropriate value to that particular variable.

NOTE

If a simple parameter is used as an argument, it must be initialized (either in the template or in a netlist)—otherwise, a netlist error will result.

If a simple parameter is used as a local parameter, the Saber simulator will automatically initialize it to undef (unless otherwise initialized in the template).

Numbers

A parameter declared as a number type requires a numeric (integer or real) value. The Saber simulator uses only real numbers, so there is no syntax that distinguishes between real numbers and integers. The form of the declaration is:

`number` *id [[= initial_value] ,id [= initial_value]...]*

where the *id*s are the argument or parameter names, and the *initial_values* are (optional) numbers or expressions specifying initial values. Such expressions must consist of constants, parameters that have been previously initialized, or template arguments.

For example, the following declares several number declarations on one line:

```
number vcc=5, dc_input, rload=10k, cload
```

Here, `vcc`, `dc_input`, `rload`, and `cload` are declared as number type parameters. In addition, `vcc` and `rload` have been assigned initial values of 5 and 10k, respectively. If these were arguments, `dc_input` and `cload` would need to be specified in a netlist entry.

If these variables were arguments, they could be assigned values in a netlist entry one of two ways:

1. Following the equals sign, just list the values in the order the arguments are listed in the header, separated by commas (... = `5, 2.7, 10k, 47n`)

2. List the names of the arguments and assign their values in any order (... = `cload=47n, vcc=5, dc_input=2.7, rload=10k`)

For clarity, you may always wish to specify the argument names, regardless of order. Arguments without initial values assigned in the template must be specified by the netlist entry.

templatename`.`*refdes connection_points* `= 5,2.7,10k,47n`

templatename`.`*refdes connection_points* `= dc_input=2.7, cload=47n`

If these numbers are parameters, you can assign them values in the Parameters section of the template. Any simple parameters can be used after they are declared in the template by referring to them by name (e.g., `vcc`, `dc_input`, `rload`, `cload`).

Enumerated Types

A parameter declared as an enumerated type (`enum`) may hold only one of a restricted set of names. This set of names must be specified within braces `{ }` when declaring the `enum` parameter.

The form of the declaration is:

`enum` {*evalue [, evalue...]*} *id [[= initial_value], id [= initial_value]...]*

where the *id*s are the arguments or parameters being declared, and the *evalues*, which are names, are the values they may contain. The *evalue* names, once declared, are meaningful in the template, and no other variables or *evalues* may have the same name. The *initial_values* are the *evalues* assigned to *id*s as initial values.

An example of an `enum` declaration is:

```
enum {_n,_p} bjt_type = _n
```

which declares `bjt_type` to be an argument or parameter that can assume values of only `_n` or `_p`. In this example, `bjt_type` has been assigned an initial value of `_n`. You can use enumerated types in assignments and in comparisons; you can also pass them to foreign routines.

If this were the declaration of an argument, you could assign it a value from a netlist entry by assigning one of the enumerated values to `bjt_type`. (Again, the argument name is necessary only if the arguments are taken out of order.) Because the declaration in the template specifies an initial value, it is not necessary to specify it in the argument list of a netlist entry if you want to use that value:

templatename`.`*refdes connection_points* `= bjt_type=_n`

If the declaration is for a parameter, you can assign it a value in the Parameters section or use the initial value by not assigning a value. You can use a simple enumerated type in a template by referring to it by name (e.g., `bjt_type`).

When passed to foreign routines, enumerated types have a numerical value that indicates their position in the enumeration declaration (1, 2, 3, etc.). In the above example, if `bjt_type` were to be passed to a foreign routine, it could have only one of the values: 1 (meaning `_n` for npn), 2 (meaning `_p` for pnp), or `undef`. For information about passing variables, refer to *Foreign Functions*.

Strings

A parameter declared as a `string` type may contain string constants. String constants are zero or more alphanumeric characters (other than the double quote or new line characters), which are enclosed in quotation marks.

The form of the declaration is:

`string` *id[[=initial_value]*, *id[=initial_value] ...]*

where the *id*s are the arguments or parameters being declared, and the *initial_value*s are the initial assignments to strings.

For example, the following statement declares the parameter `coretype` to be a string (it is also initialized to `iron`):

```
string coretype="iron"
```

Strings that are arguments can be assigned string variables or strings constants by a netlist entry. Using the name of the string variable in such an assignment is only necessary if the arguments are taken out of order.

templatename.*refdes connection_points* `= coretype = "air"`
templatename.*refdes connection_points* `= "air"`

You can assign values to strings that are parameters in the Parameters section. A string variable does not take on a fixed length; that is, it can be re-assigned a string constant with a different length. You can use a simple string variable in a template by referring to it by its name (e.g., `coretype`).

Composite Types

There are two composite parameter types: structure and union.

These parameter types allow you to group multiple parameters (either simple or composite) together, providing convenience and flexibility when working with large numbers of parameters.

structure	It is often convenient to work with many related variables as a unit. An example of such a grouping is the model argument of a semiconductor device such as a bipolar junction transistor, which is declared as a structure. This structure in turn, declares several dozen related simple parameters that can all be called by the name of the structure, `model`.
union	It may be necessary for a single argument or parameter to hold different types of information at different times, one type at a time. An example of this kind of grouping is the transient (`tran`) argument of the voltage source template, which is declared as a union. This union, in turn, declares both structures and enumerated parameters, only one of which can be active at a time. That is, assigning a value to one member of the union in a netlist entry overrides all the other members of that union.

Structures

A structure is a parameter that declares an ordered list of other parameters. The most general form of structure declaration is as follows:

```
struc [structurename] {
    declaration
    declaration
    ...
} id [= initial_value] [,id [= initial_value]...]
```

Note that the structure *id* comes after the closing brace.

Aside from the keyword `struc`, the structure declaration consists of four major components: the structure name (*structurename*), the declarations of parameters within the structure, the *id*s of the parameters being declared as instances of the structure, and, optionally, their initial values.

Following is an example of a structure declaration:

```
struc fred {
    enum {_n,_p} type
    number is,beta,cj
} bjt
```

This example declares a single argument or parameter named `bjt`. This variable is a structure of the type whose name is `fred`. It contains four subordinate parameters (`type`, `is`, `beta`, and `cj`). The parameter `type` is an enumerated type, while `is`, `beta`, and `cj` are numbers.

Using this declaration, it is now possible to refer directly to the structure name (`fred`) as a shorthand way of declaring additional structure arguments or parameters:

```
struc fred m1,m2
```

where `m1` and `m2` are declared as structures, the same as `bjt`.

It is not necessary to declare a structure name (such as `fred`). The following single declaration accomplishes the same purpose as the above two:

```
    struc {
        enum {_n,_p} type
        number is,beta,cj
} bjt,m1,m2
```

On the other hand, it is possible to separate completely the declaration of the structure name from the declarations of the arguments or parameters that have that structure, as follows:

```
    struc fred {
       enum {_n,_p} type
       number is,beta,cj
    }
struc fred bjt,m1,m2
```

You can refer to argument structures from netlist entries by assigning values to the parameters within the structure. These are enclosed within parentheses when assigned values from a netlist entry. The names of the parameters within the structure need to be specified only when they are not given in order (although for ease in reading netlist entries, you may want to specify them):

templatename.*refdes* *connection_points* =
`bjt=(type=_p,is=1a,beta=100,cj=10p)`

If the parameters of a structure are not assigned initial values, but the structure itself is assigned a value of `()` in the netlist entry, all of its parameters are given the value `undef` when simulated.

If the structure is a parameter, the value assignment in the Parameters section is similar.

You can assign initial values to the parameters within the structure in two places: within a structure declaration or following the parameter name. The next example declares a structure with three number parameters: `a`, `b`, `c`.

Initial values are assigned to a and b, but not c. Four *id*s of this structure parameter (w, x, y, and z) are then declared, with varying initial values:

```
   struc{number a=1, b=2, c
} w=(3,4,5), x=(c=3), y=(), z
```

The structure w assigns initial values of a=3, b=4, and c=5. The structure x "inherits" values a=1 and b=2 from the declaration, and initializes c=3 explicitly. The structure y also "inherits" the values for a and b, but c remains undefined, because it has no initial value in the structure and no value declared for it explicitly by y. Finally, the structure z is declared to have the same structure as the other structures, but with it has no initial values. Here a, b, and c will all be undefined.

If this structure defined arguments instead of parameters, w would have defaults of a=3, b=4, and c=5; x would have defaults of a=1, b=2, and c=3; y would have defaults of a=1, b=2, and c=undef; and z would have no defaults—it would need to be declared in the netlist entry that references this template. If the netlist entry were:

templatename.*refdes* *connection_points* = z=()

then z will also have values of a=1, b=2, and c=undef.

Unions

A union is similar to a structure, in that it groups parameters. However, the parameter within a union are "activated" at different times instead of all at the same time. A union allows you to define parameters or groups of parameters that will override the other parameters at different times—a choice function. The most general form of a union declaration is as follows:

```
   union unionname {
      declaration
      declaration
      ...
} id [[= initial_value] ,id [= initial_value]...]
```

Note that the union *id* comes after the closing brace.

Like the structure, the union declaration consists of four major components: the *unionname*, the declarations of the parameters within the union, and the *id*s of parameters being declared as instances of the union.

Unlike a structure, each instance (with name *id*) of a union parameter always has, as its value, only one of the declarations. Thus, a union type parameter presents a set of declarations as options—it will always evaluate to only one of the declarations.

The following example illustrates how unions are uniquely flexible among the parameter types:

```
  union source {
    number dc
    struc {
      number mag
      number phase
    } ac
} input1=(dc=5), input2=(ac=())
```

The declaration declares a single union type, named `source`, and two union *id*s: `input1`, `input2`. The declaration of `source` consists of two options: `dc` (a number type parameter) and `ac` (a structure type parameter with two numeric fields, `mag` and `phase`). From this, you can see that there is nesting capability—one of the members of the union grouping is a structure (`ac`) which also consists of a grouping.

You can create the same effect with the following declarations (the names of the unions and their declarations can also be specified as shown):

```
  union source {
    number dc
    struc {
      number mag
      number phase
    } ac
}
union source input1=(dc=5), input2=(ac=())
```

Initializing unions is similar to initializing structures. The difference is that you must also specify the name of the choice to be in effect. In the above example, `input1` is initialized to the choice of `dc` with a numeric value of 5. `Input2` is initialized to the choice `ac`, using initial values defined within the `ac` structure. Since `mag` and `phase` were not assigned initial values there, `input2` inherits `mag=undef` and `phase=undef`.

To reference a union argument from a netlist entry, you need to include, within parentheses, the name of the choice to be "activated" and its assigned values.

templatename.*refdes* *connection_points* =
                                        `input1=(ac=(mag=1,phase=0))`

To reference a union from inside a template, you must first determine the declaration that the union "activates." To do this, use the intrinsic function `union_type`, which is described in *Intrinsic Functions and Values*. After using this function, you can access the structures using the symbol -> (structure reference) as follows:

```
input1->dcinput2->ac->maginput2->ac->phase
```

When a union parameter is passed to a foreign routine, an indication of which choice is assigned is available to the foreign routine. The details of the passing conventions are shown in *Foreign Functions*.

Arrays

Just as it is useful to create composite type parameters to group together many parameters of possibly dissimilar types, it also desirable to keep several identical types together and to refer to them by a single name. You can do this with an array. You can declare an array to be of fixed size (like a structure), or of unbounded size (unlike a structure).

Only arguments, parameters, and states (of a fixed size) may be declared to be arrays. Use the following syntax to declare a simple array:

type id[*subscripts*]

where *type* is one of `number`, `enum`, `string`, or `state` id, and *id* is the name of the variable being declared an array, and the *subscripts* act as identifying numbers that distinguish among the members of the array. More than one set of subscripts indicates a multi-dimensional array. The subscripts themselves are a comma-separated list of simple subscripts, with each simple subscript giving an optional lower bound and an upper bound:

[*lower*:*upper*, *lower*:*upper*, ...]

The lower bound on any of the individual subscripts may be omitted, and if it is, it defaults to 1. The syntax would then be:

[*upper*, *upper*,...]

In addition, the upper bound of the first simple subscript may be specified as an asterisk (*), indicating a variable-length array whose length is determined at run-time:

[*,*lower*:*upper*,...]

It is not possible to have arrays in which the second and higher subscripts have variable lengths.

Some examples of array declarations are the following:

```
number tc[2]
number samples[*,0:1]
number x[0:50,5,-1:+1]
```

In the examples above, `tc` is a one-dimensional array of 2 numbers; `samples` is a two-dimensional array, of which the first dimension is not declared until run time, and the second dimension starts at 0 and ends at 1; `x` is a three dimensional array which has a first dimension of 0 to 50, a second dimension of 1 to 5, and a third dimension of -1 to 1.

These arrays can be initialized by including the appropriate number of comma separated entries between square brackets.

```
number tc[2]=[0,1]
number y[0:2,2,-1:1]=[1,2,3,4,5,6,7,8,9,10,11,12]
```

If you want to assign values to arrays, you must use square brackets.

The elements of multidimensional arrays are stored with the last subscript varying first (by row). This convention is the same as that in Pascal, but the opposite of that used in FORTRAN. The only time you need to know this is when passing the arrays to a foreign routine. The Chapter on *Foreign Functions* describes the details of passing variable length arrays to foreign routines.

Arrays of Composite Types

Arrays of composite or nested composite type parameters can be declared by putting the square brackets after the *id* of the parameter. For example, the following declare a structure containing two number parameters (`breakpoint`, `increment`) as an element of arrays `svbe` and `svbc`:

```
struc{
  number breakpoint, increment
} svbe[*],svbc[*]
```

or

```
struc sa_points{
  number breakpoint, increment
}
struc sa_points svbe[*],svbc[*]
```

In both these examples, `svbe` and `svbc` are one-dimensional arrays with the number of structures they contain determined at run time. Each member (structure) within these arrays has two numbers: `breakpoint` and `increment`.

You can initialize the arrays by setting each structure equal to parenthesized sets of two numbers, separated by commas and enclosed in square brackets. The following example declares `svbe` to be a variable-sized array of structures.

```
struc{
  number breakpoint, increment
} svbe[*]=[(-1k,10),(-10,.1),(0,.1),(10,10),(1k,0)]
```

Initially, its size is set to five by the five pairs of numbers enclosed in the square brackets. Because the field names are not named in the number pairs, the simulator considers the numbers in each pair to be in the same order as in the structure declaration—`breakpoint` first, `increment` second. Specifying values for these arrays as arguments or as parameters requires the same syntax as the initializer.

Nested Composite Types

The members of structures and unions may be any of the parameter types. In addition, they may be declared as fixed- or variable-length arrays.

An example of this is the following declaration of a source, similar to a voltage source. Note that the union and structure *id*s (`tran` and `source`, respectively) come after their closing braces.

```
struc {
  number dc
  union {
    struc {number off,ampl,freq,ph;} sin
    number pwl[*]
  } tran
  struc {number mag,phase;} ac
} \
source
```

This example declares `source` as a structure with three members (presumably to be used with three different analyses): `dc` (a number), `tran` (a union, meaning a choice of two values or waveforms), and `ac` (a structure). Within the `tran` union, two possibilities exist: the structure `sin`, and the variable-length array, `pwl`.

If you wanted `source` to be an array of length 3, you could achieve this by replacing the last line of the structure declaration above by `source[3]`.

This example illustrates two syntax features not yet discussed—the semicolon (;) and the backslash (\). The syntax requirements of a structure include the need for an end-of-line just before the closing brace. However, for readability you can keep the brace on the same line as the last declaration if you use the semicolon, as above. The backslash is a continuation character, meaning that the next line is to be treated as a continuation of the line with the backslash. This is simply to enhance readability.

The four syntax examples below assign values to `source`. This syntax will work when declaring initial values for source in the nested composite type declaration, when referring to an argument `source` in a netlist entry, or when referring to a parameter `source` in the Parameters section.

1. `source=(dc=5)`
2. `source=(tran=(sin=(0,1,10k,0)))`
3. `source=(tran=(pwl=[0,1,2,3]))`
4. `source=(ac=(1,0))`

In the first example, `source` is declared as `dc=5`.

In the second example, `source` is declared as a `sin tran`. The `source` structure, the `tran` union, and the `sin` structure all require parentheses.

In the third example, `source` is declared as a `pwl tran`. The `source` structure and the `tran` union require parentheses, while the `pwl` array requires square brackets.

In the fourth example, `source` is declared as `ac`. The `source` structure and the `ac` structure both require parentheses.

Declaration Operators

There are three two-character operators that provide a shorthand method of using composite parameters:

`..`	argdef
`->`	structure reference
`<-`	structure overlay

These operators allow you to declare an argument or parameter to be of the same type as in an existing template. Thus, if its declaration is lengthy, you need not write it out in full for every template that uses it.

Argdef

Argdef (argument definition) declarations let you specify an argument that is of the same type as an argument in the same or another template. It consists of two periods (`..`) and appears in the general form of the simplest argdef declaration as:

templatename`..`*argumentname id [[= initial_value]* `,`
id [= initial_value]...]

where *templatename* is the name of the template from which the argument definition is to be “borrowed”, *argumentname* is the name of an argument within *templatename*, and the *id*s are the names of the arguments or parameters being declared to be of the same type as *argumentname*. Assuming that there is a template with the name **q**, and it has an argument with the name `model`, you can declare variables named `m1` and `m2` to be of the same type with the following declaration:

```
q..model m1=(), m2=()
```

With this declaration, the two variables `m1` and `m2` can be used whenever the `model` argument of the `q` template is required. In particular, if the `model` argument of `q` is a large structure, then `m1` and `m2` are also large structures, with the same subordinate parameter names and any existing default values.

Because it is possible for template definitions to be nested, it is also possible for argdefs to refer to arguments in the nested templates. The general form is as follows:

template1`..`*template2*`..`*template3*`..`*argumentname*

with as many template names as are needed to reach the required argument. For example, there may be a template `nand`, that uses its private definition of a MOS transistor called `mos`. You could declare an argument to be of the same type as the `model` argument within `mos`, as follows:

```
nand..mos..model m1
```

Argdefs are initialized to the same parameter names and initial values as in the defining argument. It is as if the *id* were listed after the structure as follows:

data_structure {
declarations
} *id=()*

You can supply additional initialization information with the same syntax used to initialize other *id*s of the data structure. In the example using `q..model`, assume that one of the member parameters is `bf`. The following statement initializes `bf` to a value different from the default:

```
q..model m1=(bf=80)
```

Arrays of argdefs

If an argument or parameter is defined as an array of argdefs, and the argument used in the declaration is itself an array, the resulting variable will be an array with the subscripts in the arguments appended to the subscripts used on the argdef declaration. This is best explained by an example. Consider a template `lowlevel`, with the following declaration for one of its arguments:

```
number lowarg[5]
```

In the template `midlevel`, some variables may be declared by an argdef:

```
lowlevel..lowarg x,y[3]
```

which declares `x` to be just like `lowarg`, that is, an array of five numbers, and `y` to be an array of three items, each of them like `lowarg`. For `y`, the result is a variable identical to one declared as follows:

```
number y[3,5]
```

If, in the template midlevel, an argument were declared using an argdef:

```
lowlevel..lowarg midarg[2]
```

it could be used in a template `toplevel` as a declaration:

```
midlevel..midarg a[*]
```

with the result that a would effectively be declared as:

```
number a[*,2,5]
```

Notice that, although variable-length array declaration is possible with argdefs, the resulting subscripts must have the variable-length indicator in the first position. An argdef that would result in something like the following is not allowed:

```
number a[3,*,2]
```

Structure and Array Reference

Structure reference declarations let you refer to members of a structure or union within the same template. It consists of the right arrow symbol (->) and is formed according to the general rule:

variable -> *variable* [-> *variable* ...]

where *variable* is either a variable or a subscripted variable. Each structure reference operator (->) points to a field within a structure or a union. For example, if `rb` were declared within the structure `model`, using -> would allow using `rb` individually as a variable within the template:

```
struc {
  number is,rb,cj
  string type = "n_type"
} model
model->rb
```

For a union, if `phase` were declared within the structure `sin`, using -> would allow using it individually as follows (note that it is necessary to use it twice—first to reference `sin` then to reference `phase` within `sin`):

```
union {
  number dc
```

```
    struc {number vo,va,f,td,phase;}sin
} tran
tran->sin->phase
```

Similarly, you can use the same operator with an array. An array reference has the following general form:

variable[*expression*,*expression*,...]

where each *expression* must, when evaluated, be a number (real numbers are converted to integers by truncation).

An example of an array reference is shown below:

```
  struc {
    number work[32]
} m1[*]
m1[i]->work[32]
```

where array `work` is a member of array `m1[i]` (an array of arrays), and `m1[i]->work[32]` represents one of its elements.

Structure Overlay

Structure overlay declarations let you assign the values of a composite parameter to another composite parameter, and then change values of specific members. The general syntax for this assignment is:

left_hand_value=*structurename*<-*structurevalue*

where the *left_hand_value* and the *structurename* parameters have both already been declared as identical structures. The structure overlay operator (<-) is used with the *structurevalue* to indicate the members to be changed in the *left_hand_value* parameter.

For example, using an argdef to define `model1` and `model2` as structures of the same types as `model`, which is declared as:

```
  template bjt c b e s = model, area
  electrical c,b,e,s
  number area
  struc{
    number is,bf,re=0,rb=0,rc=0
} model=()
```

then, the following local declaration would start them with the same initial values as `model`:

```
  #in local declarations section
bjt..model model1,model2
```

You could then change the `is` and `rb` values for `model2` by using the `structure_overlay` with `model1`:

```
#in parameters section
model1=(is=1e-14,bf=100)
model2=model1<-(is=1e-15,rb=10)
```

In this example, this replaces the longer version of making this re-assignment, which would be:

```
model1=(is=1e-14,bf=100,rb=0,rc=0,re=0),
model2=(is=1e-15,bf=100,rb=10,re=0,rc=0)
```

The structure overlay can be used to replace the following sequence of statements:

```
model1=model2
model1->is=1e-15
model1->rb=10
```

with:

```
model1=model2<-(is=1e-15,rb=10)
```

Thus, the members not explicitly mentioned in the *structurevalue* are taken from the structure named by *structurename* instead of from the defaults defined in the structure declaration. The parameter values that are explicitly specified are assigned to directly to *left_hand_value*, overriding the contents of *structurename*.

Simulator Variables

Simvars are certain pre-defined variables that pass information from the simulator to the template or from the template to the simulator. Simvars are known to the simulator and thus do not have to be declared. You may use only the names predefined in the MAST language as the names of simvars.

If you use the name of a simvar to declare a variable of another type within a template, you will not have access to it as a simvar. If a simvar name is used for something else (such as a node in a netlist), then it cannot be used hierarchically for “child” templates or templates descending from them.

The form of a simvar declaration is as follows:

`simvar` *id,[id...]*

where the *ids* must be chosen from the following set of simulator variable names:

dc_domain	freq	next_time	time_domain
dc_done	freq_domain	statistical	time_init

`dc_init`	`freq_mag`	`step_size`	`time_step_done`
`dc_start`	`freq_phase`	`time`	`tr_done`
			`tr_start`

You may use most simulator variables only in the Values section, in When statements, and in the Equations section of the template. You can use the `statistical` simvar only in the Parameters section and in netlist statements.

Simulator variables fall into two opposing categories:

5. Those that get their values from the simulator and are available for use (but not modification) in the template.

6. Those that get their values from the template and are available for use by the simulator. They are `next_time` and `step_size`.

The simulator variables have the following meanings when declared as simvars:

Category 1

- `dc_domain` is set to 1 during DC analyses, that is, during DC, DT, and the DC portion of DCTR. It is set to 0 otherwise.

- `dc_init` is set to 1 at the start of DC analyses, meaning at the start of the DC, DT and the DC portion of DCTR. It is set to 0 otherwise. It is used primarily in When statements.

- `dc_start` is set to 1 at the start of any DC analysis and the DC portion of DCTR, even one that is restarted from a previous DC initial point. It is set to 0 otherwise. It is used primarily in When statements.

- `dc_done` is set to 1 after the DC algorithm is completed. It is set to 0 otherwise. It is used primarily in When statements.

- `freq` is set (continually updated) to the simulation frequency at which the template is being evaluated. Freq is defined only during frequency domain analyses. Freq is set to 0 in DC and time domain analyses.

- `freq_domain` is set to 1 during frequency domain analyses, that is, during frequency, distortion, and noise analyses. It is set to 0 otherwise.

- `freq_mag` is set to 1 during frequency domain analyses, that is, during frequency, distortion, and noise analyses, that compute the magnitude of complex numbers. It is set to 0 otherwise.

- `freq_phase` is set to 1 during frequency domain analyses that compute the phase of complex numbers. It is set to 0 otherwise.
- `statistical` is set to 1 when Monte Carlo and other statistical analyses are being performed. It is set to 0 otherwise. It can be used only in the Parameters section and in the netlist.
- `time` is set (continually updated) to the simulation time at which the template is being evaluated. Time progresses only during time domain analyses. Time is set to 0 in frequency and DC domain analyses in the Values and Equations sections for analog-only simulation.For templates providing mixed-mode simulation (i.e., containing When statements), the value of time is dependent on the DC Algorithm outlined in The DC Algorithm on page 9-11.
- `time_domain` is set to 1 during any transient analysis. It is set to 0 otherwise.
- `time_init` is set to 1 at the start of transient analysis. It is set to 0 otherwise. It is not reset when restarting a transient analysis from a previous one. It is used primarily in When statements.
- `tr_start` is set to 1 at the start of any transient analysis, including one restarted from a previous transient analysis. It is set to 0 otherwise. It is used primarily in when statements.
- `tr_done` is set to 1 at the end of any transient analysis. It is used primarily in When statements.
- `time_step_done` is set to 1 at the end of each time step in transient analysis. It is used primarily in When statements.

Category 2

- `next_time` can be set by the template to a future time that the simulator must reach exactly. If the template has no scheduling requirements, it should leave `next_time` undefined. A typical use of this simvar is in piecewise linear sources, where it tells the simulator when the next turning point occurs in the definition of source, or any other time-dependent template where an “abrupt” change occurs. Another use is to ensure that the simulator uses a particular time. More information about using `next_time` is provided in the subsection titled Scheduling Analog Waveform Sampling Times.
- `step_size` can be set by the template to specify a desired maximum time step size to the integration algorithm. The simulator uses the

value to limit the size of the next time step. A typical use is in sinusoid pulses.

System Variables

System variables are the dependent and independent variables in the mathematical model of the system being simulated. They include across variables, `var`s, and `ref`s.

You do not have to declare across variables (or through variables) because they are declared implicitly by the pin definition. The pin definition automatically declares a dependent through variable of the form *through(pin)* and an independent variable of the form *across(pin)*.

Var declarations

An explicitly declared `var` is a second type of system variable. The simulator assigns a value to a `var` based on a line of the following form in the Equations section:

varname`:`*expression* `=` *expression*

For each `var`, there should be one such line in the Equations section of the template. You can use `var`s in the Equations, Values, and Control sections.

The general form of a `var` declaration is:

`var` *unit id[*`,`*id...]*

where *unit* is one of the units declared in a unit declaration, and the *id*s are the names of the `var`s being declared. If a `var` is to be passed out of a template as a connection point, you must declare it in the header declarations; otherwise, declare it in the local declarations section.

Ref declarations

A third type of system variable is the `ref`. If it is necessary to refer to a `var` in another template, you may declare a `ref`. Such a declaration binds the `ref` in the current template to the `var` in the other template, and it has the following general form:

`ref` *unit id[*`,`*id...]*

where *unit* is one of the units declared in a unit declaration, and must be the same unit as in its previous declaration in the template where the `var` was defined. The *id*s are the names of the `ref`s being declared.

You may declare only connection points (of the current template) as `ref` variables. You can use a `ref` in the equations section, similar to the way you use a `var`:

refname`:`*expression* `=` *expression*

Its effect is that the right-hand side of this statement (the part to the right of the equal sign) is added to the left-hand side of the statement defining the value of the `var`. This is done even if the statement defining the value of the `var` is in another template.

An example of using a `ref` is the input current of a current-controlled source, such as **ccvs**.

State Variables

State variables are used in discrete time simulation (refer to *MAST Functions*). There are two types of state variables: digital and event-driven analog.

- A *digital signal* is discrete in the values it represents; for example, 0, 1, x, and z. It is also discrete in time.
- An *event-driven analog* signal can assume any real number as a value, but values are still discrete in time.

States can be declared and used internally in a template, they can be passed to or from a template as connection points, or they can be passed to a foreign simulator.

States passed in as connection points cannot be initialized within a template. They automatically take on initial values of `undef` for analog event-driven states, and the initial state declared in the unit definition for digital states. States declared locally in a template can be initialized, and should be initialized to conform to a zero value of any associated analog waveform (refer to Initializing Templates on page 9-10).

Digital states should be declared as foreign when they are used to relay information to foreign simulators in mixed-simulator simulation, where mixed-simulator simulation involves using the Saber simulator interfaced to a digital simulator. All foreign state declarations must be local. At present, the provided templates use the logic_4 family in mixed-simulator simulation and in hypermodels, which are templates written in the MAST language which serve as interfaces between connection points in an analog network and digital pins in a digital network. More information can be found in the mixed-simulator documentation for the appropriate combination of the Saber simulator with a digital simulator.

You must declare state variables as follows:

`state` *unit id[=initial_value[*`,` *id=initial_value]...]*

You can also declare them as fixed length arrays.

State variables can receive values only in when statements. You can use them either in when statements or on the right-hand side of statements in the Values or Equations section.

The following are examples of state declarations:

```
state logic_4 inm
state v vin
state nu handle[2] foreign
state logic_4 out
```

In the first example, `inm` is declared to be a digital signal using the logic_4 units definition. In the second example, `vin` is declared to be a voltage; it is an event-driven analog signal. In the third example, `handle` is declared to have no units (`nu`), and is a one-dimensional array of length 2. In the fourth example, `out` is declared as a foreign state, and will presumably be used in a mixed-simulator application. Foreign states can only be declared in the local declarations section.

Values

`Val` (value) declarations declare variables and specify their unit types. The simulator assigns values to `val`s only when needed, which is not necessarily at each time or frequency step.

`Val`s can act as intermediate variables that receive values in the Values section and then are used to carry those values into equations in the equations section.

The values assigned to vals depend on parameters, states, and system variables. The values of states and system variables go to the dfile (data file), while the parameter values are fixed. Extract reads the dfile and can therefore (with the help of the Values section) assign values to vals.

The general form of a val declaration is the following:

`val` *unit id[*`,`*id...]*

where *unit* is one of the units declared by a unit declaration and *id*s are the names of the `val` variables being declared.

Variables of type `val` can receive values only in the Values section of a template, and can be used in the Values, Control, and Equation sections of a template, and in When statements.

NOTE

A `val` *cannot receive a value from a subordinate template called in the Netlist section.*

Groups

It is convenient, and in some cases necessary, to group together several variables, and to refer to the group by a single name. For arguments and parameters, the usual method of combining them is to put them into a structure. However, groups differ from structures in several ways, and therefore serve different purposes. The differences are as follows:

- Groups need not consist of arguments and parameters
- A variable may be a member of multiple groups
- Groups cannot be used to pass arguments to templates

The general form of a declaration of a group is as follows:

`group {`*member*`,` *member*`,`*...*`}` *id*

where *members* are the names of the variables or other groups in this group, and *id* is the name of the group being declared.

Groups must be homogeneous. That is, they must consist entirely of arguments and parameters or entirely of system variables, `val`s, and simvars.

You can use groups to arrange vars and vals for extraction or for use with the `siglist` command. By convention, extraction groups include the following:

- `v` - voltage
- `i` - current
- `f` - flux
- `noise` - noise
- `pd` - dc power
- `pt` - instantaneous power

The following example groups current variables `ie`, `ic`, and `ib` under the name `i`:

```
group {ie, ic, ib} i
```

Although it is not necessary, you can also use groups when passing variables to and from foreign functions, and in Control section statements. Groups can be used to define pl_set, sample points, and newton steps. An example of using groups for newton steps is shown below (for further information on this syntax, refer to Purpose of Template Sections on page 7-3).

```
  #in local declarations section
  group {vbe, vce} voltages
  struc {
    number sample_point, increment
} nsteps[*]=[(0.3,3m),(0.6,1m),(0.7,0)]
  control_section{
  newton_step (voltages, nsteps)
}
```

Templates

Templates are declarations of names just like other declarations, and can be mixed with other declarations. However, because they are the most complicated constructs of the language (and also the most powerful), the details of their definition are described in the Chapter on *Templates*. Here only a simplified outline of a template declaration is given here.

The form of a template declaration is as follows:

```
[type] template templatename connection_pt_list [= arguments]
    header declarations
    {
        local declarations
        parameters section
        netlist
        when statements
        values section
        control section
        equations section
}
```

where *type* is optional, *templatename* is the name of the template, *connection_pt_list* is a list of the connection points of the template (separated by blanks or commas), and *arguments* is a comma-separated list of the arguments of the template. The body of the template, which is enclosed in braces, consists of local declarations, followed by several sections describing different aspects of the template (refer to the *Templates* Chapter).

The top-level template defining a system does not require the template header. Syntactically, it is the body of an unnamed template, without the surrounding braces:

local declarations
parameters section
netlist
when statements
values section
control section
equations section

The simplest and most commonly used top-level template is merely the netlist of a circuit or system; it uses predefined templates and does not use the other body sections.

You can include templates in other templates by writing the included template in the local declarations section. Templates declared locally are only known locally, and therefore cannot be accessed freely by outside templates.

Resolving Template Names

Whenever a template name is used in a netlist, the simulator must find its definition. In general, if there is not a declaration for a referenced template in the current template, the referenced template must be found in a higher-level template. Because template declarations may be nested to an arbitrary depth, a search for the declaration of an external template proceeds up the path of declarations until the declaration is found, or the top level is reached without finding the template. By convention, any precompiled template brought in when the simulator starts up is automatically considered to be at the top level of the template hierarchy, although this is not a requirement.

If the template declaration is not found then, the Saber simulator performs a search along the `SABER_DATA_PATH` for a file named *templatename*`.sin`. If the file is found, it is automatically included at the top level of the template hierarchy. If not, an error message is printed. In case of an error, you should examine the `SABER_DATA_PATH` and change it or the location of the template in question, or simply include the template and its full path name in the system.

Implicit Declarations

Every name used in a template must be declared to be of some type. Almost all names must be declared explicitly by one of the declarations described in the previous sections. There are, however, several cases of names that are implicitly declared when they are encountered.

Variables that are declared implicitly include the following:

- Through variables
- Across variables

- Simvars
- Connection points used in netlist statements
- External templates
- Reference designators
- Node names

Through and across variables are declared implicitly by the associated pin definition. For instance, if a `pin_name` is declared to be electrical, `i(pin_name)` is implicitly declared as a through variable, and `v(pin_name)` is implicitly declared as an across variable.

Simvars are known to the Saber simulator, and therefore you do not have to declare them.

Although you can improve template readability by declaring connection points, you do not have to declare them if they are used in netlist statements. (Presumably they will be declared further down the hierarchy.) Connection points cannot be used until they are declared, so if they are not declared by the user, they cannot be used in sections before the netlist statement that implicitly declares them.

Netlist statements implicitly declare the templates named. The system tries to locate a template in a file with names of the form templatename.sin through an automatic search using the SABER_DATA_PATH. If a template does not lie along the SABER_DATA_PATH, its full pathname should be included in the template before its use in the netlist statement.

Netlist statements also implicitly declare the reference designators and node names named in the statement.

For example, in the following netlist statement (in the absence of local declarations):

```
r.r1 n1 n2 = 1k
```

`r` is implicitly declared as an external template (**r**); `r1` is implicitly declared as a reference designator; `n1` and `n2` are implicitly declared as connection points of the same type (i.e., pin, `var`, `ref`, or state) as those in the template being referenced. It is not necessary to list a separate include statement for the template **r** since it can be found along the SABER_DATA_PATH.

External Declarations

In most cases, the names and types of variables used in a template are declared in the template and are not accessible from outside the template. Occasionally, however, it is useful to bring in a value of a variable in another template of the system, without explicitly declaring the variable in the current template.

Temperature is an example of a variable that could be brought into the template from the outside without entering it in the argument list of every template reference. Other examples are global nodes such as rail voltages or system clocks.

The mechanism for doing all of the above is to make the appropriate variable an external variable. For variables such as parameters or system variables you must declare them explicitly with an external declaration. A variable is declared to be an external variable by preceding its declaration by the keyword external:

```
external type id[,id...]
```

Some specific examples are:

```
external number temperature
external bjt..model lateral
external electrical vcc,vee,signal_ground
```

Only parameters and pins may be declared to be external.

The following example illustrates how the value of an external variable (`temperature`) is resolved through several levels of hierarchy.

Resolving Variable Names—An Example

When a variable is declared to be external, the search proceeds up the path of template references. The following example illustrates template references nested several levels deep. Only the skeleton of the templates is given here to bring out the relevant details.

```
#template example for resolving variable names
#local declarations section
number temperature=27
template r ...
  {
  external number temperature
  #...definition of a resistor by equations
  }
template bjt ...
  {
  r.b b bprime = rb
  #...further definition by equations
  }
template diffpair ...
  {
  bjt.1 ...
```

```
    }
  template opamp ... = local_t
    number local_t
    {
    number temperature
    parameters {
      temperature=local_t
    }
    diffpair.1 ...
    bjt.1 ...
    r.1 ...
    }
  # netlist section
    r.1 ...
    bjt.1 ...
    opamp.1 ... = local_t = 50
opamp.2 ... = local_t = 80
```

In this example, each reference to the **r** template is resolved at the top level, because that is the only level at which an **r** template is declared. However, the nesting of the references to the various templates is as follows:

Top Level:

- r(1)
- bjt(1)
  - r(2)
- opamp(1)
  - diffpair(1)
    - bjt(2)
      - r(3)
  - bjt(3)
    - r(4)
  - r(5)
- opamp(2)
  - diffpair(2)
    - bjt(4)
      - r(6)
  - bjt(5)
    - r(7)
  - r(8)

In the template for **r**, temperature is declared as an external number, and its value must be obtained from the higher levels of the hierarchy. Taking the eight cases of resistor references in turn:

1. The temperature is found at the top level, and is 27.

2. The temperature is looked for in bjt$^{(1)}$, is not found there, and is then found at the top level. It is 27.
3. The temperature is looked for in bjt$^{(2)}$, then in diffpair$^{(1)}$, and finally in opamp$^{(1)}$, where it is found. It is 50, as passed into the opamp by the `local_t` argument.
4. The temperature is looked for in bjt$^{(3)}$ and then in opamp$^{(1)}$, where it is found. It is 50.
5. The temperature is found in opamp$^{(1)}$. It is 50.
6. The temperature is looked for in bjt$^{(4)}$, then in diffpair$^{(2)}$, and finally in opamp$^{(2)}$, where it is found. It is 80, as passed into the opamp by the `local_t` argument.
7. The temperature is looked for in bjt$^{(5)}$ and then in opamp$^{(2)}$, where it is found. It is 80.
8. The temperature is found in opamp$^{(2)}$. It is 80.

Foreign Declarations

Foreign declarations are used for and for foreign states. There are two types of foreign subroutines: subroutines that are declared in such a way that they return a single number, and subroutines that are not restricted in what they return. The syntax for these declarations follows:

```
#Type 1...in local declarations section
foreign number subroutinename()
#Type 1...in another template section
variablename=subroutinename(input_list) [bin_operator expression]

#Type 2...in local declarations section
foreign subroutinename
#Type 2...in another template section
(output_list)=subroutinename(input_list)
```

The first type shown above is a foreign subroutine that returns a number. After it is declared, you can use that number in a statement wherever you can use a numeric value. As an example:

```
#Type 1 example...in local declarations section
foreign number fred()
#Type 1 example...in another template section
a = b + fred(c)
```

The second type is a foreign subroutine whose output is not restricted. Each member of the *input_list* and *output_list* must be declared.

Foreign state declarations are required for digital states that are intended for use by a foreign simulator, that is, in a mixed-mode simulation. Foreign states must be declared in the local declarations section, using the keyword foreign. The syntax is the same as a local connection point declaration:

```
  #Type 2 example...in local declarations section
foreign state units statevar
```

Refer to the particular mixed-simulator documentation for specific information on using the Saber simulator with a foreign digital simulator.

chapter 4

Expressions

Introduction

Expressions play an important role in templates. The main uses of expressions are to pass arguments to templates, to modify parameters, and to define values in assignment statements and in equations. They can be used in all sections. You can build expressions out any kind of variable, parameter type, strings, operator, and other characters in the MAST language. Expressions are allowed as statements, which is especially useful for messages and within the body of when statements.

Expression Types

An expression can be any of the following types:

constant
variable
array_reference
structure_reference
array_reference
unary_operator expression
expression binary_operator expression
function(expressions)

The following sections explain these types.

Constants

Constants can be numbers, enumerated, or string type values.

Variables

Variables can be numbers, enumerated types, strings, simvars, `vars`, `refs`, `vals`, states and across variables. An across variable has the form *unit(pin)* where *pin* is either a connection point or a node, and *unit* is the across variable declared in the pin definition. For example, for an electrical node `in`, the across variable is voltage; therefore, the across variable at the connection point `in` is `v(in)`. Dependent through variables of the form *unit(pin),* where *unit* is the through variable in the pin definition, can only be used in the Equations section.

Structure References and Array References

A *structure reference* is formed according to the general rule:

variable -> *variable* [-> *variable ...]*

where *variable* is either a variable or a subscripted variable. Each structure reference operator (->) points to a field within a structure or a union.

An *array reference* has the following general form:

variable[*expression*,*expression*,...]

where the *expressions* must, when evaluated, be numbers (real numbers are converted to integers by truncation).

An example of a structure reference is:

```
struc {
  number is,rb,cj
  string type = “n_type”
  } model
model->rb
```

where number `rb` is a parameter declared within the structure model.

You can also use a structure reference for nested levels:

```
union {
  number dc
  struc {number vo,va,f,td,phase;}sin
  } tran
tran->sin->phase
```

where number `phase` is a parameter within the structure parameter `sin`, which in turn is contained within the union parameter `tran`.

An example of an array reference is:

```
struc {
  number work[32]
  } m1[*]
m1[i]->work[32]
```

where the array `work` is a member of array `m1[i]` (an array of arrays), and `m1[i]->work[32]` represents one of its elements.

Expressions with Operators

Expressions may contain unary and binary operators. All binary operators are left associative except for the `**` operator, which is right associative. (The `**` operator indicates that the number to its left is to be raised to the power of the number to its right.)

Operators can operate upon any expression, assuming that the result is meaningful. The general forms for expressions with operators are:

unary_operator expression
expression binary_operator expression

All operators (except the concatenation operator `//`) may operate on numbers. Only the == (is equal to) and ~= (is not equal to) operators may operate on enumerated types and strings. The result of any operation (except string concatenation, which is a string) is a number. The boolean operators treat 0 as false and all other numbers as true; they always return 0 for false and 1 for true.

The unary operators are:

-	negation
~	boolean not

The binary operators are:

**	to the power of
*	multiply
/	divide
+	plus
-	minus
<	less than
>	greater than
>=	greater than or equal to
<=	less than or equal to
==	equal to
~=	not equal to
&	Boolean AND
_	Boolean OR
//	string concatenation

Operator Precedence

The operators are listed below in decreasing order of precedence, where operators on the same line are equal in precedence. When operators are equal, they are applied as they are encountered in a left-to-right scan of the expression (except for the ** operator and when parentheses indicate otherwise):

Unary Operators			
**			
*	/		
+	-		
//			
<	>	>=	<=
==	~=		
&			

Examples

Following are some examples of expressions. Each expression in the left column is equivalent to the corresponding parenthesized expression in the right column:

a+b-c	(a+b)-c
a**b**c	a**(b**c)
a+-b*c	a+((-b)*c)
a<b&c==2	(a<b)&(c==2)
a==b==c	(a==b)==c
-~0	-(~0)

chapter 5

Intrinsic Functions and Values

Introduction

The MAST language provides intrinsic functions to be used in expressions. These include:

- Mathematical functions

  Trigonometric and Hyperbolic Functions

`sin`	`cos`	`tan`
`asin`	`acos`	`atan`
`sinh`	`cosh`	`tanh`
`asinh`	`acosh`	`atanh`

- Logarithmic and Exponential Functions

  `ln` `log` `exp` `limexp`

- Other Mathematical Functions

`abs`	(absolute value)
`d_by_dt`	(the derivative function)
`delay`	
`random()`	
`sqrt`	(square root)

- Argument and Parameter functions

`union_type`	(present value of a union)
`len`	(length of an array)

- Event-driven functions

  ```
  schedule_event
  schedule_next_time
  event_on
  threshold
  deschedule
  ```

- Messages

  ```
  message
  warning
  error
  instance()
  ```

- In addition, MAST provides two constants which can be referred to by name:

`undef`	(undefined)
`inf`	(infinity)

Mathematical Functions

You can use all mathematical functions in all sections of a template, with the exception of the derivative and delay functions (`d_by_dt` and `delay`). The `delay` and `d_by_dt` functions are restricted to the Equations section.

Trigonometric and Hyperbolic Functions

Basic trigonometric functions and their corresponding hyperbolic functions are available for use in the parameters, values, and equations sections of a template, and in when statements.

The following table shows these intrinsic trigonometric functions. The first column shows the syntax of the function, where x indicates an expression, whose value represents an angle, in radians. The second column shows the definition of the function. The third column indicates limitations, if any, on the value of the expression *x*:

Function Syntax	Definition	Limitations
`sin(`*x*`)`	sine (x)	none
`cos(`*x*`)`	cosine (x)	none

Function Syntax	Definition	Limitations
`tan(`*x*`)`	tangent (x)	x cannot equal $n(\pi/2)$
`asin(`*x*`)`	arcsine (x)	$-1 \leq x \leq 1$
`acos(`*x*`)`	arccosine (x)	$-1 \leq x \leq 1$
`atan(`*x*`)`	arctangent (x)	returns a value between $\pm\pi/2$
`sinh(`*x*`)`	$\frac{e^x - e^{-x}}{2}$	none
`cosh(`*x*`)`	$\frac{e^x + e^{-x}}{2}$	none
`tanh(`*x*`)`	$\frac{\left(\frac{e^x - e^{-x}}{2}\right)}{\left(\frac{e^x + e^{-x}}{2}\right)}$	none
`asinh(`*x*`)`	$\ln(x + \sqrt{x^2 + 1})$	none
`acosh(`*x*`)`	$\ln(x + \sqrt{x^2 - 1})$	$x \geq 1$
`atanh(`*x*`)`	$\ln\left(\frac{1 + x}{1 - x}\right)$	$-1 < x < 1$

Log and Exponential Functions

The functions that provide the decimal logarithm and the natural logarithm are both available. In addition, the exponential function (e^x) is also available. These functions are shown in the following table, with the first column showing the syntax, and the second column showing the definition of the function, and any limitations:

Function Syntax	Definition	Limitations
`ln(x)`	natural (base e) logarithm of x	x > 0
`log(x)`	common (base 10) logarithm of x	x > 0
`exp(x)`	e^x	x ≤ 80
`limexp(x)`	a subroutine that numerically limits the value of e^x Also, `limexp(-x)=1/limexp(x)`	x > 80 This subroutine limits the value of e^x as follows: For 80 < x ≤ 88, `limexp(x) = (x-79)*exp(80)` For 88 < x ≤ 88.7, `limexp(x) = (1+1e-6*(x-88))*exp(88)` For x >88.7, `limexp(x) = exp(88.7)`

NOTE

Logarithms are expressed in MAST as follows:
base e = `ln`
base 10 = `log`

This differs from how other programming languages (such as FORTRAN, RATFOR, and C) express logarithms:
base e = `log`
base 10 = `log10`

Other Mathematical Functions

This section describes the other mathematical functions: `d_by_dt`, `delay`, `sqrt`, `abs`, and `random()`.

Derivative

The derivative function, `d_by_dt`, can only be used in the Equations section of a template. The syntax for taking the derivative of an expression with respect to time is as follows:

d_by_dt(*x*)

where *x* is any valid expression, except an expression containing `d_by_dt` (no nesting is allowed) or `delay`.

It can follow only the operators +, -, =, +=, and -=, so any constant multipliers must appear within x. Similarly, it can only be followed by +, -, and ;.

For example, to describe the effect of a linear time-invariant capacitor on the through variable, the following statement could appear in the Equations section:

```
i(p->m) += d_by_dt(cap*v)
```

where `i(p->m)` is the through variable, `cap` is the value of capacitance (a linear time-invariant capacitance declared as a number) and `v` is the voltage across the capacitor, (declared as a `val`).

Delay

The `delay` function lets you model the effects of delay in your system, as, for example, in a delay line. You specify the value that is to be delayed and the amount of delay (in seconds). Note that the `delay` function is not restricted to the time-domain analyses; it can apply to frequency-domain analyses as well.

You can use the delay function only in the Equations section of the template. It cannot be nested and it cannot contain a `d_by_dt` function.

The form of the delay function is:

delay(*reference_value*, *time_parameter*)

The *reference_value* can be a system variable (`var` or `ref`) or a linear combination of system variables. It can follow only the operators +, -, =, +=, and -=, so any multipliers must appear within the *time_parameter* expression. The *time_parameter* may be a constant, a parameter, an argument, or any expression composed entirely of constants, parameters, and arguments. A `delay` function can be followed only by operators + and -, or by a semicolon (;).

Then, in the Equations section, (in a template in which `i` is a `var`) you could include the following statement:

```
i: vindelay = delay(((vinp-vinm)/2),50u)
```

This causes the `vindelay` signal to be delayed 50 microseconds from `vin` (calculated as `(vinp-vinm)/2`).

Square Root

Use the square root function for all expressions that, when evaluated, produce a positive value of *x*. The syntax for the square root function is:

`sqrt (`*x*`)`

Absolute Value

The absolute value of any expression can be obtained by use of the following function, where x is any expression:

`abs (`*x*`):`

Random()

The `random()` function returns the next pseudo-random value in the range of 0 to 1, where 0 is included and 1 is excluded. When using the statistical environment, you can specify the seed of the pseudo-random sequence.

`random()`

This function does not take an argument.

Parameter Functions

There are two functions that affect parameters (and arguments): `union_type` and `len`.

union_type

When you define a parameter or argument to be of type `union`, you list two or more declarations that the variable can assume. At any given time, the variable assumes only one of the declarations. The `union_type` function has the form:

`union_type(`*union_name*`,  `*union_item*`)`

where *union_name* is the name of an argument or parameter of type `union`, and *union_item* is one of the items that *union_name* can assume. This function returns the value 1 if *union_name* has the value *union_item*, and it returns 0 otherwise.

Length of an Array

The `len` function returns the length of an array. It is useful particularly when the array is defined without bounds. This function has the form:

`len(`*array_name*`)`

where *array_name* is the name of the array whose length is being determined.

Event-Driven Functions

Event-driven functions are used in When statements for discrete time simulation.

The general form of the When statement is:

when (*condition***) {**
statements
}

Event_On

The `event_on` function is often used as a condition for the When statement. The `event_on` function returns 1 (true) whenever a value is assigned to a specified state variable, as previously scheduled by a `schedule_event`. The syntax of the `event_on` function is as follows:

`event_on` (*state_var* [, *old_value*])

where *state_var* is the name of a state variable to be monitored for an assignment, and *old_value* is the name of a state variable that receives the previous value of *state_var* when *state_var* is assigned a new value.

Threshold

`Threshold` is often used as a condition for the When statement. The `threshold` function returns 1 (true) whenever the value of a specified expression crosses, becomes equal to, or becomes unequal to, a specified value. It is useful for converting from analog to discrete systems, but has other uses as well. The syntax of the `threshold` function is as follows:

`threshold` (*expression*, *value* [, *before_state* [, *after_state*]])

where *expression* is compared to *value* to see if a threshold condition has been met. The *before_state* and *after_state* are output variables that can be used to determine how the threshold condition was met.

before_state equals:

1	if *expression* > *value* before the threshold
-1	if *expression* < *value* before the threshold
0	if *expression* = *value* before the threshold

after_state equals:

1	if *expression* > *value* after the threshold

-1	if *expression* < *value* after the threshold
0	if *expression* = *value* after the threshold

Schedule_event

The `schedule_event` function is often used in the statements portion of the When statement. The `schedule_event` function sets (schedules) a time at which a specified variable is to receive the value of a specified expression.

The syntax is as follows:

[scheduling_id =] `schedule_event(`*time*`,` *state_var*`,` *expression*`)`

The *scheduling_id* is an array of two unitless state variables. This array becomes a unique identifier when the event is scheduled, and can be used for de-scheduling the event. The *time* is an expression whose value indicates the time at which the assignment is to occur. Typically *time* is defined as the sum of the `time` simvar and some expression that represents a delay. The *state_var* variable is the name of the state variable that is to receive expression as its new value at time *time*.

Schedule_next_time

The `schedule_next_time` function schedules a time at which the integration algorithm samples the analog waveforms. That is, if the simulator's integration algorithm yields a time step that would cause the simulator to go beyond one or more scheduled "next" times, the simulator is required to step ahead only to the first such time. This is the means by which the discrete part of a system can affect the analog simulation. The syntax follows:

[scheduling_id =] `schedule_next_time(`*time*`)`

The optional *scheduling_id* identifier represents an array of two unitless state variables. You can use it to de-schedule the event. The time is an expression whose value indicates the time at which the assignment is to occur.

Deschedule

The `deschedule` function de-schedules a specified event or next time that had been scheduled previously by `schedule_event` or `schedule_next_time`. The syntax follows:

`deschedule(`*scheduling_id*`)`

The *scheduling_id* identifier represents an array of two unitless state variables. You can use it to de-schedule a scheduled event or *next_time*. A warning message occurs if you attempt to deschedule an un-scheduled event or time step.

Conflict resolution for event-driven digital nets

Each event-driven digital unit has an associated set of states. The supplied `units.sin` file contains two such sets, logic_4 and logic_3. In the `units.sin` file, there is, for each set, a conflict resolution routine (called `l4cnfr` for logic_4 and `l3cnfr` for logic_3). Conflict resolution routines are binary, associative operators that apply to two or more event-driven digital signals that drive the same net (node). The supplied routines resolve conflict according to the following tables:

`l4cnfr`:

	0	1	x	z
0	0	x	x	0
1	x	1	x	1
x	x	x	x	x
z	0	1	x	z

`ln3cnfr`:

	0	1	x
0	0	x	x
1	x	1	x
x	x	x	x

You can supply your own conflict resolution schemes as foreign functions. You can find out more in the foreign functions section.

Messages

There are four message functions: message, warning, error, and instance().

Message, Warning, and Error

Message, warning, and error comprise a group of three built-in functions that can be used in the Parameters section and in the bodies of When statements, but not in the Values or other sections of the template. Their format follows:

`message (`*format_string[*`,` *substitution_entities]*`)`
`warning (`*format_string[*`,` *substitution_entities]*`)`
`error (`*format_string[*`,` *substitution_entities]*`)`

Messages require a single *format_string*, usually a string constant, but which may be a string variable or expression (a concatenation of string variables and constants). The *format_string* may include ordinary text, substitution tokens, and escape sequences. All message *format_strings* have an implied new line at the end. A substitution token is a percent sign (%). It is replaced by the next available argument in the *substitution_entities*. An escape sequence consists of the backslash character (\) followed by another character. The *substitution_entities* need only be specified if there are substitution tokens in the message *format_string*.

\t	a tab character
\n	a newline character
\any	any other character taken literally; primarily to introduce the backslash (\) and percent sign (%) as ordinary text. (\\ and \%).

The `message` function simply prints the *format_string* with substitutions, and then gives a new line. Messages are intended to be used for debugging and for informational purposes.

The `warning` function prints the following annunciator line:

```
*** WARNING “TEMPLATE_WARNING” ***
```

on the screen, followed by the *format_string* with substitutions and a new line. Warnings are intended to inform template users in significant situations, such as when a user enters an invalid parameter, which is then reset to some reasonable default value in the template.

The `error` function prints this annunciator line:

```
*** ERROR "TEMPLATE_ERROR" ***
```

on the screen followed by the message format with substitutions and a new line. It is intended for errors in critical situations. It ends the analysis when encountered, and can exit the Saber simulator in some situations.

Substitution entities permitted in messages include any expressions. For debugging, it is especially useful to use the names of parameters and arguments, or to refer to their fields.

The instance() function

The `instance()` function returns the name of the template instance, including the full pathname, and thus is of particular use in messages. For example, if your top-level template had a netlist entry for `mytemplate.m1`, and if `mytemplate` contained a message such as:

```
message(“You are now using %”,instance())
```

in the Parameters section, then you would receive the message:

```
You are now using /mytemplate.m1
```

each time the Parameters section is evaluated.

NOTE

Any argument names or parameter names can be used as substitution entities for messages in the parameters section. If these are complicated data structures, then all the variables and their values from inside the structure will print out. Individual members can also be accessed.

For example, if the parameter `model` is declared as follows:

```
struc {
  enum {_n,_p} type = _n
  number is=1e-16,
    bf =100,
    vaf,
    tnom=27
  string abc=”xyz”
}model = ()
```

then the following statement in the Parameters section:

```
message(“model = %”,model)
```

produces the message:

```
model=(type=_n,is=100a,bf=100,vaf=undef,tnom=27,abc="xyz")
```

The message `“bf=% is greater than 80”,model->bf` produces:

```
bf=100 is greater than 80
```

Intrinsic numbers: undef and inf

The MAST language provides two constants, `undef` and `inf` (undefined and infinity), which you can refer to by name. They do not have to be declared to be used.

As a general rule, if parameters and arguments are not initialized, they are undefined, which means they assume a value of `undef`. You may, therefore, test in a template to see whether parameters and arguments are undefined. The `undef` number is also available in foreign subroutines because the Saber simulator passes it to foreign subroutines as one of the standard arguments.

Infinity (`inf`) is not assigned by the simulator, but can be used anywhere in a template to indicate infinity.

chapter 6

Statements

Introduction

There are several types of statements, some of which are available in only certain sections of a template as follows:

- Assignment statements, which you can use in the Parameters and Values sections, and in When statements.
- Expressions, which you can use as statements in the Parameters section, in When statements, and in the Values section.
- Foreign functions declared as numbers, which you can use anywhere a number can be used on the right-hand side of an assignment statement (either statement or field). They can be used as initializers.
- If statements, which you can use in the Parameters, Values, Control, and Equations sections, and in When statements.
- Control statements, which you can use only in the Control section.
- Equations, which you can use only in the Equations section.

The following sections describe these types of statements.

Assignment Statements

Assignment statements are similar in appearance to mathematical equations; however, assignment statements are allowed anywhere in a template except in the Equations section. They are evaluated in sequence (as in a program or routine). The expression on the right-hand side (RHS) of the equals sign is evaluated first and then assigned to the variable on the left-hand side (LHS).

In the Parameters section, only parameters can be on the LHS of assignment statements. In the Values section, only vals can be on the LHS of assignment statements. In When statements, only states can be on the LHS of assignment statements. Of these three categories of assignment statements, parameter assignment statements can be the most complex because parameters can have the most complex data structures. Parameters can be simple or composite types, or arrays, or nested combinations of these; vals are numbers or discrete values with declared units, and states are only numbers with declared units, arrays of numbers, or discrete values with declared units. For more information, refer to Parameter and Argument Declarations.)

Parameters Section

An assignment statement in the Parameters section can have one of the following forms:

1. *left_hand_value* = *expression[[*, *expression]...]*
2. *left_hand_value* = *structure_overlay[[*, *structure_overlay]...]*
3. (*id*,*id*,*id*) = *foreign_function* (*arguments*)

The *left_hand_value* is either the name of a parameter, a structure reference (->), or an array reference. The forms are described in the following subsections.

1. Left_hand_value = expression

The *expression* can be a combination of any numbers and variables. It can include intrinsic functions except for `d_by_dt` and `delay`. In Example 1 (below), the parameter `response_rate` is used to adjust the input units on the argument `slewrate`.

Example 1

```
template sample_and_hold p m = slewrate
  electrical p,m
  number slewrate       #specify in V/usec
{
  number response_rate
  parameters{
    response_rate = slewrate*1u
  }
...
```

}

In Example 2, if the value for the phase of a transient sine wave (specified within `vin`) is between 720 and 360 degrees, it is corrected to be less than 360.

Example 2

```
    template voltage p m = vin
      electrical p,m
      struc {
        number dc=0
        union {
          struc{number amp,freq,phase;}sin
          number pwl[*]
        } tran
      } vin
    {
      number phase
      parameters{
        if (union_type(vin->tran,sin)){
          phase = vin->tran->sin->phase

          if ((holder > 360) & (holder < 720)) {
            phase = phase - 360
          }
        }
      }
    ...
}
```

Example 3 contains examples of array references. First, an array reference of the fifth member of the `work` array contained in the structure `calc_model` is assigned a value of `rb*area`. Then the sample point array for `svbe` is assigned a series of breakpoint-increment pairs, `svbc` is set equal to it, and then the first sample point pair of `svbc` is changed to subtract 100 from the breakpoint and multiply the increment by 2.

Example 3

```
    template bjt c b e s = model, area
      electrical c,b,e,s
      number area
      struc{
        number is,bf,re=0,rb=0,rc=0
      } model=()
    {
      struc{
        number is,bf,work[10]
```

```
    } calc_model=()
    struc sample_point{
      number breakpoint, increment
    } svbe[*],svbc[*]
    parameters{
      calc_model->work[5] = model->rb*area
      svbe = [(-100,10),(10,1),(0,1),(10,10)]
      svbc = svbe
      svbc[1] = (breakpoint=svbc[1]->breakpoint-100,
        increment=svbc[1]->increment*2)
    }
  }
...
```

2. Left_hand_value = structure_overlay

This assignment uses the MAST operator <- (structure overlay) as a shorthand method of assigning the values of one variable to another variable, and then changing only specific field values. The general syntax for this assignment is:

left_hand_value=*structurename*<-*structurevalue*

where the *left_hand_value* and the *structurename* parameters have both already been declared as identical structures. The *structurevalue*s indicate the fields to be changed in the *left_hand_value* parameter.

In Example 4, an argdef (`..`) is used to define `model1` and `model2` to be of the same types as `model` in Example 3. Thus, they start with the same initial values as `model`. The values for `is` and `rb` from `model2` are then changed using the structure overlay.

Example 4

```
#in local declarations section
bjt..model model1,model2

#in parameters section
model1=(is=1e-14,bf=100)
model2=model1<-(is=1e-15,rb=10)
```

In this example, `model1=(is=1e-14,bf=100,rb=0,rc=0,re=0)`, and `model2=(is=1e-15,bf=100,rb=10,re=0,rc=0)`.

This form is available as a shorthand notation for the following sequence of statements:

```
model1=model2
model1->is=1e-15
```

`model1->rb=10`

Using the shorthand notation, this can be written as:

`model1=model2<-(is=1e-15,rb=10)`

Thus, the fields not explicitly mentioned in the *structurevalue* are taken from the structure named by *structurename* instead of from the defaults defined in the structure declaration. The fields that are explicitly mentioned are assigned to directly to *left_hand_value*, overriding the contents of *structurename*.

3. (id, id, id) = foreign function (arguments)

This type of assignment takes the returned value of a foreign subroutine on the LHS and assigns it to any number of variables separated by commas on the RHS. For more information on foreign functions and subroutines, refer to the chapter on *Foreign Functions*.

Foreign subroutines that are declared as numbers can be used anywhere numbers can be used. You can write a foreign function that converts degrees into radians, and then use it as in the following example:

Example 5

```
template sin_voltage p m = amp,freq,phase
  electrical p,m
  number  amp,    # amplitude in volts
         freq,    # frequency in hertz
         phase    # in degrees
{
<consts.sin
  foreign number deg2rad()
  val v voltage
  values{
    if (freq_mag){
      voltage = amp * sin(2*math_pi*freq + deg2rad(phase))
    }
...
```

In the preceding example, a supported file of math constants was included in the template using `<consts.sin`. The `math_pi` used in the assignment statement is declared in the `consts.sin` file.

A foreign function declared as a number can also be used as an initializer. Because the type of the return value is not known by the parser, it gets the type from the variable to which is assigned, and therefore can only appear on the RHS of an assignment statement.

The use of a foreign function not declared to be a number is also possible. This can be regarded as a kind of assignment statement, and is allowed in the Parameters and Values sections, and in When statements. The syntax is as follows:

left_hand_value = foreign_function_call ([input_list])
(output_list) = foreign_function_call ([input_list])
groupname = foreign_function_call([input_list])

The *left_hand_value*, the *output_list*, and the *groupname* can be different variables depending on the section in which the call is made. In the Parameters section, this type of foreign function can only return parameters, in the Values section it can only return values, and in a When statement it can only return states. The *left_hand_value* represents a single variable, the *output_list* represents a comma-separated list of variables, and the *groupname* represents the name of a set of variables declared to be a group.

The following example shows two foreign subroutine calls: one for `logsap` and one for `psrsub`.

Example 6

```
template psr p m = a,astar,vbo,l,lambda,sv
  electrical p,m
  number a,     #cross-sectional area in micron**2
    astar,      #emission constant in amp/micron**2/K
    vbo,        #barrier height at zero bias in volts
    l,          #resistor length in meters
    lambda      #grain diameter in meters
  struc sa_points {
    number breakpoint, increment
    } sv        #sample voltage points
{
  foreign logsap, psrsub
  val i current
  val p power
  val v voltage
  external number temp
  struc sa_points localsample[*]
  group {a,astar,vbo,l,lambda} psr_pars
  parameters{
    localsample = logsap(1n,1meg,1,90,sv)
  }
  values{
    v = v(p)-v(m)
    (current,power) = psrsub(psr_pars,temp,v)
  }
```

```
...
```

Here, the `logsap` foreign subroutine returns an array of number pairs in the parameter, `localsample`. The `psrsub` foreign subroutine returns two `val`s, `current` and `power`. For clarity, the output list on psrsub could also have been declared as a group, such as `psr_pars` was, and then only the name of the group would need to be used.

Values Section

An assignment statement in the Values section can have the following form:

left_hand_value = expression
(id,id,id) = foreign_function (arguments)

The *left_hand_value* is the name of a `val`; *expression* can be an expression using any combination of variables (except through variables). Intrinsic functions and foreign number functions can be used (except for `d_by_dt` or `delay`).

Example

```
template resistor p m = res
  electrical p,m
  number res
{
  val i current
  val v voltage
  values{
    voltage = v(p)-v(m)
    current = voltage/res
  }
...
```

When Statement

An assignment statement in a When statement can have the following form:

left_hand_value = expression[[, expression]...]
(id,id,id) = foreign_function (arguments)

The *left_hand_value* is the name of a state; the *expression* can be an expression using any combination of variables (except through variables). The example below shows an assignment of `next_low` and `next_high` to scheduled events. These variables, which are two-member arrays, function as scheduling_id's and would only need to be used if the events need to be de-scheduled.

Example

```
    template clock out = hightime, lowtime
      state logic_4 out
      number hightime, lowtime
    {
      state nu next_low[2],next_high[2]
      state nu notify
      when(event_on notify){
        next_low =schedule_event(time+hightime,out,l4_0)
        next_high=schedule_event(time+hightime+lowtim,
                                out,l4_1)
        schedule_event(time+hightime+lowtime,notify,0)
      }
...
```

Expressions

The MAST language accepts expressions as statements in the Parameters section and in When statements. In When statements the syntax for scheduling events and times has an optional scheduling_id, which can be used for de-scheduling. When these id's are not used, the resulting syntax is an expression. An example of this was shown in the preceding clock example.

```
schedule_event(time+hightime+lowtime,notify,0)
```

Messages are another example of expressions used as statements (refer to Messages on page 5-10). If messages were to be used as expressions within assignment statements or other sorts of statements, they would evaluate to zero. The following example shows the syntax for a message that prints out all the fields and values for a parameters named `model`.

```
message(“model=%”,model)
```

If Statements

If statements are allowed in the Parameters, Values, Control, and Equations sections of the template, and in When statements.

An If statement has the following syntax:

if(*expression***){**
  statements
}
else if(*expression***){**

```
    statements
  }
  ...

  else {
    statements
}
```

where *statements* represents one or more statements. The `else if` and `else` blocks are optional. There may be more than one `else if` block (represented by the ellipses). At most, one block of the entire if statement is executed. The expressions are evaluated, and the first true (non-zero) expression causes the corresponding block of statements to be executed. If there is an else statement, and none of the previous blocks has been executed, then the statements following the else will be executed.

If statements can be nested.

In the case where only one statement is needed in a block, double braces do not have to be used. In such a case, the statement must follow the `if`(*expression*) or the `else if`(*expression*) or the `else` directly.

Example

```
  template capacitor p m = capacitance
    electrical p,m
    number capacitance        #input value of capacitance
  {
    number cap              #local value of capacitance
    parameters{
      if ((capacitance==undef)|(capacitance<0.0)){
        cap = 0.0
      }
      else if (capacitance==inf) {
        cap = 1.0
      }
      else cap = capacitance
    }
...
```

Control Statements

A control statement may appear only in the Control section of the template. Control statements provide the simulator with specialized information that cannot be provided by any of the other sections. The general form of Control section statements is:

name (arg,arg,...)

where *name* is one of a predefined set of words, and *args* are identifiers declared in the template, expressions, or multiple identifiers or expressions. The specific control statements and their meanings are described in the chapter on *Templates*.

Equations

The Equations section describes the analog behavior of the system to the simulator in terms of through and across variables. It .also handles any `delay` or `d_by_dt` functions needed by the template.

Three types of statements may appear in the Equations section.

The first kind defines the value of a through variable as an expression in terms of other variables, and the simulator uses it to form a system equation by applying a generalized Kirchoff's Current Law (KCL) at a specified node. The second kind is a similar statement for a ref. And the third kind is an equation, corresponding to a declared var variable, that the simulator uses to find a value for the var.

The syntax for the first type of allowed statement is one of the following:

through_variable(pin_name) operator expression

through_variable(pin_name -> *pin_name) operator expression*

The pin declaration causes an implicit declaration of through variables. *Pin_names* are the names of pins used in the template. In the second case, the symbol -> indicates a flow of the through variable from the first *pin_name* to the second. Operators permitted are += and -=, which mean to add to or subtract from the equations at the node, respectively. *Expression* is any valid expression. It can contain all intrinsic functions (including `d_by_dt` and `delay`, which cannot be nested) and must follow and be followed directly by a binary operator.

The syntax for the second kind of allowed statement is the following:

ref_variable operator expression

The *ref_variable* is a `ref`, that is, a `var` passed in from another template. A `ref` may or may not need an equation in the Equations section. This depends on whether the `ref` contributes in the current equation to the `var` to which it refers.

The general form of an equation of the third kind is:

id : *expression* = *expression*

where *id* is the name of a `var` variable and *expression* has the same restrictions as described for the first and second kinds of equation.

chapter 7

Templates

Introduction

All descriptions of systems or elements in the MAST language are templates. Templates have a general form consisting of eleven different templates sections, but there is no requirement that all sections be used for all templates. The template sections used depend upon what is being modeled and whether previously defined templates are used in the model. This chapter approaches the design of a template through looking at the functionality of the different sections.

The possible template sections are as follows. The left-hand column provides a preview of the syntax and the right-hand column shows the title of each section as it is referred to in this manual.

Syntax	Title
unit definition	`# unit definition`
pin definition	`# pin definition`
header	`# header`
header declaration	`# header declarations section`
`{`	`# Begin template body`
local declarations	`# local declarations section`
`parameters {`	`# Parameters section`
statmements	`# ...`
`}`	`# ...`
netlist	`# netlist`
`when (`*condition*`){`	`# When statements`
statements	`# ...`

Syntax (continued)	**Title (continued)**
}	# ...
values{	# Values section
statements	# ...
}	# ...
control_section{	# Control section
statements	# ...
}	# ...
equations{	# Equations section
statements	# ...
}	# ...
}	# End template body

The top-level template in a hierarchical system model contains other templates or references to them, and is not referred to by other templates. In a flat system model, the whole template is top-level, by definition. The top-level template does not require the template header. Syntactically, it is the body of an unnamed template, without the surrounding braces, as shown below.

Syntax	**Title**
unit definition	# unit definition
pin definition	# pin definition
local declarations	# local declarations section
parameters {	# Parameters section
statmements	# ...
}	# ...
netlist	# netlist
when (*condition*){	# When statements
statements	# ...
}	# ...
values{	# Values section

Syntax	Title
statements	`# ...`
`}`	`# ...`
`control_section{`	`# Control section`
statements	`# ...`
`}`	`# ...`
`equations{`	`# Equations section`
statements	`# ...`
`}`	`# ...`

The simplest and most commonly-used system template describes only the netlist of a system, using pre-defined templates, and contains only the netlist section.

The template sections do not have to be in the order shown, but there is a requirement that all variables be declared before they are used. There can be as many When statements as needed. Netlist statements can appear at various places throughout the body of the template. The body is the set of sections typically enclosed in braces, starting with the local declarations section and ending with the Equations section.

Purpose of Template Sections

The sections of a template can be separated into two groups: declarative and operational. The declarative sections designate variables and other entities for use by the Saber simulator. The operational sections contain statements that are executed in various ways by the Saber simulator.

Declarative sections include the following (Also refer to the chapter on *Declarations and Data Structures*):

Sections	Description
unit defnitions	Unit definitions specify units that can be used for variables declared with units (e.g., vars, refs, states, and vals).
pin definitions	Pin definitions specify analog pin types and their associated through and across variables.

Sections	Description
header	The template header declares the name of the template, and specifies the type of template, the connection points, and the arguments. It determines how to refer to a template in a netlist entry.
header declarations section	The header declarations section specifies the types of the connection point and argument names given in the header and their default values.
local declarations section	The local declarations section specifies any other variables used locally within the template and their initial values.
Control section	The Control section has five specialized functions: 1. Collapsing nodes 2. Declaring dependencies between nonlinear dependent variables and independent variables for some templates 3. Declaring sample points for some nonlinear variables 4. Limiting the step size in Newton-Raphson iterations for some types of nonlinear variables 5. Defining noise sources

Sections	Description
Operational section	Operational sections include the following: • Parameters section -- used to manipulate parameters, to check the values of arguments passed into the template, and for speeding up other template work by performing mathematical transformations. • When statements--allow you to set up discrete time simulation, to describe digital behavior, to test for analog waveforms crossing a threshold, and to schedule events and times. • Netlist section--contains netlist entries in this section call other templates and specify their arguments. • Values section--allow you set up vals for extraction, to handle foreign functions needed for the equations sections, and to add information so the equations section is easier to read. • Equations section--describes the analog characteristics at the terminals of the element being modeled.

MAST Sections

The following pages describe each of the sections comprising a MAST template.

unit definition

Purpose

Unit definitions are declarative. They specify units that can be used for variables that are declared with units (i.e., vars, refs, states, and vals) and for pins. (It is usually unnecessary to write this section because standard units are already found in a supported file called `units.sin` that is included in `header.sin`)

Evaluation

Unit definitions are evaluated when the system is first read into the simulator.

Syntax

There are two ways to define units: the first is for analog units, the second is for digital units.

The syntax for analog unit definitions is as follows:

```
unit {"symbol","unit","definition"} identifier
```

The keyword `unit` is required. The *symbol* is used by the Scope Waveform Analyzer for assigning names to axis, `unit` is the full unit name, and *definition* is the unit description. The *identifier* is the name of the unit being defined.

The syntax for digital units is

```
unit state {MASTname,"Boolean_value","printmap","plotmap",
          ...
         MASTname,"Boolean_value","printmap","plotmap"}
          name = MASTname
```

There are as many lines in the unit definition as there are states in the unit state. Two discrete logic families are provided, named logic_4 and logic_3 in the pre-defined units. The logic_4 unit definition has four lines while the logic_3 unit definition has three lines. By convention, the name of the unit is logic_number where number refers to the number of logic states.

Each line has four fields that provide all the information needed for one state.

The first field in the definition is the *MASTname* assigned to the state. This is the name used in the MAST language as the value of the state.

The second field is the *Boolean_value* of the state, which is used by the waveform calculator in Scope. The calculator accepts only the values 0, 1, and x for digital signals, so all states must be assigned one of these three values.

The third field is the *printmap*. The *printmap* can be an arbitrary string. It will appear as the value of the state when using display or print, or while in Scope.

The fourth field is the *plotmap*. The plotmap is a string with the syntax *symbol*`.`*style*, and defines how the results will be graphicly displayed. Symbol can be `low`, `middle`, `high`, or `unknown`. The graphic meaning is shown in the following table.

Symbol	Graphic Display
low	low line
middle	mid-level line
high	high line
unknown	low and high lines

Style can be an integer from 1 through 6 as shown in the table below. This field was used in prior releases. It is currently ignored by Scope.

Style	Mono Graphic Display	Color Graphic Display
1	solid line	black
2	long-dash line	dark blue
3	dash line	red
4	2dot-2dash line	purple
5	dot-dash line	dark green
6	dot line	brown

At the end of the definition is an initializer, which must be one of the *MAST names*. This provides the default value for states declared to be of a particular logic family when they are not initialized in a template.

Description

The unit definition specifies the units used throughout the template. Once units are specified, you can use them in the declaration of ref, var, state, and val variables. You can also use them to assign units to through and across variables in pin definitions. Once a unit has been defined, it cannot be redefined. Once units are defined in any template, they are accessible by any other template in the hierarchy.

Standard unit and pin definitions are in the file `units.sin`. This file is included in the file `header.sin`. Normally, this file is included automatically in your top-level template, so if you wish to use the standard definitions, you do not need to add unit and pin definitions to your template. If you enter the `saber` command with no contradictory options, it defaults to the `-la` option. This loads the “saber load” file `analogy.sld` at the top level of a system. This is a pre-compiled file made from analogy.sin using the `saber -p` option. The standard `analogy.sld` file contains `header.sin` which includes `units.sin`.

To define new units or pins or to change definitions, there are several options as follows:

- ❑ Change `units.sin` and run the `saber -p` command to pre-compile it so it will be included under the `saber -la` option.
- ❑ Change `units.sin`, include `header.sin` at the top level of the system hierarchy, and use the `saber -ln` option. (Include `header.sin` in a template by writing `<header.sin`, with < in the first column.)
- ❑ Write definitions of previously undefined units and/or pins in the template before they are used in declarations. You can put unit definitions in either of the following places:
  - In the header declarations section
  - In the local declarations sections

Examples

The analog unit definitions for current, voltage, angular velocity, and torque are as follows:

```
  unit {"V","Volt","Voltage"} v
unit {"A","Ampere","Current"} i
unit {"rpm”,"Revolutions/minute","Angular velocity"} w
unit {"kg.m","kilogram meter","Torque"} t
```

A digital unit definition for the logic_4 family of states follows:

```
unit state {l4_0,"0","0","low.1",
         l4_1,"1","1","high.1",
         l4_x,"x","x","middle.1",
         14_z,"x","z","middle.1"
         } logic_4=l4_x
```

pin definitions

Purpose

Pin definitions are declarative. They specify analog pin types and their associated through and across variables. It is usually unnecessary to write this section because standard pin definitions are already found in a supported file called `units.sin` that is included in `header.sin`.

Evaluation

Pin definitions are evaluated only once, when the system is read into the simulator.

Syntax

There are two general forms of a pin definition:

`pin` *identifier* `through` *unit1* `across` *unit2*
`pin` *identifier* `across` *unit1* `through` *unit2*

The *identifier* is the name of the pin type being defined, and *unit1* and *unit2* are through and across units that are to be associated with the pin type. The units used in a pin definition must be defined before the pin is defined.

Description

Pin definitions specify analog pin types and their associated through and across variables. The Saber simulator uses through and across variables in generalized Kirchoff's current and voltage laws (KCL and KVL) to solve analog systems. These two laws may be stated as:

- KCL - the sum of through variables leaving a node is zero
- KVL - the sum of across variable drops in any loop is zero

Pin definitions enable the simulator to check the compatibility of connected components. For instance, it would not work to connect a resistor (electrical) directly to a motor shaft (mechanical).

Pin definitions are referred to by pin declarations in the header declarations section or in the local declarations section. When a pin is declared to be of a defined pin type in a pin declaration, the Saber simulator, using the pin definition, implicitly declares the through variable, `through` *node_name*, to be a dependent variable and the across variable, `across` *node_name* to be an independent variable.

The supplied templates usually solve analog systems using modified node analysis. In node analysis, through variables are added to and subtracted from the system matrix directly, and then the simulator solves for across variables. Therefore, through variables are dependent variables and across variables are independent variables. Node analysis must be modified to encompass all systems, since there are often situations where across variables are not functions of through variables, such as an ideal voltage source in which the current is whatever it needs to be to give the defined voltage. In such a case the through variable, current, is an independent variable, which is solved by adding an equation, or another row and column, to the system matrix.

Therefore, most through variables are dependent variables and all across variables are independent variables. The Saber simulator implicitly declares through variables of the form `through` *node_name* to be dependent variables, and across variables to be independent variables. If you need other independent variables for the system being modeled, you declare them as vars, and describe them by using an additional equation in the equations section for each var.

Pin definitions are global for a system model, meaning that once they are defined in a template, they are accessible by any template below that in the hierarchy. Once you have defined a pin, you cannot give it a different definition.

Standard unit and pin definitions exist in `units.sin`. This file is included in `header.sin`. If you wish to use the standard definitions, you typically need not do anything, because `units.sin` is usually included automatically in the top level of a template via the following mechanism. The `saber` command defaults to the `-la` option, which loads the file `analogy.sld` at the top level of a system. This “saber load” file is a pre-compiled file made from `analogy.sin` using the `saber -p` option. The standard `analogy.sld` file contains `header.sin` which includes `units.sin`.

To define new units or pins or to change definitions, there are several options:

- Change `units.sin` and run `saber -p` to pre-compile it for inclusion under the `saber -la` option.

- ❑ Change `units.sin`, include `header.sin` at the top level of the system hierarchy, and use the `-ln` option of the `saber` command. (Include `header.sin` in a template by writing `<header.sin` where `<` is in the first column.)
- ❑ Write definitions of new pins in the template before they are used in declarations. You can put pin definitions in any of the following places:
  - In the header declarations section
  - In the local declarations sections

Examples

A pin type called electrical has been predefined in units.sin. It defines an across variable of `v`, voltage, and a through variable of `i`, current. Its definition is as follows:

```
pin electrical across v through i
```

The declaration for a pin `x` uses the definition as follows:

```
electrical x
```

This causes the simulator to implicitly declare `i(x)` to be a dependent variable and `v(x)` to be an independent variable.

header

Purpose

The header is declarative. It declares the name of the template, and specifies the type of template, the connection points, and the arguments. It determines how to refer to a template in a netlist statement. It must be included in any template that you intend to call from another template.

Evaluation

Headers are evaluated when the system is first read into the simulator.

Syntax

[type] `template` *template_name connections [= arguments]*

There are two types of templates: the standard template, which does not have a specified type, and the element template, which uses the keyword `element` for type. `Template` is a required keyword that identifies the line as a template header. The *template_name* is the name you have chosen for the template being defined. This is the name used to identify the template in a netlist statement in a template on a lower or the same level in the hierarchy.

Each connection has the form:

connection_point[`:`*internal_node*]

where *connection_point* is the name of a connection point for the temple, and *internal_node,* if present, is a name used as a node in the netlist section of the template.

Connection points can be pins, vars, refs, or states. Connections can be separated by spaces or commas. The equal sign (=) is used only if there are arguments in the template. Each argument is the name of an argument to be passed in through a netlist statement higher in the hierarchy. Arguments must be separated by commas.

Description

The header declares the name of the template, and specifies the type of template, the connection points, and the arguments. It determines how to refer to a template in a netlist statement. Netlist statements are covered in the Netlist section description.

Element Templates

Declaring a template as an `element template` flattens the hierarchy one level. This speeds simulation in cases where there are few or no internal nodes. All basic templates, such as resistors, inductors, and bipolar junction transistors, are defined as element templates.

Template_name

The *template_name* is an arbitrary name for the template. Saber performs an automatic search for templates used in netlist statements. If you want the simulator to find your templates automatically, you must give their files names of the form *template_name*`.sin`. For example, the template name of a resistor is `r`. To ensure that it is found and included automatically, the file containing the template is named `r.sin`.

Connections

Connections can be `pins`, `states`, `refs`, and `vars`. `Pins` are the analog connection points of a modeled device that are used to implicitly declare the across and through variables. For example, the `pins` for a simple resistor could be given the names `p` and `m`. These pins can be referred to indirectly in the resistor template as `v(p)`, `v(m)`, `i(p)`, and `i(m)`. States can be used as discrete-time connection points of a modeled device. They can be digital or event-driven analog. `Refs` are a way to connect to a `var` in another template. When `refs` and `vars` are connected together, there must be exactly one `var`, and there may be 0 or more `refs`. A `var` can be passed out of a template as a connection point. In other templates that have connection points that are connected to the same node, declarations of the same variable must be as `refs`.

Arguments

Arguments are variables which can be passed into a template through a netlist statement higher in the hierarchy. Arguments must be declared in the header declarations section. The way they are declared determines how they can be referred to in other places within the template. Arguments can be altered within the Saber simulator using the `alter` command. Arguments can be simple, composite, or nested composite types, or arrays of these types.

Examples

```
element template r p m = r, tnom
```

This template, `r`, is declared as an element template, so its hierarchy will be flattened for simulation. It has two connection points, `p` and `m`. The simulator cannot tell, from the header, whether they are `pins`, `states`, `refs`, or `vars`. They must be declared in the header declarations section, or used in a netlist statement within the template. The template has two arguments, `r` and `tnom`. The arguments that are declared in the header declaration section determine the way in which information is passed into these arguments.

```
template amplifier inplus inminus out
```

This template, `amplifier`, is a standard template. It has three connection points: `inplus`, `inminus`, and `out`. It has no arguments, so *arguments* and the equals sign (=) are omitted.

header declarations

Purpose

The header declarations section is declarative. It specifies the types of connection point and argument names given in the header.

Evaluation

Header declaration sections are evaluated only once, when the system is first read into the simulator.

Syntax

Connection Points

The forms for the declaration of connection points are:

pin_type	(implied unit)	id, id, ...
`state`	unit	id, id, ...
`ref`	unit	id, id, ...
`var`	unit	id, id, ...

The *id*s are the names of the connection points. The *pin_type* must be previously specified in a pin definition, which may be contained within `units.sin`. The pin declaration, by using the pin definition, implicitly creates declarations of the through variable in the pin definition as a dependent variable and the across variable as an independent variable. The keywords `state`, `ref`, and `var` are needed to declare state, ref, and var connection points, respectively. *Unit* must be previously specified in a unit definition, which may be contained within `units.sin`. States, refs, and vars must be declared with units. (Note that there is a unit defined as nu (no unit) in `units.sin` for unitless states, refs, and vars.) Connection points may not be assigned initial values.

Arguments

The section on Declarations and Data Structures describes the different types of parameters and arguments. There are three simple types: numbers, enumerated types, and strings, and three composite types: structures, unions, and argdefs. Types can also be nested composite types and arrays of any of these types. The basic forms of argument declarations for the three simple and three composite types.

The syntax for declaring simple types is as follows:

`number` *id[=initializer][*, *id[=initializer]...]*
`enum` {*etype[*, *etype...]*} *id[=initializer][*, *id[=initializer]...]*
`string` *id [=initializer][*, *id[=initializer]...]*

The syntax for declaring the three composite types is as follows:

`struc{`
other declarations
} *id [=initializer][*, *id[=initializer]...]*
`union{`
other declarations
} *id [=initializer][*, *id[=initializer]...]*
template_name`..`*argument id[=initializer][*, *id[=initializer]...]*

Keywords include `number` for numbers, `enum` for enumerated types, `string` for strings, `struc` for structure, and `union` for union. The argdef does not have a keyword, but instead refers to a previously-declared *argument* in some template. *Id* refers to the argument being declared. The *initializer* is an optional initial value assigned to the argument. *Etypes* are a listing of enumerated types. *Other declarations* can include initialized composite types.

Description

The header declarations section specifies the types of connection point and argument names given in the header. The way in which these variables are declared not only determines how they can be referred to in the remainder of the template, but also determines the form of a netlist statement that refers to the template.

Connection Points

The Saber simulator checks to ensure that only like kinds of connection points are connected together. This means that electrical pins can be connected only to other electrical pins, and that logic_4 state pins can be connected only to other logic_4 state pins. Refs can have only vars passed as connection points while vars can have only refs passed as connection points.

Connection points do not have to be declared if they are used in a netlist statement in the template. They are implicitly declared from the use in the netlist, and may be used in other places in the template that follow the netlist statement which implicitly defines them. However, for clarity, we suggest that template writers declare each connection point.

Pin declarations implicitly declare the associated across variable as an independent variable, and the associated through variable as a dependent variable. State connections are the means for communicating discrete simulation information into or out of the template. Ref connections are a means for bringing a var from a higher template into a lower one. Vars can be passed out of a template as a connection point. They must be connected to refs.

Arguments

Each argument requires a declaration in the header declarations section. An optional initializer can be used for assigning a default to an argument. Arguments with defaults need not have values assigned to them in netlist statements.

There are three simple and three composite types (refer to Parameter and Argument Declarations on page 3-8). The simple types are as follows:

number Any variable of this type has a numeric value.

enum	Any variable of this type has as its value a member of a list, which is provided as part of the declaration.
string	Any variable of this type has a string variable of a string constant as its value.

Any variable of this type has a string variable of a string constant as its value.The composite types combine two or more other types, so you can refer to them as a unit.

structure	An ordered list of variables, each of which has its own type. Assigning a value to a structure involves assigning a value to each of its components. The members of a structure may either be simple or composite types.
union	A list of variables, each of which has its own type. Assigning a value to a union involves specifying one variable in the list and then assigning a value to that variable. The members may be either simple or composite types. Use unions when you want a choice between variables.
argdef	An argument definition, a variable that is to be of the same type as an argument of some template. Assigning a value to an argdef involves specifying the name of the template and the name of its argument, using the form *templatename*`..`*argument* (e.g., `q..model`) Argdefs are useful because you can define the argument in one place, and then simply refer to the argument name to define another of the same type.

In addition, you can declare an array of any of these six types. The array can either have a fixed number of elements of can be unbounded. The simplest example of an array is an array of numbers. However, it is also possible, for example, to have an array of structures, the elements of which can be any type, including arrays.

Examples

1. The following inductor template has two connection points and one argument. The connection points are declared as electrical `pins`. The argument is declared as a number. Since the argument declaration is not assigned an initial value, the user must specify `l` when referring to this template in a netlist.

```
template ind p m = l
  #declarations of connections
  electrical p,m
  #declarations of arguments
```

```
  number l
```

2. This template, a coupled inductor, has four connection points, which are all refs. Two of them have units of current and two have units of inductance. There are five arguments. The mutual inductance k has no initializer, so it must be specified in a netlist statement. Si1, si2, sl1 and sl2 are all declared as sample points, which are arrays of two numbers: breakpoint and increment. All these arguments are declared with initializers, although the initializers are undefined, so the user has a choice of specifying or not specifying them in a netlist statement.

```
element template
            coupled_ind i1 i2 l1 l2 = k,si1,si2,sl1,sl2
#declarations of connections-----
  ref i i1,i2
  ref l l1,l2
#declarations of arguments-------
  number k
  struc sa_points{
    number breakpoint, increment
  } si1[*]=[()],si2[*]=[()],sl1[*]=[()],sl2[*]=[()]
```

3. This template, trans, has four connection points, which are all specified as electrical pins. It has four arguments, all of which are specified with initializers, so the user would not have to specify any of these when using this template. Model is declared as a structure of 39 numbers, some of which have numeric values and some of which are undefined. Svbe and svbb are declared as arrays of number pairs; area is declared as a number.

```
template trans b e c s = model, svbe, svbb, area
  #declare connection points------
  electrical b,e,c,s

  #declare arguments--------------
  struc{
   number is=1e-16,bf=100,nf=1,vaf,ikf,ise=0,ne=1.5,
    br=1,nr=1,var,ikr,isc=0,rb=0,irb,rbm=0,re=0,rc=0,
    cje=0,vje=.75,mje=.33,tf=0,xtf=0,vtf,ift,ptf=0,
    cjc=0, vjc=.75,mjc=.33,xcjc=1,tr=0,cjs=0,vjs=.75,
    mjs=.33,xtb=0,eg=1.11,xti=3,kf=0,af=1,fc=.5
  } model = ()
  struc sa_points{
   number breakpoint, increment
  }svbe[*]=[(-1k,10),(-10,1),(0,.1),
            (.5,.05),(1,.1),(1k,0)],
   svbb[*] =[(-1k,1),(0,1),(1k,0)]
   number area = 1
```

local declarations

Purpose

The local declarations section is declarative. It specifies variables used locally within the template, that is, variables that are not passed in as connection points or arguments, or that are not otherwise implicitly declared.

Evaluation

The local declarations section is evaluated when the system is first read into the simulator, after the header declarations section. Initializers of parameters are evaluated again after each Saber `alter` command, once for each run of Monte Carlo, and during extraction (`extract` and `ipextract`). Initializers of states are evaluated when setting the initial value to 0.

Syntax and Description

Local declarations can include pins, states, refs, vars, parameters, vals, foreign functions, foreign states, external declarations, groups, templates and, optionally, simvars.

Pins

pin_type   *id*, *id*, ...

`Pin`s declared in the local declarations section are local pins, which can be used to declare internal nodes within a template. The *pin_type* is the identifier used in a previous `pin` definition. The *id*s are the names of the local pins being declared.

States

`state` *unit*   *id[=initializer]*, *id[=initializer]*, ...

`State`s hold information pertinent to discrete time simulation. In the declaration, `state` is a keyword for the local declaration of a state. States require a *unit* declaration, where the unit has been previously specified in a `unit` definition. The *id*s are the names of the states being declared. State variables which are declared locally can be initialized. If a local state variable has a relationship with an analog waveform, it should be initialized to conform with the analog waveform's value of zero (refer to the *MAST Functions* section.)

Foreign States

Foreign `state` declarations are required for digital states that are intended for use by a foreign simulator, that is, in a mixed-mode simulation. Foreign states must be declared in the local declarations section. The syntax is

`foreign state` *units id*

This requires two keywords, `foreign` and `state`. The *units* must have been previously specified in a units definition. The logic_4 unit declaration is typically used for units in mixed-mode simulation. The *id* is the name of the state variable being declared. Foreign states cannot have initializers. More information can be found in the mixed-simulator documentation for the appropriate combination of the Saber simulator with a digital simulator.

Refs

`ref` *unit id*`,`*id*`,`*...*

Refs are references to a `var` in another template. `Ref` is a keyword used in the declaration. The *unit* must have been previously specified in a unit definition. The *id*s are the names of refs being declared. Refs cannot be initialized. The underlying `var` is an independent variable solved for by the simulator.

Vars

`var unit id,id,...`

`Var`s are a type of independent variable for which the simulator solves. There must be an equation in the equation section for each `var` of the form: *id*`:` *expression*=*expression*. In the declaration, `var` is a keyword. The *unit* must have been previously specified in a unit definition. The *id*s are the names of the `var`s being declared. Since the simulator solves for `var`s, they cannot be initialized by a template.

Parameters

The forms for parameter declarations are covered in Parameter Types on page 3-9. There are three simple types (number, enumerated, string) and three composite types (structures, unions, argdefs). Types can also be nested composite types and arrays of any of these types. The syntax for declaring simple types is as follows:

`number` *id[=initializer][*`,`*id[=initializer]...]*
`enum` {*etype[*`,`*etype...]*} *id[=initializer][*`,`*id[=initializer]...]*
`string` *id [=initializer][*`,`*id[=initializer]...]*

The syntax for declaring the three composite types is as follows:

local declarations

1. `struc{`
   other declarations
   `}` *id [=initializer][*`,` *id[=initializer]...]*

2. `union{`
   other declarations
   `}` *id [=initializer][*`,` *id[=initializer]...]*

3. *template_name*`..`*argument id[=initializer][*`,` *id[=initializer]...]*

Keywords include `number` for numbers, `enum` for enumerated types, `string` for strings, `struc` for structure, and `union` for union. The *argdef* does not have a keyword, but instead refers to a previously declared argument in some template. *Id* refers to the parameter being declared. The initializer is an optional initial value assigned to the *id*. *Etypes* are a listing of enumerated types. *Other declarations* can include composite types.

Values

`val` *unit id*`,`*id*

`Val`s are used to hold temporary information during simulation, and to supply information not otherwise available during post-simulation processing.

The keyword is `val`. The *unit* must have been previously specified in a unit definition. The Saber simulator does not check the `val` unit type for consistency. The *id*s are the names of the `val`s being declared. `Val`s cannot be initialized; their values are supplied by the simulator.

Foreign Functions

There are two types of foreign functions: those that are declared to return only a single number, and those that are not restricted in what they return. Those that are declared to return a single number may be used as part of a valid expression, whereas those that are not restricted can be used only in an assignment statement.

`foreign number` *subroutinename*`()`

This is the declaration for a foreign subroutine which returns only a single number. It requires two keywords: `foreign` and `number`. The keyword is followed by the chosen subroutine name and a pair of empty parentheses.

`foreign` *subroutinename*

This is the declaration for a foreign subroutine that is not restricted in what it returns. It uses the keyword, `foreign`, and the chosen *subroutinename*. Having declared either of these foreign subroutine types, you may write the routines and use them as described in the chapter on *Foreign Functions*.

External Declarations

Variables declared as `external` get their values from the declaration of that variable in some higher template in the hierarchy of the system being modeled. For example, if temperature is declared at the top level, and all templates using temperature declare it as external, then the temperature need only be changed at the top level to affect all the templates in the system. Parameters, `pin`s, and `var`s can be declared as external.

`external` *type id*[`,` *id...*]

This declaration uses a keyword of external. *Type* can be a parameter, a `var`, or a `pin`. *Id* refers to the variable being declared.

When a variable is declared to be external, the search for its value proceeds up the hierarchical path of template references.

Groups

Group declarations group variables together for extraction and other purposes.

`group {`*member*`,` *member*`,`*...*`}` *id*

This declaration uses the keyword `group`. The members are the names of variables or other groups in the group, and *id* is the name of the group being declared. Groups must be homogeneous. That is, they must consist entirely of arguments and parameters or entirely of system variables, `val`s, and `simvar`s.

Template Declarations

[**`element`**] **`template`** *templatename connection_points* [**`=`***arguments*]
[header declarations]
{
[local declarations]
[parameters section]
[when statements]
[netlist]
[values section]
[control_section]
[equations section]
}

Entire templates can be declared in the local declarations area. They need different sections based upon their modeling function. Since templates declared in a local declarations area will not be top-level templates, they need headers, and will be referred to by netlist statements that must follow the template declaration. Any templates declared in the local declarations template are local to the current template, meaning they are not recognized in higher-level templates.

Simvars

`simvar` *simvarname*

It is optional to declare `simvars`. If you wish to declare them, use `simvar` as a keyword and give the *simvarname* exactly as it is known to the Saber simulator. If you declare a *simvarname* as something other than a `simvar`, e.g., `number time`, then you will not be able to use that `simvar` in the template.

Examples

The following are examples of local declarations (the # sign indicates a comment, which is ignored by the simulator):

Pins

```
#declare internal pins for transistor model --------------
electrical bp,ep,cp
```

States

```
#declare internal state ---------------------------------
state logic_4 notify
```

Foreign States

```
#declare foreign state, dout, to be used with foreign ----
#simulator
foreign state logic_4 dout
```

Refs

```
#declare ref --------------------------------------------
ref l inductance
```

Vars

```
#declare var --------------------------------------------
var i current
```

Parameters

```
#declare work array of 33 numbers -----------------------
number work[33]
```

```
#declare enumerated type of n or p channel --------------
enum{n_channel,p_channel}type

#declare coretype as string, and initialize to null string
string coretype=""

#declare structure of sample points and initialize -------
struc{
  number breakpoint, increment
}svbe = [(-100,1),(0,.1),(100,0)]

#declare union and initialize to off --------------------
union{ number   off
      struc {number vo=0,va,f,td,theta;} sin
}tran=(off=1)
#use argdef to declare local parameter based on bjt ------
#argument "model"
bjt..model localmodel
```

Vals

```
#declare val with units of p, for power -----------------
val p power
```

Foreign Functions

```
#declare foreign function as number --------------------
foreign number deg2rad()

#declare foreign function without restricted output type -
foreign foo
```

External Declarations

```
#declare external parameter of temperature, named "temp" -
external number temp

#declare external argdef from bjt, and name it lateral --
external bjt..model lateral

#declare external pins ----------------------------------
external electrical vc, vee, signal_ground
```

Groups

```
#group bjt voltages for extraction ---------------------
group {vbe,vbc,vce,vbei,vbci,vsi,vbx,vbb} v
```

Templates

```
#declare local template --------------------------------
template localres p m = lres
  electrical p,m
  number lres=10k{
    equations{i(p->m) += (v(p)-v(m))/lres
  }
}
```

Simvars

```
#declare simvar time -----------------------------------
simvar time
```

Parameters section

Purpose

The Parameters section is an operational section used to manipulate parameters. It can be used to “bullet-proof” templates by testing the input values of arguments for validity, to model complex distributions for Monte Carlo analyses, and to perform intermediate operations on parameters to speed up simulation time.

Evaluation

The expressions in the Parameters section are evaluated once just after the input file is read into the simulator, again after each Saber `alter` command, and once for each run of a Monte Carlo simulation. The Parameters section is evaluated from top to bottom, without interruption, in the same way that a subroutine is evaluated.

Syntax

parameters{
  statements
}

The Parameters section uses the keyword of `parameters` followed by a left brace ({). Following the *statements*, the section is terminated with a right brace (}).

Assignment statements, if statements, and foreign functions are allowed in the Parameters section. Intrinsic functions (with the exception of `d_by_dt` and `delay`) are also available. Only parameters can appear on the left-hand side of assignment statements and in the output list for foreign functions within the Parameters section. Another limitation is that you may use only parameters, arguments, and constants in statements in the Parameters section. In addition, you may use expressions and messages in the Parameters section. The only `simvar` you can use in the Parameters section is `statistical`.

Description

The Parameters section is used mainly for transforming values in arguments (which are meaningful to the user of the simulator) to values used by the simulator (which are meaningful to the designer of the template). It can be used to speed up simulation through including repetitious mathematical calculations, to “error-proof” a template by testing the input values of arguments for validity, to allow alternative values of parameters, to model complex distributions for Monte Carlo analyses, and to perform intermediate operations on parameters to speed up simulation time.

Parameters may depend only on other parameters, on arguments, and on constants: no `var`s, `ref`s, `val`s, or `simvar`s (other than `statistical`) may appear in the Parameters section. Only parameters can be on the left-hand side of assignment statements. Only parameters may be returned from foreign functions used in the Parameters section. The Parameters section can include assignment statements, foreign functions, If statements and messages. Intrinsic functions, with the exception of `d_by_dt` and `delay`, can be used in this section.

Examples

```
parameters{
  if(model->type==_n){
    p=1
  }
  else if(model->type==_p){
    p=-1
  }
  rb = area * model->rb
```

```
    work = spq(1,model,rb,temperature)
    message("work=%",work)
}
```

In this example, If statements are used to determine a parameter `p` based on the model `type`. Then a parameter `rb` is found from the two arguments area and `model->rb`. A foreign subroutine, `spq`, is called with an input list of `1`, `model`, `rb`, and `temperature`. The results are put into the array parameter `work` and are printed to the screen and to the `.out` file using the `message` function.

Netlist section

Purpose

The Netlist section calls other templates and specifies their arguments.

Evaluation

This section is loaded during pre-processing (after the declarations), after an `alter` command, for each run of a Monte Carlo analysis, and during extraction, just as for the Parameters section.

Syntax

templatename`.`*refdes connection_pt_list [=argument_assignments]*

The *templatename* is the name by which the template is identified in its header. The *refdes* is the reference designator, which is a unique name that distinguishes this reference to this template from all other such references. The *connection_pt_list* is a space-separated list of nodes in the system to which the connection points of the template are joined. The *argument_assignments* is a comma-separated list of assignments of values to the arguments of the templates. The most general format is *argument=assignment*.

Description

The Netlist section contains references (called netlist entries;) to templates that have already been defined. The netlist components section is required only in templates that refer to other templates. In fact, the presence or absence of this section determines whether there is hierarchy below the current template.

The *refdes* must be unique among all reference designators for each component of type templatename. For example, in a given circuit, an npn transistor may be labeled `q1`. If this transistor is described by the template named `npn`, the *templatename*`.`*refdes* for this element would be `npn.q1`. Then `npn.q2` would refer to a different instance of the same transistor model implemented by `npn`. However, the netlist could not have more than one transistor named `npn.q1` on any hierarchical level.

The *connection_pt_list* is a space-separated list of nodes in the system to which the connection points of the template are joined. Each connection point specified in a template must be assigned to a node. One or more connection points may be connected to each node. Nodes can be specified in two ways: either the nodes associated with each connection point can be listed in order of association, or the colon convention can be used. In the colon convention, the connection point name is listed, followed by a colon (:), and then the node name. Connection point assignments using the colon convention can occur in any order.

Specification of arguments can be very complicated, as seen from the examples . Argument names (and their equals signs, =) are not needed if the arguments are simple types (refer to Parameter and Argument Declarations on page 3-8) and are specified in the order they appear in the template header. The names for composite types must be specified, and each structure and union must be specified using a set of parentheses, while each array specification must be enclosed within brackets (`[ ]`). To change one or more of the argument fields in a structure, all arguments to be changed are enclosed in a single set of parentheses (( )), and are assigned to the major parameter name with an equal sign (=).

The values assigned to arguments can be numbers or parameters. Values need to be specified for arguments if they have no initializers specified in the argument declarations. If initializers are specified, values specified in the *argument_list* will over-write them.

You can also assign distributions to arguments that will be used during Monte Carlo analyses.

Examples

The following example uses the template `v` from the MAST Template Library to illustrate argument declarations and how they can be assigned in a netlist to other template instances:

```
element template v p m = dc,tran,ac
electrical p,m
number dc=0
union {
  number off
  struc {number vo=0,va,f,td,theta;}      sin
  struc {number v1,v2,td,tr,tf,pw,per;}   pulse
  struc {number v1,v2,td1,tau1,td2,tau2;} exp
  struc {number vo,va,fc,mdi,fs;}         sffm
  number  pwl[*]
  number   ppwl[*]
  struc {
    number v1,v2,period,rtime,width,ftime,delay,sdelay;
  }clock
}tran=(off=1)
struc {number mag=0,phase=0}              ac=(0,0)
v.v1 p:a m:0 = dc=5
v.v2 b 0 = 5
v.v3 c 0 = vcc
v.v4 d 0 = tran=(sin=(va=1,f=10k,td=0,theta=0))
v.v5 e 0 = tran=(sin=(0,1,10k,0,0))
v.v6 f 0 = tran=(pwl=[0,0,10n,0,11n,5,20n,5,21,0])
v.v7 g 0 = tran=(sin=(0,1,10k,0,0)),ac=(mag=1,phase=0)
```

The *connection_pt_list* for `v.v1` is specified using the colon convention. `P` and `m` are the pin names in the template; `a` and `0` (ground) are the nodes to which these are joined. The `p:a` and `m:0` can occur in the opposite order without affecting the polarity of the voltage source.

The *connection_pt_list* for `v.v2` is specified by noting the node names in the same order as the pins in the template. Thus, `b` would be connected to `p`, and `0` would be connected to `m`. The phrase `dc=` does not have to be used to specify the DC value because `dc` is the first argument specified in the template header and is a simple number.

In `v.v3`, the parameter `vcc` (which was presumably previously defined) is used instead of specifying a number for `dc`.

In `v.v4`, the first set of parentheses is used for the `tran` union. The second is used for the sin structure inside the `tran` union. In this example, there is no value specified for `vo`, which was specified with an initializer, so the names of all the succeeding argument fields must be listed. Because they are named, they can occur in any order.

In `v.v5`, the arguments for the `sin` structure are not named, so they will be assigned in the order in which they are listed in the argument declaration. In this case, the value of vo must be specified because it precedes the other arguments of the `sin` structure.

In v.v6, a piece-wise linear voltage source is specified. It is declared as a variable-length array of numbers. The set of parentheses is used when specifying the tran union, while brackets ([ ]) specify the pwl array.

In v.v7, both tran and ac arguments are specified, while dc is not.

When statements

Purpose

When statements are operational sections. These are used to perform discrete time simulation, to describe digital behavior, to test for analog waveforms crossing a threshold, and to schedule events and times.

Evaluation

The conditions for When statements are monitored, where the frequency of monitoring depends upon the specific condition. When a condition is met, the statements within the body of the When statement are evaluated in order, from top to bottom, as in a subroutine.

Syntax

```
when(condition){
 statements
}
```

The When statement uses the when keyword. The when(*condition*) must be followed by a brace, as in the Parameters, Values, Control, and Equations sections. The *condition* is a logical expression involving one or more of the intrinsic functions named threshold and event_on or simvars such as dc_done and time_step_done. The statements are a collection of one of more statements, usually with one or more being calls to the intrinsic scheduling functions such as schedule_event, schedule_next_time, and deschedule.

Unlike most of the other sections, there may be as many When statements as needed to model the device.

Assignment statements, if statements, foreign functions, expressions and messages can be used within When statements. Intrinsic functions (with the exception of d_by_dt and delay) and are also available. Only states can appear on the left-hand side (LHS) of assignment statements and in the output list for foreign functions within When statements. Other than this limitation, simvars, across variables, vars, refs, vals, states, parameters and arguments can be used in statements in When statements.

Description

When statements are used to perform discrete time simulation, to describe digital behavior, to test for analog waveforms crossing a threshold, and to schedule events and times. for more information on the conditions and statements used with When statements, refer to the chapter on *MAST Functions*.

The simulator processes analog and digital events separately, but communicates between them using the following functions:

- An analog waveform may cause a digital event through the `threshold` function.
- A digital event may cause an analog reaction through the `schedule_next_time` function.

Examples

This example is a .digital clock. It has a digital output pin named `out`. The state of this pin is not be reported to the template because, as an output, its state may depend on other states connected to its node. For this reason, the simulator uses a local state variable, `wake_up`, to relay the information about out to the template.

There are three When statements in the example.

1. `when(dc_init)` is used to set the clock to a high state during DC domain analyses.
2. `when(time_init)` is used to schedule the first occurrence of the local state variable.
3. `when(event_on(wake_up))` provides the changes of state at the out pin and to schedule the next wake_up event.

```
template clock out = hightime, lowtime
# this template models a square wave whose duty cycle
# can be input using hightime and lowtime

  state logic_4 out
  number hightime    # time in seconds when signal is high
  number lowtime     # time in seconds when signal is low
{
  state nu wake_up
  when(dc_init){
    schedule_event(time,out,l4_1)
```

```
  }
  when(time_init){
    schedule_event(time,wake_up,0)
  }
  when(event_on(wake_up){
    schedule_event(time+hightime,out,l4_0)
    schedule_event(time+hightime+lowtime,out,l4_1)
    schedule_event(time+hightime+lowtime,wake_up,0)
  }
}
```

Values section

Purpose

The Values section is both a declarative and an operational section. It is used to set up `vals` for extraction, to handle foreign functions needed for the Equations sections, to describe noise sources, and to provide clarity in the Equations section.

Evaluation

Simulator evaluations after the initial reading are optimized, so that only statements needing further consideration are evaluated. Templates that are time-dependent are evaluated at least once per time step. Nonlinear portions of templates are evaluated during nonlinear iterations, if needed. After simulation, `vals` can be extracted from a data file. They are evaluated only as needed. Certain information about dependencies of `vals` on system variables is discovered during compilation; this is the “declarative” nature of the section.

Syntax

`values {`

statements

}

The Values section uses the `values` keyword, followed by a left brace ({), which encloses the body of the Values section.

Assignment statements, If statements, and foreign functions are allowed in the Values section. Intrinsic functions (with the exception of `d_by_dt` and `delay`) and are also available. Only `vals` (and the `simvars` named `next_time` and `step_size`) can appear on the left-hand side of assignment statements and in the output list for foreign functions within the Values section. Other than this limitation, simvars, across variables, `vars`, `refs`, `vals`, `states`, parameters and arguments can be used in *statements* in the Values section. Expressions and messages cannot be gainfully used in the Values section because statements are evaluated only when necessary, and output statements are generally discarded.

Description

The Values section sets up `vals` for extraction, handles foreign functions needed for the Equations section, and promotes clarity in the Equations section. The Values section can also be used to assign values to the `simvars` named `next_time` and `step_size`, although the use of `schedule_next_time` in When statements is more efficient because it does not have to be reset for each time step.

All `vals` can be extracted from the data file after simulation, although they are not stored there. They can be extracted and put into the plot file using the Saber `extract` command. `Vals` can be grouped so that several may be easily extracted at once.

Foreign functions that are not declared to return a single number cannot be used in the Equations section. Therefore, any other manipulation required of a foreign subroutine for the Equations section must be done in the Values section.

Noise sources, to be used with the Saber small-signal noise analysis, are defined in the Values section. `Vals` can also be used when it would be clearer to declare and define intermediate variables in your templates.

Examples

1. `Vals` can be used to define values useful for extraction. The above voltages can be extracted using the group `v`, which was declared in the local declarations section, or individually.

```
#in the local declarations section
val v vbe,vbc,vce,vbei,vbci,vsi,vbx,vbb
group {vbe,vbc,vce,vbei,vbci,vsi,vbx,vbb}v

#in the values section
vbe = v(b) - v(e)
vbc = v(b) - v(c)
vce = v(c) - v(e)
vbei = v(bp) - v(ep)
vbci = v(bp) - v(cp)
vbx = v(b) - v(cp)
```

```
vbb = v(b) - v(bp)
```

2. The next example shows a foreign function, `diodesub`, being used in the Values section to find `vals` `idi` and `qd`, which are then used in the Equations section.

```
# pins and arguments in header declarations section
electrical p,n
struc{
  number  is,rs,n,tt,cjo,vj,m,eg,xti,kf,af,
          fc,bv,ibv,tnom,gmin,reltol
}model=()
number area=1

# in local declarations section
electrical pi
val i idi,id
val q qd
val v vdi,vres
number work[17]
external number temp
foreign diodesub

# in values section
vdi = v(pi) - v(n)
vres = v(p) - v(pi)
id = vres/model->rs
(idi,qd)=diodesub(work,model,temp,area,vdi)

# in equations section
i(p) += id
i(pi) += idi + d_by_dt(qd) -id
i(n) -= idi + d_by_dt(qd)
```

3. The following example is a noise source. This example models six noise sources. They include pairs that represent the constant and the frequency-varying portions of noise for a voltage noise source and two current noise sources. The `vals` for the noise sources are declared in the local declarations section, described and evaluated in the Values section, and then specified as noise sources in the Control section.

```
#in local declarations section
val nv nsv,nsvf
val ni nsim, nsip, nsipf, nsimf
var i i
```

```
#in values section
nsv = 20n  nsip = .71  nsim = .71  p
if (freq ~= 0.0) {
  nsvf = 200n/freq
  nsipf = 70p/freq
nsimf = 70p/freq
}
else {
  nsvf = 0.0
  nsipf = 0.0
  nsimf = 0.0
}

#in control_section
noise_source (nsv,i)
noise_source (nsvf,i)
noise_source (nsim,n2)
noise_source (nsimf,n2)
noise_source (nsip,n3)
noise_source (nsipf,n3)
```

Control section

Purpose

The Control section is a declarative section. It is used for five specialized functions:

1. Collapsing nodes.
2. Declaring dependencies between dependent nonlinear variables and independent nonlinear variables for some templates.
3. Declaring sample points for some independent variables.
4. Limiting the step size in Newton-Raphson iterations for some types of independent variables.
5. Specifying noise sources.

Evaluation

The different types of statements in the Control section are evaluated after everything else is read in. The statements are consulted as necessary.

Syntax

```
control_section{
   statements
}
```

The Control section uses the keyword `control_section` followed by the left-hand brace ({). Statements in the Control section can be the following:

```
collapse(node1,node2)
noise_source(val, pin_or_var[, pin_or_var])
sample_points(variable, sapoints)
sample_points((variable, variable...), sapoints)
pl_set((dep_id[,dep_id...]),(indep_id[,indep_id...]))
newton_step(variable, nsteps)
newton_step((variable, variable...), nsteps)
dc_help (node1, node2)
```

The `collapse` statement collapses two nodes, *node1* and *node2*. This is used to speed up simulation in cases where, for instance, there is no resistance specified between two nodes.

The `noise_source` statement applies to the small-signal noise analysis simulation. The `val` is the name of a noise source declared as a `val` and described by an expression in the values section of the template. The *pin_or_var* is a reference to a `pin` or a `var`.

A `sample_points` statement is required for each independent variable in nonlinear functions. *Variable*, a `var` or `ref`, is the nonlinear independent variable, while *sapoints* represents an array that holds the sample points for the variable. If there is more than one nonlinear independent variable that has the same set of sample points, a list of the variables, in parentheses, is substituted for variable.

The `pl_set` statement declares which dependent variables depend upon which nonlinear independent variables for the purposes of piece-wise linear evaluation. The *dep_ids* are the nonlinear dependent variables, while the *indep_ids* are `var`s or `ref`s or the differences between `var`s and/or `ref`s. There can be as many *dep_ids* and *indep_ids* as needed. The parentheses around the *dep_ids* and the *indep_ids* are necessary only if more than one *dep_id* or *indep_id* is given.

The `newton_step` statement limits the step size for Newton-Raphson iterations. This limitation is needed for some nonlinear functions that have rapidly changing slopes. *Variable* identifies a single, linear independent variable (or a list of them in parentheses), and nsteps represents an array of ordered pairs containing breakpoints and increments.

The `dc_help` statement specifies two template connection points that will be affected by Gmin_Ramping selections found in SaberGuide by traversing the following path:

4. In the Operating Point Analysis form
(Analyses > Operating Point > DC Operating Point...)
5. Under the Algorithm Selection tab
6. Under the Ramping Algorithm Settings... button
7. In the Algorithm Selection for dcanalysis form.

This provides a minimum conductance (leakage path) between the specified connection points that is gradually reduced as a solution is approached.

Description

Collapsing Nodes

Under some conditions nodes may be collapsed to speed up simulation. For example, if you set the ohmic collector resistance of a bipolar transistor to zero, you can create a statement that collapses the internal and external nodes. You cannot collapse or “uncollapse” nodes using the Saber simulator’s `alter` command, because to do so would change the topology of the system being simulated.

Noise Sources

If the noise source is a through variable, then the specification should be

`noise_source(`*val_name*`,`*pin[*`,`*pin]*`)`

where the *val_name* is the name of the noise source (which must be declared as a `val`), and the pins are the names of the pins or internal nodes it is connected between. If it is connected between a pin and ground, then the ground pin, 0, need not be specified.

If the noise source is an across variable, then the specification is:

`noise_source(`*val_name*`, `*var_name*`)`

where the *val_name* is the name of the noise source and the *var_name* is the name of the `var` that defines the noise source, appearing in the Equations section as: *var_name*`:` *expression* = *expression*.

Appropriate units are defined in the file *units.sin* for noise voltage and current sources.

Sample Points

There are several techniques used by simulators for linearization. Of these, three are described as follows:

- Taking the slope of the characteristic at the guessed point

  This technique is used by some simulators. Because the guess is not known beforehand, you must provide the simulator with both the characteristic equations and their slopes. This usually results in a requirement that the slopes are continuous, thus disallowing step functions.

- Piece-wise linear approximation

  This technique depends on the model itself being composed of piece-wise linear segments between selected points. The simulator does not linearize because the model itself is linearized. The accuracy of this model depends entirely on the selection of the points in the model, and cannot be changed without changing the points in the model.

- Piece-wise linear evaluation

  This technique is the one the Saber simulator uses, and is distinct from piece-wise linear approximation. In this method, the model describes the nonlinear characteristics, but the simulator approximates the curve at certain points with a set of straight lines. These points apply to the independent variables, and are called sample points. For this technique the simulator needs the characteristic equations and associated sample points.

In general, there is a trade-off between accuracy and speed. For accuracy, you should specify sample points close together where the rate of change of the characteristic is large. They can be further apart where the rate of change is small.

Sample points should be defined as pairs of numbers in an array. They can be arguments or parameters.

```
  struc{
    number breakpoint, increment
}sample_array = [*]
```

The first number in each pair, the *breakpoint*, is the value of the independent variable that is the starting point for the *increment*. The increment is the spacing at which sample points are to be taken until the next breakpoint. The number 0 must be one of the breakpoints. Values assigned to sample points may be parameterized, that is, made dependent on some parameters and assigned in the Parameters section. The last *breakpoint/increment* pair tells where sample point specification stops, by specifying the last increment as 0.

The syntax for describing sample points in the Control section needs the name of the independent variable, and the name of the sample point array, or the array itself. If there is more than one nonlinear independent variable that has the same set of sample points, the variable name may be replaced by a group name, where the group is declared in the local declarations area.

NOTE
The Saber simulator will provide default sample points for any nonlinear template that does not specify them in a Control section.

Logsap

The subroutine `logsap` was created to help generate *logarithmic* sample points. You can declare this as a foreign subroutine in the local declarations section and call it in the Parameters section, according to the following format:

```
logsap = (min, max, step_per_decade,
      density_per_decade, [sample_points])
```

where:

`min` = absolute value of minimum breakpoints (symmetric about 0)

`max` = absolute value of maximum breakpoints (symmetric about 0)

`step_per_decade` = logarithmic spacing of breakpoints within a decade (must be greater than zero)

`density_per_decade` = used to calculate the increment between specified breakpoints (`increment(i)= breakpoint (i+1)- breakpoint(i)/density`)

Logsap Examples

1. `logsap (1u,1meg,1,x)` would yield the following breakpoints (disregarding increments):

```
  -1e6, -1e5, -1e4, -1e3, -1e2, -10, -1, -1e-1, -1e-2,
-1e3, -1e-4, -1e-5, -1e-6, 0, 1e-6,...,1e6
```

2. `logsap (1u,1meg,3,x)` would yield the following breakpoints (again disregarding increments):

```
  -1e6, -4.641e5, -2.154e5, -1e5, -4.641e4, -2.154e4,
-1e4,...0,  1e-6,-4.641e-5, -2.154e-5,...,1e6
```

3. `logsap (1u,1meg,0.5,x)` would yield the following breakpoints (again disregarding increments):

```
-1e6, -1e4, -1e2, -1, -1e-2, -1e-4, -1e-6, 0, 1e-6,...,1e6
```

4. `logsap (1u,1meg,1,90)` would yield the following complete sample point specification, including increments:

```
(-1e6,1e4) (-1e5,1e3), (-1e4,1e2), (-1e3,1e1),
(-1e2,1e0), (-1e1,1e-1), (-1e0,1e-2), (-1e-1,1e-3),
(-1e-2,1e-4), (-1e3,1e-5), (-1e-4,1e-6), (-1e-5,1e-7),
(-1e-6,1e-8), (0,1e-8), (1e-6,1e-7), (1e-5,1e-6),
(1e-4,1e-5), (1e-3,1e-4), (1e-2,1e-3), (1e-1,1e-2),
(1e0,1e-1), (1e1,1e0), (1e2,1e1), (1e3,1e2), (1e4,1e3),
(1e5,1e4), (1e6,0)
```

Pl_Set

The “piece-wise linear set” specification defines dependencies of nonlinear variables on linear combinations of system variables. It is rarely required, because the Saber simulator can find these dependencies, but its use is suggested because it defines what can be extracted from a distortion analysis, and ensures that dependent variables remain dependent, even when they can be optimized to be independent variables. (Independent variables may not be changed with the Saber `alter` command.)

If you have not specified a `pl_set` statement, and you have such dependencies, you can find out what the Saber simulator is using as a `pl_set` by using the `saber -d pl_set` option. This option displays the `pl_set` to the screen and places it in the `.out` file, after the files are read in, as the topology is being analyzed.

Newton Steps

Newton steps give a way of limiting the size of steps taken in the Newton-Raphson algorithm. Some nonlinear systems, such as systems with exponential characteristics, require them to reduce the time the simulator takes for convergence. However, if they are not needed they slow simulation, so you should avoid using them unless you know that they are needed. A recommended approach is to model without Newton steps and then add them to the model if convergence takes excessive time (or if too many time-step iterations prevents convergence).

As with sample points, you specify Newton steps using breakpoint-increment pairs. Increments should be small when the slope of a function is large, and large when the slope is small. Newton steps need cover only the part of a function where step-size limitation is helpful. You can terminate a Newton step sequence at any point by specifying an increment of 0.

DC_help

The `dc_help` statement specifies two template connection points that will be affected by GMIN_Ramping selections found in SaberGuide by traversing the following path:

1. In the Operating Point Analysis form (Analyses > Operating Point > DC Operating Point...)
2. Under the Algorithm Selection tab
3. Under the Ramping Algorithm Settings... button
4. In the Algorithm Selection for dcanalysis form.

It is used for models with connection points that appear as an open circuit at DC (e.g., a MOSFET), thus causing the simulation not to converge.

When GMIN_Ramping is selected, the Saber simulator inserts a leakage path across the specified connction points. The conductances of these paths are then gradually reduced as a solution is approached. You can specify the beginning and ending values and the rate of reduction in the appropriate tab.

Examples

Collapse Nodes

```
   #in control section
if(model->rb <= 0) collapse (b,bp)
```

If the parameter `rb` in `model` is less than or equal to 0, the two nodes `b` and `bp` are collapsed.

Noise Source

```
   #header declarations
   electrical p,m
   #local declarations
   val ni nsr
   #in control section
noise_source(nsr,p,m)
```

`Nsr` is a current noise source attached between pins `p` and `m`. It is assigned a value in the Values section.

```
   #local declarations
   val nv nsv
   var i i
```

```
   #in control_section
   noise_source(nsv,i)
   #in equations section
i: v(a)-v(m)=0
```

`Nsv` is a voltage noise source dependent upon `i`, a `var` that is defined such that the voltage between nodes `a` and `m` is 0.

Sample points

```
   #in local declarations section
   struc{
     number point, increment
   }sp[*]=[(-100,10),(-10,1),(0,.2),(10,0)]
   val v vd
   #in control section   sample_points(vd,sp)
   #or, simplifying,
#sample_points(vd,[-100,10,-10,1,0,.2,10,0])
```

Pl_set

5. In this example, `id` is declared to be a function of the independent variable `vd`.

```
pl_set(id,vd)
```

6. In the second example, `idi` and `qd` are declared to be dependent upon `vdi`.

```
pl_set((idi,qd),vdi)
```

Newton steps

In this example, the `val` named `vdi` is given Newton steps from 0.2 to 1 volts. At 0.2 volts the Newton step is limited to 20mV, at 0.6 volts the Newton step is limited to 1 mV, and after 1 volt the Newton step is turned off. This would indicate that `vdi` had a sharp rise in slope between 0.2 and 1 volts, and then settled out again.

```
   #in local declarations section
   struc{
     number breakpoint, increment
   }nv=[(0.2,20m),(0.6,1m),(1,0)]
   val v vdi

   #in control section
   newton_step(vdi,nv)
#or    newton_step(vdi,[0.2,20m, 0.6,1m, 1,0])
```

DC_help

```
control_section{
  dc_help (gate, drain)
}
```

Equations section

Purpose

The Equations section is for describing the analog characteristics at the terminals of the element being modeled. Statements in the Equations section either define the dependent through `vars` or `refs` in the system in terms of the across variables or other variables of the system, or the equations necessary for each `var` declared in the template.

Evaluation

The Equations section is evaluated when the system is read into the simulator, to make a "system matrix." The system matrix is evaluated, as needed, throughout simulation.

Syntax

equations{
    statements
}

The Equations section uses the keyword equations followed by a left-hand brace ({). Four types of syntax are allowed in the Equations section. The syntax for the first two are:

through_variable(*pin_name*) *operator expression*
through_variable(*pin_name* -> *pin_name*) *operator expression*

Through variables are declared implicitly by the pin declaration. *Pin_names* are the names of pins used in the template. In the second case, the symbol "->" indicates a flow of the through variable from the first *pin_name* to the second. Operators permitted are += and -=, which mean to add to or subtract from the node, respectively. *Expression* is any valid expression. It can contain any intrinsic functions, including `d_by_dt` and `delay`, with the limitations that `d_by_dt` and `delay` cannot be nested and the only binary operators are + and -.

The syntax for the third kind of statement is the following:

```
ref_variable operator expression
```

The *ref_variable* is a `ref`, that is, a `var` passed in from another template. Operators permitted are += and -=, which, respectively, mean to add to or subtract from the `var`. Expression is any valid expression. It can contain all intrinsic functions, including `d_by_dt` and `delay`, with the limitations that `d_by_dt` and `delay` cannot be nested and the only binary operators are + and -.

The syntax of the fourth kind of statements is the following:

var_variable : *expression* = *expression*

The *var_variable* is the name of a `var`. The *expression* = *expression* is an equation, where the simulator is to determine a value of the `var` such that the equation can be true. The `var` must be declared in the header or local declarations sections.

If statements whose conditions depend on parameters and arguments, and foreign functions that are declared as numbers are allowed in the Equations section.

Description

The Equations section is used to describe the analog characteristics at the terminals of the element being modeled. Statements in the equations section either define the dependent through `vars` or `refs` in the system in terms of the variables of the system, or the equations necessary for each `var` variable declared in the template.

Expressions

Expressions used in the Equations section can contain all intrinsic mathematical functions, including `d_by_dt` and `delay`, with the restrictions that `d_by_dt` and `delay` cannot be nested and the only binary operators permitted are + and -. If the argument of either of these functions has a constant multiplier, it must appear within the argument, rather than as a multiplicand. Thus, you can write `d_by_dt(3*xo)`, but not `3*d_by_dt(xo)`.

Refs

`Refs` may or may not need an equation in the Equations section. This depends on whether the ref in the template contributes to its referenced `var`. When `refs` are used in the Equations section, they are summed into the equation for the referenced `var`. Therefore, the signs of the operators are opposite of what they may intuitively seem, because they are summed into the left-hand-side of the originating `var` equation, rather than on the right-hand-side.

Vars

There are three reasons to use a `var`. The first is when the through variable is not a function of the across variable, as in a voltage source or in an inductor. (The MAST language offers only derivatives, not integrals, so the equation for an inductor cannot be written in one of the two forms for the through variables. Therefore, we declare the current as a `var`, and express the equation in the fourth form so we can take the derivative of current.)

The second use of `vars` is to add them as needed to use the `d_by_dt` and `delay` functions appropriately. These cannot be nested, so to take a second derivative, for instance, we must declare the first derivative as a `var`, and write an equation for it, and then take the derivative of the first derivative.

The third use of a `var` is to declare a variable which will be used as a `ref` in another template.

Examples

1. The first example shows an equation section for a simple resistor.

```
#in header declarations section
electrical p,m
number resistance
#in equations section
i(p->m) += (v(p)-v(m))/resistance
```

   This could also be written in this way:

```
#in equations section
i(p) += (v(p)-v(m))/resistance
i(m) -= (v(p)-v(m))/resistance
```

2. The second example shows equations based on refs in the Equations section. Note that the signs on the operators are -= instead of += because `refs` are summed into the left-hand-side of the `var` equation rather than the right-hand-side.

```
template ml i1 i2 = m
ref i i1,i2
number m{
  equations{
    i1 -= d_by_dt(m*i2)
    i2 -= d_by_dt(m*i1)
  }
}
```

3. The third example shows a `var` used in taking the second derivative of a function with respect to time. This models a four-port behavioral

model for a second-order differential equation. Note that an intermediate `var` named `dvo` is used to avoid nesting the `d_by_dt` function.

```
#in equations section
i(vop->vom) += i
dvo: dvo=d_by_dt(vout)
i: x1*vin = x2*vout + x3*dvo + d_by_dt(x4*dvo)
```

Description

chapter 8

Foreign Functions

Introduction

The MAST modeling language lets you use subroutines, called foreign functions or foreign subroutines that are outside of all templates. These can be especially useful when a lot of mathematical manipulation is required.

You can write foreign functions in most high-level languages, but they must have a special interface because they are not called directly from MAST, but from the Saber simulator's interpretation of MAST. This chapter describes the interface necessary for foreign subroutines written only in C or FORTRAN.

Generally, you must compile foreign subroutines into the Saber environment. The Saber simulator supports dynamic loading, so you can merely compile subroutines and make sure the directory they are in the SABER_DATA_PATH. The Saber simulator loads the routines as needed at run time.

Calling Foreign Subroutines

Whenever you use a foreign subroutine, you must both declare it and call it.

Declaring Foreign Subroutines

There are two types of foreign subroutines in MAST: foreign subroutines that return a single number, and foreign subroutines that are not restricted in what they return. These are declared in the local declarations section and require different syntax. Foreign subroutines can also be declared globally in the local declaration section of the top-level template.

The syntax for declaring a subroutine that returns a single number is as follows:

`foreign number` *subroutinename*`()`

The syntax for an unrestricted subroutine is:

`foreign` *subroutinename*

Foreign subroutines that return a single number use two keywords, `foreign` and `number`, and a set of parentheses. The *subroutinename* is the user selected name of the subroutine. Foreign subroutines that are not restricted in what they return use only one keyword, `foreign`. In either case, the name of the file containing the compiled subroutine must be *subroutinename*`.o`. If a foreign subroutine itself calls a foreign subroutine in another file, that file name must also be declared as foreign in the local declarations section. If the additional subroutine is contained within the same file as the first subroutine, no additional declaration is necessary.

Calling the Subroutine

Foreign subroutines that return a single number can be used within an expression. They evaluate to the number. They are called with an input_list as follows.

subroutinename(*input_list*)

The *input_list* consists of a comma-separated list of inputs to the subroutine. The variable names in the *input_list* must have been previously declared. They can be used within an expression as shown in the following example, where `vout`, `initial`, and `vin` are declared variables, and `gain` is the subroutine returning a number.

```
vout = initial + gain(vin)
```

This kind of subroutine can be used wherever a built-in mathematical function can be used.

Foreign subroutines that are not restricted as to what they return are called with an input and output list as shown:

(*output_list*) = *subroutinename*(*input_list*)

The statement that calls this type of foreign subroutine is similar to an assignment statement. The *input_list* and *output_list* are comma-separated lists of values. They can also contain the group name of variables declared as a group. The *output_list* can consist of only parameters in the parameters section, states in when statements, and `val`s in the values section. The simvars `step_size` and `next_time`, can also appear in the *output_list* in the values section and in when statements.

There can be one or more returned values. If there are more than one, the names of the returned values should be enclosed in parentheses. Otherwise, the parentheses are optional. The *input_list* can consist of constants, or any variables which can normally be used in the template section containing the foreign subroutine call. The names of variables in the *input_list* and *output_list* must be declared before the subroutine is called.

Examples of calls to foreign subroutines follow:

```
err = bjt(1,1,work,model)
(svbel,svbcl,svscl,svbxl,svbbl) = sample(temp,area)
```

In the first case, the foreign subroutine `bjt` has one variable `(err)` in the *output_list* and four values in the *input_list* (consisting of two constants and two variables). In the second case the subroutine sample has five variables in the *output_list* and two variables in the *input_list*. All variables would need to have been previously declared, and of acceptable types in the template section containing the subroutine call.

Writing Foreign Subroutines

In general, the name of a foreign function should be the same as the name of the file containing it. Thus, the name must follow the conventions of both the operating system under which you run the Saber simulator and the programming language in which you write the foreign function. In particular, this means that the number of characters in the name of the function (or subroutine) must conform to the limits set by the operating system.

In FORTRAN, the header line of the file must have the following form:

`subroutine` *name*(*arguments*)

where *name* is the name by which the foreign function is called in templates, and *arguments* is a specific comma-separated list of argument names, which are described in Required Interface Arguments on page 8-4.

NOTE

If you want to use an OPEN statement to assign an output device, set UNIT to a number between 10 and 59 (preferably between 20 and 30). Numbers 1-9 and 60-99 are reserved for use by the Saber simulator.

In the C language, depending on the platform, the header line has one of these two forms:

name(*arguments*)
*name_*(*arguments*)

The purposes of *name* and *arguments* are the same as in FORTRAN. For a list of platforms requiring the second form, i.e., *name* followed by an underscore(_), refer to the *SaberDesigner Inatallation* manuals; "Using C or FORTRAN Routines Called by Templates".

When using other languages, *name* and *arguments* have the same purposes and requirements. Otherwise, you need only follow the normal conventions for those languages.

Required Interface Arguments

The MAST language does not directly call the subroutines; instead, the Saber simulator calls them. The Saber simulator takes the *input_list* from the template and translates it into arguments used in the subroutine. The simulator takes the arguments from the foreign subroutine, and translates them into the *output_list* or the single number returned from a foreign number subroutine. Therefore, since the Saber simulator adds an additional interface between the templates and the foreign subroutines, the arguments in the subroutine will not match the input and *output_list*s in the calling template.

There are two levels of translation:

7. The inputs and outputs from the templates are translated into ten arguments (listed below)

8. Because only numerical values are actually passed from template to foreign subroutine, in many cases the foreign subroutine must decode the numerical values to discover the variables passed

The arguments are 10 dummy arguments, in a specific order, through which values are passed between the Saber simulator and the foreign function. You must use the entire list of 10 arguments whenever you write a foreign function for use by templates written in the MAST language. The argument names are arbitrary, but the place for each is fixed. The meanings of these arguments, listed in order, are:

`in` is an array of the inputs to the routine. Saber passes only numerical arguments to foreign subroutines. Therefore, most variable types are encoded, as described later in this chapter. Foreign subroutines must be written in such a way as to decode the input array. All arguments are passed by value, meaning that even if they are modified, the modification will not appear in the corresponding variables in the template.

`nin` is the number of elements of the array in

`ifl` (reserved for use in a future release)

`nifl` (reserved for use in a future release) `out` is an array of the outputs from the routine. On entry to the routine, it contains undefined values. Only numerical arguments are passed out, but these can be decoded by the Saber simulator to match various variables in the calling statement's *output_list*.

`nout` is the length of out

`ofl` (reserved for use in a future release)

`nofl` (reserved for use in a future release)

`aundef` is a special constant that indicates undefined quantities. Any variable that is undefined in the template will have this value in the in array. This is the same as undef in the MAST language

`ier` (reserved for use in a future release)

NOTE

Even though only `in`, `nin`, `out`, `nout`, *and* `aundef` *are currently used, you must declare all 10 arguments in the subroutine header.*

Interface Examples

In FORTRAN, the foreign function will have the form:

```
subroutine name(in,nin,ifl,nifl,out,nout,ofl,nofl,
                aundef,ier)
  integer nin,nifl,nout,nofl,ier
  integer ifl(nifl),ofl(nofl)
  real*8 in(*),out(nout),aundef
   body of function
   ...
end
```

In C, the function will have the form:

```
name(in,nin,ifl,nifl,out,nout,ofl,nofl,aundef,ier)
  int *nin,*ifl,*nifl,*nout,*ofl,*nofl,*ier
  double *in, *out, *aundef
{
   body of function
```

```
        ...
    }
```

As described earlier, some platforms require *name* to be followed by an underscore (_).

Argument Passing

When passing arguments to a foreign subroutine, the simulator enters numerical values that represent the arguments into the `in` array, in the same order in which the arguments are listed in the subroutine call. When returning a result from a function declared to return only a single number, the simulator takes its value from the single value in the `out` array. When returning results from a subroutine that is not restricted in its output, the simulator translates the numerical values in the `out` array into the *output_list*, in the order in which the outputs are listed. If a group is used as the output, the numerical values are translated into those variables in the order in which they are declared as a group.

Determining the Translation of the In and Out Arrays

It is important, when writing a foreign function, to know the translation of the in and out arrays. The inputs and outputs of foreign functions can each be any of the types permitted in the MAST language. The simulator encodes arguments in specific ways when placing them into the in array of a foreign function and decodes them in the reverse way when taking them out of the out array. The following paragraphs describe, by argument type, how the simulator encodes arguments from a function call to produce a representation of the argument into the in array of the foreign function -- the reverse of the encoding process is the decoding process (how it interprets the contents of the out array. Examples of translations follow this.

`number` is represented as itself. This holds for variables of types `number`, `state`, `val`, `var`, `ref`, and `simvar`

`enum` is represented by an index number, which indicates the position (starting with 1) of the argument in the enum declaration

`string` is represented as a number, which is a string descriptor that can be used with some Analogy-supported subroutines to pass the string

`struc` is represented as a list of the translations of its fields, in the order in which they are declared in the structure `union` is represented by two sets of numbers: the first is an index number, which indicates the position (starting with 1) of the current choice in the definition of the union; the second is the translation of the fields of the choice

`array` is represented by two sets of numbers: the first is a single number indicating the number of items in the array; the second is a sequence of translations, one for each array item. The translations are in row-dominant order (last index varies first). An array item may be of any number of fields, and not all array items need be of the same number of fields. For example, an array of unions may consist of different types of array items. Undefined arrays have the length indication stored as undefined (set to `undef`).

Because there is no restriction on the types of the inputs and outputs of foreign functions, they can be declared as nested composite types. If this is the case, the elements of each composite type are translated one at a time, taking into account the fact that for arrays, the first element stored is the number of members in the array, and for unions, the first element stored is the index number of the choice used.

The following examples illustrate how to determine the translation of the `in` and out arrays. In each example, the first column shows a sample declaration, and the second column shows the resulting array. These are shown here as *input_list* and the `in` array, but the same translation would apply to the *output_list* and `out` array.

Number

```
   #in local declarations section
   number fred=7
   foreign foreignsub
   #in template section              #in in array
output_list = foreignsub (fred)    [7]
```

Because numbers are entered into the array as they are, the `in` array has a single member, the number 7, and `nin` (the number of elements in the array) is 1.

Enum

```
   #in declarations section
   enum {plus, minus} fred = plus
   foreign foreignsub
   #in another template section      #in in array
```

```
output_list = foreignsub (fred)   [1]

#in declarations section
enum {plus, minus} fred = minus
foreign foreignsub
#in another template section      #in in array
output_list = foreignsub (fred)   [2]
```

For enumerated types, each member is associated with a number that indicates its position in the enumerated list. The number passed to the array is the number associated with the member selected. In each case `nin` is 1.

Strings

```
#in declarations section
string fred=""
foreign foreignsub
#in another template section      #in in array
output_list = foreignsub (fred)   [a number]
```

A string descriptor is passed for a string, where a descriptor is defined to be a number which acts like a memory pointer, but which does not point to the memory location where the string is stored, but to some encoded location known by the Saber simulator. The string can be decoded in the foreign subroutine by using an Analogy-supported subroutine, as explained later in this chapter. Here `nin` equals 1.

Struc

```
#in declarations section
struc {
  number a=6,b=5
}fred=()
foreign foreignsub
#in another template section      #in in array
output_list = foreignsub (fred)   [6,5]

#in declarations section
struc {
  enum {plus, minus} mary = minus
  number a=6,b=5
}fred = ()
foreign foreignsub
#in another template section      #in in array
```

```
output_list = foreignsub (fred)   [2,6,5]
```

For structures, the members of the structure are passed to the in array in order. Therefore, in the first structure, which consists of two numbers, the in array has two members, consisting of the values of the two numbers, and `nin` equals 2. The second structure has three members, the first number corresponding to the enumerated type, and the other two to the numbers.

Union

```
#in local declarations section
union {
  number a=10,b=20,c=30
} fred=a
foreign foreignsub
#in another template section      #in in array
output_list = foreignsub (fred)   [1,10]
```

A union represents a choice of values (here, `a`, `b`, or `c`). Associated with each choice is its index number (indicating which choice it is, as with enumerated types), and additionally, its value. Both the index number and the value are passed to the array. In the example above, the selection is `a`, so the index (1) and the value (10) are passed to the array. If `b` had been selected, the array would hold [2,20]. If `c` had been selected, the array would hold [3,30]. In all cases, `nin` equals 2.

Arrays

Arrays can contain any of the types or combinations thereof. Following are examples for some types. There is an example for one of each type. When an array is passed through, the first number is the size of the array, and the rest are values for the array elements.

```
#in local declarations section
number fred[2]=[7,5]
foreign foreignsub
#in another template section      #in in array
output_list = foreignsub(fred)    [2,7,5]
```

Because this is an array of numbers, the result consists only of the size of the array and the two array numbers, and `nin` equals 3.

```
#in local declarations section
enum {plus, minus} fred[2]=[plus,plus]
foreign foreignsub
#in another template section      #in in array
output_list = foreignsub(fred)    [2,1,1]
```

The values for this array of enumerated types consist of the size of the array (2), followed by the indexes of the selection (1,1), and `nin` equals 3.

```
#in local declarations section
struc {
  number a=6,b=5
} fred[2]=[(),()]
foreign foreignsub
#in another template section     #in in array
output_list = foreignsub (fred)  [2,6,5,6,5]
```

Because this is an array of structures, and the structure consists of two numbers, the resulting consists of the size of the array (2), followed by the value of the structure (6,5) twice, since the array is initialized to contain the same values for each member of the array, and `nin` equals 5.

```
#in local declarations section
union{
  number a=10,b=20,c=30
} fred[3]=[a,b,c]
foreign foreignsub
#in another template section     #in in array
output_list = foreignsub(fred)   [3,1,20,2,20,3,30]
```

This array of unions is a three member array, so the first member of the resulting array is a 3. The array contains members `a`, `b`, and `c`, in turn. This means that the result array must contain the index number for each, followed by its value (1,10 for `a`, 2,20 for `b`, and 3,30 for `c`), and `nin` equals 7.

Multi-dimensional arrays

As mentioned previously, multi-dimensional arrays are passed such that the last index varies first. That is, the elements of a given array are passed in sequential order according to their subscripts (preceded by the total number of elements in the array). This is best illustrated by an example for the 2-dimensional array, `d[3,4]`, which would have 12 elements as follows:

```
d11  d12  d13  d14

d21  d22  d23  d24

d31  d32  d33  d34
```

This would be passed as the following sequence:

```
[12 d11 d12 d13 d14 d21 d22 d23 d24 d31 d32 d33 d34]
```

Mixtures of Types

There are no restrictions on mixing types, as long as the rules for each type are observed. This example is a structure which is composed of an enumerated type, two numbers, an array of numbers, a structure, and a union:

```
#in local declarations section
struc{
  enum {plus, minus} mary = plus
  number a=7,b=6
  number mike[4]=[10,11,12,13]
  struc{
    number c=20,e=40
  }jim=()
  union {
    number e=60,f=70
  }ian = e
}fred = ()
foreign foreignsub
#in another template section                #in in array
output_list = foreignsub(fred)
                        [1,7,6,4,10,11,12,13,20,40,1,60]
```

The members of this structure go into the resulting array in order, following the rules for each of the types, with each member simply placed into the resulting array as defined. The first member is 1, the index of the enumerated type. The second and third members (7,6) are the values of the numbers. The next five values are for the array of four numbers: the size of the array followed by the four values (4,10,11,12,13). The next two values are for the members of the structure `jim` containing two numbers (20,40). The final two numbers represent the index and value for the selected member of the union (1,60); `nin` equals 12.

Increasing the Size of the Output Array

In some cases, it is not known in advance how many values will be returned by the foreign subroutine; for example, if it returns a variable-length array. In such cases, the foreign subroutine must negotiate with the Saber simulator to obtain an output array out of appropriate size.

To be able to negotiate with the Saber simulator, the foreign routine must declare its argument `nout` as an array of two integers, which, in FORTRAN, requires the subroutine header to be of the following form:

```
subroutine name(in,nin,ifl,nifl,out,nout,ofl,
                nofl,aundef,ier)
integer nin,nifl,nout(2),nofl,ier
integer ifl(nifl),ofl(nofl)
real*8 in(*),out(nout),aundef
   body of function
   ...
end
```

No change is needed in the header of C routines. The two entries of `nout` have the following form:

FORTRAN	C	Meaning
`nout(1)`	`nout(0)`	The number of values the routine returns in out. This number should be set by the foreign routine before it returns.
`nout(2)`	`nout(1)`	The size of the array `out`, which is the maximum number of values the routine can return in `out`. This number is set by the Saber simulator before it calls the foreign routine.

A foreign routine returning variable length arrays must be written such that it first determines the number of values it intends to return and saves this value in `nout(1)` (or `nout[0]`). It then should compare this value with `nout(2)` (or `nout[1]` in C) and return if the `out` array is too short. The Saber simulator calls the routine a second time with identical arguments, except that the `out` array is now as long as requested. The following is a sample code fragment in FORTRAN to illustrate this procedure:

```
subroutine foreign(in,nin,ifl,nifl,out,nout,ofl,
                   nofl,aundef,ier)
   declarations
   code to determine number of values to return
nout(1) = storage_need
if (nout(1) .gt. nout(2)) return
   code to complete the return values
return
end
```

Passing Strings

It is possible to pass strings both to and from foreign subroutines. Analogy provides special subroutines to facilitate this.

Passing Strings to Foreign Subroutines

You can cause string values to be passed to foreign subroutines by including a string expression in the input argument list. A double-precision real number, the string descriptor, is then passed as part of the foreign subroutine's input array. By calling a special Analogy-supported subroutine, you can cause a string to be recovered within the foreign subroutine.

A string descriptor acts like a string pointer, but it does not point to the actual memory location where the string resides, as a pointer would. Instead, it directs the Analogy-supported subroutines discussed below to the place where the strings are stored.

In FORTRAN the string variable that will receive the string must be declared as a character variable, and then the program must make the following call:

```
call getstr(descriptor,buffer,nch)
```

where the input is `descriptor`, a string descriptor variable name assigned to the appropriate double precision number from the input array, and the outputs are `buffer`, the Fortran character variable which will receive the string, and `nch`, the number of characters in the string. The string is truncated or blank padded to fit the buffer.

In C, the `cgetstr` function is declared as follows:

```
   char *cgetstr(descriptor)
double   descriptor;
```

It returns a pointer to a null terminated string. *Do not* modify the string; the results can be quite unpredictable.

Passing Strings from Foreign Subroutines

Strings are passed from foreign subroutines to templates by using a set string function (`setstr()`) on the string, which returns a double precision real number descriptor that can be passed to the template in the foreign subroutine's output array. This descriptor will be automatically interpreted as a string within the template.

In FORTRAN the descriptor must be declared as a double precision real number, and then the routine can call `setstr`.

```
call setstr(buffer,nch,descriptor)
```

where the inputs are `buffer`, the FORTRAN character variable containing the string to be passed to MAST, and `nch`, the number of significant characters in the string. The output is `descriptor`, a double precision real number to be used in the output array.

The C function `csetstr(string)` accepts a null terminated string and returns its descriptor.

```
    double csetstr(string)
char *string;
```

Implementing Statistical Distributions

You can use foreign routines to define statistical distributions of your own, to supplement the Saber simulator's collection of built-in statistical distributions. Typically, such routines return a single number and should therefore be declared as such (e.g., `foreign number mydist()`). The interface of foreign routines implementing statistical distributions is identical to other foreign routines, but they may want to get additional information from the Saber simulator, such as random numbers or an indication whether they should return statistical or deterministic values.

chapter 9

MAST Functions

Introduction

NOTE

Except where otherwise indicated, variables referred to in this chapter are state variables.

Saber's time-domain analyses are continuous, in the sense that the simulator chooses the size of a time step to be as large or as small as necessary for accuracy. However, by letting a model schedule exact times for variables to take on new values or for the integration algorithm to sample the analog waveforms, the simulator provides discrete time simulation. Scheduling can dramatically speed up simulation, because the simulator has to check effects of state changes only at scheduled times, instead of after each time step. You can easily model many components, especially digital components, using discrete time simulation. The key to discrete time simulation is the When statement.

This chapter describes the When statement and the conditions that can satisfy it. It also describes how you can schedule the assignment of a value to a variable (an event) and how you can schedule times at which the integration algorithm samples the analog waveforms. In addition, it describes how to de-schedule events that are scheduled. For more information on the When statement and digital modeling in general, refer to *Guide To Writing MAST Templates, Book I*.

State variables are an integral part of the When statement and scheduling. Because the initial values of state variables are important in the DC analysis and the resulting DC initial point, the chapter gives the DC algorithm used in mixed-mode simulation.

Finally, the chapter gives some examples that illustrate the main ideas of the chapter.

The following statements, functions, and simvars are the "tools" of this chapter:

- The When statement waits for a specified condition to be true. The condition may be simple or complicated, but it normally includes one or more of the functions and simvars described in the next four paragraphs. The when condition is monitored and, when satisfied, the statement causes a block of one or more specified statements to be executed immediately and in order.
- The `schedule_event` function sets (schedules) a time at which a specified variable is to receive the value of a specified expression.
- The `event_on` function returns "true" whenever a value is assigned to a specified state variable, by a `schedule_event`.
- The `threshold` function returns "true" whenever the value of a specified expression crosses, becomes equal to, or becomes unequal to, a specified value. It is useful for (but not limited to) converting from analog to digital.
- The `schedule_next_time` function schedules a time at which the integration algorithm samples the analog waveforms. That is, if the integration algorithm yields a time step that would cause the simulator to go beyond one or more scheduled `next_times`, the simulator is required to step ahead only to the first such time. This is the only means whereby the digital part of the system can change something in the analog part. It is useful (but not limited to) converting from digital to analog.
- The `deschedule` function deschedules a specified event or `next_time` that had been scheduled previously by `schedule_event` or `schedule_next_time`.
- The `dc_init` simvar becomes true at the start of DC analyses, that is, at the start of DC operating point analysis (`dc`), DC transfer analysis (`dt`), and the DC operating point analysis portion of the combined DC operating point and transient analysis (`dctr`).
- The `dc_start` simvar becomes true at the start of `dc` and the DC portion of `dctr` only. It is not true at the start of the `dt` analysis. It becomes true after the `dc_init` simvar has become true and then reset to false.
- The `dc_done` simvar becomes true at the end of a `dc` analysis, or the DC portion of a `dctr` analysis. (It is set to true after the DC algorithm is completed.)

- The `time_init` simvar becomes true at the start of transient analysis. It does not become true when a transient analysis is re-started from a previous transient analysis.
- The `tr_start` simvar becomes true at the start of any transient analysis, including one re-started from a previous transient analysis.
- The `tr_done` simvar becomes true at the end of any transient analysis.
- The `time_step_done` simvar becomes true at the end of each time step. It is usually possible to avoid use of `time_step_done` by using the `threshold` condition, and, when possible, it is desirable to do so.

NOTE

You can use the binary operators “&” and “|” with these simvars as Boolean AND and OR functions (refer to Expression Types on page 4-1). This allows you to stipulate the conditions for two or more simvars in a single When statement (e.g., `when dc_init|time_init {`*...).*

When Statement

The general form of the When statement is as follows:

```
when ( condition ) {
     statements
}
```

where:

condition	is a logical expression involving one or more of the intrinsic functions named `threshold` and `event_on` or the simulator variables named `dc_init`, `dc_start`, `dc_done`, `time_init`, `tr_start`, `tr_done`, and `time_step_done`.
statements	is a collection of one or more statements, usually with one or more being calls to the intrinsic scheduling functions named schedule_event, `schedule_next_time`, and `deschedule`.

The Saber simulator monitors *condition*. Whenever the simulator finds *condition* to be true, the statements are executed.

Two characteristics of a When statement to keep in mind are:

- They are evaluated in the order of appearance in a template
- They cannot be nested

Conditions for the When Statement

The `condition` of the When statement usually involves at least one of the `event_on` and `threshold` intrinsic functions and the `dc_init`, `dc_start`, `dc_done`, `time_init`, `tr_start`, `tr_done` simvars.

Event_on Condition

The `event_on` function becomes true when a scheduled assignment (scheduled using schedule_event) to the specified variable takes place.

The format of the `event_on` condition is as follows:

`event_on (` *statevar [*`,` *oldvalue]* `)`

where:

statevar is the name of a state variable to be monitored for an assignment. When the assignment takes place, the `event_on` condition is true.

oldvalue (optional) is the name of a state variable, which, when *statevar* receives a value, it in turn receives the previous value of *statevar*. Thus, *oldvalue* is an output variable.

An example is: `event_on(flag)`.

An output state used as a connection point cannot be used as a *statevar* in the `event_on` statement to detect a scheduled assignment in the template.

Threshold Condition

The `threshold` condition occurs when an analog waveform crosses a threshold.

The format of the `threshold` function is as follows:

`threshold (` *expression*`,` *value[*`,` *beforestate[*`,` *afterstate]]* `)` where:

expression	Is an expression that, with *value*, determines when the threshold condition is met. The threshold condition is met under the following conditions: When *expression* changes from less than value to more than value When *expression* changes from more than value to less than value When expression becomes equal to value When expression changes from being equal to value to being unequal to value. The *expression* can involve system variables, such as vars, across variables, and refs. The threshold condition is evaluated most efficiently if *expression* is a linear combination of system variables. Vals are also available for use in *expression*. Vals are evaluated only when necessary. If they are used as expressions in when statements, they must be evaluated much more often, so it is often very costly in simulation time to use vals as conditions.
value	Is an expression whose value is the threshold value, i.e., the reference value to which the `threshold` function compares *expression*.
beforestate	(optional) is: 1 -- If the value of *expression* was greater than value before the threshold condition was met. -1 -- If the value of *expression* was less than value before the threshold condition was met. 0 -- If the value of *expression* was equal to value before the threshold condition was met. Thus, this variable is an output variable.
afterstate	(optional) is: 1 -- If the value of *expression* was greater than value after the threshold condition was met. -1 -- If the value of *expression* was less than value after the threshold condition was met. 0 -- If the value of *expression* was equal to value after the threshold condition was met. Thus, this variable is an output variable.

If you are using `threshold` to detect a rising or falling edge, then *beforestate* and *afterstate* provide that information, and more. The following table gives the meanings of the eight possible combinations of beforestate and afterstate:

beforestate	afterstate	Meaning
-1	0	rose to equal value
-1	1	rising edge
0	-1	fell from value
0	1	rose from value
1	-1	falling edge
1	0	fell to value
-1	-1	rose to value, then fell again
1	1	fell to value, then rose again

Simvars

The simvars become true as explained previously in this chapter. Their formats for inclusion in the condition statement are simply the simvar names. Simvars do not have to be declared in templates. For example, the format of the `dc_init` simvar is:

```
dc_init
```

Statements

The *statements* portion of the When statement can include any MAST language statements, such as assignment statements, If statements, foreign function calls, and expressions such as message functions. Only states can be on the left-hand side of assignment statements, or be returned from foreign subroutines. Other than these limitations, states, vars, refs, across variables, parameters, arguments, and vals are available for use in When statements. However, because the primary purpose of the When statement is to schedule discrete time simulation functions, *statements* will usually contain at least one or more of the `schedule_event`, `schedule_next_time`, or `deschedule` functions.

You can schedule assignments of specified values to specified variables using the `schedule_event` function, and you can schedule times at which the integration algorithm is to sample the analog waveforms using the `schedule_next_time` function. Each of these functions can return a unique identifier that distinguishes the scheduling from all other schedulings. If necessary, you can use these unique identifiers to cancel schedulings with the deschedule command.

Scheduling Assignments

To schedule an assignment, use the `schedule_event` function, usually in the *statements* portion of the when statement, as follows:

[scheduling_id =] `schedule_event (` *time*, *statevar*, *expression* `)`

where:

scheduling_id	(optional) is an array of two state variables. This array becomes a unique identifier when the event is scheduled, and can be used for descheduling the event.
time	is an expression whose value indicates the time at which the assignment is to occur. Typically time is defined as the sum of the `time` simvar and some expression that represents a delay (the delay may be zero). If the *time* value is less than the current simulation time, the simulator ignores the expression.
statevar	is the name of the state variable that is to receive *expression* as its new value at time *time*.
expression	is an expression whose value *statevar* is to receive. The assignment of *expression* to *statevar* is the assignment being scheduled.

Scheduling an assignment to take place immediately (*time* = `time`) is different from merely making an assignment, because a scheduled assignment causes the integration algorithm to sample the analog waveforms, and thus can affect the state of the system.

The following example (when accompanied by the other required parts of a template) causes the variable named out to be negated every 10 nanoseconds:

```
when( event_on( out ) ) {
     schedule_event ( time + 10n, out, ~out )
}
```

Scheduling Analog Waveform Sampling Times

There are two ways to schedule times at which the integration algorithm is to sample analog waveforms:

- Using the `next_time` simulator variable

- Using the `schedule_next_time` function

As explained in the chapter on declarations, `next_time` is a simulator variable that templates can use in the `values` section to specify a time beyond which the simulator must not go in its next time-step. The disadvantage of `next_time` is that, at each time-step the simulator checks to make sure that it is not crossing the `next_time` barrier, and then the simulator effectively clears the value of the `next_time` variable. Thus, the template has to reassign a value to `next_time` before the simulator takes its next time-step. The advantage of `next_time` is that it gives the template the opportunity to re-evaluate circumstances, and then assign to `next_time` either the same value, a new value, or no value.

With the `schedule_next_time` function, you have the opportunity to overcome the disadvantage of `next_time` without sacrificing its advantage. `Schedule_next_time` lets you provide the simulator with a value that it can schedule (and therefore not forget). The `schedule_next_time` function has the following syntax:

[scheduling_id =] `schedule_next_time (`*time*`)`

where:

scheduling_id	(optional) is an array of two state variables. This array became a unique identifier when `next_time` is scheduled, and can be used to deschedule the `next_time`.
time	is an expression whose value is a time at which the simulator is to evaluate the system. That is, unless this scheduling is subsequently de-scheduled, the simulator will not “step over” this scheduled time, even if the time-step algorithm indicates that it should. If you are running an analysis with `mon`(itor) set to a positive value (every *n* steps), scheduled time steps are displayed with an “x” in column 1 of the display. If the *time* value is less than the current simulation time, the simulator ignores the expression.

To see the usefulness of the `schedule_next_time` function, suppose an input waveform “turns” suddenly and without warning (i.e., its time derivative changes abruptly). Normally the simulator has to backtrack and search around to get back “on track.” Worse, if the sudden turn is the beginning of a spike, then the simulator might “step over” the spike without even detecting it.

To keep the simulator close to the waveform, you can schedule time steps at the places where the time derivative of the waveform changes abruptly. In particular, telling the simulator to take time-steps precisely at the “turning” points of the input function, that is, by scheduling time-steps there, you can keep it close to the curve and thereby increase its efficiency.

DeScheduling

To deschedule a scheduled event, use the `deschedule` command, which has the following format:

`deschedule (` *schedule_id* `)`

where:

schedule_id	is an array of two unitless state variables which becomes a unique identifier when an event of a next_time was scheduled.

Initializing Templates

In mixed-mode simulation, state variables in a template can depend upon an analog waveform or analog waveforms can depend upon a state variable, or both. For example, in digital-to-analog converters, analog output depends upon digital state input, whereas in analog-to-digital converters, digital state output depends upon analog input. Because of this relationship and the nature of the DC algorithm, state variables should be initialized as if the corresponding analog waveforms are equal to 0.

When performing discrete time simulations, the Saber simulator treats the analog and digital portions of the system separately. The analog portion has continuous time and analog signals, while the digital portion has discrete time and either discrete or analog signals, represented as states. The declaration of a discrete state is similar to the following:

`state logic_4` *statevar*

where the `logic_4` unit definition permits only four distinct states. In contrast, the declaration of an analog state is similar to:

`state v` *statevar*

where the `v` (voltage) unit definition permits a continuum of states.

State variables used as connection points in a template cannot be initialized within the template. These “external” states are automatically initialized to the initial value specified in a digital state unit definition for digital states, or to `undef` for analog states.

If an initializer is not specified, local digital states will be initialized to the initial value specified in the digital state unit definition and analog states will be initialized to `undef`, just as with states used as connection points. However, there are several cases in which local states should be initialized to alternate values.

First, state variables used locally within a template should be initialized so that they correspond to zero values of any associated analog waveforms. This is required so that the DC algorithm (described later in this section), which is used to find an initial point in a circuit containing both analog and discrete circuitry, will work correctly.

An example of this might be a template that performs a local analog to digital conversion and an analog to analog-event-driven conversion in the course of modeling some device. The digital state should be initialized to be whatever it should be if the incoming analog waveform were to equal zero. (If a zero input would produce an `l4_1` digital state, then the digital state should be initialized to `l4_1`.) Similarly, the analog event-driven state should be initialized to whatever it should be if the incoming waveform were to equal zero. (If a zero input would produce an analog state of 2.0 volts, then the analog state should be initialized to equal 2.0 volts.)

Second, if a local state variable does not have an associated analog waveform, then it does not have to be initialized, but in many cases the model would be more realistic if it were initialized. For instance, it would be more pragmatic for a digital clock to be initialized to some state during a DC analysis, rather than allowing it to remain indeterminate until the start of ensuing analyses.

For instance, in a clock template you need to decide what its initial value should be, a logic 0 or a logic 1. There is no reason to propagate a logic x on a clock for dc, since the purpose of dc is to come up with initialization. The following is an example of a clock:

```
   when(dc_init){
     schedule_event(time,out,l4_1)
}
```

It is important that templates that generate events, such as clocks, only generate events during transient analyses, and either not at all or (in some cases) only once during the DC analyses. This implies that there might be places in templates where you have to put “if (time_domain)” around events you plan to schedule.

Initialize state variables in the local declarations section using the following syntax:

`state` *unit statevar*=*initial_value*

where `state` is a keyword, the *unit* refers to a unit definition, the *statevar* is the name of the state being declared, and the *initial_value* is the initial value to which the *statevar* should be initialized.

The DC Algorithm

The DC algorithm for mixed-mode simulation proceeds as follows:

1. Take the initial point, `ip`, from the `dcip` variable of the `dc` or `dctr` command.

2. Initialize variables. Set all analog system variables to zero. Set external digital states to their initial values as specified in the appropriate unit definitions, set external event-driven analog states to `undef`, and set internal discrete states to any specified initial values, or, if no initial value is specified, to the initial value specified in the unit definition for digital states and to `undef` for analog states.

3. Do all `when(dc_init)` sections in all templates. Evolve discrete system appropriately. Do all `when(dc_start)`.

4. Find the effects of the analog subsystem on the discrete subsystem by observing all threshold conditions as analog signals go from zero to their ip values. For any threshold conditions that become satisfied, execute the statements portion of the corresponding when statement.

5. Evolve the discrete subsystem, ignoring scheduled next times, until either there are no more scheduled events or until oscillation is detected. The purpose of “next times” is to convey discrete effects into the analog subsystem. They are ignored at this point because the analog signals have not yet advanced beyond their ip values.

6. Solve the analog subsystem, starting with `ip` and ending with a `new_ip`.

7. Again find the effects of the analog system on the discrete subsystem by observing all threshold conditions as analog signals go from their `ip` values to their `new_ip` values. For any threshold conditions that become satisfied, execute the associated statements. If no threshold is crossed, the DC algorithm is complete and ends.

8. Otherwise, evolve the discrete subsystem to find the effects of the discrete subsystem on the analog subsystem, noting scheduled next times. If no next times are scheduled, then the DC algorithm continues until either there are no more scheduled events or oscillation is detected. If there are no scheduled next times, the DC algorithm is complete and ends.

9. Otherwise, let `ip` be `new_ip`. If the number of DC iterations is less than the limit, return to step 6. Otherwise, the DC algorithm is complete and ends, and the remaining scheduled next times go into the DC endpoint file (`dcep`).

In step 4, the simulator checks for any effects that the analog subsystem might have on the discrete subsystem as the analog signals move from zero to their `ip` values. The template writer must ensure that the initial values of the state variables correspond to the initial values of the analog signals, so that the DC algorithm will produce the correct result.

State Outputs Used as Connection Points

One very important concept to understand about using states and discrete time simulation is the following:

> The value at an output state connection point is not known to a template internally. You must use an internal state to schedule a state output as an event.

This is the case because of requirements to resolve conflicts and to include bi-directional pins. As an example of conflict resolution, imagine two digital devices with outputs connected. The output of one model is HIGH and the output of the other is LOW. The actual resulting state at the node may be low, high, or some other value.

As an example of bi-directional pins, imagine a digital device that usually drives an analog device. However, if it has a digital state of logical z (high impedance), it cannot drive the analog device. The analog device can then map onto the digital device and force the digital part to have a specific logic state.

In both cases, it is not possible to say, in general, that the output state found by a template will be the resulting state at the connection point. Therefore, output states are not made known to the template as events, because they may be incorrect. If you want to model a device in which you rely on an output state connection point, you must declare an internal state that changes with the output state, and use the internal state for scheduling. This is shown in Examples on page 9-14. The example shows a digital clock model.

Mixed Simulator Simulation

The Saber simulator can perform mixed analog and digital simulation using discrete time simulation. It can also perform mixed-simulator simulation, in which Saber, as the analog simulator, models the analog sub-system, and another simulator, as a digital simulator, models the digital sub-system.

There are three new concepts that must be understood for mixed-simulator simulation: time resolution, foreign states, and hypermodels.

Time resolution refers to the granularity of time in a digital simulator. Events can happen only at multiples of minimum time resolution. The menus used in Saber for mixed-simulator simulation have a time resolution variable which should be set to less than or equal to the digital simulator's time resolution.

Foreign states are the states that the Saber simulator passes to the digital simulator. These must be declared locally as foreign states on the analog side. The invocation of the mixed-simulator sets up the communications path on which these foreign states are passed between the simulators. One of the first steps in the mixed-simulator simulation is to check for agreement in the foreign state declarations.

Hypermodels are the models of devices that are simulated by both an analog and a digital simulator. Any device with pins common to both simulators must have those pins modeled for both analog and digital characteristics. Each of these pins requires two hypermodels -- one for the digital simulator and one for the analog simulator (Saber).

Examples

The following examples show templates that use discrete time simulation.

Clock Template

This template models a simple clock, producing the type of waveform shown above. It is set to a HIGH state (l4_1) in the DC analysis using the `when(dc_init)` statement. An internal state, `wake_up`, is used to report the output state of the template, `out`, within the template, because an output state used as a connection point will not have its event made known to the template internally. The `wake_up` is used to schedule the clock timing and the next `wake_up` event.

```
  #******************************************************
  # l4_1               +---------+              +-----
  #                    |         |              |
  # l4_0 +-------------+         +--------------+
  #
  #      |<-----td1---->|<--td2-->|<-----td1---->|
  #         lowtime      hightime      lowtime
  #******************************************************
template clock out = hightime, lowtime
 # this template models a square wave whose duty cycle is
 # input using hightime and lowtime
  state logic_4 out
  number hightime     #time when signal is high
  number lowtime      #time when signal is low
{
  state nu wake_up
  when(dc_init){
    schedule_event(time,out,l4_1)
  }
  when(time_init){
    schedule_event(time,wake_up,0)
```

```
      }
      when(event_on(wake_up)){
        schedule_event(time+hightime,out,l4_0)
        schedule_event(time+hightime+lowtime,out,l4_1)
        schedule_event(time+hightime+lowtime,wake_up,0)
      }
  }
```

Ideal Sample and Hold Template

This template models an idealized sample and hold with a digital gate input (which could be connected to the previous clock example, for instance). When the input gate is HIGH, the template holds the previous `out` waveform. When the input gate is LOW, the template samples the in waveform. The template has two arguments, a delay time, `dt`, and a rise/fall time, `rt`.

```
template smplhold in gate out gnd = dt,rt

  electrical in, out, gnd
  state logic_4 gate
  number dt=1n          #delay time in seconds
  number rt=1n          #rise & fall time in seconds
{
  var i iout            #output current
  state v held          #held voltage
  state time next=0     #time when sample is valid
  state nu sample       #flag: 1 => sample, 0 => hold
  val v vout,           #output voltage
  vin                   #input voltage
#detect event on gate - an input state connection point
  when (event_on(gate)){
    if (gate == l4_1)
      schedule_event(time+dt+rt,sample,0)
    else if (gate == l4_0)
      schedule_event(time+dt,sample,1)
  }
#sample input waveform
  when (event_on(sample)){
    schedule_next_time(time)
    if (sample == 1) {
      next = time + rt
      schedule_next_time(next)
    }
    else held = v(in) - v(gnd)
```

```
  }
  values{
    vin = v(in) - v(gnd)
    if (time < next & sample==1){
      vout = vin - ((next-time)/dt)*(vin-held)
    }
    else vout = vin
    if (sample == 0) vout = held
  }

  equations{
    i(out->gnd) += iout
  iout: v(out) - v(gnd) = vout
  }
}
```

The `smplhold` template has a digital state variable, `gate`, used as an input pin. Note that a state connection point which is an input can be used as an event in a When statement. Only events which are based on output state connection points cannot be known within a template; thus, they often require the use of an additional local state variable.

The Values section is used to provide a way for the voltage at the output to progress continuously from a previously held state to a new sampled state. It uses the `schedule_next_time` function to schedule when the change from holding to sampling should take place, and when full sampling has been achieved. The template uses event-driven analog states, `held`, `time`, and `sample`.

Digital Inverter

This template models a digital inverter that can produce states of logic_4 states of `l4_0`, `l4_1`, and `l4_x`. Since there are no analog signals, no Equations section is necessary to model the device.

```
template inverter in out

  state logic_4 in, out
{
  when(event_on(in)){
    if (in == l4_1) schedule_event (time,out,l4_0)
    else if (in == l4_0) schedule_event (time, out, l4_1)
    else schedule_event(time,out,l4_x)
  }
}
```

Index