Saber® MAST Language User Guide

Version B-2008.09, September 2008

SYNOPSYS®

Contents

Preface

What You Need to Know to Use This Manual

This manual is for Saber users who are familiar with netlists and plan to write their own models for simulation. It is largely tutorial in its approach and assumes no prior experience with the MAST modeling language. It does assume that you know how to use a MAST template with the Saber simulator. Some programming experience would be helpful, though not absolutely necessary.

This manual also assumes that you are familiar with the following:

- How to view the contents of a file.
- How to use a text editor to create a file or edit the contents of a file.
- How to delete files.
- How to create or delete a directory.
- How to list the contents of a directory.
- How to move from one directory to another.

If you are not familiar with these procedures, consult the user's guide for your operating system.

What This Manual is About

This manual shows how to use the MAST modeling language by presenting a small number of concepts at a time, with each presentation based on a modeling example.

It has two objectives:

- To describe some general purpose modeling techniques.
- To show how to use the MAST language to implement simple models of systems or subsystems for use by the Saber simulator.

The models described in the beginning chapters are kept simple. As the model complexity increases with succeeding chapters, every effort has been made to keep the conceptual “jumps” manageable. Most examples start by listing the template of the model and describing the characteristic equations.

Because the MAST language has the features and capabilities of an Analog Hardware Description Language (AHDL), it is suited for the diverse and complex modeling requirements of simulation in general. This manual is not meant to serve as a textbook on modeling—it is intended to be a survey (by example) of MAST capabilities. Therefore, achieving full MAST expertise requires time and practice in addition to reading this manual. Fortunately, for immediate use, you need only learn the portions that apply to your own simulation, which is not so difficult.

This manual explains not only what, but why. The MAST Language Reference Manual also explains what, but in greater detail—it provides more detailed information on topics that are introduced in this manual.

Basic Versus Advanced Modeling

This manual presents a sequence of graduated examples that demonstrate how to incorporate many commonly-used features of the MAST modeling language. These example templates are presented in order of increasing complexity to introduce advanced features. They have been selected to provide the following:

- An easy starting point (the first two examples are models for a current source and a resistor)
- Simplified models
- Balanced length—short enough to read and scan, but long enough to demonstrate significant features and capabilities
- Moderate differences from one example to the next
- Modularity of chapters, to minimize having to read “cover-to-cover”

Note:

Most of the templates used as examples are simplified versions of templates that already exist in the Saber libraries. Be aware that, for simplicity, these example templates do not include the error-checking features contained in the library templates.

Schematic Entry Versus Netlist

As stated above, one objective of this manual is to show how to implement a mathematical model in the MAST modeling language—how to write a template. Some of the constructs and requirements for doing this are based on using a template in a netlist (which is a textual description of your design provided as input to the simulator). Consequently, explanations sometimes refer to topics and procedures that are related to a netlist.

If you are using a schematic entry program, information in this manual regarding netlists still applies to writing a MAST template. This is because a schematic entry program still produces a Saber netlist—this type of program is basically an interface that lets you avoid having to edit a netlist directly. In general, such information is not extremely detailed and is easily applied to using a schematic entry program.

For example, specifying a template in a netlist is effectively the same as placing a symbol in a Saber Sketch schematic and specifying its properties, from which a netlist is created. There is no difference in simulation.

1 Fundamental Modeling Concepts

This chapter presents an overview of the MAST modeling processes, and explores essential topics in computer simulation and modeling.

Guide to Writing MAST Templates

MAST is a description language for use by modelers as a way to describe the behavior and structure of hardware in mathematical terms. Hardware can range from discrete devices to complete system designs. Among the types of devices that can be modeled are electronic devices such as resistors and mechanical devices such as hydraulic valves. A design can be a combination of more than one functional block to comprise a complete system.

Designers can create their own MAST models or rely on a comprehensive library of pre-existing MAST models.

Once a device or design has been described with MAST, the behavior of the model can be simulated with the Saber simulator.

MAST provides two different styles for creating models: structured and unstructured.

The structured approach divides the model into sections. Each section contains specific model functionality. This modeling style is recommended for complicated models. Most models fall into this category.

The unstructured modeling style is beneficial for use with simple models and can be created with less lines of code. Although the term “unstructured” implies a lack of model structure, this is not the case. The model’s structure is made simpler by eliminating the need to partition different model functionality into specific model sections.

The following topics describe general modeling and simulation concepts:

General Modeling Process Overview

The process of replacing a system with a simplified model and then investigating its properties by controlled experiments is called simulation. In order to obtain meaningful simulation results, the model must accurately represent the original system within the constraints of those experiments. Beyond that, the model and the original system may exhibit completely different behavior. This means that a system can be represented by a variety of different models, each having a specific purpose and a set of related limitations.

In computer simulation, the model typically consists of a mathematical description of the system properties, and each controlled experiment is an analysis of a particular type. For example, a transient analysis is typically used to investigate the response of the system to a time-dependent excitation. To yield meaningful simulation results for all types of excitation, the system model must include accurate descriptions (such as nonlinearities and time constants) of the system's dynamic properties. On the other hand, a linear approximation of the system (if it exists) can be sufficiently accurate for feasibility studies. Most simulators also use a linearized model for small-signal AC analysis to determine the frequency response of the system. They generate such a model by linearizing a more accurate nonlinear model about the operating point of the system.

Among the different modeling languages, the MAST modeling language can represent a model in terms of linear or nonlinear algebraic or integro-differential equations. The Saber simulator can then use these models to perform simulation analyses. The interaction of the MAST language and the Saber simulator provides a powerful method of investigating systems containing a wide variety of models.

Simulation Concepts

To create a model for computer simulation, you must understand a few basic concepts about computer simulation, especially how a simulator operates and how it uses a model in a system. These concepts are universal and therefore independent of the Saber simulator. Other concepts apply only to the Saber simulator, which is unique in that it supports continuous analog, event-driven (including digital), and data flow (control system) models.

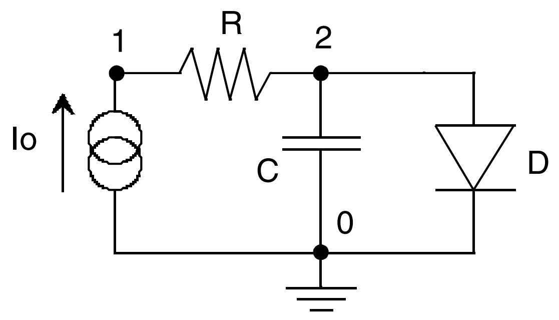
Simulation of Continuous Analog Systems

When simulating continuous analog systems, a simulator repeatedly solves a set of simultaneous nonlinear differential equations. These equations represent the system being simulated—they are the model of the system. The system model consists of ordinary differential equations if it can be represented by lumped elements.

In electrical systems, examples of such elements are resistors, capacitors, or transistors. The elements are connected at the nodes of the system. There is a voltage and a current associated with each node in the system. The voltage is measured at the node with respect to the reference node; the current flows through each element connected to the node.

Electrical Network

For example, consider the following simple network, which consists of a current source, a resistor, a diode, and a capacitor. The text to the right of the diagram is known as a netlist—it specifies the models to be used in the system and how they are connected to each other. A netlist is usually provided as an input file to a simulator.



```
i.in 0 1 = Io
r.12 1 2 = R
c.2 2 0 = C
d.2 2 0 = Is
```

Sample network and its netlist

The following equations represent resistor current, capacitor current, and diode current in this circuit:

R: $IR = V/R$
C: $IC = d/dt(CV)$
D: $ID = IS(e^{V/Vt} - 1)$

Using these equations, it is straightforward to apply Kirchhoff's Current Law (KCL) to nodes 1 and 2 (i.e., the sum of all currents entering a node equals the sum of all currents leaving that node):

i(1): $(v(1)-v(2))/R - I0 = 0$
i(2): $-(v(1)-v(2))/R + d(v(2)*c)/dt + IS*(e^{V(2)/Vt} - 1) = 0$

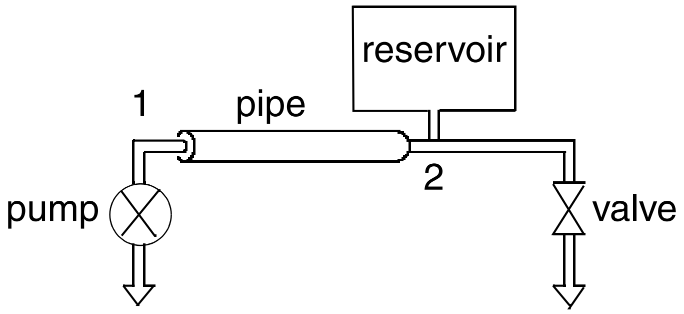
where v(1) and v(2) are the voltages at nodes 1 and 2, with respect to the reference node. IS is the diode saturation current and Vt is the thermal voltage. These equations describe the behavior of the network completely. That is, they are a complete mathematical model of the circuit shown in the previous diagram.

Although it is possible to describe this system directly using the above equations, it is preferable to provide a model for each system element, together with a netlist describing their interconnections.

That way, the simulator obtains the equations from these models and sets them up according to how the netlist describes their interconnections. It then solves the equations for the unknowns, which are the system variables v(1) and v(2)—the voltages at nodes 1 and 2 with respect to the reference node ("0" in this netlist). It may do so repeatedly, in order to determine the system variables as a function of time or frequency.

Continuous Analog Systems - Hydraulic Network

Now consider the simple hydraulic system in the following figure. It consists of a pump delivering a constant water flow through a pipe into a reservoir. A valve protects the reservoir from excess pressure.



Assuming that the valve opens gradually with exponential behavior and that no water spills at the ends (1 and 2) of the pipe, the equations describing the dynamics of the system are:

flow(1): (p(1)-p(2))/pipe_res - pump_flow = 0

flow(2): -(p(1)-p(2))/pipe_res + d(p(2)*rsvr_cap)/dt + vlv_lkg*(e $^{p(2)/P0}$ - 1) = 0

where p(1) and p(2) are the pressures at the two ends of the pipe, and P0 is a reference pressure. You can see that Equations 6 and 7 are mathematically equivalent to Equations 4 and 5 of the electrical network. In the hydraulic system, the pressures p(1) and p(2) are the system variables for which the simulator has to solve.

The similarities between these two examples illustrates that the fundamental concepts used in continuous analog simulation are not restricted to electrical networks—they apply equally to a variety of other types of physical systems, including hydraulic, mechanical, and magnetic.

Each discipline has its own type of system variables, but there is no requirement for a system model to be purely electrical or purely hydraulic. The simulator simply determines the system variables (which can be any combination of voltages, pressures, torques, or system variables of other disciplines) by solving the simultaneous nonlinear differential equations describing the system.

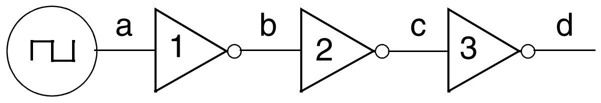
It is important to note that, despite analogous similarities between disciplines, the Saber simulator does not require a non-electrical, continuous analog model to be written in terms of equivalent electrical elements.

For example, it can simulate a model of the system shown in the previous figure directly—there is no need to convert flow to current, pressure to voltage, pump to current source, etc.

Simulation of Event-Driven Systems

The simulation of a continuous analog system described by differential equations yields accurate waveforms for the system variables, provided that the system model is accurate. In many applications, a rough approximation of the waveforms may be sufficient. A typical example is the timing simulation of a digital system, where the waveform for the transition from one logic level to another is much less important than the time when the transition occurs.

Consider the simple inverter chain shown in the following figure. In a timing simulation, the logic levels at nodes a, b, c, and d are of interest, as well as how a change at node a propagates through the chain, given a particular delay t for each inverter.



Chain of inverters

Using binary logic, assume that the system's initial state is LOW, HIGH, LOW, HIGH for nodes a, b, c, and d respectively. A transition to HIGH at node a causes node b to go LOW after the inverter delay of t. After another delay of t, c will go HIGH, because its input changed. Finally, after the third delay of t, d will go LOW, and the system will have settled.

Because the state of the system responds to a change in input only at multiples of t, there is no need to simulate the complete system between the changes; and even then only the parts that change require attention.

Systems like the chain of inverters are called event-driven. During a simulation, each part of the system (here each inverter) simply monitors its inputs; it is otherwise inactive. A part becomes active only when there is an event, such as a transition, on one of its inputs. It then processes the event, which may involve scheduling another event (a transition) on one of its outputs, either at the same time or in the future. Once the event is processed, the system part becomes inactive again and continues monitoring its inputs.

Consider the following chain of events:

1. For the inverter chain in the previous figure, the pulse source schedules the level of node a to change to high at a certain time.
2. Inverter 1 is monitoring this node and becomes active when the event time arrives. This inverter recognizes that node a went HIGH, and schedules node b to go LOW after a delay of t.
3. Inverter 1 then becomes inactive, and returns to monitoring node a.
4. After the delay time has elapsed, inverter 2 becomes active to process the event at its input, node b. Because node b went LOW, inverter 2 schedules node c to go HIGH after a delay of t and then becomes inactive.
5. Finally, inverter 3 becomes active and schedules its output to go LOW after another delay period. The system settles after node d goes LOW, because node d is not monitored.

Event-driven systems differ from continuous analog systems in several respects:

- They are always time-based, whereas continuous analog systems can be simulated in either the time or frequency domain.
- Models of event-driven systems (Strictly event-driven models—excepting Hypermodel analog/digital interface templates) must have clear distinctions between inputs and outputs. That is, the models always have a direction. Although such distinctions are possible in continuous analog models, none are required.
- Event-driven models, by nature, are discontinuous in time.
- An event-driven simulator is basically an event manager that activates selected parts of the system sequentially, in order to process events at their inputs. In contrast, a simulator for continuous analog systems is a solver of simultaneous differential equations.

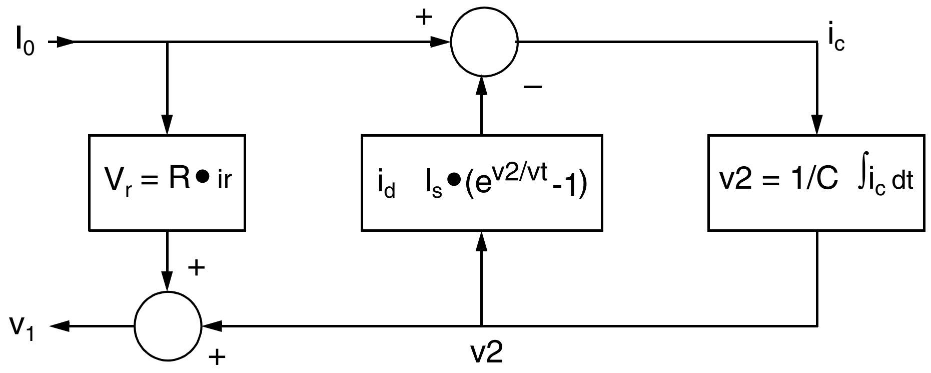
Simulation of Data Flow Systems

A third kind of system, the data flow system, has some characteristics of both the continuous analog system and the event-driven system.

Data flow systems have the following characteristics:

- They can be simulated in either the time or frequency domain.
- Models of data flow systems distinguish clearly between inputs and outputs. That is, the models always have a direction.
- Data flow models, by nature, are continuous.
- Data flow models do not require conservation of through and across quantities (such as current and voltage)
- A data flow simulation involves the solution of simultaneous differential equations. However, because through and across quantities are not conserved, data flow simulation presents the system of equations in a different form than do continuous analog systems.

The following diagram shows a data flow representation for the circuit shown in the topic titled "Electrical Network".



Data flow representation of electrical schematic

The resulting differential equations for the network and the data flow are identical. Only the manner in which they are presented is different. It is difficult to make a one-to-one correspondence between the functional blocks in the data flow representation and the related circuit diagram. It is even more difficult to derive the data flow schematic given the circuit diagram. If the current source were changed to a voltage source, the data-flow schematic would change completely. It is apparent that the continuous analog approach presented in the topic titled "Electrical Network" is a more natural way to describe the electrical system application. However, there are many systems where the data flow approach is more natural (e.g., describing control systems using data-flow schematic diagrams).

The diagram shown in the topic titled "Behavioral Level of Modeling Abstraction" represents a schematic for a system that uses models written with data flow constructs.

General Modeling Concepts

A model of a system is “good” only if it accomplishes the specific purposes for the person using the model. Some keys to good modeling are to create models that:

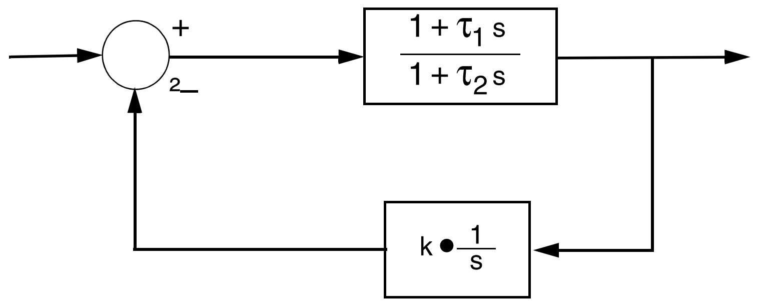
- Accurately reproduce the effects of interest
- Do not waste time with effects that are not of interest
- Provide flexibility to allow the modeler to change his or her mind about the effects that are of interest

A good model answers a suitable number of the “right” questions. A modeling system that just solves differential equations does not answer enough of the right questions. A good modeling system must be able to manipulate input quantities to form the right differential equations, solve the differential equations, and manipulate the results.

Behavioral Level of Modeling Abstraction

In general, most models are behavioral models. With VHDL, a model could describe a design structure (structural description). Although at some level of a design hierarchy, the behavior of each structural piece must be defined. All MAST models are behavioral.

A behavioral model is one that describes an observed behavior with an appropriate set of equations and coefficients. These characteristics are shared by all mathematical models—the distinction between different models in a system lies in the levels of detail that they model.



Control system schematic using data flow models

For example, you can model a bipolar transistor as a switch, with an off and on resistance. The resistance model lets you observe current and voltage effects at the external pins of the transistor. If you need more accuracy, it is available in the Gummel-Poon model. The Gummel-Poon model lets you observe more accurate current and voltage effects at the pins, as well as letting you observe internal voltages, charges, and currents.

If you need still more accuracy, you can use a finite-element device model. This type of model lets you observe even more accurate current and voltage effects at the pins and more accurate internal voltages, charges, and currents. In addition, you can observe internal three-dimensional electric field distributions and temperature gradients. All of these models describe behavior with equations. You could go to lower levels still with quantum mechanical descriptions of the transistor materials. There appears to be no real limit. In practice, however, there are always approximations and simplifications that you must make, regardless of the depth of detail.

What is the appropriate level of abstraction to use? That depends on two things:

- What questions are to be answered using this model?
- What information is available to fit the real device to the model?

For example, if you are trying to optimize the emitter efficiency of a bipolar transistor design, then the finite-element model might be appropriate. If, however, you are trying to verify the functionality of a large logic block on a digital integrated circuit implemented with bipolar transistors, then the finite-element model is clearly excessive. At most, you would use the Gummel-Poon model, and you might even find the switch model to be sufficient. A good model doesn't take the time to provide information that isn't required. That is, it is not appropriate to worry about three-dimensional electric field distributions when the goal is to verify logic design.

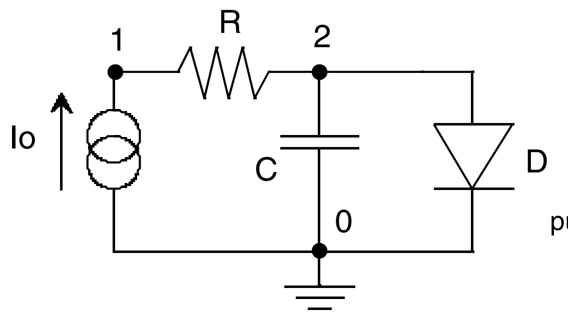
A behavioral description includes the algebraic and differential equations needed to describe the physical system, as well as values for the equation coefficients (also called model parameters). An accurate model requires both accurate equations and accurate parameters. For example, if you want to model a bipolar transistor and all you have is a power supply and an ohmmeter to measure its characteristics, it is better to use a resistor model than a finite-element model. Simulation results based on a well-characterized resistor model will be more accurate than results based on a poorly-characterized finite-element model, even though the finite-element model is capable of more accurate results.

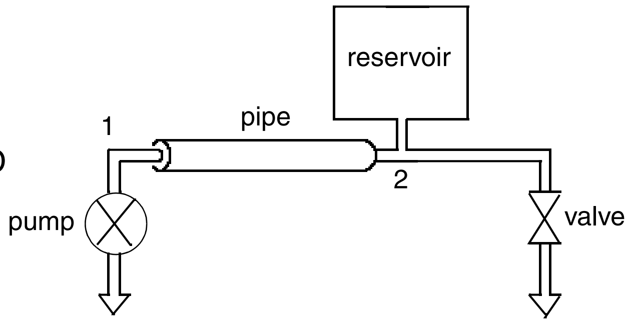
Finally, a model implemented with a high level of behavioral abstraction is not necessarily less accurate than one implemented with a lower level of abstraction. For example, a complicated integrated circuit, such as an analog-to-digital converter, uses hundreds or thousands of transistors to implement a very well-defined input-to-output functionality. If you use one of these devices in your design, you typically would not care about internal node voltages and currents, but only about the external ones. The external behavior can be implemented much more efficiently, without losing accuracy, by a “behavioral” model that considers only external behavior. Integrated circuit designers, in fact, increase internal complexity to make external functionality even simpler and more ideal, making external modeling even more attractive, as compared to internal modeling.

MAST Modeling Modularity and Hierarchy

A “good” model not only implements the appropriate level of abstraction. It is constructed so that it can easily be replaced with a model of either higher or lower abstraction. You can achieve this flexibility by using modularity and hierarchy. Central to the notion of modularity is the concept of the netlist.

Many physical systems can be described as a network of connected components. This applies to electrical systems, but it applies equally to many other systems as well. The following examples illustrate this concept.





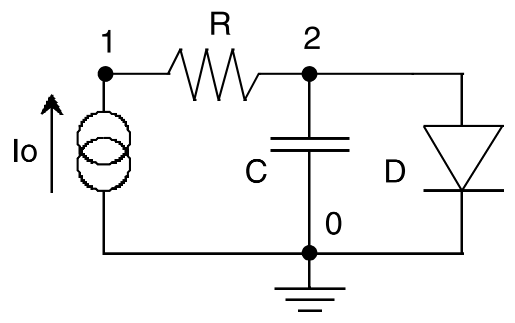
In all such systems, the elements can be considered to have through and across variables, where the through variable is the quantity conserved at a given connection (or node). Generally, the across variable is the driving force and the through variable is the driven force.

For example, in electrical networks, voltage is the driving force and current flow results. In a hydraulic network, pressure is the driving force and fluid flow results. In a thermal network, temperature is the driving force and heat flow results. A mechanical network, on the other hand, uses force as the driving force that is the through variable (position is the across variable). However, regardless of whether it is the driving or driven force, the through variable must sum to zero at each node. Consequently, models operating in such systems are often called conservation models.

Because of the convenience of describing conservation models with through and across variables, the MAST language provides special constructs to make implementing such models easier (see the topic titled "MAST System Variables".

The Saber Netlist Overview

The following figure shows a network and its corresponding netlist—a list that represents a network of models functioning as a system.



```
i.in p:0 m:1 = Io
r.12 p:1 m:2 = rnom = R
c.2 p:2 m:0 = C
d.2 p:2 m:0 = Is
```

Sample network and its netlist

This list is provided as an input file to the simulator. There is a one-to-one correspondence between the elements in the network and the entries (lines) in the netlist.

A netlist entry consists of the following parts:

- the name of the model
- a period (.)
- the name of the instance of that model (this name is often called a reference designator)
- the names of the nodes to which this model instance is connected
- an equals sign (=) (Not necessary in a netlist entry where user-specified parameter values are omitted.)
- values for the model parameters

The objective of this manual is to illustrate how to create MAST models for use in netlists. Refer to the Analyzing Designs manual, the “Saber Netlister Command Reference” appendix for more information on creating netlists from Saber.

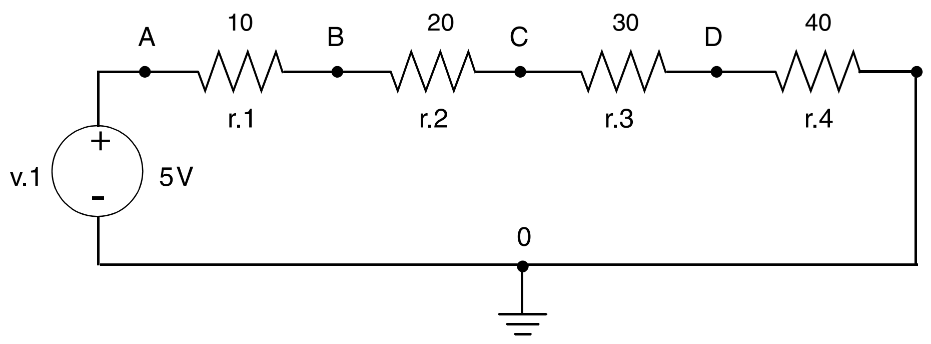
Note:

If you use a schematic capture/entry program such as Saber Sketch, generally, you will not be aware of netlists. However, all such schematic programs create netlists, because nearly all simulators use some kind of netlist to describe the system.

The template is a text file containing the MAST description of the model of a system element (although it can also contain the description of an entire system or of several system elements). It is the basic unit of modularity—there is a MAST template that describes each element in every netlist. Each netlist entry has a corresponding template with which it is associated. Conversely, any given template can be used in any number of netlist entries; each such use of a template in a netlist entry is called an instance of that template.

Because it is a unit of modularity, a template can also be composed of further netlist entries, which introduces the concept of hierarchy. If you are familiar with SPICE, you should realize that, although a MAST template is similar to a subcircuit (.SUBCKT), a template can also be extended to contain algebraic and differential equations directly—a profound distinction.

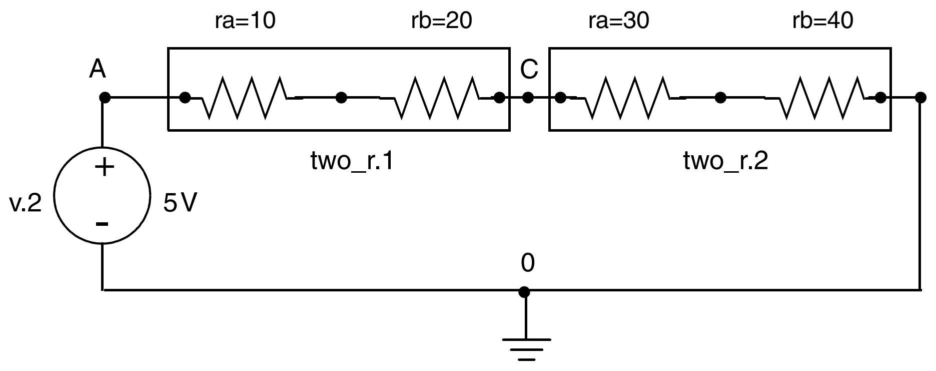
Consider, for example, the string of resistors shown in the following diagram. This circuit can be represented with the netlist shown below, consisting of five entries. Four of the entries (r.1, r.2, r.3, r.4) are instances of the r template and one entry (v.1) is an instance of the v template.



Example 1: String of Resistors

v.1	**p:A**	**m:0**	**= dc = 5**
r.1	p:A	m:B	= rnom = 10
r.2	p:B	m:C	= rnom = 20
r.3	p:C	m:D	= rnom = 30
r.4	p:E	m:0	= rnom = 40

Alternatively, the following diagram shows the same circuit with the first two resistors combined into an instance of a template named two_r. The second pair of resistors combined into another instance of the two_r template. This is an example of hierarchy.



Example 2: Using hierarchy to create resistor pairs

The following two_r template provides a model of two resistors in series:

```
 template two_r a b = ra, rb
  number ra, rb
{
  r.ra   p:a       m:middle = rnom = ra
  r.rb   p:middle  m:b      = rnom = rb
}
```

The first line of this template lists two arguments (ra, rb) by which you can specify the resistance for each of the series resistors; thus, the two_r template can replace two resistor instances in series. The circuit in Example 2 uses two instances of the two_r template, as shown in the following netlist:

```
v.2           p:A      m:0      = dc = 5

two_r.1       a:A      b:C      = ra=10, rb=20

two_r.2       a:C      b:0      = ra=30, rb=40
```

Simulation results for the implementation of Example 2 would be identical to those for the implementation of Example 1. This example actually shows three levels of hierarchy, since the resistor is implemented with a template, r.

In addition to simplifying implementation, hierarchy can provide increased computational efficiency. The numerical methods used by the Saber simulator to solve systems of equations can take advantage of extra information in the specification of a hierarchical design. The internal matrix representation of the system of equations preserves and exploits the hierarchy. The efficiency benefits can be substantial, especially in large, highly-repetitive systems.

By contrast, some simulators do allow the hierarchical specification of a netlist, but then they “flatten” the netlist before simulation. This means that the internal matrix representation of the set of differential equations is the same as if the design were entered in a one-level netlist with no hierarchy.

MAST Modeling Modularity and Hierarchy Summary

The use of modularity and hierarchy are good modeling practices. Just as in writing software, it breaks a large problem into small, easily-identifiable pieces. The use of modularity and hierarchy has several advantages:

- Ease of reuse in other designs
- Ease of replacing one model with another of different behavioral abstraction
- Ease of debugging when there is a problem
- Ease of top-down design
- Ease of partitioning design tasks

Modeling Objectives

The objective of modeling is to answer specific questions—a good modeling system must do more than just implement systems of equations. It must be able to convert inputs into appropriate forms, solve the resulting equations, and manipulate the results to produce the information required.

The implementation of the model must include the mathematical description of the system. However, it should also include any additional information that the model writer may happen to possess. For example, the writer of a diode model (such as the model shown in the topic titled "Electrical Network" may know that the vt model parameter should never be negative. (The model is invalid if vt is negative.) The model writer could then include parameter-checking in the model, to prevent a user-specified negative value from being used.

In the example of the reservoir model shown in the topic titled "Continuous Analog Systems - Hydraulic Network", you may not know the exact meaning of “reservoir_capacity.” The writer of this model can require a user to provide the reservoir’s cross-sectional area, fluid density, and gravitational acceleration. Once the model has this data, it can calculate the resulting reservoir capacity automatically.

A user of a model may wish to use simulation results to perform further calculations. For example, a user of the reservoir model may be interested in finding out how high the level in the reservoir gets when subjected to the conditions shown with the reservoir model. The user may not know (or even care) that the height can be calculated only after the pressure is determined. In a good modeling system, the modeler can design the model to provide the information for calculating the height from the system solution (pressure).

In another example, a user may wish to calculate the power dissipated in the resistor in the diode model. The power dissipation does not have to be calculated explicitly in order to solve the system of equations. However, it is quite reasonable for a user to request its calculation. In a good modeling system, the modeler can include the power dissipation calculation in the resistor model to be provided upon the user's request.

A modeling system answers questions by performing the following functions:

- Before simulation—check for errors and convert parameters.
- During simulation—apply suitable algorithms to the design (such as solving equations repetitively or by processing events).
- After simulation—manipulate simulation results for additional information that is not directly available from the simulation.

2 MAST Overview

The following topics describe some of the specific constructs used to write models in the MAST modeling language:

- MAST Template Description
- Model Implementation Using the MAST Language
- Walkthrough of a Simple MAST Template
- General MAST Conventions

MAST Template Description

The basic unit of modularity for a MAST implementation of a model is the template. A template is the mathematical description of a subsystem and is contained in a "MAST Template File". The characteristic equations implemented in a template can be any combination of linear or non-linear algebraic or differential equations. No integral expressions are allowed; however, by differentiating both sides of an integral equation, you can convert it into a differential equation.

MAST Template File

A template is a MAST model contained in a text file that generally has the same name as the template, plus the extension .sin. This enables the Saber simulator to find the file upon invocation. Although the distinction is subtle, there is a difference between a template and the file in which it resides. That is, the template is the contents and the file is the container. Unless otherwise indicated, the term template refers to a model and the term template file to refer to the file containing the template.

For more information on using a template file, see the MAST Reference Manual.

Note:

The template files for all example templates in this manual are provided in the following directories:

`saber_home/example/MASTtemplates/structured` or
`saber_home/example/MASTtemplates/unstructured`

Template Header

The first line of a template is known as the header, which appears in the following general form:

```
template templatename connectionpoints = arguments
```

where template is a reserved word that identifies the contents of this file as a template, templatename is the “official” name of the template for use in a circuit description (i.e., a netlist), connectionpoints are the names of connections to the template, and arguments are the names of user-specifiable parameters. The actual names given to templatename, connectionpoints, and arguments are all selected by the writer of the template using MAST identifier and string rules.

In the following two_r template:

```
template two_r a b = ra, rb
number ra, rb
{
  r.ra   p:a      m:middle = rnom = ra
  r.rb   p:middle m:b      = rnom = rb
}
```

ASCII text of this example is located in:
saber_home/example/MASTtemplates/structured/two_r.sin

- two_r is the templatename
- a, b are the connectionpoints
- ra, rb are the arguments

MAST Declarations

Most variables within a template must be explicitly identified so that they can be recognized by the simulator—this is called declaring a variable. Variables include the template name, arguments, connection points, and other variables described in subsequent chapters.

MAST Connection Points

There are several different kinds of models that the Saber simulator uses:

- Simulation of Continuous Analog Systems (physical systems)
- Simulation of Event-Driven Systems (such as digital)
- Simulation of Data Flow Systems (control)

Typically, each category of model has its own type of connection point (event-driven uses states, for example), which communicates the characteristics of that type of model to the rest of the system. Note that it is possible to combine model types within a template, thus a template can have different types of connection points (digital and analog, for example).

Note:

The terms “pins” and “connection points” are often used interchangeably. In the MAST language, pins are a specific type of connection point (i.e., not all connection points are pins). Refer to the MAST Reference Manual for more information on connection points.

MAST Template Body

The template body is a mandatory partition in all example templates; the body must be explicitly begun with an opening brace ({) and ended with a closing brace (}). It is recommended to place each brace on its own separate line as shown in the following example. The body contains all template statements, sections, and declarations, except for the header and header declarations that precede it.

```
template two_r a b = ra, rb
  number ra, rb
{ # Template body starts here
  r.ra   p:a      m:middle = rnom = ra
  r.rb   p:middle m:b      = rnom = rb
} # Template body ends here
```

ASCII text of this example is located in:
saber_home/example/MASTtemplates/structured/two_r.sin

MAST Parameters

For a model of a physical system, coefficients of the characteristic equation are called the parameters of the model. Parameters for models of other types of systems provide similar functions—they fill in information that the model requires for a specific simulation (which may or may not appear in a characteristic equation). An argument (such as ra or rb) is a parameter whose value can be specified by a user in a netlist. Other parameters are local—the template uses them as it does arguments, but their values are not specified in a netlist.

Each parameter in a template (and thus each argument) must be declared as a particular type. The simplest parameter type is a number, which can assume the value of either an integer or a real number. Other parameter types introduced in later chapters are strings, structures, arrays, and unions.

```
template two_r a b = ra, rb
  number ra, rb       # Parameters of type number
{
  r.ra     a    middle  = ra
  r.rb    middle     b  = rb
}
```

ASCII text of this example is located in:
saber_home/example/MASTtemplates/structured/two_r.sin

MAST Template Equations

Template equations are the characteristic equations of the model, as implemented in a template. Some templates may require additional modifications to get the equations into a usable form, as described in the topic titled "MAST Equation Modifications".

MAST Sections - Optional but Recommended

The template body may be further partitioned into various sections that contain statements providing specific functionality.

With the exception of the control section, declaring sections within the template body is optional—it is most often done for clarity within large, complicated templates. The use of sections is recommended with most models. Using sections in a model is called the “structured” modeling approach. Sections help document the template by identifying segments where specific tasks are isolated.

The following example shows the general form of a template that uses sections.

```
template header
header declarations
{
 local declarations - must appear first
 parameters {
  parameter assignments and argument testing
  }
 netlist components
 values {
  value assignments
  }
 control_section {
  simulator-dependent control statements
  for non-linearities
  }
 equations {
  equations describing behavior
  }
 when (expression) {
  event-dependent assignments and scheduling
  }
}
```

The sections required vary depending on the model. Template sections can be in any order (except for the local declarations section, which must appear first).

Sections can also be used to optimize simulation efficiency.

Note:

One situation where you must use one of these explicit sections (the equations section) is explained in the topic "Modeling an Idealized Op Amp with MAST" which is an operational amplifier template with one output pin.

Variable Types in MAST Sections

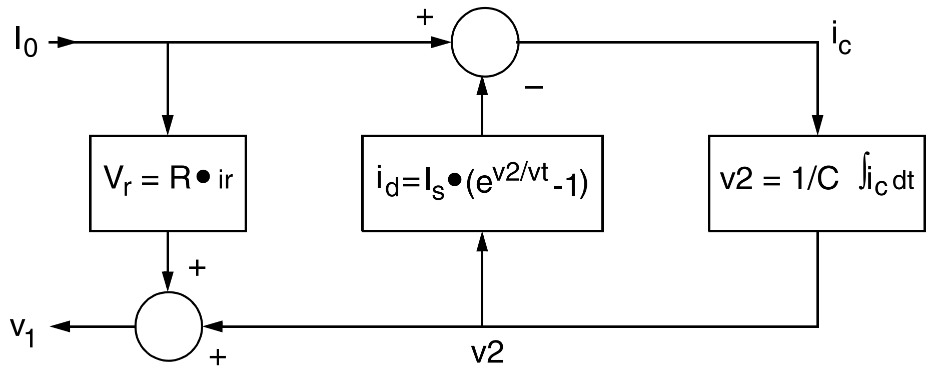
The MAST language has several types of constants and variables. The table below summarizes the different types, showing where in the template they may be declared, where their values can be assigned (left side of assignment statement), and where their values can be referenced (right side of assignment statement).

Type of Variable	Section Declared	Section Assigned	Section Referenced
number enum struc array union string	Header declaration or Local declarations	Parameter (if declared locally)	Parameters Netlist when statement Values Control Equations
val	Local declaration	Values	when statement Values Control Equations
var ref	Header declaration (var and ref) or Local declarations (var only)	none (determined by the simulator)	Netlist when statement Values Control Equations
simvar	none	none*	when statement Values Equations
state	Header declaration or Local declarations	when statement	Netlist when statement Values Equations

*Simvar variables step_size and next_time can be assigned in Values section.

MAST System Variables

The concept of through and across variables makes it possible to specify network systems in the convenient form of a netlist, as shown in the electrical network sample. Without this concept, the system could be specified only in a much less convenient form, as shown in the following figure.



Data flow representation of electrical schematic

Through and across variables are also instrumental for an efficient implementation of continuous analog systems using the MAST language.

The across variable in an electrical system is the voltage at each node with respect to the reference node. Similarly, the through variable is the current flowing through each element connected to each node. The simulator solves the simultaneous equations for the system variables, which include the across variables of the system. It does this based on the conservation of through and across variables, which, for electrical systems, are Kirchhoff's Current Law (KCL) and Kirchhoff's Voltage Law (KVL), respectively.

Branch

A conservation model (such as an electrical model that observes KCL and KVL) provides a branch when the model can connect between two nodes in a system and model a path from one node to the other.

For example, the model of an electrical resistor (having two pins) provides a single branch in a circuit—from p to m. Similarly, the model of a bipolar transistor (having three pins) provides three branches—from base to emitter, from base to collector, from collector to emitter.

This concept of branch from electrical circuit theory can be generalized to any conservation model and used as a MAST construct for templates. The use of branches is referred to as part of the unstructured modeling approach.

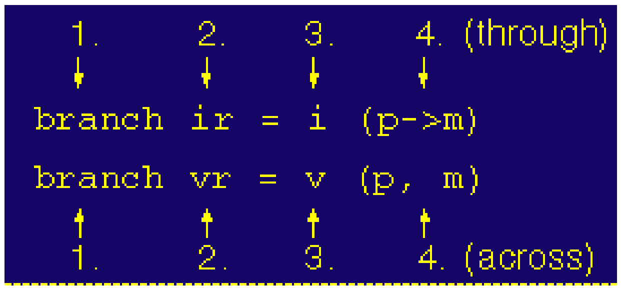
When writing a template for a conservation model, you can define the through and across variables for each branch, either explicitly or implicitly, as explained in the following topics:

Explicit Declaration

The explicit specification of a branch consists of the following:

1. The reserved word branch.
2. The branch name, which is either a through branch or an across branch.
3. The declaration of either the through or the across variable.
4. The names of the two pins defining this branch, which must match the pin names in the header declaration. For a through variable, pin names are separated by the combination of a hyphen and a right angle bracket (->). For an across variable, pin names are separated by a comma.

For example, the branch current and branch voltage of a resistor might be declared as shown below. Each number refers to the list given above.



```
branch ir = i (p->m)
branch vr = v (p, m)
```

Branches may be declared in any order within the template body. You may combine branch declarations on one line, separated by a comma:

```
branch ir=i(p->m), vr=v(p,m)
```

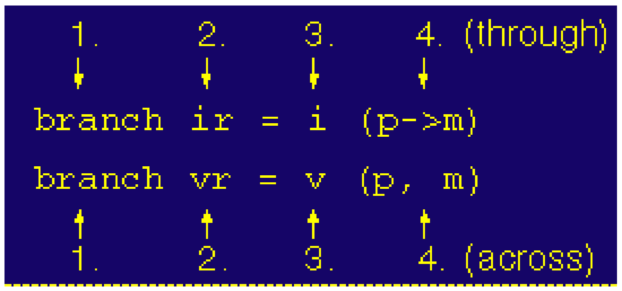
Given this declaration, you could then use branch variables ir and vr in a template equation, as shown below.

```
ir = vr/res
```

Implicit Declaration

Although not recommended, you can omit the word branch and the local name of the branch variable and just use the branch through or across variable along with the corresponding pins.

For example, in the following resistor template, the branch declarations:



```
  1.      2.     3.     4. (through)
branch  ir  =  i  (p->m)
branch  vr  =  v  (p, m)
  1.      2.     3.     4. (across)
```

could be omitted and the following template equation could be used:

```
i(p->m) = v(p,m)/res
```

Note:

Using i(p->m) or v(p,m) in this way more than once in the same template would define a different branch each time; therefore, it is recommended to use the explicit declaration method instead.

Creating a MAST System Variable

From the perspective of a template, the across variables are “given” and can be used to compute the through variables (e.g., branch currents).

Note:

For optimum template efficiency, you should algebraically rearrange the characteristic equations so that the through variables are expressed as explicit functions of the across variables.

Case 1

A resistor template should implement the following characteristic equation:

```
I = V/R   (rather than  V = I•R )
```

This is done using branch declarations as follows:

```
branch ir=i(p->m)
branch vr=v(p,m)
ir = vr/res
```

In the third line, res is the template argument. It assumes the value of resistance specified by the user in a netlist.

Case 2

Unfortunately, the choice shown in "Case 1" is not always available—a template equation may not be able to express the through variable as a function of the across variable.

Two basic examples of this are the unstructured examples for the voltage source and inductor models.

Case 3

It is possible for a conservation model to have a pin that does not define a branch. This is a situation where the through variable cannot be declared as a branch, but it is still needed in the characteristic equation. This is done by declaring the through variable as a var variable (see the MAST Reference Manual for more information).

A var variable is a local declaration that is required for the following conditions:

- The simulator cannot determine the through variable explicitly as a function of the across variable, and
- The through variable cannot be declared as branch variable

This technique is described more completely in the template for a simple, ideal operational amplifier with two input pins (ip, im) and a single output pin (out). Because no branch characteristic is modeled between the output pin and either input, the output connection does not constitute a branch. Thus, the current at out cannot be declared as a branch current.

The remedy for this situation is to declare a local variable for current at the output pin (iout) as a var variable and use it in the equations section. This declaration would be made as:

```
var i iout
```

where iout is the name of the variable and i identifies it as a current (for display units).

Case 4

The Saber simulator can solve simultaneous algebraic or differential equations that are internal to the template (i.e., equations that are not necessarily characteristic equations for a particular model). For example, the simulator can solve the following simultaneous equations, which are not specific to any model:

$x = (y - 4)/3$

$y = 4x + 3$

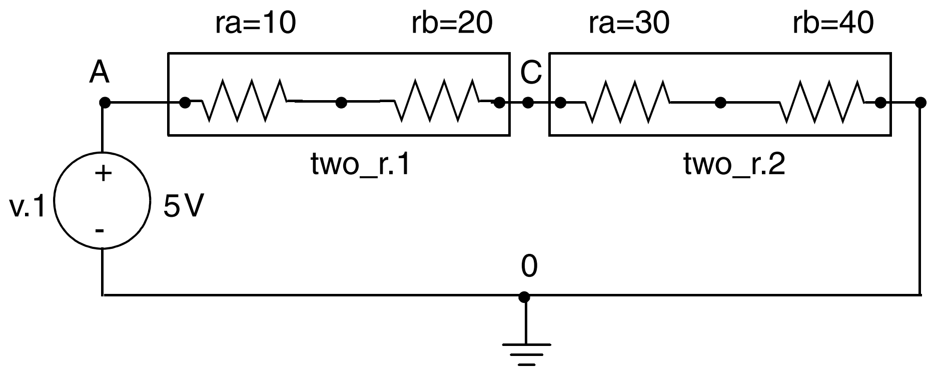
The preceding set of equations can appear in a MAST template (or as a template by itself), as follows:

```
var nu y
x = (y-4)/3
y = 4*x + 3
```

The line var nu y declares y as a var variable with no units (nu). You can perform a DC analysis or transient analysis on this template, and the simulator will solve the equations to find that x=1 and y=7. You can use this technique whenever you need to solve a system of simultaneous algebraic or differential equations.

MAST Reference Node

The topic titled "The Saber Netlist Overview" describes conservation models and provides a simple example of using templates containing such models in a netlist (a string of resistors and a voltage source). The following diagram of this netlist, lists the names of the nodes of this simple circuit: A, C, and 0.



An electrical circuit with a reference node

Note that the node connected to circuit ground is named “0”. This conforms to a convention that is used by the Saber simulator, as described below.

One of the requirements of a conservation template is that, when used in a design, the value of the across variable at any of its pins must be measurable with respect to some reference value in the design. This reference value is provided by a node in the design whose across value is always zero—the reference node of the design.

In electrical designs (circuits), the reference node is usually referred to as the ground node. However, the requirement of a reference node extends to all designs that have conservation templates (electrical, mechanical, magnetic, etc.); it serves as a generalized source/sink for all through variable quantities (current, torque, flux, etc.). Although templates of event-driven or data flow models do not have this requirement, the Saber simulator makes sure that every design has a reference node, as follows:

- If there are conservation templates in the netlist, and a node has been explicitly assigned the name 0, then it is used as the reference node.
- If there are conservation templates in the netlist, but no node has been explicitly assigned the name 0, then the Saber simulator arbitrarily selects a node to be used as the reference node. In addition, a warning message is displayed that this has been done.
- If there are no conservation templates in the netlist, then the Saber simulator automatically adds a reference node named 0 (without displaying warning message). Note that it is not possible (or necessary) to add this reference node yourself—it is just done for the convenience of programming.

Model Implementation Using the MAST Language

The following topics summarize some essential principles of implementing a MAST model, indicating the appropriate MAST constructs for each task:

1. Determine Characteristic Equations
2. MAST Equation Modifications
3. Pre-Simulation Calculations and Error Checking
4. Equation Implementation
5. MAST Model Verification and Testing

Determine Characteristic Equations

The most difficult aspect of writing a template is determining the appropriate characteristic equations. Once you have a satisfactory set of equations, the MAST implementation is relatively straightforward. Bear in mind that the objective of a large number of technical papers is to produce equations that describe the operation of some process or device—someone else may have already developed some or all of the equations you need. Conducting a search of the literature is usually preferable to developing and deriving the equations yourself.

MAST Equation Modifications

Once you have a basic set of equations, you may have to make some modifications to get the equations into a usable form.

1. Select the appropriate through and across variables. Although this is often easy, it can sometimes require some thought. The across variable of an element is the one that, when a connection point of the element is connected to a connection point of another element, has to have the same value at both connection points.

   For example, when two resistors are soldered together, their voltages are the same at the connection node. Therefore, voltage is the across variable. The through variable is the one that sums to zero at any node. In the resistor example, this is the current, so current is the through variable.

2. Rearrange the equations so that each through variable is expressed as an explicit function of the across variables. If only an implicit relationship can be expressed, then it is necessary to declare the through variable as a branch or to add one or more equations, with the through variable declared as a var variable for each new equation (see the topic titled "MAST System Variables".
3. Examine the equations for pitfalls:
   - Avoid potential singularities (division by zero). In particular, avoid division as much as possible and check denominators carefully when they do occur.
   - Avoid taking the difference of two large numbers. Digital computers can run out of resolution when taking the difference of large numbers.
   - Watch out for functions that are not well behaved, especially around zero. Troublesome functions can often be rewritten or approximated with a truncated series around unstable points.
4. Replace non-time derivatives with time derivative equivalents, because the time derivative is the only available derivative function. For example:

   $$y = \frac{dm}{dh}$$

   can be replaced with

   $$y = \frac{\frac{dm}{dt}}{\frac{dh}{dt}}$$

5. Consider whether the equations can be characterized. Many promising models fail because their parameters are impossible to determine. A simple model that can be accurately characterized is better than a complex model that cannot. Complete the characterization phase before starting the implementation phase.

Pre-Simulation Calculations and Error Checking

The next step in implementing a MAST model is to check the equations for quantities that can be calculated before the simulation begins. For example, if your equations use the square of an argument value, it is better to square it once, rather than each time it is used during the simulation. You can do this by declaring a local parameter of type number and setting it equal to the square of the argument. The squared value is then used in the model evaluation, which improves the computational efficiency of your model.

You can also perform other calculations that precede the simulation, such as unit conversions or characterization functions.

You can also examine input arguments and check for range errors when you write the model by using instance (), error (), and warning (), message functions. This way the model itself can catch these types of violations and report them to the screen.

Equation Implementation

In the simplest situations, the characteristic equations of the model can be implemented directly as a template equation. However, model equations may require the results of if-else statements, which you can specify separately from the template equations and then have them refer to the if-else values, so this requirement causes no difficulties.

If the model has discrete or event-driven characteristics, then you must use at least one when statement in the template.

Post-Simulation Calculation

It is often desirable to calculate some quantities after the simulation is complete. For example, the power dissipated by a resistor is easily calculated from the solution of the equations involving the resistor, even though the value of power dissipation is not required in the solution of the system. To obtain this value using the MAST language, it is declared as a local variable (a val variable) and calculated accordingly. You can do this for any val variable, var variable, parameter, state, etc.—the value is calculated after the simulation is complete, but only if requested, so it doesn't cost extra to define it in the template.

MAST Model Verification and Testing

A model is not complete until it has been tested and its accuracy and functionality verified. A common mistake in template development is to develop several new templates as a batch and then throw them all together to see if they work. It is much more productive to test each template individually and verify it before proceeding. In fact, it is often appropriate to develop each template in stages, implementing the bare essence of functionality first, and then proceeding with additional features, testing each one before going on to the next.

Walkthrough of a Simple MAST Template

This topic provides the actual contents of the file for a template that models an electrical resistor. The same model is shown twice, once using the structured, and once using the non-structured modeling approach. It is provided as a “walkthrough” to illustrate the most common principles of writing a MAST template and to briefly contrast the two modeling approaches.

The following text shows two MAST descriptions of the same resistor model. Either of these modeling approaches can be used to create a template that is used by the Saber simulator.

	Un-Structured	**Structured**
1	`template resistor p m = res`	`template resistor p m = res`
2	`electrical p`	`electrical p`
3	`electrical m`	`electrical m`
4	`number res`	`number res`
5	`{`	`{`
6	`  branch cur=i(p->m)`	`  equations {`
7	`  branch vlt=v(p,m)`	`  i(p->m)+= (v(p)-v(m))/res`
8	`  cur=vlt/res`	`  }`
9	`}`	`}`

By convention, the text of one of these models would be in a file called resistor.sin, so that the Saber simulator can find the file upon invocation.

Note:

It is often possible to combine declarations of multiple arguments, connection points, and branches on a single line (separated by commas). For the sake of explanation, this has not been done here.

Line 1: Template Header

The word resistor determines the “official” name of the template for use in a circuit description (i.e., a netlist). The letters p and m identify the connection points for the system variables used by the model. The equals sign (=) separates connection points from the arguments. The word res is the name of the single argument for this template.

Thus, when the simulator encounters resistor in a netlist entry, it takes the rest of the information in that entry and applies it to the model described in the template file named resistor.sin.

Lines 2 and 3: Connection Points

The word electrical implicitly defines the connection points as “pin-type”, which means they use the through and across variables of a physical system (in this case, an electrical circuit uses current and voltage). This word automatically defines the units (amperes and volts) that will be displayed in CosmosScope for these variables. There are a number of other physical systems that have corresponding words similarly defined in the units.sin file that is automatically loaded when the Saber simulator is invoked. See the MAST Reference Manual for information on the units.sin file. For example, the words magnetic, thermal, rotational_vel, and rotational_ang in this file define through and across units for magnetic, thermal, and mechanical systems.

The letters p and m are the names of the points at which the current and voltage for this resistor model are provided to/from the rest of the circuit. In an implicit netlist, these are actually mnemonic placeholders that will be assigned the names of circuit nodes in a netlist. An explicit netlist uses these names.

Note that lines 2 and 3 are usually combined into one line, as follows:

```
electrical p, m
```

Line 4: Argument Declaration

The word number defines the type of parameter that follows. See the MAST Reference Manual for information on parameter types. In this case, the parameter can assume real numerical values. The word res is the name of the argument to the template. An argument is a parameter whose value can be specified in a netlist. This user-specified value is used by the template to evaluate its characteristic equation.

Line 5: Opening Brace

This character is the required syntax for beginning the body of the template.

Line 6 - Un-Structured: Branch Through Variable

Branch through variable—The keyword branch is a MAST term for the characteristics of a conservation model between two connections in a circuit (system). In this case, the model of an electrical resistor is a single branch in a circuit. The definition of electrical connection points (declared in Lines 2 and 3) includes current, i, as the through variable for the branch between p and m.

The word cur following branch is the name of the branch variable. The equals sign (=) signifies that a definition of this branch variable is to follow. The i following the equals sign defines the branch variable as current. The hyphen followed by a right angle bracket, ->, is the notation for the through variable (current) from one pin (p) to another (m). The indicated positive direction of the variable is from p to m.

Line 6 - Structured: Equation Section

The keyword equations is used at the start of an equations section, followed by an opening brace. The equations section contains the terminal (connection point) equations of the model. Relationships involving the through and across variables must be defined in this section.

Line 7 - Un-Structured: Branch Across Variable

As with the through branch, the definition of electrical connection points (declared in Lines 2 and 3) includes voltage, v, as the across variable for the branch between p and m. The word vlt following branch is the name of the branch variable. The equals sign (=) signifies that a definition of this branch variable is to follow.

The v following the equals sign defines the branch variable as a voltage. The comma separating p and m is the notation for the across variable (voltage) between one pin (p) and another (m).

Line 7 - Structured: Characteristic Equation

This equation describes the relationship between the through and across variable.

Note:

For the simplicity of this example, no provision was made to check for res=0. This is called parameter checking and is explained in a subsequent chapter.

Line 8 - Un-Structured: Characteristic Equation

This is an expression of the terminal (connection point) equations as the relationship of the through and across branch variables. The through variable (cur) should be computed as a function of the across variable (vlt) whenever possible.

Thus, the characteristic equation for this resistor template is

```
cur=vlt/res
```

instead of

```
vlt=cur*res
```

Note:

For the simplicity of this example, no provision was made to check for res=0. This is called parameter checking.

Line 8 - Structured: Closing Brace

This character is the required syntax for ending the equations section.

Line 9: Closing Brace

This character is the required syntax for ending the template body, indicating to the simulator that it has reached the end of template.

General MAST Conventions

- All across variables are determined with respect to a reference value. In a netlist, the MAST convention for specifying the reference point is the special node name 0. For more information, see the topic titled "MAST Reference Node".
- The Saber simulator solves the system of equations such that the through variables at every node sum to zero (KCL is a special case of this). It is important, therefore, to know what the Saber simulator considers to be a positive contribution and a negative contribution to the through variable of a node. The convention is that the through variable contribution is positive when it goes into a template and negative when it comes out of a template.
- Typically, each template is contained in its own file that has the same name as the template (as it appears in the header). However, the file name must be followed by a .sin extension so that the simulator will automatically find and include the template whenever it finds a reference to it in a netlist.

  For example, the file containing the two_r template should be named two_r.sin (although the file could be named something like twores.sin and it would still work).

- When possible, it is a good idea to define connection points in the header of a template such that the inputs (if any) to the model are specified first, followed by the outputs. This is not absolutely necessary, but it helps make the templates more intuitive for most people.
- The Saber simulator is capable of simulating continuous analog, event-driven, and data flow models (i.e., it is not restricted to electrical models). Generally, you can look at the declaration of connection points in a template to determine which type of model it is. The units.sin file specifies units for several types of connection points.

  For example,

  The word electrical preceding a connection point name means that it is a pin-type connection with current and voltage as through and across variables, respectively.

  The words rotational_vel or rotational_ang preceding the connection point name means that it is a pin-type connection with torque or angular velocity as through and across variables, respectively.

  The word state preceding a connection point name means that it is an event-driven connection (no through and across variables). There is also a unit declaration following the word state, such as nu (no units) or logic_4 (4-state logic units), defining the unit of the state for display in Saber Sketch.

3
Basic Modeling

The topics in this chapter introduce concepts of the MAST modeling language. Templates of electrical elements are used as examples to introduce new concepts. The topics are ordered so that the first one introduces concepts that the later topics build on. The later topics add new concepts.

MAST Modeling Examples - Electrical Elements

The MAST Template Library includes templates for most of the functionality modeled in these examples.

- Modeling a Constant Current Source (isource)
- Modeling a Linear Resistor with MAST (resistor)
- Modeling a Linear Capacitor with MAST (capacitor)
- Modeling a Constant Voltage Source with MAST (vsource)
- Modeling a Linear Inductor with MAST (inductor)
- Modeling a Current-Controlled Voltage Source with MAST (cvt)
- Modeling Mutual Inductance with MAST (mutind)
- Preserving Hierarchy - MAST Template with Netlist (rlc1)
- Flattened Hierarchy - MAST Template with Equations (rlc2)
- Mixed Hierarchy - MAST Template with Netlist and Equations (rlc3)
- Modeling Extractable Capacitor Voltage and Charge with MAST (capacitor_1)
- Modeling Multiple-Mode Voltage Source with MAST (vsource_1)
- Modeling a Linear Transformer with MAST (xformer)
- Modeling a Temperature-Dependent Resistor with MAST (resistor_1)
- Modeling an Idealized Op Amp with MAST (opamp)

For simplicity, some of these templates model ideal elements that ignore certain types of information (such as noise or temperature effects) that might otherwise be included.

MAST Modeling Concepts using Electrical Elements

Some of the concepts introduced in one or more of the templates listed in the previous topic titled "MAST Modeling Examples - Electrical Elements" are as follows:

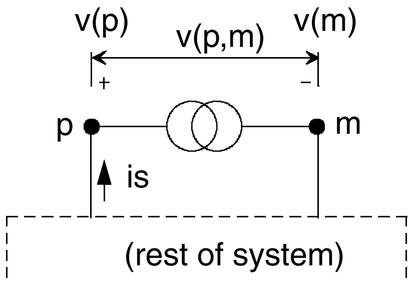
- Starting the isource MAST Template -- shows the general form of a template
- isource Netlist Example -- shows the correspondence between the template header and an instance of the template in a netlist (netlist entry)
- Characteristic Equation for a Linear Resistor -- describes selecting a template equation so that the through variable (current) is computed as a function of a system variable (voltage)
- Equations Section -- describes the time derivative operator, d_by_dt, used in the template equation
- Equations Section and Local Declarations -- describes the local declarations section of a template, describes explicitly declaring var variables, a new system variable type, and describes assigning an equation to a var variable, where there must be exactly one equation for each var variable in the equations section of a template
- Equation Section -- shows how to use a var variable to change an integration formulation into a differentiation formulation
- Solving for Across Variables at a System Node -- provides an example that is intended to clarify why a template equation should express the through variable in terms of the across variable
- Using MAST System Variables Between Models -- provides two examples to show how a template can use the value of a system variable from another template without an explicit pin connection between them by using a ref connection point
- The chapter on Modeling Hierarchical Systems -- introduces the following examples:
    - Preserving Hierarchy - MAST Template with Netlist -- refers to the previously-created resistor, capacitor, and inductor templates

- Flattened Hierarchy - MAST Template with Equations -- describes the whole system by its equations, which results in a flat description of the system
- Mixed Hierarchy - MAST Template with Netlist and Equations -- describes element templates and their properties and describes rules for deciding whether a template should be an element template or an ordinary template
- Modular vs. Non-Modular MAST System Descriptions -- provides a summary of the characteristics of the hierarchical system and the flat system

▪ The chapter on Variables and Arguments -- offers topics describing ways of using internal variables and writing arguments to provide more template versatility. They also demonstrate how to check the validity of argument values, use message functions, and improve template performance by precalculating expressions.

Modeling a Constant Current Source

The example for a source of constant current (or of any “through” variable) is one of the simplest analog templates that can be written in the MAST language. The following figure shows the symbol used for the constant current source, as well as the variable names used in the template.



Constant current source

The contents of the constant current source template are shown below, followed by an explanation of the contents.

```
template isource p m = is
  electrical p,m
  number is
{
  equations {
    i(p->m) += is
  }
}
```

```
ASCII text of this example is located in:
install_home/example/MASTtemplates/structured/isource.sin
```

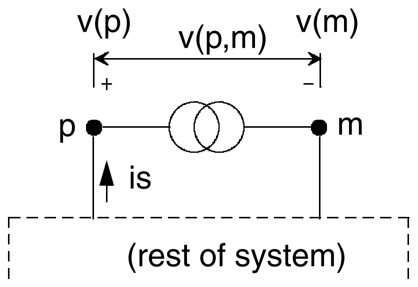
Concepts Introduced Using this Example

The description of the constant current source template (isource) introduces the following concepts:

- Characteristic Equation For a Constant Current Source
- Starting the isource MAST Template, shows the general form of a template
- Template Header
- isource Netlist Example, shows the correspondence between the template header and an instance of the template in a netlist (netlist entry)
- isource Header Declarations, which include "Declaring Connection Points" in the isource MAST Template and "Declaring Arguments in the isource MAST Template"
- Template Body
- Equations Section
- Syntax Guidelines for isource Template

Characteristic Equation For a Constant Current Source

As shown in the following figure, the current is enters the current source at connection point p and leaves the source at connection point m. The value of is is user-specified for an instance of this template in a netlist; it is therefore an argument for the source model. That is, if isource is used in a netlist, the current flowing between the system nodes to which pins p and m are connected is determined by the value of current specified for is.



Constant current source

In an ideal constant current source, the current supplied is independent of the voltage across the connections. A current source model expresses this relationship as follows:

- Let is = constant
- Choose v(p,m) such that Kirchhoff's Current Law is satisfied. (That is, the current between nodes p and m has a value of is).

Starting the isource MAST Template

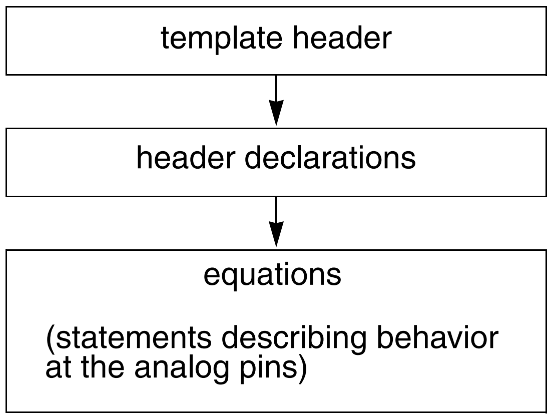
You create the template file using a text editor that can create an unformatted text file. You should give the file the same name as the template it contains, with the addition of the .sin extension. Because the name of this template is to be isource, this file must be named isource.sin.

Giving the file the same name as the template is a convention, not a requirement. Using this convention provides the following significant advantage:

> When the Saber simulator encounters a reference to a previously undefined template x, it tries to find the template in a file named x.sin and, if successful, automatically includes the template. If you do not follow this convention, you may need to include the name of the file containing template x in the netlist.

This topic develops the parts of the template in the order shown below, to clarify the use of declarations.

The syntax for this template appears in the following sequence.



When the simulator reads a template, it ignores white space (spaces, tabs, and blank lines), except when the white space separates two names. In that case, it replaces any amount of white space with a single space.

When you write a template, you must give each variable (including connection points) a unique name that is not a reserved word. Such variable names are local to a template; they distinguish the constituent parts of a template, but they do not have semantic meaning to the simulator (for example, you could name a voltage variable current). It is therefore good practice to choose variable names that reflect variable usage.

As shown in the isource template, the following names are used:

isource	name of the template
p	the plus connection point
m	the minus connection point
is	the argument specifying the constant current

```
template isource p m = is
  electrical p,m
  number is
{
  equations {
    i(p->m) += is
  }
}
```

Template Header

The template header declares the name of the template, connection points, and any arguments through which particular characteristics can be assigned to the template in a netlist.

The form of the standard template header is:

```
template template_name connectionpoints = arguments
```

where template is a required reserved word, templatename is the name you give the template, connectionpoints are the names you give the connection points, and arguments are names of parameters to which you can assign specific values (arguments must be separated from connectionpoints by an equals sign, =). Note that arguments and connectionpoints are optional, although all templates in this manual have at least one of each.

The name of this template is isource, and the names of its connection points are p and m. It has only one argument, is, which is the user-specified value of current to be provided by the source.

Therefore, the template header for the template isource is:

```
template isource p m = is
```

On the left-hand side of the equal sign (=), space separate all names. (An alternative is to use commas to separate connection point names.) On the right-hand side of the equal sign (=), commas must separate all names. For example, if two arguments were to be used in the template, such as is and tc, the header would be:

```
template isource p m = is, tc
```

The template header tells how the template can be used in a system, as defined by a template instance. A template instance is a specific usage of a template by the simulator, as specified in a netlist. A template instance defines which argument values to use and where to place the connection points in the design.

isource Netlist Example

Suppose a system description (netlist) contains an instance of this constant current source template, isource, specified as follows:

unique identifier	i1
connected to nodes	a, b
argument value (amperes)	2

The following statement, a netlist entry, places this instance of the isource template into the system description (design), naming both the nodes to which it is connected and its argument value:

isource.i1 a b = is=2

Note the correspondence between this netlist entry for the isource template and its template header:

```
isource.i1 a b = is=2            ◄──────────────  netlist entry
template isource p m = is          ◄──────────  template header
```

The netlist entry designates this instance of isource as i1, connects pin p of the template to node a, connects pin m to node b, and specifies the value of is as 2.

isource - MAST Header Declarations

The header declarations section is where you declare all the names used in the template header, except the template name. For the isource template, the two kinds of declarations in the header declarations section are:

- connection points, p and m
- argument, is

```
template isource p m = is    # Template Header
  # Header declarations - connection points
  # Header declarations - argument declarations
{
  #equations section
}
```

Declaring Connection Points in the isource MAST Template The connection points (or pins) of a template are of a particular type. Depending on the system being modeled, the type of pin could be mechanical, thermal, electrical, etc., or any combination thereof. Because the current source is an electrical model, declare its pins to be electrical, using the following statement (comma-separated names of the same type can appear in the same statement):

```
electrical p, m
```

The pin type (electrical) appears first, followed by the pin names (p, m). The word electrical is a pin definition provided in the units.sin file that identifies the type of node to which these pins may be connected. This is an “include” file that is automatically used by the Saber simulator upon invocation. Included in this definition are the through and across variables that the simulator solves for at this type of node, along with their respective units. In this case, the through variable for an electrical node is defined as current in amperes; the across variable is defined as voltage in volts.

The across variables are those system variables that are equalized by the simulator when two or more pins are connected to the same node. An across variable adheres to a generalization of Kirchhoff’s Voltage Law: the sum of across variables (e.g., voltage) around a closed loop is zero. The simulator makes the voltage v(p) at pin p and the voltage v(m) at pin m available (or “known”) inside the template.

The through variables are those system variables that are conserved at a node, adhering to a generalization of Kirchhoff's Current Law: the sum of through variables (e.g., current) flowing out of a node is zero. The current i(p) at pin p and the current i(m) at pin m are not available inside the template, which can only define current contributions to these nodes. In this example, the source current contributes to both i(p) and i(m).

Declaring Arguments in the isource MAST Template According to the isource characteristic equation, this constant current source has one parameter (is) that can be assigned a value each time it appears in a netlist. By including is in the template header and declaring it in the header declarations, it becomes an argument of the current source template.

Template arguments are parameters that can receive values when the template is used in a netlist statement. This distinguishes them from parameters that are declared locally in templates. The topic titled "Modeling an Idealized Op Amp with MAST" introduces local parameters.

Each parameter in a template (and thus each argument) must be declared as a particular type. Because the source current in this example (is) is a single numerical value, its type is declared as number. This is the simplest parameter type—the simulator does not distinguish between integers and real numbers. Other parameter types are introduced in later chapters. For a complete description of all types and how to use them in declarations, refer to the MAST Reference Manual.

Thus, is is declared as an argument as follows:

```
number is
```

In this declaration, no default value has been assigned to is (although there could be one). Consequently, whenever isource appears in a netlist entry, the value of the is argument must be specified.

The following lines of this template have been covered so far:

```
template isource p m = is  # template header
  electrical p, m          # pin declarations
  number is                # argument declaration
```

Template Body

All MAST templates in this manual contain a template header, header declarations, and a body. The body is begun with an opening brace, {, and ended with a closing brace, }. In this template, the body contains the equations section:

```
template isource p m = is
  electrical p,m
  number is
{           # Beginning of Template Body
   # equation
   # section
   # goes here
}           # End of Template Body
```

Equations Section

The next step is to express the characteristic equation of the constant current source using the constructs of the MAST language. This is done with the template equation, which describes the effect of the continuous analog portion of a template at its connection points.

In the constant current source figure, the source current is enters at pin p and leaves at pin m. That is, if isource is used in a netlist, the current flowing between the nodes to which pins p and m are connected is modified by the source current from isource. In the MAST language, the simplest way to express this is:

i(p->m) += is

The terms in this equation include:

i(p->m)	the current flowing from the node connected to pin p to the node connected to pin m.
is	the user-specified current contributed by the source.

+=	a MAST language operator meaning “is added to”. Its counterpart is -=, which means “is subtracted from”

A general form of a template equation expressing terminal characteristics is shown as follows:

```
  through(pin1->pin2) = operator expression
```

through	is the name of through variable associated with pin1 and pin2, depending on their declaration (for instance, i (current) is the through variable for pins declared as electrical).
pin1->pin2	is the notation for the through variable of the branch from pin1 to pin2.
operator	is either += or -=
expression	is a MAST expression that specifies the through variable of the branch. See the MAST Reference Manual for more information on expressions. The expression can include variables of different types (usually arguments and parameters), mathematical functions, the algebraic operators +, -, *, /, and ** (for addition, subtraction, multiplication, division, and exponentiation, respectively), parentheses (()), and, with restrictions that are described in subsequent chapters, the special operators d_by_dt (time derivative) and delay (ideal delay).

This statement has the following interpretation:

> The value of the variable flowing through the template between the node connected to pin1 and the node connected to pin2 becomes equal to the amount specified by expression.

For a constant current source, this means that a current of the amount is is provided between the node connected to pin p and the node connected to pin m. Because an independent current source is not affected by its branch voltage, no equations need to be provided for voltage across the source (i.e., the branch voltage becomes whatever is required to maintain the specified level of branch current, is).

Therefore, the following equation is a complete description of a constant current source:

```
equations {
  i(p->m) += is
}
```

Syntax Guidelines for isource Template

When putting a template together in final form, remember the following general syntax guidelines:

- Use extra spaces, indentation, and blank lines at will.
- Use comments. Comments begin with a pound sign (#) and continue to the end of the line. They can begin anywhere on a line.
- Place complete statements on separate lines.
- Place the body of the template (all sections below the header declarations) between braces.

With these rules in mind, the current source template is repeated below, with a comment on each line following the # sign.

```
template isource p m = is
  electrical p,m
  number is
{
  equations {
    i(p->m) += is
  }
}
```

Modeling a Linear Resistor with MAST

This topic defines a simple resistor template as shown below:

```
template resistor p m = res
  electrical p,m
  number res
{
  equations  {
    i(p->m) += (v(p)-v(m))/res
  }
}
ASCII text of this example is located in:
install_home/example/MASTtemplates/structured/resistor.sin
```

The description of the resistor template is divided into the following topics:

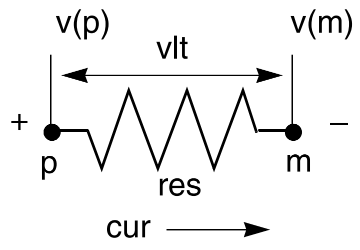
- Characteristic Equation for a Linear Resistor -- describes selecting a template equation so that the through variable (current) is computed as a function of a system variable (voltage)
- Selecting Names for a MAST Linear Resistor Template
- Creating the template header and template header declarations
- Equations Section

Characteristic Equation for a Linear Resistor

The voltage, VR, across an ideal resistor is equal to the current through the resistor (IR) multiplied by the value of resistance, R. Thus, the characteristic equation of an ideal resistor is that of Ohm’s law, namely:

Vr = Ir * res

The resistance, res, characterizes each instance of a resistor and is therefore written as an argument of the template. The following figure shows the variables contained in the resistor template.



Ideal resistor

The equation would be expressed in terms of these template variables as:

vlt = cur * res

where:

vlt	is the voltage across the resistor (from p to m -- v(p)-v(m))
cur	is the current through the resistor (from p to m-- i(p->m))
res	is the non-zero resistance value, the template argument

However, because the value of the across variable at the system node is known inside the template, this equation should be written in the template such that the current is a function of the voltage, namely:

```
cur = vlt/res
```

Selecting Names for a MAST Linear Resistor Template

In this example, the following names have been chosen for the user-specifiable items in the template:

resistor	template name
p	plus pin
m	minus pin

res	argument (resistance)

```
template resistor p m = res
  electrical p,m
  number res
{
  # equations section here
}
```

Template Header and Header Declarations

The template header and declarations for the resistor template are similar to those for the isource template, the current source template, as shown below:

Resistor Template

```
template resistor p m = res
  electrical p,m
  number res
{
  # equations section here
}
```

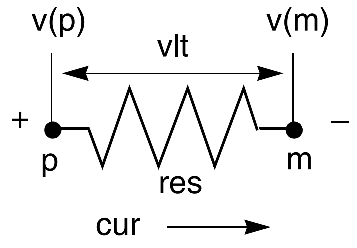
Current Source Template

```
template isource p m = is
  electrical p,m
  number is
{
  # equations section here
}
```

Equations Section

The equations section contains the characteristic equation of the resistor written as a MAST template. Referring to the following Ideal resistor figure, the voltage across the resistor is given as the difference between the voltages at its pins:

vr = v(p) - v(m)



Ideal resistor

Because both p and m are electrical, v(p) and v(m) are implicitly declared as system variables. That is, the simulator supplies their values. Therefore, the current ir through the resistor is given by:

```
cur = (v(p) - v(m))/res
```

This is the amount of current that the resistor contributes to the current flowing from pin p to pin m. (From now on, this manual uses this form, rather than the more complicated, but exact statement: the current flowing from the node connected to pin p to the node connected to pin m.)

Thus, the equations section of the template is:

```
equations  {
  i(p->m) += (v(p)-v(m))/res
}
```

Note:

For simplicity, no provision was made in this example to check for res=0. This is done with conditional statements in the resistor_1 template in the topic "Modeling a Temperature-Dependent Resistor with MAST".

Modeling a Linear Capacitor with MAST

The template for a linear capacitor uses the same sections as the current source and resistor templates. The capacitor template appears as follows:

```
template capacitor p m = cap
  electrical p,m
  number cap
{
  equations  {
    i(p->m) += d_by_dt(cap*(v(p)-v(m)))
  }
}
ASCII text of this example is located in:
install_home/example/MASTtemplates/structured/capacitor.sin
```

The description of the capacitor template is divided into the following topics:

- Selecting Names for a MAST Linear Capacitor Template
- Template Header and Header Declarations
- Characteristic Equation for a Linear Capacitor
- Equations Section -- describes the time derivative operator, d_by_dt, used in the template equation

Selecting Names for a MAST Linear Capacitor Template

In this example, the following names have been chosen for the user-specifiable items in the capacitor template:

capacitor	template name

p	plus pin
m	minus pin
cap	argument (capacitance)

```
template capacitor p m = cap
  electrical p,m
  number cap
{
  # equations section here
}
```

Template Header and Header Declarations

The template header and declarations for the capacitor template are similar to those for the isource template (the current source template) and the resistor template as shown below:

Capacitor Template

```
template capacitor p m = cap
  electrical p,m
  number cap
{
  # equations section here
}
```

Current Source Template

```
template isource p m = is
  electrical p,m
  number is
{
  # equations section here
}
```

Resistor Template

```
template resistor p m = res
  electrical p,m
  number res
{
  # equations section here
}
```

Characteristic Equation for a Linear Capacitor

The current, Ic, through any capacitor is defined as the derivative of the charge (Qc) on the capacitor with respect to time:

Ic = dQc/dt

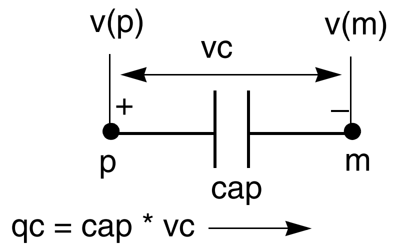
For a linear capacitor, the charge is defined as the product of the capacitance, C, of the capacitor and the voltage (Vc) across it:

Qc = C•Vc

Therefore, capacitor current can be expressed in terms of capacitance and voltage:

Ic = d(CVc)/dt

The capacitance (C) in Equations 2 and 3 characterizes each instance of a capacitor and is therefore provided as an argument of the template (cap). The following figure shows the symbol and the relevant characteristics of a linear capacitor, including the variables to be included in the ideal capacitor template.



Linear capacitor

Equations Section

To write the equations section of the template, you need to express the voltage across the capacitor in terms of system variables. Because both p and m are electrical pins, this becomes:

```
vc = v(p) - v(m)
```

The simulator solves for the values of v(p) and v(m) as system variables, so they are known to the template.

Referring to the characteristic equation, the current (ic) of the capacitor is given as the time derivative of its charge. In the MAST language, taking the time derivative of an expression is represented by applying the d_by_dt operator to the expression.

In this example, this becomes:

```
ic = d_by_dt(cap * (v(p) - v(m)))
```

The d_by_dt operator can operate on any expression that does not include a delay operator or another d_by_dt operator. It can appear only in a template equation, and you can only add terms to it and subtract terms from it. Therefore, the expression cap * d_by_dt(v(p) - v(m)), while valid in conventional calculus, is not valid in the MAST language. Higher order derivatives are implemented with multiple d_by_dt statements.

With the contribution of the capacitor template to the current flowing from pin p to pin m, the equations section is as follows:

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

It is sometimes useful to specify, in the equations section, the current contribution of a component to each pin individually, rather than to a current flowing between two pins. Using the equations section of the capacitor as an example, the statement:

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

can also be interpreted as the following:

- Add the current defined by the expression to the current at pin p

  AND
- Subtract the current defined by the expression from the current at pin m

The MAST language lets you express these two facts separately:

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

For the Saber simulator, these two statements are equivalent to the single statement above in every respect, including the amount of work it must do to set up and solve the equations. Which formulation you use usually depends only upon your personal preferences. The single-statement formulation automatically guarantees that current is conserved in the template. The pin-oriented formulation places this burden on the template writer. On the other hand, the pin-oriented formulation provides more flexibility, which can be important in advanced cases.

Modeling a Constant Voltage Source with MAST

The template describing a constant voltage source is as follows:

```
template vsource p m = vs
  electrical p, m
  number vs
{
  var i ivs
  equations  {
    i(p->m) += ivs
    ivs: v(p) - v(m) = vs
  }
}
ASCII text of this example is located in:
install_home/example/MASTtemplates/structured/two_r.sin
```

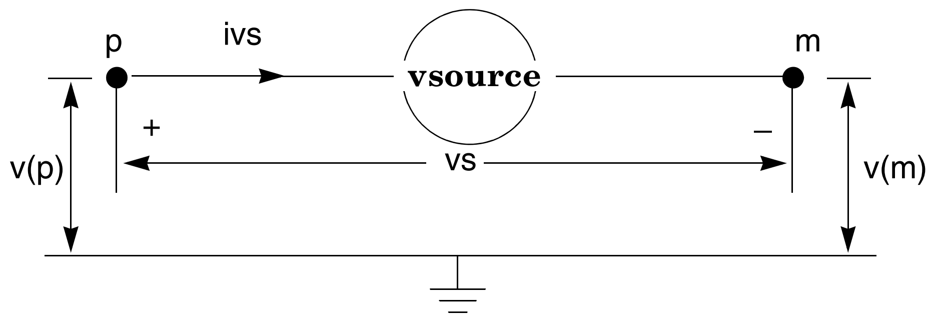
Constant Voltage Source Topics

The description of the vsource template is divided into the following topics:

- Characteristic Equation for a Constant Voltage Source
- Equations Section and Local Declarations - describes the local declarations section of a template, describes explicitly declaring var variables, a new system variable type, and describes assigning an equation to a var variable, where there must be exactly one equation for each var variable in the equations section of a template

Characteristic Equation for a Constant Voltage Source

In an ideal constant voltage source, the voltage supplied is independent of the current passing through the connections. The following figure illustrates the voltage source; it provides a constant voltage (vs) across pins p and m. The value of vs is user-specified for an instance of this template in a netlist; it is therefore an argument for the source model.



Voltage source

A voltage source model expresses this relationship to the simulator as follows:

Let vs = constant.
Choose ivs such that Kirchhoff's Voltage Law is satisfied, meaning the voltage between nodes p and m has a value of vs.

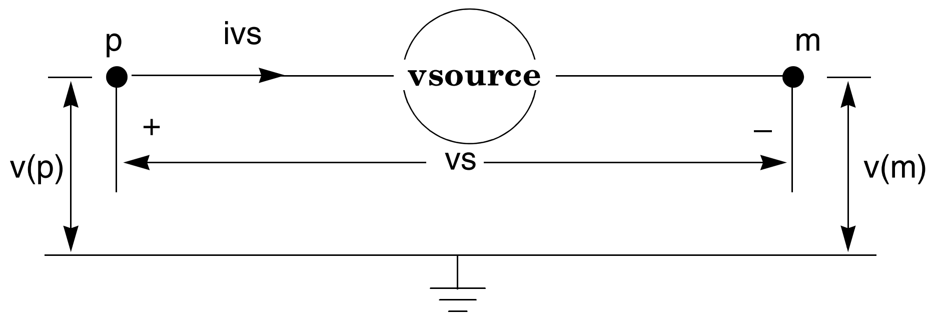
The first statement defines the vs parameter as characterizing the voltage source; it is declared as an argument to the template. The second statement recognizes that ivs cannot be determined by the source template alone.

Instead, its value depends upon the system to which the constant voltage source is connected.

That is, if vsource is used in a netlist, the voltage across pins p and m is held at the value specified. Because there is no dependency between current and voltage in an independent voltage source, a provision must be made for the current through the source.

Equations Section and Local Declarations

Given the following figure and characteristic equation:



Voltage source

Let vs = constant.
Choose i(p->m) such that Kirchhoff's Voltage Law is satisfied, meaning the voltage between nodes p and m has a value of vs.

The equations for the connection points are as follows:

```
i(p->m) += ivs
ivs: v(p) - v(m) = vs
```

One piece is missing with just these two lines. The simulator needs to know that it must solve for ivs such that Kirchhoff's Voltage Law (KVL), the second equation, is satisfied. This means that ivs must be a system variable. The simulator solves for system variables, such as the across variables at the pins.

Do this with the following statement in the local declarations section of the vsource template:

```
var i ivs
```

In general, the statement:

```
var unit name[, name ...]
```

declares one or more system variables with the specified unit.

Many such units are provided in the Saber simulation environment. You can define additional units for your own application (refer to the MAST Reference Manual, “Unit Definition”). Units are currently used only to ensure consistency of connection points and for the labeling the axes of graphs. In addition, they help you to associate simulation results with physical interpretations.

A var variable is a system variable, much like implicitly declared system variables. When you declare pin p to be electrical, this implies that there is a system variable v(p). In particular, a var variable can be used anywhere an implicitly-declared system variable can be used. The important difference between implicitly-declared system variables and var variables (which are declared explicitly) is that the template writer must tell the simulator which equation to use to solve for the var variable. The simulator uses KCL to solve for implicitly declared system variables.

The syntax for associating an equation with a var variable is as follows:

```
var_variable : expression1=expression2
```

where:

var_variable	is the name of a var variable declared earlier in the template in a var variable declaration.
expression1 expression2	are two expressions formed from variables of different types (but not through variables); mathematical functions; the algebraic operators +, -, *, /, and **; parentheses (()); and the special operators d_by_dt and delay

The simulator interprets this statement as the following:

Find the value of var_variable such that expression1 equals expression2. For the constant voltage source, the statement associating an equation with ivs is as follows:

```
ivs: v(p) - v(m) = vs
```

which is declarative and means:

Find ivs such that the voltage across the voltage source equals the specified vs value.

This now specifies the voltage source equation completely, and the equations section is as follows:

```
equations {
  i(p->m) += ivs
  ivs: v(p) - v(m) = vs
}
```

Modeling a Linear Inductor with MAST

The template describing a linear inductor is as follows:

```
template inductor p m = ind
  electrical p, m
  number ind
{
  var i il
  equations  {
    i(p->m) += il
    il: v(p) - v(m) = d_by_dt(ind*il)
  }
}
ASCII text of this example is located in:
install_home/example/MASTtemplates/structured/inductor.sin
```

Linear Inductor Topics

The description of the inductor template is divided into the following topics:

- Characteristic Equation for a Linear Inductor
- Equation Section -- shows how to use a var variable to change an integration formulation into a differentiation formulation
- Header and Header Declarations

Characteristic Equation for a Linear Inductor

The voltage, VL, across an inductor is defined as the derivative of magnetic flux (f) with respect to time (Faraday's Law):

VL = df/dt

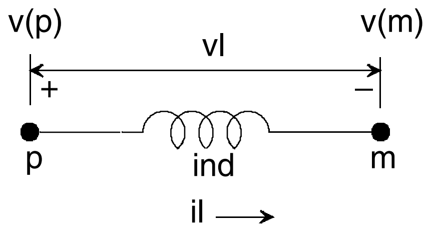
For a linear inductor, the flux is defined as the product of the inductance, L, of the inductor and the current (IL) through it:

f = L•IL

Therefore, inductor voltage can be expressed in terms of inductance and current:

VL = d(LIL)/dt

The inductance (L) characterizes each instance of an inductor and is provided as an argument of the template (ind). The following figure shows the symbol and the relevant characteristics of a linear inductor, including the variables to be contained in the ideal inductor template.



Ideal inductor

Equation Section

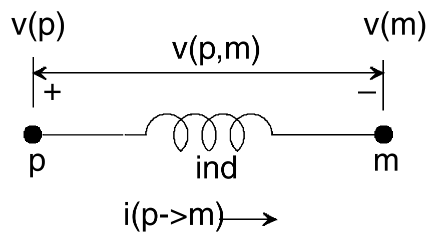
As with the resistor template, you should try to express the inductor current (the through variable contribution) as a function of the voltage across the inductor, which would be:

$$il = 1/ind \int vl\ dt$$

Although this equation serves as a compact implementation of the model, it requires the use of an integral, which the MAST language does not support. Therefore, it is necessary to use a new approach that differs from the one used to implement the resistor and capacitor templates.

Note:

When you encounter an integral expression as in the previous equation, you need to differentiate both sides of the equation to eliminate the integral.



Ideal inductor

As with the constant voltage source, create a new system variable by declaring a var variable as follows:

```
var i il
```

You do this in the local declarations section of the template.

In the equations section, the equation associated with il is the characteristic equation of the inductor, shown as follows:

```
il: v(p) - v(m) = d_by_dt(ind*il)
```

This statement can be interpreted as follows:

> The current through the inductor, il, is to be such that the voltage drop is the derivative of ind*il, this flux.

Consequently, by introducing another system variable, you replace the integration formulation for the inductor current with a differentiation formulation for the inductor voltage.

In the equations section, you must still define the contribution of the inductor branch current. With il as a var variable that receives this value, you can use the following statement:

```
i(p->m) += il
```

The entire equations section of this template is as follows:

```
equations  {
  i(p->m) += il
  il: v(p) - v(m) = d_by_dt(ind*il)
}
```

Header and Header Declarations

The template header and the header declarations are very similar to those in other templates (ind has been chosen as the name for the user-specifiable value of inductance):

```
element template inductor p m = ind
  electrical p,m
  number ind
```

Note that the word element has been inserted as the first word of the header. This has been done to make the branch current available for mutual inductance. Additional reasons for using the word element in a template header are explained in the topic titled "Using MAST Element Templates".

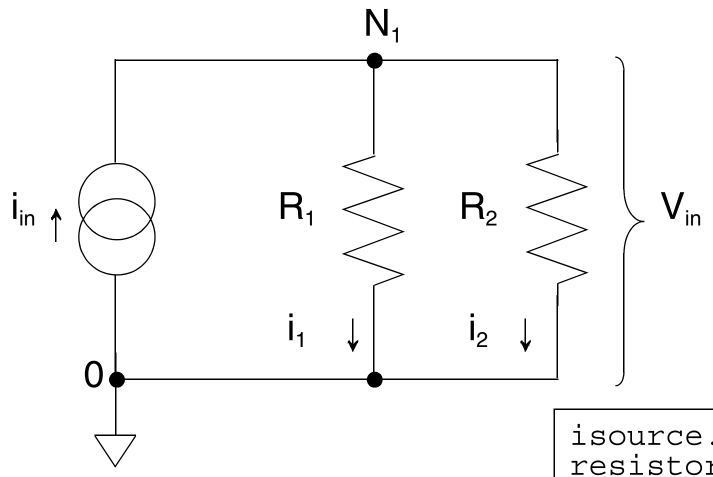
Solving for Across Variables at a System Node

This topic provides an example that is intended to clarify why a template equation should express the through variable in terms of the across variable. Basically, it results from the principle shown as follows:

- The simulator calculates the value of an across variable at a system node and makes this value available to templates connected to that node—This is not done for the through variable.
- The known across value (voltage) at the system node is then "internalized" by the template to solve for the through value (current).

Consider node N1 of the simple circuit shown in the following figure, consisting of the isource and resistor. The characteristic equation for the resistor template is written to express current as a function of voltage:

IR = V_R/R



```
isource.I1in 0 N1 = 1
resistor.R1 N1 0 = 1
resistor.R2 N1 0 = 1
```

Circuit showing through and across variables

This is done although it would be equally valid to express the same relationship with voltage as a function of current:

VR = I_R•R

However, it is Equation 7 that defines the through variable contribution of each resistor model to the circuit shown in the figure above. This is the optimum representation for use by the Saber simulator. Why is this so?

Examining node N1, the system must observe Kirchhoff's Current Law (KCL) as follows:

S i = 0   or $i_{in} + (-i_1) + (-i_2) = 0$

Therefore, the simulator is solving the following problem:

Find V_{in} such that $i_{in} - i_1 - i_2 = 0$

It is extremely important to note that, because $i_1 = V_{in}/R_1$ and $i_2 = V_{in}/R_2$, the simulator makes the following substitution for i_1 and i_2 in Equation 9:

$i_{in} - (V_{in}/R_1) - (V_{in}/R_2) = 0$

That is, to observe KCL at N1, the simulator solves an equation for V_{in} (by rewriting to obtain Equation 11). Notice that i_1 and i_2 are no longer in the equation—the solution to Equation 11 produces only the across variable, V_{in}, at node N1.

$$\begin{tikzcd} \text{Equation 9} & i_{in} + (-i_1) + (-i_2) = 0 \arrow[d, "\text{Substitute}"] \\ \text{Equation 10} & i_{in} - (V_{in}/R_1) - (V_{in}/R_2) = 0 \arrow[d, "\text{Rewrite}"] \\ \text{Equation 11} & V_{in} = \dfrac{i_{in}}{1/R_1 + 1/R_2} \end{tikzcd}$$

Simplified view of how the simulator solves for through variables only

This results in the following conclusions:

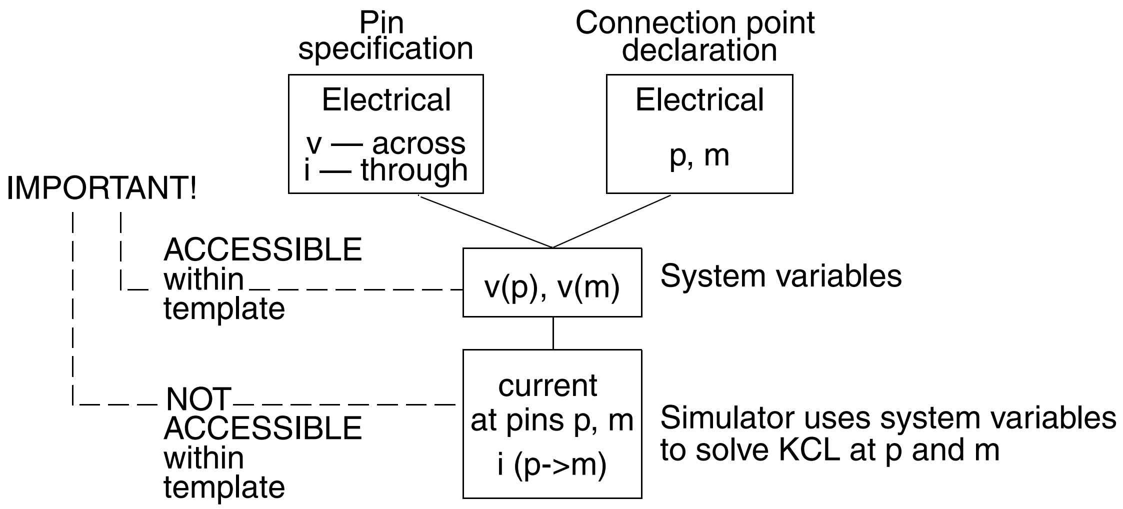
1. The across variables in the system (such as V_{in}) can be treated as “known” in the resistor template (or any other template connected to N1)—where it is provided as the voltage at p (abbreviated as v(p)). Thus, Vin is referred to as a system variable because it is known at system node N1.
2. The through variables in the system (such as i_1 and i_2) cannot be treated as known in the resistor template (or any other template connected to N1). The current at p (abbreviated as i(p)) is not provided, because the simulator doesn’t solve for it.

   Note:

   The values for the through variables i1 and i2 can be calculated from the across variable, Vin However, this is a post-simulation process (i.e., these currents are not calculated unless explicitly requested).

Because Equation 10 is linear, Vin can be determined directly. In general, however, the simulator solves multiple nonlinear equations by making iterative guesses at values for the across variables.

The following figure summarizes the relationships among voltages, currents, and connection points for electrical templates.


Pin specification
Connection point declaration
Electrical
v — across
i — through
Electrical
p, m
IMPORTANT!
ACCESSIBLE within template
v(p), v(m)
System variables
NOT ACCESSIBLE within template
current at pins p, m
i (p->m)
Simulator uses system variables to solve KCL at p and m

Relating across and through variables

4

Using System Variables Between Models

The topics in this chapter explain how a template can use the value of a system variable from another template without an explicit pin connection between them by using a ref connection point:

- Modeling a Current-Controlled Voltage Source with MAST
- Modeling Mutual Inductance with MAST

Introduction to MAST ref Connection Points

When a template declares a through variable as a system variable, it is made available to other templates in the system. One way that these other templates can then use this variable is to declare a new kind of connection point called a ref variable. A ref variable connection point allows the template declaring it to “import” the through variable directly, without using pin connections.

Modeling a Current-Controlled Voltage Source with MAST

Declaring a pin implicitly declares a through variable and an across variable, according to the type of the pin (for example, an electrical type of pin implicitly declares current and voltage). The MAST language also lets you declare other types of connection points in a template besides pins. This section introduces one such type of connection point by using a current-controlled voltage source (CCVS), also called a current-to-voltage transducer (CVT).

The template describing a CCVS is shown as follows:

```
template cvt ci p m = k
  ref i ci
  electrical p, m
  number k
{
  var i i
  equations  {
    i(p->m) += i
    i: v(p)-v(m) = k*ci
  }
}
ASCII text of this example is located in:
install_home/example/MASTtemplates/structured/cvt.sin
```

The description of the cvt template is divided into the following topics:

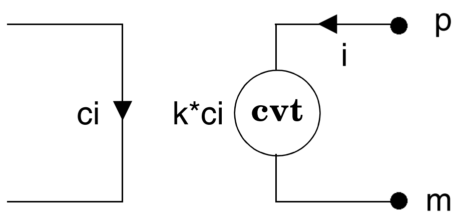
- Characteristic Equation for a Current-Controlled Voltage Source
- Equations Section and Local Declarations
- Header and Header Declarations -- describes how to dealer a ref variable, which refers to a var variable defined elsewhere
- Using a CCVS Template

Characteristic Equation for a Current-Controlled Voltage Source

The characteristic equation for a CCVS is similar to that of a constant voltage source, except that the output voltage is determined differently.

v = k*ci
choose i such that KVL is satisfied.

In this equation, ci is the controlling current, k is the transimpedance characterizing an instance of the cvt template, v and i are the output voltage and current, respectively.



Current-controlled voltage source

Equations Section and Local Declarations

The equations section differs only slightly from that of the template for the constant voltage source, vsource.

```
equations  {
  i(p->m) += i
  i: v(p)-v(m) = k*ci
}
```

Similarly, you must make i a system variable by adding the following statement in the local declarations section:

var i i

This statement declares i to be a var variable with unit i. (The first i is the unit name, and the second is the name of the var variable.)

Header and Header Declarations

Because the controlling current (ci) comes from elsewhere in the system to which the cvt template is connected, the simulator must determine it. That is, ci is a system variable that is declared in some other template in the circuit, either as a branch current or as a var variable. See the MAST Reference Manual for more information on var variables and ref variables. The cvt template brings in the value of this current to its ref variable connection point, which is specified as the node name in a netlist. However, unlike node names for pins, this type of node name consists of two parts:

- The name of the branch current or var variable from the external template
- The instance name of that external template

The ref variable declaration declares ci as a system variable with unit i (current, a through variable). Its purpose is to declare current as a system variable that this template refers to. Both refs and vars require declaration in any template that uses them. The difference is that a var variable is defined in the same template in which it is declared. That is, the equations that the simulator uses to determine the value of a var variable must reside in the template containing the declaration of the var variable. On the other hand, if a variable is declared as a ref variable in a template, then its defining equation(s) must reside in another template (where it is declared as a var variable and probably has a different name).

The name of this ref variable connection point (ci) is included in the header just like the names of pin connection points:

```
template cvt ci p m = k
```

This reinforces the fact that ci, like the across variable at pins p and m, provides a connection for a system variable. As with pins p and m, the ref variable declaration appears with the header declarations.

Thus, the three connection points (ci, p, m) for cvt are declared as:

```
ref i ci
electrical p, m
```

The declaration of the template argument (k) is similar to previous examples:

```
number k
```

The argument k models the transimpedance of the source, so that when its value is multiplied by the controlling input current (ci), the product is an output voltage (i.e., A•W = V).

Using a CCVS Template

You could use the cvt template in a netlist as follows:

```
cvt.1 i(v.1) a b = 1k
v.1 c d = 5
```

The v template in the Standard MAST Template Library declares its branch current i as a var variable. Therefore, i(v.1) is a var variable that can be passed to the cvt template, where it is given as a value the ref variable connection point, ci. Note that i(v.1) serves as the node name to which ci is connected, just like a and b are the node names to which pins p and m are connected. As stated earlier, this node name, i(v.1), consists of two parts:

- i, the name of the var variable in the external template
- v.1, the instance name of that external template

Also note that, in this example, you cannot use vsource, the constant voltage source template. You must use the v template instead, because the current, i, is not available outside of the vsource template. The next chapter shows how to make this current available to other templates in the system (by declaring vsource as an element template).

Modeling Mutual Inductance with MAST

This example shows how a template can modify the equation associated with a branch variable or var variable from another template in the same design. The template describing mutual inductance (mutind) is as follows:

```
template mutind i1 i2 = m
  ref i i1, i2
  number m
{
  equations  {
    i1 -= d_by_dt(m*i2)
    i2 -= d_by_dt(m*i1)
  }
}
```

ASCII text of this example is located in:
install_home/example/MASTtemplates/structured/mutind.sin

This template can be used in a netlist to provide electromagnetic coupling between two inductors as shown in the following example:

```
inductor.l1 p:p1 m:m1 = ind=1
inductor.l2 p:p2 m:m2 = ind=2
mutind.1 i1:il(inductor.l1) i2:il(inductor.l2) = \
  m=0.98*sqrt(ind(inductor.l1)*ind(inductor.l2))
```

where inductor.l1 and inductor.l2 are two instances of the inductor template, with inductance values of 1H and 2H, respectively. Notice that the mutind template is not electrically connected to this circuit.

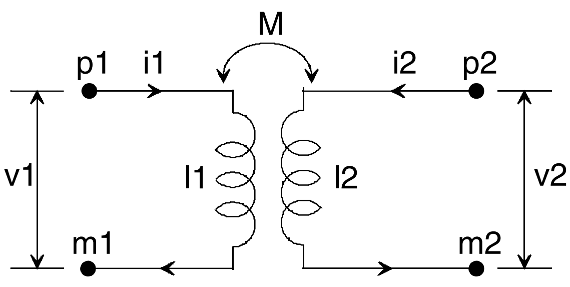
The description of the mutind template is divided into the following topics:

- Characteristic Equations for Modeling Mutual Inductance
- Setting up the Equations Section -- shows that a template with a ref variable connection can modify the equation associated with the current to which it refers (usually in another template)—this current must be a branch current or a var variable.

- Header and Header Declarations
- Using the Mutual Inductance (mutind) MAST Template -- shows that an element template is required when providing current to a ref variable of another template.

Characteristic Equations for Modeling Mutual Inductance

Mutual inductance does not exist on its own. Instead, it describes how two inductors are coupled by a magnetic field, which transfers energy from one inductor to the other. The figure below shows two mutually coupled inductors and the variables that pertain to this example.



Coupled inductors

The two equations below describe this system of two coupled inductors:

$V_1 = d(L_1 I_1)/dt + d(MI_2)/dt$

$V_2 = d(L_2 I_2)/dt + d(MI_1)/dt$

The first term in each equation is the self-inductance term—it is equivalent to the characteristic equation of a single, simple inductor. The second term in each equation is the mutual inductance term—the contribution of the mutual inductance template of this example.

Mutual inductance, represented here as m, is given by the expression:

$m = k \bullet \sqrt{L_1 \cdot L_2}$

where L_1 and L_2 are the inductance values for the respective inductors and k is their coupling factor (where -1 £ k £ 1).

Setting up the Equations Section

The template equations for mutind use the equation in the inductor template. There, the associated equation was essentially the first term shown in Equations 1 and 2 as follows:

$V_1 = d(L_1I_1)/dt + d(MI_2)/dt$

$V_2 = d(L_2I_2)/dt + d(MI_1)/dt$

The mutind template modifies the equation in the inductor template that is associated with the var variable, i, of the inductor. That is, when used in a netlist, the mutual inductance template “searches out” the equations of the two inductors and couples them as specified in the mutind model.

The following MAST construct in the equations section modifies an equation associated with an explicitly-declared system variable (i.e., a var variable):

```
system_variable    operator    expression
```

where:

system_variable	is the name of a branch variable, var variable, or a ref variable (which is a var variable from another template)
operator	is either += or -=, indicating that expression is added to or subtracted from the left side of the equation defining system_variable
expression	is a MAST expression formed from variables of different types; mathematical functions; the algebraic operators +, -, *, /, and **; parentheses; and the special operators d_by_dt and delay

Using this construct, the characteristic equations of mutual inductance yield the following equations section for mutind:

```
equations  {
  i1 -= d_by_dt(m*i2)
  i2 -= d_by_dt(m*i1)
}
```

It is important to understand that d_by_dt(m * i2) is not subtracted from i1, but rather from the left side of the equation that defines i1 (and, because i1 is a ref variable in this template, its defining equation is in the inductor template).

This is illustrated using the following example netlist:

```
inductor.l1 p:p1 m:m1 = ind=1
inductor.l2 p:p2 m:m2 = ind=2
mutind.1 i1:il(inductor.l1) i2:il(inductor.l2) = \
  m=0.98*sqrt(ind(inductor.l1)*ind(inductor.l2))
```

Because the inductor template declares il as a var variable, its associated equation is:

```
il: v(p)-v(m) = d_by_dt(ind*i)
```

This corresponds to the first term of either Equation 1 or Equation 2. Considering inductor.l1, the current from inductor.l2 is passed as ref variable i2 into the mutind template, which, in turn, modifies the equation in inductor.l1, using its var variable, il, and inductance, ind. The effective equation in inductor.l1 would be:

```
il: v(p)-v(m)-d_by_dt(m*i2) = d_by_dt(ind*i)
```

which, after algebraic rearrangement, corresponds to Equation 1. A similar modification is applied to the equation associated with the var variable il and inductance, ind in inductor.l2 to yield Equation 2. Thus, the mutind template, together with two instances of the inductor template, provide the inductance coupling shown in the “Coupled inductors” figure, shown previously, under the topic titled "Characteristic Equations for Modeling Mutual Inductance".

Header and Header Declarations

The mutind template has mutual inductance, m, as its only argument. The connection points of this template are the currents from the two coupled inductors, which you must declare as refs. Template header and header declarations, therefore, are as follows:

```
element template mutind i1 i2 = m
ref i i1, i2
number m
```

Note that the word element has been inserted as the first word of the header, as was done for the inductor template. This was done for mutind to make its reference current available for the transformer template (xformer).

Using the Mutual Inductance (mutind) MAST Template

Using the mutind template in a netlist is demonstrated using the following example netlist:

```
inductor.l1 p1 m1 = ind=1
inductor.l2 p2 m2 = ind=2
mutind.1 i(inductor.l1) i(inductor.l2) = \
  m=0.98*sqrt(ind(inductor.l1)*ind(inductor.l2))
```

Note, however, that in order to use the inductor template with the mutind template, the branch current had to be made available outside of the inductor template. This was done by declaring inductor to be an element template, which is explained in more detail in the topic titled "Using MAST Element Templates" in a subsequent chapter.

5

Modeling Hierarchical Systems

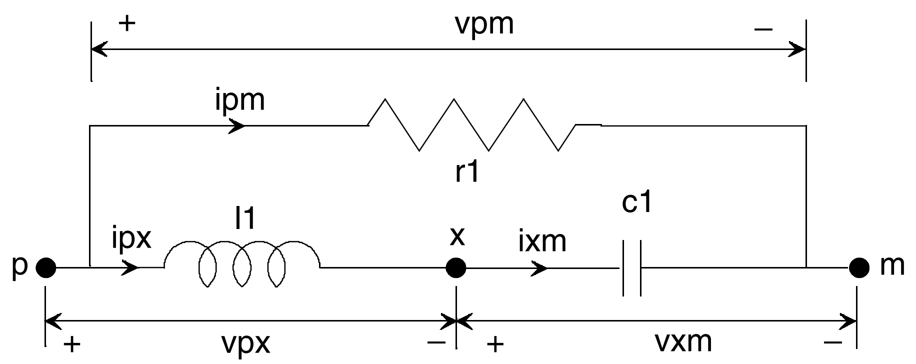
The following topics describe four ways to use the MAST language to model hierarchical systems:

- Preserving Hierarchy - MAST Template with Netlist -- refers to the previously-created resistor, capacitor, and inductor templates
- Flattened Hierarchy - MAST Template with Equations -- describes the whole system by its equations, which results in a flat description of the system
- Mixed Hierarchy - MAST Template with Netlist and Equations -- uses a combination of the first two methods
- Modular vs. Non-Modular MAST System Descriptions-- provides a summary of the characteristics of the hierarchical system and the flat system

Introduction to Modeling Hierarchical Systems

The topic titled "MAST Modeling Modularity and Hierarchy" introduces the concept of system hierarchy. The topic "The Saber Netlist Overview" in the earlier chapter, “Fundamental Modeling Concepts”, Example 2, shows a simple resistor network (the two_r template), which illustrates the concept of hierarchy.

The topics in this chapter expands this idea of using the MAST language to describe hierarchical systems using a simple RLC circuit as an example as shown in the following figure. This circuit has two external connection points (p and m), and one internal node (x).



The RLC network

The rlc1, rlc2, and rlc3 examples also introduce the following new concepts:

- Modular system description
- Element templates
- Internal nodes
- Default values for arguments
- Modular description and hierarchy as different concepts. In particular, it is possible to have a modular description that is flat

Simulation Efficiency of Hierarchical Systems

Unlike other simulators that flatten a hierarchical system description, the Saber simulator exploits system hierarchy when it solves the system equations. Thus, the computational effort to simulate a hierarchical system grows sublinearly with increasing system size—with most other simulators, the growth rate is close to quadratic. Therefore, hierarchical system descriptions generally improve simulation efficiency.

This rule, however, is not absolute. If a subsystem described hierarchically is very small, the overhead required to handle the hierarchy may exceed the benefit of having hierarchy. It is best to model such systems without introducing a new level of hierarchy.

In the MAST language, you can do this by declaring the template to be an element template. The following convention is recommended:

> You should declare a template to be an element template if the number of its connection points is greater than the number of system variables (local pins, vars, additional branches) declared in the template.

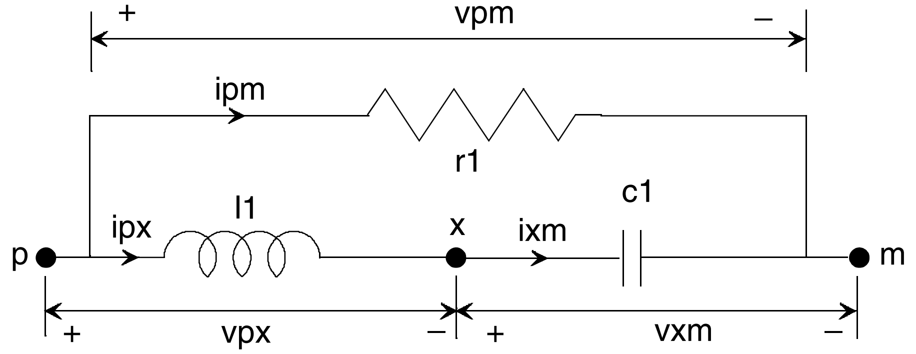
Preserving Hierarchy - MAST Template with Netlist

With resistor, inductor, and capacitor templates already defined, the simplest way to describe an RLC circuit is by creating a template that consists primarily of an internal netlist of other templates. Each reference in this netlist creates an instance of the corresponding template and gives values to its arguments. Thus, the netlist section is identical to a top-level system template that just refers to other templates (i.e., they are circuit netlists with no additional modeling information).

The following template describes the RLC network in terms of the resistor, capacitor, and inductor templates to preserve the hierarchy in the system:

```
template rlc1 p m = r,l,c
  number r = 10k,   # resistance arg. w/default
         l = 1m,    # inductance arg. w/default
         c = 1u     # cap. arg. w/default

{
  resistor.r1  p m = r
  inductor.l1  p x = l
  capacitor.c1 x m = c
}
ASCII text of this example is located in:
install_home/example/MASTtemplates/structured/rlc1.sin
```

The rlc1 template uses a netlist for the circuit shown in the following figure:



The RLC network

rlc1 Template Topics

The description of the rlc1 template is divided into the following topics:

- Template Header -- shows implicit declarations of connection points and the use of an internal node, thereby implicitly declaring its type
- Header Declarations -- shows the assignment of default values to template arguments
- Netlist Section

Template Header

This template named rlc1 provides two connection points to an external circuit: p and m. The template has three arguments: r, l, and c, which represent the values of the internal elements. When rlc1 is used in a circuit netlist, any values that a user would specify for these arguments are passed to the arguments of the internal templates (see the topic titled "Netlist Section").

Therefore, the template header becomes:

```
template rlc1 p m = r, l, c
```

Note that, on the right-hand side of the equals sign (=), the multiple arguments are separated by commas.

The header declaration typically includes the declaration of the pins and the arguments. However, in this case it is sufficient to declare only arguments, because the simulator can determine from the usage of p and m that their pin type is electrical. This is because they are nodes of r1, l1, and c1—the simulator finds this information in the resistor, inductor, and capacitor templates.

Thus, the following rule applies to references to connection points in templates:

> If, in a template, the first reference to a connection point is as a node in an internal netlist, then the simulator implicitly declares the type of that connection point. It will be of the same type as found in the netlist template.

The same is true of an internal node, such as x in this template. It needs to be declared only if the simulator cannot determine its type from its usage in a netlist entry. In this example, x is implicitly declared to be electrical by its use as a node between an inductor and a capacitor. In general, it is good practice to declare all nodes even if declaration is not required.

Header Declarations

Declaring the arguments for this template introduces the idea of providing initializers as default values. An initializer is a value assigned to an argument in the header declarations. It becomes the value of that argument if none is explicitly specified in a netlist entry.

For this template, all arguments are of type number, so their declarations (including initializers) are as follows:

```
number r = 10k,
    l = 1m,
    c = 1u
```

The commas at the end of the first two lines are used to separate the arguments and to provide line continuation. If a line ends with a comma, that indicates that the line is incomplete, and the next line is automatically interpreted as a continued line. (For a complete list of the line continuation characters, see the topic titled "MAST Line Continuation" in the MAST Reference Manual.) This declaration could be written on a single line, as follows (note the commas):

```
number r=10k, l=1m, c=1u
```

Netlist Section

The netlist section of a MAST template consists of one or more netlist entries, each describing an instance of an internal template. Internal templates are sometimes referred to as components. However, this term is not used here because we also provide a Component Library.

The general form of a netlist entry is:

```
templatename.refdes node_list [= argument_list]
```

where:

templatename	is the name of a template describing the model to which this netlist entry refers.
refdes	is the reference designator of this template instance—thus, templatename.refdes is the full name of the template instance (note the period as separator). This full name should be unique in the netlist, although the same reference designator may be used for different templates.
node_list	lists the names of the nodes to which the connection points of the template instance are connected. This list is separated by spaces. Note that the names of these nodes correspond to the connection point names of the top-level template (e.g., the rlc1 template).
=	separates node_list from argument_list.
argument_list	lists the values characterizing this template instance. The simulator assigns these values to the arguments of templatename. Depending upon the template, the argument_list and the equals sign (=) may be optional.

The rlc1 template should be designed such that it can be used in many places. Therefore, it does not assign numerical values to the arguments of “internal” templates in its netlist section. Instead, the argument names of the rlc1 template (r, l, c) are assigned to the argument_list of each internal template:

```
resistor.r1 p m = r
inductor.l1 p x = l
capacitor.c1 x m = c
```

Thus, any value assigned to an rlc1 argument in a netlist will be passed to the corresponding argument in one of its internal templates. In other words, to override the default values assigned to these internal templates, you specify an instance of rlc1 in a netlist and provide appropriate argument values there.

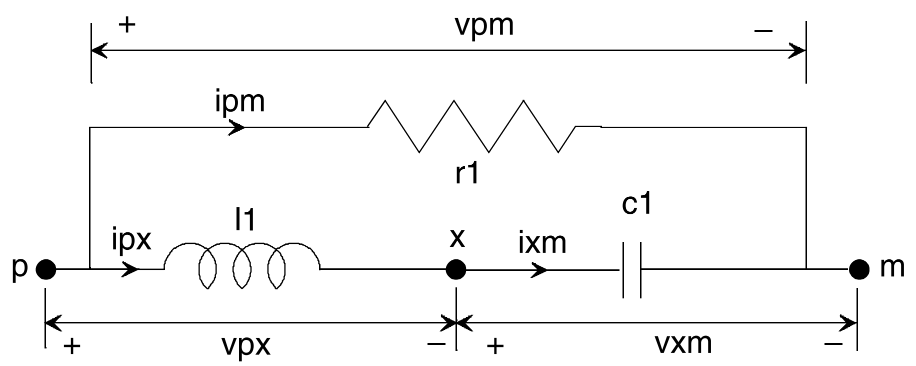
Flattened Hierarchy - MAST Template with Equations

Another method for defining an RLC template is to include all the equations for solving the network without referring to any other templates. This approach requires the model descriptions from the resistor, inductor, and capacitor templates to be defined in one template. In addition, their relationships to one another and their relationships to the pins must also be included. The template that does this (rlc2) is shown as follows:

```
template rlc2 p m = r,l,c
  electrical p, m
  number r = 10k,   # resistance arg. w/default
         l = 1m,    # inductance arg. w/default
         c = 1u     # capacitance arg. w/default
{
  electrical x      # internal node x
  var i il
  equations  {      # current through r, l, c
    i(p->m) += (v(p)-v(m))/r
    i(p->x) += il
    i(x->m) += d_by_dt((v(x)-v(m))*c)

   # tell simulator how to find il
    il: v(p)-v(x) = d_by_dt(il*l)
  }
}
```

ASCII text of this example is located in:
install_home/example/MASTtemplates/structured/rlc2.sin

This template contains a flat description of the RLC circuit shown in the following figure:



The RLC network

rlc2 Template Topics

The description of the rlc2 template is divided into the following topics:

- Header and Header Declarations
- Equations Section -- shows multiple equations that can appear in any order
- Template Body with Local Declarations -- shows the declaration of a local node

Header and Header Declarations

The template header is declared the same as rlc1, but with the name rlc2:

```
template rlc2 p m = r, l, c
```

Giving the same default values to the header arguments as in rlc1, the header declarations are:

```
  electrical p, m
  number r = 10k, l = 1m, c = 1u
```

Here you must explicitly declare the connection points p and m, because they are used only in the equations section, so there is no way for the simulator to determine their type (i.e., they do not appear elsewhere in a netlist entry).

Equations Section

The template equations for the rlc2 template consist of the equations from the resistor, inductor, and capacitor templates described in previous chapters, using the variable names from the rlc2 template. Note that these statements can appear in any order.

```
ipm = vpm/r
vpx = d_by_dt(ipx*l)
ixm = d_by_dt(vxm*c)
```

Template Body with Local Declarations

As with the connection points p and m, you must explicitly declare internal node x because there is no way for the simulator to determine its type (i.e., it does not appear elsewhere in a netlist entry). Node x it is a local internal node. It must be declared following the opening brace before it can be used in the equations section:

```
{
electrical x
var i il
```

The variable il is declared to handle the inductor current.

Mixed Hierarchy - MAST Template with Netlist and Equations

Another approach for modeling an RLC network is to create a template that contains equations for some of the elements and netlist entries for other elements—a mixture of the methods used in the rlc1 and rlc2 templates. For instance, you can define the capacitor and the resistor by their equations, but reference the inductor template in a netlist entry.

```
template rlc3 p m = r,l,c
  electrical p,m
  number r = 10k,     # resistance arg. w/default
         l = 1m,     # inductance arg. w/default
         c = 1u      # capacitance arg. w/default
{
  inductor.l1 p x = l    # use inductor template


  equations {
    i(p->m) += (v(p)-v(m))/r
    i(x->m) += d_by_dt((v(x)-v(m))*c)
  }
}
```

ASCII text of this example is located in:
install_home/example/MASTtemplates/structured/rlc3.sin

The equations for the inductor branch are described in the inductor template. Note that you could have omitted the declaration of connection point p; its type (and the type of x) can be determined from the inductor template in the netlist entry. Because the inductor template is included in an internal netlist, rlc3 is hierarchical. It could be made flat by making inductor an element template.

Element template is described in the following topics:

- Introduction to MAST Element Templates
- Properties of MAST Element Templates
- Using MAST Element Templates

Introduction to MAST Element Templates

It is preferable to model small subsystems so that they do not introduce a new level of hierarchy into the system description.

Note:

> A subsystem is "small" when the number of connection points is greater than the number of its locally-declared system variables (local nodes, branch declarations, and var variable declarations).

This definition is satisfied by most templates containing equations for a single element (such as a resistor or a capacitor) and some templates containing netlists (such as rlc2). In particular, all templates given as examples in previous chapters are "small."

The MAST language contains a construct for specifying that a template should not introduce a new level of hierarchy. This construct declares it to be an element template, rather than a template, in the template header. For example, the inductor template would have the following header:

```
element template inductor p m = ind
```

Properties of MAST Element Templates

An element template flattens the hierarchy locally. This has three main consequences when referencing an element template in an internal netlist:

- An element template does not introduce a new level of hierarchy.
- The simulator merges the equations of the internal element template with that of the calling (containing) template, for the purpose of solving the simultaneous differential equations.
- Any vars and branch through variables declared in the element template become available in the calling template.

The third consequence is the key to using the voltage source template (v) with the current-controlled voltage source, and to using the inductor template with the mutual inductance template. Because v and I are declared as element templates, their currents become available to a calling template; they can be passed as refs to the current-controlled voltage source template (cvt) and mutual inductance template (mutind), respectively.

Using MAST Element Templates

You can use element templates in the same way you use regular templates. Therefore, if the resistor, capacitor, and inductor templates had been declared as element templates, then the rlc1 and rlc3 templates, which contain netlist references to those templates, would describe flat subsystems. In other words, rlc1 and rlc3 would not be hierarchical to the simulator, even though they reference other templates internally.

This demonstrates that you cannot determine, by looking at a netlist, whether it represents a flat or hierarchical system—you must look at each template referenced in the netlist.

Modular vs. Non-Modular MAST System Descriptions

You can decide whether to model a system or subsystem hierarchically, based on the number of local pins and additional branches. Further, you also can choose between a modular description (like rlc1), a completely local description (like rlc2), or a combination of the two (like rlc3). A comparison of the characteristics of each are described in the following topics:

- Modular (hierarchical) MAST System Descriptions - Summary
- Non-Modular (flat) MAST System Descriptions - Summary

Modular (hierarchical) MAST System Descriptions - Summary

The characteristics of modular system descriptions are:

- Modular descriptions are easier to develop and maintain
- Modular descriptions provide a library of building-block templates that can be reused in other designs
- Modular descriptions can be hierarchical or flat (by using element templates)
- Improvements in one of the templates referenced in a modular description become available immediately
- Changes in one of the templates in a modular description may require updating the system description

Non-Modular (flat) MAST System Descriptions - Summary

The characteristics of non-modular system descriptions are:

- Non-modular descriptions are always flat
- In a non-modular description, it is simpler to extract a variable (such as the inductor current) in the system using Saber's extract command
- Modeling a system in a non-modular way may duplicate some efforts spent to model another system (e.g., repeating the entire model description for a resistor instead of calling the resistor template).

6

Variables and Arguments

The following topics describe ways of using internal variables and writing arguments to provide more template versatility. They also demonstrate how to check the validity of argument values, use message functions, and improve template performance by precalculating expressions:

- Modeling Extractable Capacitor Voltage and Charge with MAST -- shows how to obtain charge in a capacitor, which is not available in a template as a system variable; it also shows how to write arguments that provide a nonzero voltage across the capacitor at DC (initial condition).
- Modeling Multiple-Mode Voltage Source with MAST -- shows how to use conditional statements (if-else) to describe a voltage source that can be used as both a constant supply and a variable-input stimulus.
- Modeling a Linear Transformer with MAST -- shows how to create a transformer by using the inductor template and the mutind template.
- Modeling a Temperature-Dependent Resistor with MAST -- shows how to check for a value of zero for the resistance argument and issue an error message when it happens.
- Modeling an Idealized Op Amp with MAST -- demonstrates a three-pin operational amplifier.

Introduction to MAST Variables and Arguments

The Saber simulator solves (i.e., finds values of the system variables for) the simultaneous nonlinear differential equations that are given by the system description. In electrical systems, system variables are typically node voltages, except for models such as inductors and voltage sources, in which case current is the system variable.

During simulation, certain information about the system (such as system variables) is automatically extracted from the raw simulation results and made available for display. The amount of such information is typically controlled by the siglist variable of the simulation command.

After simulation, you can extract this information from raw simulation results using the extract command (also known as extraction).

The example templates, isource through rlc3, in the chapter “Basic Modeling”, topic titled "MAST Modeling Examples - Electrical Elements", have the following two characteristics in common:

- Argument values are assumed to be valid
- Arguments are used directly in the template equation(s)

In general, a template should be prepared to handle invalid argument values. For example, the resistor template should check for res=0 to prevent division by 0. In addition, if template arguments are not directly usable by the model, the template has to convert them to appropriate local parameters. The topics in this chapter explain how to use local variables and conditional statements to accomplish these tasks. These constructs are incorporated into templates such as resistor and opamp.

Modeling Extractable Capacitor Voltage and Charge with MAST

The capacitor template directly expresses the characteristic equation of the capacitor. This topic describes how to modify the capacitor template (now called capacitor_1, listed below) such that the voltage across the capacitor and its charge become available for extraction.

```
element template capacitor_1 p m = cap, ic
  electrical p, m
  number cap, ic=undef
{
  val q qc
  val v vc
  values  {
    vc = v(p) - v(m) # voltage across cap.
    qc = vc * cap    # charge stored in cap.
  }
  control_section{
    initial_condition(vc,ic)
  }
  equations  {
    i(p->m) += d_by_dt(qc)# current through cap.
  }
}
ASCII text of this example is located in:
install_home/example/MASTtemplates/structured/capacitor_1.sin
```

The description of the capacitor_1 template is divided into the following topics:

- Header and Header Declarations -- shows how to write arguments that provide a nonzero voltage across the capacitor at DC (initial condition).
- Values Section -- describes how to define variables that can be extracted
- Equations Section -- describes how the equations section works with the values section
- Local Declarations
- Control Section -- shows how to insert a statement that allows an initial conditions argument

Header and Header Declarations

The template header declares capacitor_1 to be an element template. This means that the capacitor_1 template does not introduce a new level of hierarchy into the system description.

It also contains a new argument, ic, that allows you to specify an initial voltage across the capacitor when a DC analysis is performed (instead of being set to 0).

```
element template capacitor_1 p m = cap, ic
  electrical p, m
  number cap, ic=undef
```

The default value for ic has the initializing value undef, a special numeric constant representing an undefined value in the MAST language. You can assign it as a value to a template parameter (such as ic=undef). However, you should not use it as a value in an expression (such as qc=undef/6), although you can use it in a conditional statement to test equivalence (such as if (ic==undef)).

This is because the results of evaluating the expression are unpredictable if the simulator has to operate on undefined values (i.e., any parameter set to undef should be given a numerical value when the template is used). However, in this case, ic=undef means that ic is ignored.

Values Section

While the MAST language can describe the characteristic equations of most physical systems in a template, you can use the values section for the following:

- Define a variable (called a val) specifically for use with the extract command or the siglist variable. A val variable is usually used as an intermediate variable; you can assign expressions to it. Once a val variable is declared and defined, it is available for extraction—even if it does not appear in a template equation.
- Simplify other template sections. You can use vals not only in the values section, but also in the equations section, the control section, and when statements.

- Call foreign routines that return more than one value
- Define the values of noise sources

The values section consists of assignment statements and conditional statements. The syntax of a assignment statement is as follows:

variable = expression

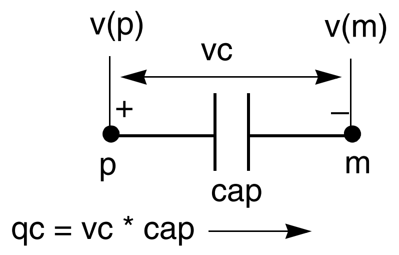
where:

variable	is a val variable
expression	is a MAST expression defining variable—the value of expression is assigned to variable (this is not a mathematical equality). An expression can include variables of different types (except for through variables); mathematical functions; the +, -, *, /, **, and parentheses (()).

The values section is both procedural and declarative. Because it is procedural, it must be able to execute its statements in sequence, so you must order the statements such that each variable is defined before it is used. Because it is declarative, each of its statements might be executed only if the simulator requires variable, for example if a variable is a val variable that is to be extracted or is needed for solution of the system variables.

If variable is not needed, the simulator skips the statement during evaluation of the values section, although it might "absorb", during initial setup, certain information provided by a statement. Therefore, including a values section in a template does not impose any performance penalties if no variables are extracted. This means that a values section should be part of almost every structured template.

One objective stated earlier was to make the voltage across the capacitor and the charge of the capacitor available for post-processing. The following figure shows the relationship of these quantities



Linear capacitor

The following values section expresses the relationship of the voltage across the capacitor and the charge of the capacitor:

```
values  {
  vc = v(p) - v(m) # voltage across cap.
  qc = vc * cap    # charge stored in cap.
}
```

The quantity vc is defined as the difference of the across variables at pins p and m; qc is the product of vc and the capacitance value. As required, the definition of vc precedes the definition of qc.

Equations Section

The following equations section for the capacitor_1 template is very similar to the capacitor template, except that now the capacitor charge is available as a val variable from the values section:

```
equations  {
  i(p->m) += d_by_dt(qc)# current through cap.
}
```

The Saber simulator uses the equations and values sections jointly to set up the system equations. It uses the values section only to the extent necessary to determine the through variable contribution at the resistor pins as a function of the across variables at the pins. It does not, however, evaluate the values section, except when simulator users specify that certain information is to be extracted, and even then it evaluates only the equations needed to provide the requested information.

This does not mean that the values section is evaluated for solution of the system variables. Rather, the simulator "absorbs" certain information when it sets up the simultaneous equation describing the system. Consequently, the equation that the simulator solves for with the capacitor_1 template is identical to the one for the capacitor template.

Local Declarations

It is required that all variables defined in the values section must be declared among the template's local declarations. Both vc and qc are vals. These two vals, the voltage vc and the charge qc, are declared as follows:

```
  val q qc
  val v vc
```

The declaration of a val variable is similar to that of a var variable. The statement:

```
val unit name [,name, name, ...]
```

This declares one or more variables to be vals having the specified unit.

You can assign a value to a val variable only in a values section of a template or an assignment statement in the local declarations section of an unstructured template. However, you can use a val variable in the template equation, the control section, and in when statements.

There is a collection of predefined units in the units.sin file, which is provided with the Saber simulator. The simulator, by default, includes the units.sin file with each system description.

Control Section

The control section of a template provides the simulator with information that is specific to the system being analyzed but is not directly part of the component model. An example of such information is the initial voltage across the capacitor. This is implemented by using the initial_condition statement within the control section.

The control section consists of the keyword control_section, followed by a sequence of control section statements, enclosed between braces, { }.

```
control_section {
  initial_condition(vc,ic)
}
```

Such statements are special in the sense that they can occur only in an explicitly-declared control section. A complete list of these statements is given in the MAST Reference Manual. Here, only an initial_condition statement for the voltage across the capacitor (v) is used.

When you specify a value for the argument ic, that value is used by the branch voltage, vc, when a DC analysis is performed (using either the dc or dctr command).

Initial Conditions

For some templates, you may want to specify the initial condition of a system variable or difference of two system variables. The initial_condition statement allows you to assign a value or values to such system variables (for instance, by means of a template argument). The general form for this statement in the control section is as follows:

initial_condition (variable, value)

where variable is a system variable or difference of system variables (such as across branch variables), and value is the name of the initial condition parameter (i.e., variable=value at time=0).

If there are multiple system variables to receive initial conditions, each one must have a separate initial_condition statement in the control section. In this template, value is named ic. It is provided as an argument to the template and assigned the value of undef. However, other implementations are possible, and it is up to the writer of the template to select from them (including assigning names).

Modeling Multiple-Mode Voltage Source with MAST

The vsource template models a source that provides a constant voltage output, independent of time and frequency. Such voltage sources are common, but many systems have voltage sources with waveforms that are time- or frequency-dependent. The purpose of this example is to show how to use conditional statements (if-else) to describe a voltage source (vsource_1, listed below) that can be used as both a constant supply and a variable-input stimulus.

This voltage source can model the following:

- A constant voltage, to be used as a supply.
- An exponential voltage, as a simple example of a time-dependent waveform. This overrides the constant voltage if both are specified.
- A frequency-dependent AC signal, to be used for frequency-domain (small-signal) analyses.

The following topics describe how to activate each of these outputs, depending on the analysis being run. An extended version of this voltage source (which provides several additional waveforms using these same techniques) is described in the vsource_2 template.

```
element template vsource_1 p m = supply, tran, ac
  electrical p, m      # header declarations
  number supply=0

  struc {              # start of tran structure
  number v1=0,         # initial voltage
         v2=0,         # voltage at time=inf
        tau=0.000001   # time constant
  } tran=()            # end of tran structure

  struc {              # start of ac structure
    number   mag=0,    # AC magnitude
           phase=0     # AC phase
  } ac=()              # end of ac structure
```

```
{                  # start template body
 var i is              # local declarations
 val v vs
 values {
  if (dc_domain | time_domain) { # If large signal
   if ((tran->v1~=0 | tran->v2~=0) & tran->tau>0){
                               # If waveform is defined
    vs=tran->v1 + (tran->v2-tran->v1) *(1-exp(-time/tran-
>tau))
   }                    #   source voltage = waveform
                        #   at current time
   else {               # otherwise
    vs=supply           #   source voltage = supply
   }
  }
  else if (freq_mag) {   # or if source is ac magnitude
   vs=ac->mag            #   source voltage = magnitude
  }
  else if (freq_phase) { # or if source is ac phase
   vs=ac->phase          #   source voltage = phase
  }
 }# end of values section

 equations {           # start of equations section
  i(p->m) += is        # add current contributed by source
  is: v(p)-v(m)=vs     # determine current contributed
                       #  by source
 }                     # end of equations section
}                      # end of template body
```

ASCII text of this example is located in:
`install_home/example/MASTtemplates/structured/vsource1.sin`

The description of the vsource_1 template is divided into the following topics:

- Constant DC Supply Output -- shows the header and header declarations
- Time-Dependent, Exponential Output -- shows the characteristic equation and introduces the declaration of a structure parameter, which is further described in the following topics:
  - Initializing a Structure - Method 1 Using vsource_1
  - Initializing a Structure - Method 2 Using vsource_1
  - Summary of Structure Initializers
  - Template Body - shows the template body, which include the following concepts:

    Introduction to MAST Simulator Variables (simvars)

    Introduction to MAST Conditional Statements

    Using simvars and Conditional Statements

    Using a Structure Reference

    Completing the Values Section for tran

    Equations Section
- Frequency-Dependent, AC Output -- shows how to model the small-signal analysis behavior for the vsource_1 template using the following topics:

  Small-Signal Structure Type Parameter

  Small-Signal simvar Variables

  Small-Signal Conditional Statements

Constant DC Supply Output

Template Header and Header Declarations

Because this voltage source models three different output functions, the template header contains three arguments (supply, tran, ac), one for each function. It is also declares vsource_1 as an element template.

```
1 element template vsource_1 p m = supply, tran, ac
```

In the header declarations, connection points (p and m) are declared as electrical pins and supply is declared as a simple number parameter. To make vsource_1 operate as a constant DC supply, the argument supply replaces the argument vs (but is used the same way). Further, supply is assigned an initializer (0), which becomes the default value if no value is specified for it in a netlist:

```
3   number supply = 0
```

This default value of 0 causes the voltage source to act as a short circuit between p and m for a DC analysis.

Time-Dependent, Exponential Output

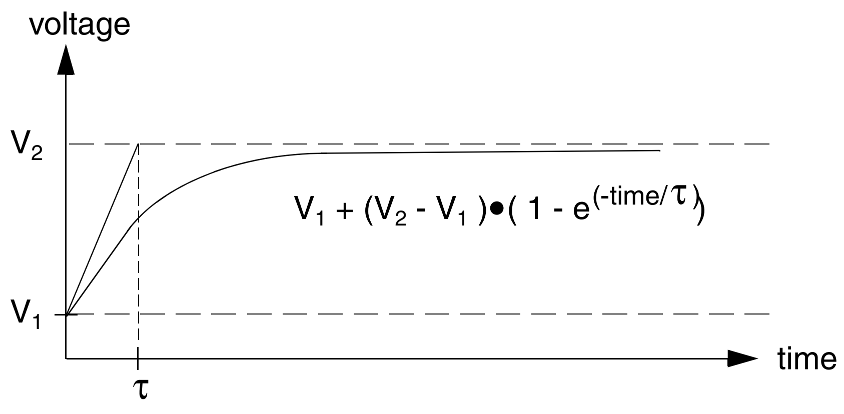
Characteristic Equation

The exponential waveform to be modeled by the voltage source template is shown in the following figure and is defined by the following equation:

$V_{out} = V_1 + (V_2 - V_1) \bullet (1 - e^{(-time/\tau)})$

where the following quantities are specifiable arguments to the template:

V1	the initial voltage (at time=0)
V2	the voltage at time = infinity
t	the time constant (t > 0) of the exponential function



Exponential waveform

Note that the three arguments (v1, v2, tau) that characterize the exponential waveform do not appear in the template header. They are declared in the template as part of the argument tran, which does appear in the template header. This is a commonly-used arrangement.

Header Declaration Using a Structure Type Parameter

To emphasize that they belong together, the three arguments used in the characteristic equation (v1, v2, tau) are declared so that the exponential waveform can be referred to by a single name (tran). In the MAST language, you do this by declaring this name as a structure parameter, which is an ordered list of member parameters (here, v1, v2, tau). Each structure member may be a number, an enum, another structure (which would then make a nested structure), or any of the other types that a parameter can be. See the MAST Reference Manual for more information.

In its most general form, the syntax for declaring a structure is (optional declarations are enclosed by italicized square brackets, [ ]):

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

where:

struc	is a required keyword that declares the structure.
structurename	is an optional name identifying the structure. For information about how to use this, see the MAST Reference Manual.
member	is the declaration of a structure member—a “subordinate” parameter whose declaration is identical to what it would be if it were outside of a structure. This declaration can include initializing values, as described in the topic titled "Initializing a Structure - Method 1 Using vsource_1".
id	is the name of a variable having the type described by the structure. There can be multiple ids, separated by commas, that declare duplicate versions of this particular structure with different names.
initial_value	is an optional initializer for the structure. It interacts with the initializers for the structure members, depending on the initialization method selected.

You can declare the exponential waveform using a structure declaration in various ways. Two possible methods are described in the following topics, both initializing the structure (whose id is tran) so that all of its members are 0—unless otherwise specified in a netlist entry:

- Initializing a Structure - Method 1 Using vsource_1
- Initializing a Structure - Method 2 Using vsource_1

Initializing a Structure - Method 1 Using vsource_1

Specify the initializer for each structure member separately from the structure initializer:

```
struc {
  number v1 = 0,
         v2 = 0,
        tau = 0.000001
} tran= ()
```

This declares a parameter named tran to be a structure consisting of number type parameters v1, v2, and tau. Each of these members is initialized to 0. The initializer for the structure is simply a pair of parentheses preceded by an equals (=) sign. This means that, in a netlist entry referring to vsource_1, if the tran structure is not defined, then its members take on the values specified by their initializers. If this initializer for the structure (tran=( )) were missing, the structure would have to be defined (given a value) in a netlist entry, even though the individual members have initializers.

Initializing a Structure - Method 2 Using vsource_1

Specify initializers for structure members within the structure initializer:

```
   struc {
     number v1, v2, tau
} tran = (0, 0, 0.000001)
```

Again, this declares tran to be a structure consisting of three members and initializes them to a value, so it is not necessary to specify tran in a netlist entry because it is initialized in its declaration.

In a netlist entry referring to the vsource_1 template, these two declarations of the tran structure are equivalent under the following conditions:

- The tran structure is not specified in a netlist entry. This initializes two of the structure members to 0.
- The tran structure is specified in a netlist entry, and values are specified for all three structure members. This initializes all three structure members to their specified values.

These two methods behave differently when the tran structure is only partially defined in a netlist entry. Suppose, for example, that a netlist entry specifies values for v1 and v2 but not for tau. Then v1 and v2 take on the specified values in both structure declarations. On the other hand, tau takes on the value 0 in the first case but is undefined in the second.

Summary of Structure Initializers

The following summarizes how structure initializers, member initializers and values defined in a netlist entry interact to define the value for the structure members:

- A structure receives initializing values either from a structure initializer or from values provided in a netlist entry. If both are present, the simulator ignores the structure initializer and uses the values from the netlist entry. If neither is provided, an error message is displayed.
- Each member of a structure takes on the value provided for it in the structure initializer or netlist entry (whichever applies, as described above). If no value is provided, the structure member is set to the value specified in the member initializer. If the member initializer is missing, its value is undefined.

A structure initializer has the following form:

```
id = (valuelist)
```

The member initializers in the vsource_1 template are v1=value, v2=value, and tau=value.

Considering these differences, the first method of declaring the exponential waveform is recommended, so that all of its members will always be defined. The example in the topic titled "Modeling Extractable Capacitor Voltage and Charge with MAST" shows how to check for undefined parameters or invalid values.

Template Body

To summarize, the three arguments of this template—supply, tran, ac—are provided to allow a user to select whether the output will be a constant voltage, an exponential voltage, or an AC signal.

It is important at this point to distinguish between large-signal and small-signal responses of a system:

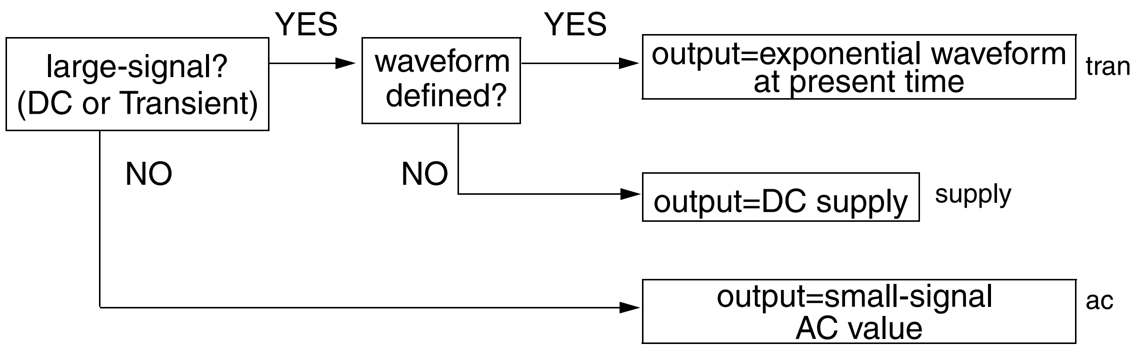
- A large-signal response, such as that obtained from a DC or transient analysis, incorporates all nonlinearities of the system. Except for the DC response (i.e., the operating point of the system), which is not related to time or frequency, large-signal responses are typically evaluated in the time domain.
- A small-signal response, such as that obtained from an AC analysis, assumes a linearization of the system about its operating point. It is a first-order approximation of the large-signal response at that operating point and is typically evaluated in the frequency domain. Modeling this functionality is further described in the topic titled "Frequency-Dependent, AC Output".

  Note:

  The vsource_2 template extends this voltage source to provide a large-signal sinusoidal waveform. This is not to be confused with the small-signal sinusoid provided for a small-signal, AC analysis.

Given these distinctions, the constant and exponential voltages are to be used for large-signal analyses (DC and transient analyses), while the AC voltage is to be used for small-signal analyses (such as an AC analysis). Therefore, the template must obtain, from the simulator, information about the simulation being used.

Once the template has this information, it has to define the output voltage according to the logic illustrated in the following figure:


large-signal? (DC or Transient)
YES
waveform defined?
YES
output=exponential waveform at present time
tran
NO
NO
output=DC supply
supply
output=small-signal AC value
ac

Selecting an output

In order to provide these multiple modes, the vsource_1 template must have the following capabilities:

- Information from the simulator regarding the kind of analysis being performed
- Based on that information, the ability to decide which argument is assigned to the output

These capabilities are provided by simulator variables and conditional statements, which are described in the following topics:

- Introduction to MAST Simulator Variables (simvars)
- Introduction to MAST Conditional Statements
- Using simvars and Conditional Statements
- Using a Structure Reference
- Completing the Values Section for tran

Introduction to MAST Simulator Variables (simvars) To pass analysis information from the simulator to a template, the MAST language supports what are called simvars (simulator variables). A simvar is a predefined MAST variable that does not need to be declared. Simvars provide information about the nature of the analysis being carried out, various stages of the analysis, the present simulation time or frequency, and the present mathematical environment (deterministic or statistical). They also let a template interact with the simulator's integration algorithm. With two exceptions simvars cannot be modified by a template.

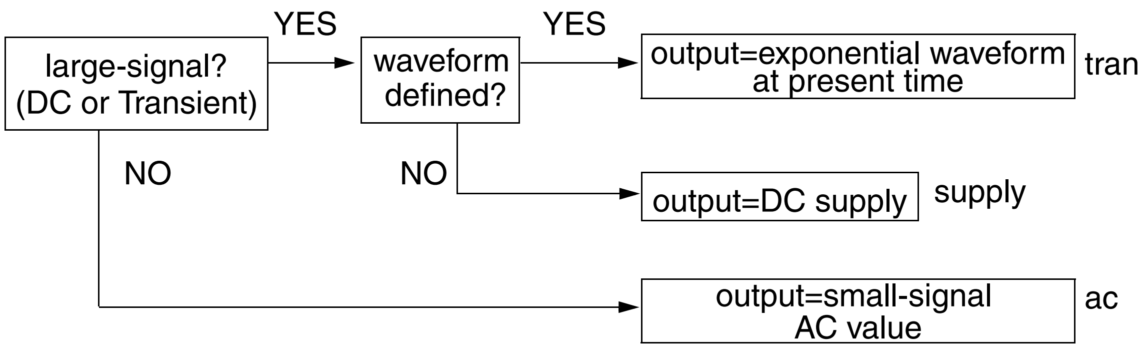
A complete list of simvars is given in the MAST Reference Manual. In vsource_1, only the simvars related to the nature of the analyses and simulation time are necessary.

Simvars used by tran in the vsource_1 template are:

dc_domain	has a non-zero value when a DC analysis or DC transfer analysis is being carried out. It is equal to 0 otherwise.
time_domain	has a non-zero value when a transient analysis is being carried out. It is equal to 0 otherwise.

time	contains the current simulation time when a time-domain analysis is being done. It is equal to 0 otherwise.

Introduction to MAST Conditional Statements The following logic diagram illustrates the decisions that the vsource_1 template makes in defining its output voltage based on simvar information. It makes these decisions using conditional statements.


large-signal? (DC or Transient)
YES
waveform defined?
YES
output=exponential waveform at present time
tran
NO
NO
output=DC supply
supply
output=small-signal AC value
ac

Selecting an output

In the MAST language, conditional statements are implemented as if-else constructs, as follows:

```
if (condition1) statement1
else if (condition2) statement2
...
else statementN
```

where:

if, else	are MAST keywords
condition	is a MAST expression that is false if its value is 0 or true if its value is not 0
statement	is either a simple MAST statement or a group of MAST statements enclosed in braces ({})

When processing an if, else if,..., else compound statement, the simulator evaluates the conditions in the order given. When it finds a true condition, then it executes the associated statement and ignores the rest of the compound statement. When all conditions are false, it executes the statement associated with the last else keyword, if there is one. If there is no else keyword, it ignores the entire compound statement.

In if statements, you can build conditions using the following building blocks:

- Arithmetic expressions, such as the expressions that can be on the right-hand side of an assignment statement, with the restriction that the expression must return a single value. An arithmetic expression is true if it has a non-zero value.
- Relational expressions, which consist of two expressions separated by one of the relational operators listed below. The value of such an expression is 1 if the relation is true. It is 0 otherwise.

==	equal to
~=	not equal to
>	greater than
>=	greater than or equal to
<	less than
<=	less than or equal to

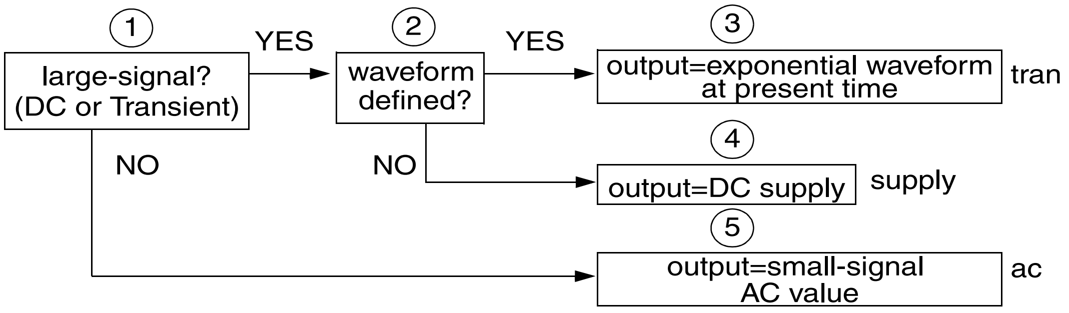
- The complement of an expression, that is, the ~ (tilde) operator followed by an expression. If expression is true, then ~expression equals 0. Otherwise, ~expression equals 1.
- A compound expression, that is, two expressions separated by a logical operator (& for “and”, | for “or”):
  - expression1 & expression2 has a value of 1 if both expression1 and expression2 are not equal to 0. Otherwise, expression1 & expression2 has a value of 0.
  - expression1 | expression2 has a value of 1 if either expression1 or expression2 (or both) is not equal to 0. Otherwise, expression1|expression2 has a value of 0.

It is important to understand that the condition in an if statement can include system variables and vals that depend upon system variables.

Using simvars and Conditional Statements Values Section

The following figure shows a partial implementation of the outline shown in the previous figure by using the information just presented on simvars and conditional statements. The following figure numbers each of the blocks and provides statements beneath them that describe how the conditional portions of the template associate decision Blocks 1 and 2 with output Blocks 3, 4, and 5. These statements use a val variable named vs that represents whichever output is selected for vsource_1.

Following the figure are descriptions of each correspondingly-numbered block.


1
large-signal? (DC or Transient)
YES
NO
2
waveform defined?
YES
NO
3
output=exponential waveform at present time
tran
4
output=DC supply
supply
5
output=small-signal AC value
ac

```
if (dc_domain | time_domain) {                  # Block 1
  if (waveform defined) {                       # Block 2
    vs = waveform at current time               # Block 3
  }
  else (if waveform not defined) {
    vs = supply
  }                                             # Block 4
}
elseif (freq_mag) {
  vs = magnitude of complex value               # Block 5
}
elseif (freq_phase) {
  vs = phase of complex value
}
```

Selecting an output (continued)

1. The first condition is true if dc_domain or time_domain is true (i.e., if a either a DC or a transient analysis is being performed). See Block 2.

   The first condition is false if the simulator is executing a small-signal analysis. See Block 5.
2. The second condition is true if the waveform is defined. See Block 3. (The waveform is considered defined if either v1¼0 or v2¼0 (see the figure that refers to the exponential waveform) and the value of tau is positive.)

   The second condition is false if the members of tran have not been specified. See Block 4.
3. The output of vsource_1 (vs) is given the waveform value, tran.
4. The output of vsource_1 (vs) is given the constant DC value, supply.
5. If the simulator is performing an AC analysis, it asks for either the magnitude (freq_mag) or the phase (freq_phase) of the complex value specified by ac. In such cases (the else if cases), the simulator takes the appropriate action, depending on the values of freq_mag and freq_phase.

Using a Structure Reference To convert the following conditional statements to those actually appearing in the template, it is necessary to gain access to the parameter within the ac and tran structures.

```
if (dc_domain | time_domain) {                   # Block 1
  if (waveform defined) {                        # Block 2
    vs = waveform at current time                # Block 3
  }
  else (if waveform not defined) {
    vs = supply                                  # Block 4
  }
}
elseif (freq_mag) {
  vs = magnitude of complex value                # Block 5
}
elseif (freq_phase) {
  vs = phase of complex value
}
```

In the MAST language, the “structure reference” operator (->) is used to do this, as follows:

```
name1->name2
```

Refer to Section 3.4.4 of the MAST Reference Manual for more information on the structure reference operator.

Here, the structure named name1 contains a member parameter named name2. For example, the three parameters within the tran structure are indicated by:

tran->v1

tran->v2

tran->tau

The type of name1->name2 is the same as the type declared for name2. Therefore, because v1 is a number parameter within the tran structure, tran->v1 is also a number, which you can use just like any number parameter.

Completing the Values Section for tran By using simvars, conditional statements, and structure references, the template can now define the tran argument in such a way that it activates the exponential waveform output for either a DC or transient analysis.

First, the members of tran are evaluated to see if they have defined the waveform. The waveform is considered defined if an instance of vsource_1 (in a netlist) has set the value of tau > 0 and either v1¼0 or v2¼0. If these conditions are met, tran will override the constant value specified by supply. This ensures continuity in the waveform at time 0.

Second, the waveform must be expressed as a function of time. The MAST language provides a collection of mathematical and other functions that you can use in expressions. The MAST Reference Manual lists all such functions. This example requires the exponential function ex, which is implemented in the MAST language by exp(x).

The resulting expression is:

```
v1 + (v2 - v1)*(1 - exp(-time/tau))
```

where time is a simvar. Note that time is set to the current simulation time during a transient analysis and to 0 during a DC analysis.

Third, the conditional statements of the vsource_1 template use simvars (time, dc_domain, time_domain, freq_mag, freq_phase) and the structure reference operator (->) to decide whether the conditions for tran are satisfied. Compare the actual condition testing of whether the waveform is defined (from the values section, below) with the informal versions which follow it. Also, compare the final statements defining vs (below) with the informal versions.

```
values {
 if (dc_domain | time_domain) { # If large signal
  if ((tran->v1~=0 | tran->v2~=0) & tran->tau>0){
                                # If waveform is defined
   vs=tran->v1 + (tran->v2-tran->v1) *(1-exp(-time/tran->tau))
  }                    #   source voltage = waveform
                       #   at current time
  else {               # otherwise
   vs=supply           #   source voltage = supply
  }
 }
 else if (freq_mag) {   # or if source is ac magnitude
  vs=ac->mag            #   source voltage = magnitude
 }
 else if (freq_phase) { # or if source is ac phase
  vs=ac->phase          #   source voltage = phase
 }
}# end of values section
```

```
if (dc_domain | time_domain) {                      # Block 1
  if (waveform defined) {                           # Block 2
    vs = waveform at current time                   # Block 3
  }
  else (if waveform not defined) {
    vs = supply                                     # Block 4
  }
}
elseif (freq_mag) {
  vs = magnitude of complex value                   # Block 5
}
elseif (freq_phase) {
  vs = phase of complex value
}
```

Informal Presentation of Condition Testing

Equations Section The template equation describes how the model that the template represents interacts with the rest of the system. The voltage source modifies the current at its pins p and m by its contribution is. The is current contribution is to be determined by the simulator such that the voltage drop across the source equals the specified source value. As with the vsource template, you must declare the source current as a var variable. The resulting equations section is as follows:

```
equations {             # start of equations section
 i(p->m) += is          # add current contributed by source
 is: v(p)-v(m)=vs       # determine current contributed
                        #  by source
}                       # end of equations section
```

Frequency-Dependent, AC Output

The definition for the ac argument is very similar to that of the tran argument. Simvar variables, conditional statements, and the structure reference operator (->) are used to define conditions for which the ac argument determines the output. The principal difference is that ac defines a small-signal response, such as that obtained from an AC analysis, which assumes a linearization of the system about its operating point. It is a first-order approximation of the large-signal response at that operating point and is typically evaluated as function of frequency.

The following topics describe how the small-signal functionality is modeled in the vsource_1 template:

- Small-Signal Structure Type Parameter
- Small-Signal simvar Variables
- Small-Signal Conditional Statements"

Small-Signal Structure Type Parameter

The ac argument represents a complex number specified by magnitude and phase. Because the MAST language does not include complex arguments and parameters, you declare this argument as a structure consisting of two real numbers (which represent magnitude and phase) and initialize them both to 0, as follows:

```
struc {
  number mag=0,
         phase=0
} ac = ()
```

Note that this corresponds to Method 1 of initializing a structure described for the tran argument (see the topic titled "Initializing a Structure - Method 1 Using vsource_1"). It declares a parameter named ac to be a structure consisting of number type parameters mag and phase, each of which is initialized to 0.

The initializer for the structure is simply a pair of parentheses preceded by an equals (=) sign. This means that, in a netlist entry referring to vsource_1, if the ac structure is not defined, then its members take on the values specified by their initializers (here, 0 for each). If the structure's initializer were missing, the structure would have to be defined (given a value) in a netlist entry, even though the individual members have initializers.

Small-Signal simvar Variables

Recall that the three arguments of the vsource_1 template—supply, tran, ac—are provided to allow a user to select whether the output will be a constant voltage, an exponential voltage, or an AC signal. The supply and tran arguments provide output voltages that are to be used for large-signal analyses (DC and transient analyses), while the ac argument provides an AC voltage for small-signal analyses (AC analysis). Therefore, the template must obtain, from the simulator, information about the simulation being used.

As with the tran argument, the ac argument uses simvars to determine whether or not the simulator is performing an AC analysis. The ones used by ac are:

freq_mag	has a non-zero value when a small-signal (frequency domain) analysis is being carried out and the simulator is requesting the magnitude of a complex number. It is equal to 0 otherwise
freq_phase	has a non-zero value when a small-signal (frequency domain) analysis is being carried out and the simulator is requesting the phase portion of a complex number. It is equal to 0 otherwise.

```
values {
 if (dc_domain | time_domain) { # If large signal
  if ((tran->v1~=0 | tran->v2~=0) & tran->tau>0){
                                # If waveform is defined
   vs=tran->v1 +(tran->v2-tran->v1) *(1-exp(-time/tran-
>tau))
  }                     #   source voltage = waveform
                        #   at current time
  else {                # otherwise
   vs=supply            #   source voltage = supply
  }
 }
 else if (freq_mag) {    # or if source is ac magnitude
  vs=ac->mag             #   source voltage = magnitude
 }
 else if (freq_phase) { # or if source is ac phase
  vs=ac->phase           #   source voltage = phase
 }
}# end of values section
```

Small-Signal Conditional Statements

The if else statements for the ac argument check to see whether the simulator is requesting the magnitude of a complex number (i.e., whether freq_mag ¼ 0). If so, then the output of vsource_1 (vs) is assigned the value of mag within the ac structure by using the structure reference operator:

```
30    vs = ac->mag
```

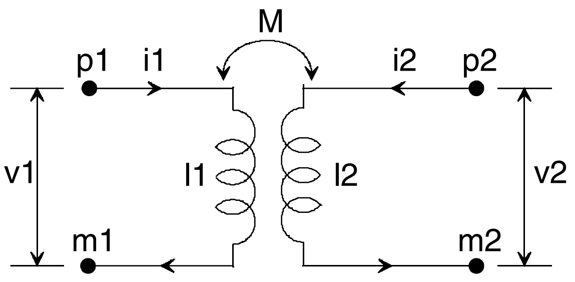
If freq_mag =0, the template checks to see whether the simulator is requesting the phase of a complex number (i.e., whether freq_phase ¼ 0). If so, then the output of vsource_1 (vs) is assigned the value of phase within the ac structure by using the structure reference operator:

```
33   vs = ac->phase
```

```
18   values {
19    if (dc_domain | time_domain) { # If large signal
20     if ((tran->v1~=0 | tran->v2~=0) & tran->tau>0){
21                                  # If waveform is defined
22      vs=tran->v1 + (tran->v2-tran->v1) *(1-exp(-time/tran-
>tau))
23     }                   #   source voltage = waveform
24                         #   at current time
25     else {              # otherwise
26      vs=supply          #   source voltage = supply
27     }
28    }
29    else if (freq_mag) {   # or if source is ac magnitude
30     vs=ac->mag            #   source voltage = magnitude
31    }
32    else if (freq_phase) { # or if source is ac phase
33     vs=ac->phase          #   source voltage = phase
34    }
35   }# end of values section
```

Modeling a Linear Transformer with MAST

In general, a transformer is modeled as two or more magnetically-coupled inductors. The linear transformer template (xformer, listed below) consists of two inductors and their mutual coupling, as shown in the following figure.



Coupled inductors

The xformer template uses the inductor and mutind templates in a netlist section. The arguments of xformer are the two inductance values and their coupling factor, whose user-specified values are passed to inductor.1, inductor.2, and mutind.12.

```
element template xformer p1 m1 p2 m2 = l1, l2, k
  electrical p1, m1, p2, m2
  number l1, l2, k = 1
{                          # start body of template
  number m                 # local declaration
  parameters {             # start parameters section
    if (k < -1 | k > 1) {
      error("%:coupling factor must be between -1 and 1:k=%",
        instance(), k)     # if error, display message
                           #  and terminate simulation
    }
    else {
      m = k * sqrt(abs(l1 * l2))
                           # otherwise, compute mutual
                           #  inductance
    }
  }
  # Use following netlist to make a transformer from
  # two mutually-coupled inductors
  inductor.1 p1 m1 = l1     # inductor netlist entry
  inductor.2 p2 m2 = l2     # other inductor netlist entry
  mutind.12 i(inductor.1) i(inductor.2) = m
                           # mutual inductance netlist entry
}
```

ASCII text of this example is located in:
install_home/example/MASTtemplates/structured/
xformer.sin

The description of the xformer template is divided into the following topics:

- Header and Header Declarations
- Netlist Section
- Local Declarations

- Parameters Section
- Error Reporting
- Template Equation

Header and Header Declarations

Three values characterize a model of a linear transformer: two self inductances and a mutual inductance. However, from a user's point of view, it is more convenient to supply a mutual coupling factor, k (which has a value between -1 and 1) instead of a mutual inductance (which must be calculated from k and the two inductance values). Each inductance is connected between a pair of electrical pins, so the template header and header declarations are:

```
element template xformer p1 m1 p2 m2 = l1, l2, k
  electrical p1, m1, p2, m2
  number l1, l2, k=1
```

Note that the arguments are declared such that k has a default of 1, while l1 and l2 have no default values. Thus, when xformer appears in a netlist, its l1 and l2 values must be specified, whereas k defaults to 1 (which models ideal coupling, so k need not be specified if this is acceptable).

Netlist Section

This transformer template uses an internal netlist to connect instances of the inductor and the mutind templates, which have been declared as element templates:

```
inductor.1 p1 m1 = l1
inductor.2 p2 m2 = l2
mutind.12 i(inductor.1) i(inductor.2) = m
```

It is important to understand that this netlist can work correctly only if both the inductor and mutind templates are element templates. The reasons are as follows:

- If the inductor template is not an element template, the inductor branch currents (which are vars) are not available outside the template. Thus, declaring them as refs in the mutind template would not be sufficient to provide access to them.
- The mutind template must be declared an element template because it modifies the equations associated with the inductor currents, and this is possible only at the same level of hierarchy.

Local Declarations

The only item that requires local declaration is the mutual inductance m, which is a numeric parameter:

```
number m
```

In general, local declarations of parameters are very similar to argument declarations. They can take on the same types as arguments and they can have initializing values. The following statement:

```
type name [=initial_value]
```

declares the parameter name as type type and, if the optional part of the statement is present, sets its value to the specified initial_value. If initial_value is not present, name has the initializing value undef, a special numeric constant representing an undefined value in the MAST language. You can use this constant wherever you can use a numeric constant, although the results are unpredictable if the simulator has to operate on undefined values.

If an initial value of a local parameter is specified, you can alter it from the Saber simulator, using the alter command. You cannot alter a local parameter that has no initializer. Note that you cannot alter the value of a local parameter--you can alter only its initializer. This is because you can modify parameters in the parameters section.

Parameters Section

The parameters section of a template is useful for manipulating parameters, such as template arguments, that are variables or structures that do not change during simulation.

The parameters section is used primarily for the following purposes:

- Validating template arguments
- Converting template arguments into parameters needed by a model
- Specifying, for statistical analysis, correlation between parameters
- Setting up static information, such as sample points, that depends upon template arguments

The xformer template example addresses only the first two points.

The parameters section is a procedural section--meaning that, when executed, its statements are executed in the sequential order. It consists of the keyword parameters followed by a sequence of statements between braces ({}).

The simulator executes the parameters section as follows:

- Once during initialization of the system description
- Once for each alter command (including alter commands generated by the vary command), but only for the template instances affected by the alter command
- Once for each Monte Carlo run, but only for the template instances having statistical distributions

Statements in the parameters section follow the same syntactical rules as values section statements, except that the operations can involve only parameters, that is, template arguments and parameters declared locally in the template. There are also several routines, most for error reporting, that you can use only in the parameters section and in when statements.

Template arguments and parameters declared locally in a template are collectively called parameters. They can assume the same types (numbers, structures, etc.), can take on the same values, and can be used identically in a template, except that a template argument cannot be changed inside a template. That is, it is not possible to assign a value to a template argument in an assignment statement.

The parameters section in the xformer template checks to see if the user-specified value for coupling factor, k, lies between -1 and 1:

```
if (k < -1 | k > 1) {
```

Note the opening brace at the end of the line; it introduces the error reporting statements that follow, as explained in the topic titled "Error Reporting".

The else clause is used, when there is no error detected, to compute the mutual inductance and assign the result to m, which is later used in the netlist section.

```
parameters {              # start parameters section
  if (k < -1 | k > 1) {
   error("%:coupling factor must be between -1 and 1:k=%",\
     instance(), k)# if error, display message
                          #  and terminate simulation
  }
  else {
    m = k * sqrt(abs(l1 * l2))
                          # otherwise, compute mutual
                          #  inductance
  }
}
```

Error Reporting

If a user has specified invalid argument values to a template instance in a netlist, there should be a clear error message to indicate the situation. In the xformer template, you should report an error if the user-specified value for coupling factor, k, does not lie between -1 and 1, because such a factor represents a physically impossible situation.

The MAST language supports a set of functions that can be included in a template that will inform the user which instance has the bad value, what the bad value is, and what must be done to correct the problem:

- The instance( ) function returns a string containing the full pathname of the template instance, describing its exact position in the design hierarchy.
- The error(format [, argument...]) function reports error conditions that the template cannot correct. It issues an error message specified by the arguments of the function, which are:

format	a string holding the message text and, optionally, placeholders for information that is can vary, such as the instance name. A string is a sequence of characters enclosed between quotation marks (" ") but not including quotation marks. Percent signs (%) in the string are interpreted as placeholders, unless preceded by a backslash (\). In the printed message, they are replaced with the remaining arguments of the error() function.
argument	an optional numeric or string quantity, such as a parameter or function value. When the error message prints, the i’th argument replaces the i’th placeholder in format. If the i’th argument is a variable of type structure, the whole structure is printed in place of only the i’th placeholder. There must be as many arguments as there are placeholders in the format string.

  Calling the error() function causes the condition causing the error to be discarded. If this happens during simulator start-up, the simulator quits. If it happens because of an alter statement, the alter statement is discarded.

- The warning(format [, argument...]) function is for reporting an abnormal condition that the template can correct. It issues a warning message, where format and argument are as described for the error() function, above. Calling the warning() function has no effect on simulator operation.
- The message(format [, argument...]) function is for debugging and for printing out general information. It simply issues the message described by format and argument. Calling the message() function has no effect on simulator operation.

It is typical to use these functions with conditional statements, although you can also use them in when statements.

For the xformer template, the error( ) function is required, because the template cannot correct a user-specified argument value. The resultant message should display the current instance, the bad value, and the range of valid values for the user to correct. This message is implemented as follows:

```
error("%:coupling factor must be between -1 and
1:k=%",instance(),k)
```

Note that the format string has two placeholders (the percent (%) signs), which are filled in with the instance name and the value of k, respectively.

For example, the following netlist entry specifies an invalid coupling factor of 2:

```
xformer.bad a b c d = 1m, 2m, 2
```

This would result in the following error message:

```
** ERROR "TEMPLATE_ERROR" ** xformer.bad:coupling factor must be between -
1 and 1:k=2
```

Template Equation

The xformer template combines existing models (inductor and mutind) into one template and provides some error checking—it has no characteristic equation of its own. However, it does compute the mutual inductance value (to be used by mutind) from the arguments of coupling factor (k) and the two inductance values (l1, l2).

The mutual inductance value is given by the equation:

$$m = k \bullet \sqrt{L_1 \cdot L_2}$$

This is implemented in the template as:

```
m = k* sqrt(abs(l1*l2))
```

where m is the argument of the mutind template instance, sqrt is the square root function, and abs is the absolute value function. The sqrt and abs functions are intrinsic to the MAST language (sqrt requires a non-negative argument). Although you could use conditional statements to restrict l1 and l2 to positive values, it is more efficient to use the abs function and take the square root of their absolute value.

Thus, this statement assigns a value to m resulting from an expression using the xformer arguments k, l1, and l2.

Modeling a Temperature-Dependent Resistor with MAST

Materials used in electronic components often change their characteristics with temperature variations. You can distinguish two different causes of temperature-related effects:

- Operation at an ambient temperature different from the nominal one
- Self-heating effect in a component

The template in this section models a resistor that can operate at various ambient temperatures. This template, called resistor_1, also checks for a value of zero resistance and allows power to be extracted, is listed below:

```
element template resistor_1 p m = res, tc, tnom
  electrical p, m
  number res, tc[2]=[0, 0], tnom=27
  external number temp
  export val p power            # make power available
{
  number r                      # local declaration
  val v v
  val i i

  parameters {
   r = res * (1 + tc[1]*(temp-tnom) + tc[2]*(temp-tnom)**2)
                                # computation of resistance
   if (r==0) error("%: resistance value is zero",instance())
                                # message returned if error
  }

  values {
    v = v(p,m)
    i = v/r
    power = v*i                 # calculate power
  }

  equations {
    i(p->m) += i                # current based on res.
  }
}
```

The description of the resistor_1 template is divided into the following topics:

- Header and Header Declarations -- shows how to declare, initialize, and use arrays
    - External Variable for Temperature
- Temperature Dependence of the Resistance Value
- Equations Section
- Parameters Section and Local Declarations
    - Error Checking and Message
- Altering an External Parameter
- Export a Variable

Header and Header Declarations

The arguments of the resistor template are the nominal resistance, the temperature coefficients, and the nominal temperature:

```
element template resistor_1 p m = res, tc, tnom
  electrical p, m
  number res, tc[2]=[0, 0], tnom=27
```

Note that this template declares all arguments on the same line, shown as follows:

```
  number res, tc[2] = [0,0], tnom=27
```

The nominal resistance, res, is not initialized and must be specified by the user. Temperature coefficients, tc, are contained in an array of length 2; its entries are both initialized to 0. The nominal temperature, tnom, is initialized to 27×C.

It is preferable to have just one argument, tc, for the temperature coefficients, by declaring it to be an array of length 2. An array is an ordered set of parameters that are all of the same type.

The elements of an array need not be number parameters, although they are here. Array elements can be of any valid parameter type, such as structures or enums. Arrays can be of fixed size, as in this example, or of unbounded size, which you can specify by declaring them with an asterisk (*), for example, as in tc[*]. If not declared otherwise, array subscripts run from 1 to the array size. The i'th element of array is array[i]. For information about multi-dimensional arrays and subscripts not starting at 1, refer to the MAST Reference Manual.

For the resistor_1 template, if both array elements tc[1] and tc[2]are left at 0, the resistance value is independent of temperature.

Note the following use of initializers for template arguments:

- res has no initializer, so anyone using the resistor_1 template must specify a resistance value.
- tc[1], tc[2], and tnom have initializers or default values that are satisfactory in most cases, because temperature effects are rarely used.

The topic titled "Export a Variable" describes how to use power, which is declared as an export variable in the header declarations:

```
export val p power
```

External Variable for Temperature

In order to compute the actual resistance value (see the topic "Temperature Dependence of the Resistance Value"), you need to use the value of the ambient simulation temperature, temp. This is a "global" variable that is used by all templates; it is specified in the provided header.sin file. The Saber simulator, by default, includes the contents of this file at the top of each system description. (For more information on which files are automatically activated by the Saber simulator, see the topic "Information Common to all Generic Templates" in the Saber online documentation.) Therefore, a template can refer to the value of temp by declaring it external in the header declarations section:

```
external number temp
```

This ambient temperature is initialized in header.sin to a suitable value, 27×C.

In general, a parameter declared as external is declared as a type that is found elsewhere in the design (not necessarily within the same template). If a template instance, say x.y, refers to an external parameter in a hierarchical design, the simulator resolves the external reference by "walking up" the hierarchy from x.y, going first to the "parent" of x.y, then to its "grandparent," and so on. This is continued until either the specified name is found or until the top level of the design is reached. If the simulator finds the name, it uses its value in x.y. Otherwise, it reports an error. This mechanism for resolving external references allows you to have different values for the same variable in different parts of the design. Refer to the MAST Reference Manual for a more extensive example of this resolution process.

By declaring a variable as external in the header declarations section, you can change its value for any instance of that template in a design—as if the variable were an argument for that template. For example, the netlist entry listed below changes the value of temp for resistor_1.r1 to 58×C; all other templates in the design (including other instances of resistor_1) use the value of temp specified in the header.sin file (27×C).

```
resistor_1.r1 n1 b10 = 5k, temp=58
```

Note:

You cannot modify the value of an external variable from within a template.

Temperature Dependence of the Resistance Value

This model implements first-order and second-order temperature dependence according to the following:

$$R = R_{nom} \bullet [1 + K_1(T - T_{nom}) + K_2(T - T_{nom})^{2}]$$

where K_1 is the linear coefficient and K_2 is the quadratic temperature coefficient. The nominal resistance value, R_{nom}, is assumed to have been measured at the nominal temperature, T_{nom}.

The actual resistance value, R, is therefore a function of the difference between the actual simulation temperature, T, and Tnom.

Note that Equation 3 is not the characteristic equation of the template—it is only a modification of the user-specifiable nominal resistance, Rnom; however, the result of this modification (R) is used in the characteristic (template) equation.

Equations Section

The equation for the resistor_1 template is similar to the one used in the resistor template, except that it uses the calculated resistance value (r) from the equation shown in the topic titled "Temperature Dependence of the Resistance Value", rather than the argument value (res).

Refer to the highlighted sections in the following code:

```
element template resistor_1 p m = res, tc, tnom
  electrical p, m
  number res, tc[2]=[0, 0], tnom=27
  external number temp
  export val p power            # make power available
{
  number r                      # local declaration
  val v v
  val i i

  parameters {
    r = res * (1 + tc[1]*(temp-tnom) + tc[2]*(temp-tnom)**2)
                                # computation of resistance
    if (r==0) error("%: resistance value is zero",instance())
                                # message returned if error
  }

  values {
    v = v(p,m)
    i = v/r
    power = v*i                 # calculate power
  }

  equations {
    i(p->m) += i                # current based on res.
  }
}
```

Parameters Section and Local Declarations

The resistor_1 example shows how to compute the actual resistance value (by modifying the nominal resistance with temperature effects) and report an error if this value is 0. This prevents division by 0 in the template equation.

With access to the actual temperature established, the equation shown in the topic titled "Temperature Dependence of the Resistance Value", is implemented in the template as shown highlighted in the following parameters section:

```
number r                         # local declaration
# ...
parameters {
  r = res * (1 + tc[1]*(temp-tnom) + tc[2]*(temp-tnom)**2)
                                 # computation of resistance
  if (r==0) error("%: resistance value is zero",instance())
                                 # message returned if error
}
```

where array elements tc[1] and tc[2] correspond to temperature coefficients K1 and K2, respectively. This is a direct translation into the MAST language of the equation describing the temperature dependence of the resistor (note the exponentiation operator, **). The actual resistance value is assigned to a local parameter r—you cannot simply modify res, because a template cannot change the value of an argument.

Error Checking and Message

A conditional expression (beginning with if) is included to test whether the actual resistance value (r) is 0. Using a relational expression in an if statement to check whether a variable is equal to a number requires that the double equals sign (= =) be used as shown highlighted in the following example:

```
number r                  # local declaration
# ...
parameters {
  r = res * (1 + tc[1]*(temp-tnom) + tc[2]*(temp-tnom)**2)
                          # computation of resistance
  if (r==0) error("%: resistance value is zero",instance())
                          # message returned if error
}
```

If r is calculated to be 0, an error is returned using the error() and instance() functions. These indicate the erroneous value and the template instance in which it occurred. Note that the error() function call is on the same line as the if statement.

Altering an External Parameter

You can include an alter command within a netlist file to change a value of a declared parameter that is available at the same level of hierarchy. The altered parameter must be initialized and available locally. It may be declared within any of the following:

- a template in the netlist (e.g., within resistor_1)
- a template not in the netlist but that is automatically included when the Saber simulator is invoked (e.g., within header.sin)
- the netlist itself (e.g., by including a line such as number von=5)

The most useful occurrence of this is to alter the default temperature of the system (temp, which is declared within header.sin) at the top of a netlist.

In that case, you cannot simply change the initializer on the declaration, because you have no local access to the declaration in header.sin. However, in the netlist, you can specify one of the following and change the default temperature in the system for that simulation. With the first method, the value of temp in the alter statement overrides the value of temp in header.sin. With the second method, you specify a temperature instance-by-instance in a netlist as shown in the following example:

```
alter temp=37
r.load 0 aout = rnom=1k, tc=[0.4,  0.5]
. . .
or
r.load 0 aout = rnom=1k, tc=[0.4,  0.5], temp=37
```

Export a Variable

As was done for charge of the capacitor_1 template, you can specify the power dissipation of the resistor_1 template as a val variable. Like charge, power is a quantity that is not required by the model, but the simulator can provide it with no performance penalty.

This is normally done by declaring a val variable named power in the template header and then calculating it as a function of other template variables (i, v) that have been previously declared. The computation for i, v and power is done in a values section. Refer to the highlighted statements in the following example:

```
element template resistor_1 p m = res, tc, tnom
  electrical p, m
  number res, tc[2]=[0, 0], tnom=27
  external number temp
  export val p power            # make power available
{
  number r                      # local declaration
  val v v
  val i i

  parameters {
    r = res * (1 + tc[1]*(temp-tnom) + tc[2]*(temp-tnom)**2)
                                # computation of resistance
    if (r==0) error("%: resistance value is zero",instance())
                                # message returned if error
  }

  values {
    v = v(p,m)
    i = v/r
    power = v*i                 # calculate power
  }

  equations {
    i(p->m) += i                # current based on res.
  }
}
```

The letter p in the declaration establishes the variable named power as power in watts, as defined in the units.sin file, which is provided with the Saber simulator. The simulator, by default, includes the units.sin file with each system description (netlist). (For more information on which files are automatically activated by the Saber simulator, see the topic “Information Common to all

Generic Templates" in the Saber online documentation.) The unit for power (watt) is then automatically used when power is displayed in the Scope Waveform Analyzer after simulation.

However, you can make this value of power available to another template hierarchically above an instance of the resistor_1 template, such as rlc1. (See the Example subtopic, below). This is done by declaring power as an export variable in the header declarations section of resistor_1:

```
export val p power
```

An export variable is simply a val variable, a var variable, or a branch whose declaration appears in the header declarations and is preceded by the keyword export. This makes the value of the export variable upwardly available in a hierarchy. That is, it can be used by a "parent" template at the next higher level in the hierarchy (i.e., a template that calls an instance of resistor_1). However, an export variable cannot be passed to other templates at the same level of hierarchy (i.e., within the same circuit).

The main restrictions on an export variable are the following:

- An export variable can be used only by the parent template (however, it can be exported upward from the parent template using another export declaration).
- An export variable cannot be used in the template equation of another template.
- An export variable must be declared in the header declarations section.

Example

Consider the rlc1 template, which uses hierarchy by including an internal netlist of three templates. Suppose resistor_1 were specified within rlc1, instead of resistor. Then rlc1 could declare a val variable that used the value of power from resistor_1.

This is shown as follows, using pwrd as the name of the val variable in rlc1:

```
val p pwrd                       # hypothetically in rlc1
pwrd = power(resistor_1.r1)      # hypothetically in rlc1
```

Of course, if the capacitor and inductor templates within rlc1 had also declared power as an export variable, the power from capacitor.c1 and inductor.l1 would also be available to rlc1. You could then write an expression within rlc1 to add the power from all three of these subordinate templates. This sum could then be specified in a signal list for display in Scope.

Modeling an Idealized Op Amp with MAST

The template for this operational amplifier is shown as follows:

```
element template opamp ip im out = a
  electrical ip, im, out # header declarations
  number a = inf
{                        # start of template body
  var i iout             # local declarations
  number x1, x2
  parameters {
    if (a==inf | a==undef) { # if gain is infinite
      x1=1; x2=0             # or undefined, then input
    }                        # voltage is 0;
    else {                   # otherwise, output voltage
      x1=a; x2=1             # is gain times input voltage
  }                          # end of parameters section
  equations {                # start of equations section
    i(out) += iout           # current contribution at output
    iout: x1 * (v(ip) - v(im)) = x2 * v(out)
                             # equation associated with iout
  }                          # end of equations section
}
```

The description of the opamp template is divided into the following topics:

- Current Contribution for Each Pin -- shows a modeling technique that lets you combine several apparently different model equations into one, so that altering a parameter value does not change the topology of the system
- Three-Pin Topology
- Characteristic Equations

- Header and Header Declarations -- shows the use of the special MAST constant inf
- Local Declaration of Local Parameters -- shows how to declare local parameters, along with how to initialize and alter them
- Parameter Section -- shows how to put multiple statements on the same line, separated by semicolons (;)
- Equations Section

Current Contribution for Each Pin

It is sometimes necessary to specify the current that a template contributes to each pin individually, rather than to specify the current flowing between two pins (i.e., the branch current). This is the case for the opamp template.

For example, the equation from the capacitor template can be expressed in a single line as follows:

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

This can be interpreted as a consolidation of the following instructions:

- Add the current defined by the expression to the current at pin p
- Subtract the current defined by the expression from the current at pin m

In the MAST language, you can express these two instructions separately, pin-by-pin:

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

For the Saber simulator, these two statements are equivalent to the single statement above in every respect—including the processing required to set up and solve the equations. The single-statement formulation guarantees that current is conserved in the template. The pin-by-pin formulation places this responsibility on the template writer, but it provides more flexibility. This is important in advanced or exceptional cases, such as the opamp template.

Which formulation to use is largely a matter of personal preference.

Three-Pin Topology

An important concept in simulation is the topology of the system to be simulated. Simply put, this is a description of how the through variables depend upon the across variables at each pin. In this context, the complexity of the dependence (linear, nonlinear, etc.) is irrelevant. All that matters is that there is dependence. The Saber simulator obtains this information from the template equations of all template instances connected to a node when it sets up the system equations. After that, the topology is fixed and may not be changed without exiting and restarting the simulation.

There are cases where the model equations change depending upon a model parameter, yielding different topologies for different values of that parameter. In such cases, you can modify the parameter using the alter command only in a value range that would not change the topology.

The opamp example introduces a modeling technique that lets you combine different model equations such that a single equation satisfies all values of the template parameters. This template allows specifying either finite or infinite gain.

This example uses different constructs than those of the other electrical element examples. The most notable distinction is the fact that this model provides a two-terminal input and a one-terminal output, which means that current iout cannot be found as a branch current—it must be declared as a var variable. Because of this, the template equation uses the pin-oriented formulation described in the topic titled "Current Contribution for Each Pin".

Characteristic Equations

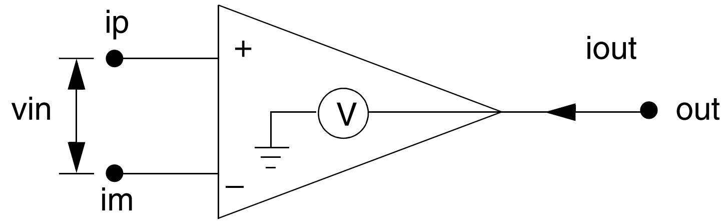
An idealized op amp (shown in the figure below) is modeled as a voltage-controlled voltage source, whose output voltage satisfies the following equation:

$V_{out} = A \bullet V_{in}$

where A is the gain. If the gain is infinite, it is more appropriate to use the form:

$V_{in} = V_{out} /A$

which immediately yields the result that, for an ideal op amp having infinite gain, the input voltage is 0. There is an additional equation, just as in any model having a voltage output—select the output current such that KVL is satisfied (i.e. Equations 4 and 5).



Simple, idealized operational amplifier

Header and Header Declarations

This op amp model is a three-terminal device with gain, a, as its only argument. The template models an ideal op amp with a default value of infinite gain. Therefore, the template header and header declarations are as follows:

```
element template opamp ip im out = a
  electrical ip, im, out
  number a = inf
```

Note that the third line initializes a to inf, a special numerical constant in the MAST language (similar to undef) that can be used to indicate an infinite value in the template. Although the Saber simulator does not use an infinite value for inf, you can assign inf as a value to a template parameter (such as a=inf). However, you should not use it as a value in an expression (such as x2=inf/6), except in a conditional statement to test equivalence (such as if (a==inf)).

Local Declaration of Local Parameters

The following equations describe the voltage relations in this op amp. The first equation is valid for all values of A except infinity; the second equation is valid for all values of A except 0.

```
Vout = A*Vin
Vin = Vout/A
```

This is a typical situation, in the sense that implementing just one of these equations does not completely describe the model.

For example, a naive implementation would incorporate the first equation for all values of A except infinity. It would treat A=infinity as a special case and set Vin=0 explicitly. This means that for most values of A, Vout depends on Vin, while for A=infinity, no such dependency exists. Therefore, by the definition of topology, altering A from a finite value to infinity changes the topology of the system. This is not a valid implementation, as explained at the beginning of the topic “Three-Pin Topology”.

You can combine the two equations into one by introducing two auxiliary (local) variables x1 and x2, with values that depend on the value of A (the template argument, a):

```
x1 * vin = x2 * vout
```

Because x1 and x2 depend only upon the template argument, a, they can be defined as local parameters:

```
number x1=1, x2=0
```

In general, a local parameter is declared in much the same way that an argument is declared. A local parameter can be any of the same types as arguments and they can have initializing values. The generalized form of a declaration:

```
type name  [=initial_value]
```

declares the parameter name as type type and sets its value to initial_value, if present. If initial_value is not present, name has the initializing value undef, a special numeric constant representing an undefined value in the MAST language.

The main reason for using undef is that if an initial value of a local parameter is specified, you can alter it from the Saber simulator, using the alter command. You cannot alter a local parameter that has no initializer. Note that you cannot alter the value of a local parameter—you can alter only its initializer. This is because you can directly modify a parameter value in the template.

Also, the output current must be declared as a var variable instead of as a branch current because the template has only one output pin (out):

```
var i iout
```

Declaring iout as a var variable allows it to be used as a system variable in the template equations.

Parameter Section

The following if-else statement in the parameters section expresses the dependency of x1 and x2 on the value of the argument, a:

```
parameters {
  if (a==inf | a==undef) {
     x1=1; x2=0
  }
  else {
    x1=a; x2=1
  }
}
```

The “|” character in the if statement denotes an OR condition—if a is specified as inf OR undef, then use the following values of x1 and x2: x1=1; x2=0.

Inserting the values of x1 and x2 into the equation:

x1 * vin = x2 * vout

and comparing it with the characteristic equations of the op amp, you can see that this single equation is valid for all values of the gain a.

Note that each of the lines listed below contains two statements that are separated with a semicolon (;). The semicolon marks the logical end of a line, so using one to separate two statements has the same effect as placing the statements on different lines.

x1=1; x2=0

x1=a; x2=1

Also, changing the value of the gain from a finite value to infinity does not change the topology of the system. This is because vin depends on vout in both cases (although for infinite gain, the dependence has zero value).

Equations Section

The following equations describe the voltage relations in this op amp. The first equation is valid for all values of A except infinity; the second equation is valid for all values of A except 0.

```
Vout = A*Vin
Vin = Vout/A
```

The += operator is used to add the current of the system variable, iout, to the current at the output pin of the op amp, i(out):

```
i(out) += iout
```

The simulator has to determine the value of iout in order to satisfy the voltage gain equation.

Consequently, to express Vin and Vout in terms of the across variables at the pins, the template equation becomes:

iout: x1 * (v(ip) - v(im)) = x2 * v(out)

Thus, apart from describing the characteristic equations of the op amp, there are two additional facts of note:

- There is no need to specify explicitly that there is no current between pins ip and im, although you could write i(ip->im) += 0. The default contribution of a template to the through variable at a pin is 0.
- There is no need to specify the contribution of a through variable to the reference (ground) node, 0. The way the current contribution of the template is specified implies that the current flows to ground. In general, if the contribution of a through variable is not balanced, it is assumed to flow into or out of the reference node.

The complete equations section is as follows:

```
equations {             # start of equations section
  i(out) += iout        # current contribution at output
  iout: x1 * (v(ip) - v(im)) = x2 * v(out)
                        # equation associated with iout
}                       # end of equations section
```

7
Modeling Digital Systems

Many models are written with the assumption that time-domain analyses are continuous. This means that the Saber simulator chooses the size of a time step to be as large or as small as necessary for accuracy. However, later topics shows that it is possible to limit time steps using the next_time and step_size simvar variables—this is done for the step output of the vsource_2 template.

Digital modeling also uses this ability of a template to schedule the exact time for variables to take on new values. Further, systems of digital models are concerned primarily with the occurrence of transitions from one logic level or state to another at connection points. A change of state on a connection point is known as an event. Thus, a digital model does not need to account for all continuously changing analog information; it needs only to schedule transitional events. As a result, 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.

This process, known as discrete time simulation, enables you to create digital models in a way that is an extension of time-domain modeling. The key to discrete time simulation with MAST is the when statement. In addition, digital modeling optimizes simulation speed without sacrificing simulation accuracy.

The following topics defines the terminology and some concepts of discrete time simulation:

- Digital Terminology
- Connection Points
- Time
- Values
- Events
- Scheduling With the when Statement
- Event Queue

The following topics describe the special approach to initialization with digital modeling:

- Initializing Connection Points - Digital MAST Modeling
- Initializing Internal Variables - Digital MAST Modeling

The following topics provide examples of digital modeling techniques:

- Modeling an AND Gate - MAST Template
- Initialization and Internal Events - MAST clock Template

Digital Terminology

Digital models often employ (but are not limited to) logic functions defined in terms of Boolean algebra. These functions are mathematical ways of representing propositional conditions based on binary choices referred to as logic states (or levels). There are several equivalent ways of expressing a pair of logic states:

- true—false
- HIGH—LOW
- on—off
- closed—open
- 1—0

For example, the Boolean AND function specifies two inputs and an output with the following conditions: if both inputs are HIGH (or 1, or true, etc.), then the output is also HIGH. Otherwise, the output is LOW.

The MAST language provides constructs to model these types of propositions, define logical levels, and track changes in logic levels. However, the capability for digital modeling incorporates a different approach to simulation from that used for analog models. Consequently, this requires a closer look at terminology.

Connection Points

In previous chapters, connection points have been declared as electrical. When connection points are declared as an electrical type, it is implied that they are pin-type (i.e., electrical is a specific kind of analog, pin-type connection point). Pin-type connection points define associated through and across variables for the simulator to solve as part of a continuously changing analog system.

Digital modeling uses a different type of connection point, called state, which does not define through or across variables. A state connection point is restricted to a finite set of values, which are assigned only at discrete times.

The terms connection points and pins are often used interchangeably. However, a pin is a specific kind of connection point that implicitly declares through and across variables for a model.

Time

A digital model does not need to account for all continuously changing analog information. A digital model only needs to account for scheduling transitions from one logic level, or state, to another. For digital models, the simulator has to check only the effects of state changes and only at scheduled times. As a result, the differences between analog and digital models lie within two fundamental quantities:

- time—continuous for analog; discontinuous for digital
- values—continuous for analog; discrete for digital

A state connection point declares that the values at those points change discontinuously in time. Such points have no through and across variables associated with them—there are no quantities that must “sum to zero.” They are simply connection points that allow state transitions (see the topic titled "Events") to pass between templates. A state connection point establishes a discrete time model, for which the simulator has to check effects of state changes only at scheduled times instead of after each time step.

Note:

It is possible to declare states for a continuous-valued (analog), but event-driven model.

Values

In addition to declaring connection points to be discrete in time (by using the term state), a digital model also declares them to assume only discrete values. This is done by providing an additional term in the declaration that defines a set of values the connection points can assume. Two such sets are provided in the units.sin file: logic_4 and logic_3. The logic_4 set contains MAST values for logic states 0, 1, X, Z; logic_3 contains MAST values for 0, 1, X. The available values for logic_4 and logic_3 are user-definable, just as those for electrical are user-definable within the units.sin file.

The default values for the set named logic_4 has the following MAST values (shown with their boolean interpretations):

l4_0	Logic 0
l4_1	Logic 1
l4_x	Logic X, unknown state
l4_z	Logic Z, high impedance state

Thus, a connection point declaration for a digital model appears in the header with two terms (such as state and logic_4) that declare discrete time and discrete values, respectively. For example,

```
state logic_4 cp1, cp2,...
```

This differs from the one-word declarations, such as electrical, used in template examples from previous chapters.

The definition for logic_4 is contained in the units.sin file and is expressed 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
```

Each line gives, respectively, a value that these pins can assume, the Boolean equivalent of the value, how the value appears when printed, and how the value appears when displayed graphically (as with the Scope Waveform Analyzer). The name of the unit is logic_4 and the default value is l4_x. Such unit constructs are documented in more detail in the Mast Reference Manual.

Events

An event occurs at a connection point when the value changes from one state value to another, at a scheduled time. It is very important to note that the mere occurrence of a state transition does not constitute an event—it must be scheduled.

An event is the result of associating a change in discrete value with a scheduled discrete time—if this change is not scheduled, it is not an event.

Because digital templates are concerned only with events on their connection points, they do not have through and across quantities to calculate (that is, they do not contain template equations). Thus, they do not need to be declared as element templates.

Scheduling With the when Statement

Scheduling an event at a connection point is absolutely necessary for the Saber simulator to perform discrete time simulation with a circuit (or system) of digital models. That is, you need to schedule events to “kick off” discrete time simulation among templates—if no transitions (events) are scheduled, there is no propagation of information from one digital template to the next.

Scheduling an event requires using the when statement, which is used in conjunction with the following intrinsic, event-driven functions:

- schedule_event
- handle
- event_on
- deschedule

The combination of the when statement with one or more of these functions is the basis of digital modeling, as shown in the examples in the topics titled "Modeling an AND Gate - MAST Template" and "Initialization and Internal Events - MAST clock Template". For reference information on the when statement and its intrinsic functions, see the MAST Reference Manual.

It is not always necessary to use a when statement to specify a value for a state variable. In the case of an internal (locally declared) variable, you can just use an assignment statement as follows:

```
int_state = l4_1
```

However, the assignment of this value will not be propagated externally (that is, to other templates) as a scheduled event.

It is possible to change state values without scheduling them (by using assignment statements in the template). However, these changes will not be propagated externally to other templates that are waiting for scheduled events.

As a general rule, use the when statement to schedule state values of connection points; use an assignment statement to specify state values of internal variables.

Event Queue

The distinguishing characteristic of discrete time simulation is the ability to detect and schedule events at connection points. The Saber simulator performs discrete-time analyses based on an event queue rather than on time steps. This event queue is simply the result of the Saber simulator automatically ordering scheduled events from all digital templates that need to be evaluated. During simulation, the simulator places events into the queue in a temporal sequence, moves them up, and executes them in that order when they reach the front of the queue.

The importance of the event queue is that the when statement —the principal MAST construct for discrete time simulation—depends entirely upon events being in the event queue.

The following topic titled shows a simplified representation of how an event queue works at the time of simulation.

A Typical Event Queue

The following table indicates that the Saber simulator has determined that there is a sequence of 227 events from all the digital templates in the circuit to be processed (for example, “#1: at time 3ms, the value at node out1 changes from its present value to l4_0; #2: at time 4ms, the value at connection point out3 changes...”). Once the Saber simulator executes an event, the simulator shifts all unexecuted events up one position until all events in the queue have been executed.

event #	node name	time	value
1	out1	3ms	l4_0

event #	node name	time	value
2	out3	4ms	l4_1
3	in1	6ms	l4_0
...			
227	in2	261ms	l4_x

The contribution of a digital template to the event queue is determined by using the when statement and intrinsic event-driven functions. For instance,

1. Event #1 was placed in the queue by including either of the following statements in a template:

```
schedule_event(3u,out1,l4_0)
handle = schedule_event(3u,out1,l4_0)
```

2. Once in the queue, event #1 was detected with the following when statement in the template:

```
when (event_on(out1))
```

3. Event #1 can be removed from the queue with the following statement in the template:

```
deschedule (handle)
```

4. Event #1 will not be placed in the queue with an assignment statement in the template, such as:

```
out1 = l4_0
```

The MAST Reference Manual and the examples in topics titled "Modeling an AND Gate - MAST Template" and "Initialization and Internal Events - MAST clock Template" provide more details on the syntax of the when statement.

Note:

Multiple events scheduled at the same time are executed in an order that is not user controllable.

Initializing Connection Points - Digital MAST Modeling

A digital model is mainly concerned with changes in discrete values of state variables (events) occurring on its connection points. An analog model is concerned with changes in continuous values of through and across system variables. Because of this, the digital model requires a different approach to initialization than that of an analog model.

analog:	A circuit consisting of analog templates is initialized by setting all nodes to a DC voltage of 0, treating all capacitors as open, and all inductors as shorts. The simulator then performs a DC analysis to find the resulting node voltages (operating point).
digital:	A circuit consisting of digital templates is initialized by first setting all nodes to a logic_4 value of l4_x (undetermined X state) and then checking templates to see which connection points have scheduled a value other than l4_x. The simulator then performs a DC analysis and executes the event queue.

The functional difference between these approaches is that nodes of an analog circuit do not require explicit initialization by the constituent templates, whereas nodes of a digital circuit do.

If you want to initialize a connection point of a digital template to a value other than l4_x, you must use the when statement to schedule an event.

Initializing Internal Variables - Digital MAST Modeling

Digital templates often use internal state variables. These can be initialized in one of two ways:

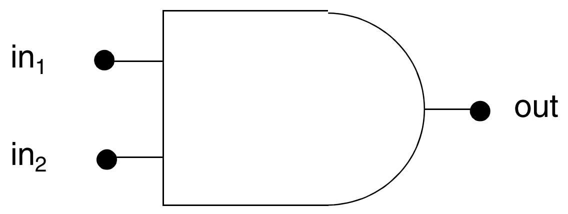
1. With a when statement—schedule an event to set the desired value
2. In the variable declaration—use an initializer to set the desired value

Note that the first method places the initialization of the internal variable in the event queue—which is not always necessary. Recall that setting the value of a state variable needs to be scheduled only if it is necessary to be in the event queue (that is, if other templates are “looking for” scheduled events). For internal state variables, this is often not the case—you can use the second method above to set their values.

The examples in topics titled "Modeling an AND Gate - MAST Template" and "Initialization and Internal Events - MAST clock Template" demonstrate the initialization of connection points and internal variables.

Modeling an AND Gate - MAST Template

The AND gate used in this example has two inputs and one output. The output is the logical (Boolean) AND of the two inputs, as shown in the following figure.



Logical AND Gate

The and template (shown below) implements the AND gate in the MAST language. The characters “|” means OR, “&” means AND, “~=” means “is not equal to.”

```
template and in1 in2 out = td
  state logic_4 in1, in2, out   # state connection points
  number td=0
{
  state logic_4 out_state      # internal state declaration
  when (event_on(in1) | event_on(in2)) {
    if ((in1==l4_1) & (in2==l4_1)) { # AND logic
      out_state = l4_1
    }
    else out_state = l4_0

    if (driven(out)~=out_state) {   # update AND gate output
      schedule_event(time+td, out, out_state)
    }
  }
}
```

ASCII text of this example is located in:
install_home/example/MASTtemplates/structured/and.sin

The AND gate template description is divided into the following topic:

- MAST AND Gate Model Logic States
- Header Declarations
- Local Declarations
- The when Statement in the MAST AND Gate Template - shows how to schedule and detect digital events and how to use a local variable to internally hold a logic_4 value for further event scheduling
- Initializing the AND Gate MAST Template
- MAST Template Conflict Resolution - the driven Function - show how to use the driven(out) statement and the l4cnfr subroutine
- A Netlist Example - MAST AND Gate Template

MAST AND Gate Model Logic States

The following table shows how an AND gate determines the output at pin out, according to the input at pins in1 and in2. The model provides the Boolean AND function as indicated in the truth table. In addition, it incorporates a non-negative delay parameter named td for delay between input and output.

This truth table shows that discrete states appear at the in1, in2, and out connection points. This implies a fundamental difference from the analog electrical pins in other examples.

in1	**in2**	**out**
0	0	0
0	1	0
1	0	0
1	1	1

Header Declarations

The first indication of the major difference between analog and digital models appears in the following header declarations of the and template:

```
state logic_4 in1, in2, out
```

It is important to note that the word state declares that the input and output connection points assume discrete state variables (for discrete time simulation). The term logic_4 declares the units for discrete values that these connection points will assume. The word state is reserved, while logic_4 is user-definable.

Local Declarations

The only local declaration for this template is out_state, which is a state variable (state) that can assume discrete values (logic_4):

```
state logic_4 out_state
```

This declaration creates a local state variable that can be assigned logic_4 values in a when statement (see below).

The form of this declaration is identical to the declarations of the connection points in the header.

The when Statement in the MAST AND Gate Template

The when statement is a fundamental construct for digital modeling. It provides the primary means of detecting and scheduling events on state connection points and state variables.

A when statement can be used in a netlist (at the top level of hierarchy) as well as in a template.

The syntax for the when statement is as follows:

```
when (condition) {
   statements
}
```

The above syntax means, “When condition occurs, execute the following statements.” The power of this construct comes from the full set of simulator variables, conditional statements, and intrinsic (built-in) functions that are part of the MAST language. Refer to Section 5.4 of the Mast Reference Manual for more information on event-driven functions.

The following when statement from the and template illustrates its usage in digital modeling:

```
when (event_on(in1) | event_on(in2)) {
  if ((in1==l4_1) & (in2==l4_1)) {
    out_state = l4_1
  }
  else out_state = l4_0

  if (driven(out)~=out_state) {
     schedule_event(time+td, out, out_state)
  }
}
```

This means that when at least one of the inputs to the AND gate changes, then the output state might need to change. The event_on intrinsic function detects any change in the value at either input at the time of the change.

Examine the following lines:

```
if ((in1==l4_1) & (in2==l4_1)) {
  out_state = l4_1
}
 else out_state = l4_0
```

This statement uniquely determines the next output state and specifies that it be stored in a local variable called out_state. Its logic can be interpreted as follows:

> If in1 has value logic 1 and input in2 has value logic 1, then the next output state will be logic state 1; otherwise, the next output state will be logic 0.

Examine the following lines:

```
if (driven(out)~=out_state) {
   schedule_event(time+td, out, out_state)
}
```

This statement means that the result of the AND between in1 and in2, as computed by the previous if-else statement, is propagated to the output only if the output has changed. Its logic can be interpreted as follows:

> If the output pin (out) has been scheduled by this template to have a value not equal to the value of out_state, then schedule an event on the output pin at the present time plus the delay time, td. The value of the scheduled event is out_state.

For example, if the two inputs to the AND gate were logic 1 and 0, respectively, then the output would be logic 0. If the inputs had changed to logic 0 and 0, the output would be still be 0 and it would not be necessary to use the schedule_event function to change the output of the template.

The general form of the schedule_event intrinsic function is:

```
schedule_event (schedule_time, state, value)
```

where:

schedule_time	is the time at which the event is to occur
state	is the state variable whose change-of-state constitutes the event
value	is the new value to which state is to change

Initializing the AND Gate MAST Template

In this example, in1, in2, out_state, and out are not explicitly initialized by the template—they are allowed to remain at l4_x, as set by the simulator. Note that the internal state variable, out_state, serves as an intermediate placeholder whose value is established by the if-else statements regarding in1 and in2.

The remainder of the when statement schedules a logic_4 value to be assigned to the connection point out, but only if there is a change in out_state resulting from events on in1 and/or in2.

The MAST Reference Manual lists several simvar variables that are used in initializing a digital model before DC, DC Transfer, and Transient analyses. These simvar variables were not used in this example because the template uses the fact that in1, in2, out_state, and out are initialized to l4_x by default (i.e., no event scheduling is required to determine that they are initialized to l4_x).

In the example “Initialization and Internal Events - MAST clock Template”, however, two of these simvar variables (dc_init, time_init) are used to schedule an event to override the default initialization value of l4_x.

MAST Template Conflict Resolution - the driven Function

It would seem that the following code:

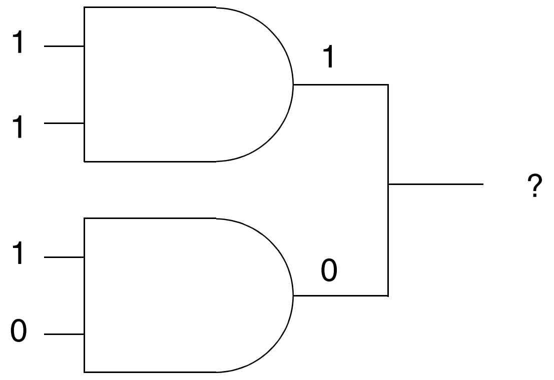
```
if (driven(out)~=out_state) {
  schedule_event(time+td, out, out_state)
}
```

could have been accomplished with:

```
if (out~=out_state) {
  schedule_event(time+td, out, out_state)
}
```

What was the purpose of the driven intrinsic function? The driven function is necessary because it is possible for a pin to be driven by more than one gate.

The following figure shows two logic gates driving the same node. The output of one gate is logic 1 and the output of the other is logic 0. What is the correct value at that node? It's ambiguous. It might be 1, it might be 0, and it might be something in between. This conflict is resolved by the conflict resolution mechanisms discussed in the topic titled "Defining Conflict Resolution in a MAST Template". The value of out may be different from the value of driven(out).



Conflict at a digital node

The driven function lets you proceed without having to worry about what is happening outside the template.

The value of out reflects the value to which the output pin is finally set, whereas the value of driven(out) reflects the most recent value that the template has scheduled for the connection point, out.

Defining Conflict Resolution in a MAST Template

When two templates drive a single pin, there is potential for conflict. The following mechanism is provided to resolve this conflict:

> A resolving subroutine is defined for each type of logic units declared (e.g., l4cnfr is defined for logic_4). When the simulator detects conflict, it automatically passes all conflicting values to this subroutine. The subroutine automatically resolves conflicting values on a pairwise basis and returns the results.

The subroutine that resolves conflicts is declared together with the state unit to which it applies (such as logic_4). Typically, this is in the units.sin file, which is automatically included for simulation. The declaration for the logic_4 state unit looks something like the following:

```
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\
{CONFLICT_RESOLUTION:foreign l4cnfr}
```

This declares that the conflict resolution subroutine is a foreign function named l4cnfr. This subroutine is called to resolve any conflicts that occur at logic_4 state connections. The l4cnfr subroutine is provided to the Saber simulator in the same manner as any other foreign subroutine.

The conflict resolution subroutine l4cnfr selects the first two template output pins (logic 0 and another logic 0). For logic_4, the resolution of these two is defined as logic 0, so the subroutine determines a logic 0 to be paired with the next output. Thus, the result of the first conflict resolution (logic 0) is paired with the output of the third template (logic Z). For logic_4, the resolution of logic 0 with logic Z is logic 0. (The high impedance state is like an identity operator when used in conflict resolution.) Therefore, the final result of conflict resolution is logic 0, which is returned to that node.

For example, suppose that three different templates with logic_4 connections have their outputs connected to a single pin. Suppose further that the output of the first template is logic 0, that of the second is logic 0, and that of the third is logic Z (high impedance). The conflict resolution mechanism would then proceed as follows to determine the correct value of the node.

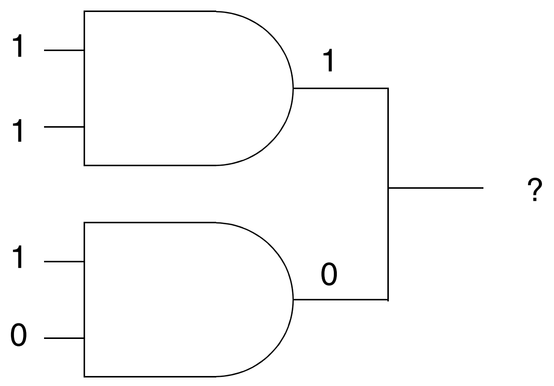
The body of the following table shows the resultant logic_4 value when a state along the left appears simultaneously with a state along the top. This is the truth table for conflict resolution for logic_4 unit states.

	l4_0	**l4_1**	**l4_x**	**l4_z**
l4_0	l4_0	l4_x	l4_x	l4_0
l4_1	l4_x	l4_1	l4_x	l4_1
l4_x	l4_x	l4_x	l4_x	l4_x
l4_z	l4_0	l4_1	l4_x	l4_z

A Netlist Example - MAST AND Gate Template

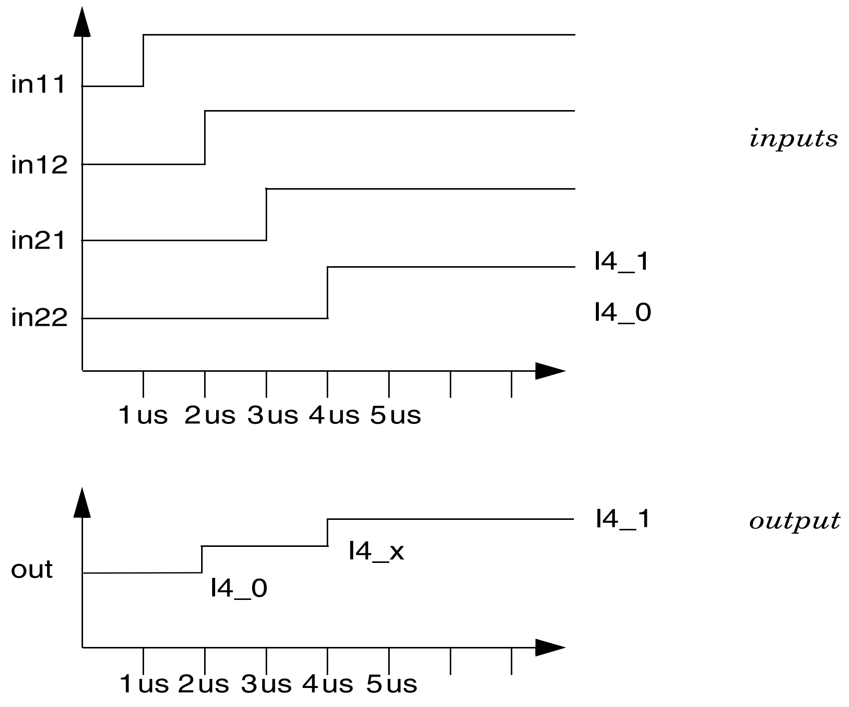
The following netlist example shows two netlist instances of an AND template whose outputs are connected to the same node, as shown in the figure that follows.

```
and.1 in11 in12 out = td=10n
and.2 in21 in22 out = td=10n
when (dc_init) {
  schedule_event(time, in11, l4_0)
  schedule_event(time, in12, l4_0)
  schedule_event(time, in21, l4_0)
  schedule_event(time, in22, l4_0)
}
when (time_init) {
  schedule_event(1u, in11, l4_1)
  schedule_event(2u, in12, l4_1)
  schedule_event(3u, in21, l4_1)
  schedule_event(4u, in22, l4_1)
}
```



Conflict at a digital node

The netlist also uses multiple when statements to generate delayed inputs to the AND gates, as shown in the following figure.



Inputs and output of netlist example

Note:

> Multiple when statements are allowed in a netlist; however, they cannot be nested.

Alternatively, the AND templates could be driven with event-generating templates such as the clock template in the topic titled "Initialization and Internal Events - MAST clock Template". The outputs of the AND gates are tied together to exercise the conflict resolution function.

Note that the when statement can be used in a netlist as well as in a template. The when (dc_init) statement initializes all of the inputs to logic 0 (l4_0), for the DC analysis. The when (time_init) statement schedules l4_1 as events, starting at 1 ms and proceeding through the inputs at 1 ms intervals. These events are scheduled at the beginning of the transient simulation (using time_init), but do not become active until their scheduled times in the transient simulation.

Initialization and Internal Events - MAST clock Template

The following clock template shows the use of internal events and also shows the initialization of states.

```
template clock  ckout  =  freq, duty
  state logic_4 ckout
  number freq=0,   # clock frequency
         duty=0.5  # clock duty cycle (time pulsed/period)
{
  state nu tick    # internal "wake-up" state

  number ton=0,    #clock on-time
         toff=0    #clock off-time

  parameters  {    # calculate off and on time
    if (freq > 0) {
      ton  = duty/freq
      toff = 1/freq - ton
    }
  }
  when (dc_init)  {
    schedule_event(time,ckout,l4_0)
  }
    # start clock ticking after delay time
  when (time_init) {
    if (freq > 0) schedule_event(time,tick,1)
  }
  when (event_on(tick))  {
    if (driven (ckout)==l4_0)  {
        # turn clock on (set to 1)
      if (ton > 0) {
        schedule_event(time,ckout,l4_1)
        schedule_event(time+ton,tick,1)
      }
    }
    else {                # turn clock off (set to 0)
      if (toff > 0) {
        schedule_event(time,ckout,l4_0)
        schedule_event(time+toff,tick,1)
      }
    }
  }
}
```

```
ASCII text of this example is located in:
install_home/example/MASTtemplates/structured/
clock.sin
```

The clock template description is divided into the following topics:

- Header Declarations
- Local Declarations - shows the use of a local variable to serve as an internal counter for generating clock pulses
- When Statements1 - shows the use of multiple when statements in a template, shows the use of digital modeling simvar variables (dc_init, time_init) to initialize a template

Header Declarations

The header and declarations of clock are as follows:

```
template clock  ckout  =  freq, duty
  state logic_4 ckout
  number freq=0,    # clock frequency
         duty=0.5   # clock duty cycle (time pulsed/period)
```

This template has only one connection point (ckout), providing the clock's output. This type of connection point is declared as a logic_4 state. The freq parameter determines clock frequency; the duty parameter determines the fraction of the clock cycle that the clock signal is at logic 1 (l4_1) for each period.

Local Declarations

The first local declaration for this template is the internal variable, tick as follows:

```
state nu tick
```

This local variable is declared as a state with no units (nu). It is not being used to assign a logic state to a connection point (as is the case for out_state in the and template). In this template, tick serves as an internal counter to propagate clock pulses. Thus, there is no need for it to assume a logic_4 value.

The values of ton and toff are based on the values specified for the arguments freq and duty as follows:

```
number ton=0,
       toff=0
```

These values for on time and off time are then used in the when statements to determine the scheduling of each portion of the clock cycle. Note that the when (time_init) statement will not allow template operation unless you specify freq > 0.

Note:

For simplicity, this template does not include error checking for duty > 1 or duty < 0.

When Statements

This template contains the following three when statements:

```
#1----------------------------------------------
when (dc_init)  {
  schedule_event(time,ckout,l4_0)
}
#2----------------------------------------------
  # start clock ticking after delay time
when (time_init) {
  if (freq > 0) schedule_event(time,tick,1)
}
#3----------------------------------------------
when (event_on(tick))  {
  if (driven (ckout)==l4_0)  {
      # turn clock on (set to 1)
    if (ton > 0) {
      schedule_event(time,ckout,l4_1)
      schedule_event(time+ton,tick,1)
    }
  }
  else {                # turn clock off (set to 0)
    if (toff > 0) {
      schedule_event(time,ckout,l4_0)
      schedule_event(time+toff,tick,1)
    }
  }
}
```

Note:

Multiple when statements are allowed in the template, but they cannot be nested.

Initialization

The Mast Reference Manual lists several simvar variables that are used in initializing a digital model before DC, DC Transfer, and Transient analyses. Two of them are used here to schedule an event to override the default initialization value of l4_x:

- The dc_init simvar variable in the first when statement 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 time_init simvar variable in the second when statement 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 three when statements listed above have the following functions:

1. when (dc_init)—DC analysis initialization. This uses the dc_init simvar variable, which initializes the template at the start of a DC operating point analysis. The result is that ckout is assigned an initial value of l4_0 (because using schedule_event puts this assignment in the event queue).
2. when (time_init)—Transient analysis initialization. This uses the time_init simvar variable, which initializes the template at the start of the transient analysis. This enables timing for the clock template so that it can use the simvar time variable to synchronize with simulation timing and the internal counter, tick, to begin generating multiple clock cycles. Again, using schedule_event puts time and tick in the event queue.
3. when (event_on(tick))—Clock pulse propagation. Because ckout has been initialized to l4_0 and tick has been initialized to 1, the transient analysis can begin to generate clock pulses. The first if statement checks the present value of ckout—if it is l4_0 (off), it turns the clock on (ckout==l4_1). Otherwise, it yields to the else statement, which turns the clock off. The event_on function then schedules another event for tick and the process is repeated for the duration of the transient analysis.

8

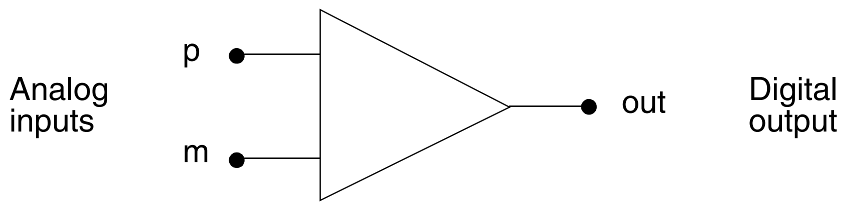
Modeling Mixed Analog-Digital Systems

The following topics combine time-domain analog modeling with digital modeling—mixing analog and digital models in the same system:

- Modeling a Voltage Comparator with MAST -- shows the use of the threshold intrinsic function to detect threshold crossings of analog signals
- Modeling A Digitally-Controlled, Ideal Switch with MAST -- shows how to schedule analog time steps with the schedule_next_time intrinsic function and describes issues about reconsidering the DC analysis
- Using Interface Models in Mixed Analog-Digital Simulation -- describes the use of interface models to connect analog and digital templates
    - A MAST Analog-to-Digital (a2d) Interface Model
    - A MAST Digital-to-Analog (d2a) Interface Model
    - Analog-to-Digital and Digital-to-Analog Summary
- MAST Interface Models and Foreign Simulators -- shows how to use interface models to connect analog and discrete subsystems in mixed-simulator applications

Modeling a Voltage Comparator with MAST

The comparator (shown in the figure below) used in this example has two inputs and one output.



Voltage comparator

This model compares the instantaneous analog voltage at the two input nodes and produces a digital output of logic 1 if the voltage at p is greater than the voltage at m, and logic 0 otherwise. A time delay parameter, td, specifies a delay between the time the input voltage changes polarity to the time the output changes state.

The text for this simple voltage comparator template (comparator) is listed below:

```
element template comparator p m  out = td # header
  electrical p, m                    # pin declarations
  state logic_4 out                  # pin declaration
  number td=0                        # argument declaration
{
  state nu before, after             # local variables
  when (dc_init) {                   # DC initialization
    schedule_event(time, out, l4_0)

  when (threshold(v(p), v(m), before, after)) { # comparison
    if (after >= 0)   schedule_event(time+td, out, l4_1)
    else              schedule_event(time+td, out, l4_0)
  }
}
ASCII text of this example is located in:
install_home/example/MASTtemplates/structured/
comparator.sin
```

comparator Gate Topics

The AND gate template description is divided into the following topic:

- Header Declarations
- Local Declarations
- When Statements - shows how to detect threshold crossings of analog signals with the threshold intrinsic function
- DC Initialization - shows DC analysis considerations, particularly DC initialization
  - Initialization Example

Header Declarations

The header declarations show that this voltage comparator template has two analog inputs and one digital output.

```
electrical p, m
state logic_4 out
```

This portion of the template indicates that it is a mixed analog-digital template, because the input pins are analog (electrical) and the output connection is digital (state logic_4). The input pins are analog and electrical; the digital output is a discontinuous state type. Thus, the output of the comparator template could be used as an input to the and template. The time delay parameter is also similar to the one introduced for the and template.

Local Declarations

The local variables for this template are before and after, declared as follows:

state nu before, after

These variables determine which direction the input voltage is going after crossing the threshold of 0 volts. They are simultaneously state variables and analog variables, as mentioned the topic titled "Time". As analog variables, they are not limited to a discrete set of values—they can take on any real-numbered values. As state variables, they can change their values discontinuously in time. This type of variable is known as event-driven analog (or analog state).

The units of before and after are declared as nu, which stands for no units. In general, the units for analog state variables can be any of those in units.sin (or its equivalent)—the same way a val or var variable can assume these units. The topic titled "Modeling A Digitally-Controlled, Ideal Switch with MAST" shows an example of an ideal switch example that has an analog state variable (res) whose unit type is resistance (r).

The before and after variables must be of type state because their values are set in a when statement. Any variable that is set in a when statement (either by an assignment statement or by schedule_event) must be of type state.

When Statements

In this template, there are two when statements. The first is for DC initialization (described in the topic titled "DC Initialization"); the second detects the crossing of the two input voltages, which is described below.

The following when statement from the comparator template detects the crossing of the two input voltages:

```
when (threshold(v(p), v(m), before, after)) {
```

This when statement means the following:

> When the voltage at p crosses the voltage at m, set the values of before and after to their appropriate values (as described below), and execute the if-else statements that follow.

The threshold intrinsic function is simply stated but powerful—it pinpoints the exact time when two changing quantities become equal and it provides information about the past and the future of those quantities. In this instance, it monitors the voltages at connections p and m, then triggers an event during the simulation at the precise time that the voltages become equal to each other. This functionality is independent of analog time steps. (Refer to the MAST Reference Manual for more information on the threshold function.)

This threshold checking sets the values of before and after according to the following (see the table below):

- If the first argument of the threshold statement, v(p), is crossing the second argument, v(m), from negative to positive, then before is set to -1 and after to +1.
- If the first argument of the threshold statement is crossing the second from positive to negative, then before is set to +1 and after to -1.
- If v(p) and v(m) remain equal after becoming equal, then after is set to 0.

before	after	meaning
-1	0	v(p) rose from below v(m) to equal v(m)
+1	0	v(p) fell from above v(m) to equal v(m)
0	-1	v(p) fell from equal to v(m) to below v(m)
0	+1	v(p) rose from equal to v(m) above v(m)
+1	-1	v(p) fell from above v(m) to below v(m)
-1	+1	v(p) rose from below v(m) to above v(m)
-1	-1	v(p) rose to equal v(m) then fell below v(m)
+1	+1	v(p) fell to equal v(m) then rose above v(m)

Now that there is a way to find the voltage crossing point, completion of the comparator function is simple. Checking the after variable lets you determine whether v(p) is crossing positive or negative. If the crossing is positive, then out, the output pin, is scheduled to be l4_1 (after time delay td). Otherwise, out is scheduled to be l4_0.

Note:

It is possible to implement conflict resolution for this model using the driven function (see the topic titled "MAST Template Conflict Resolution - the driven Function"), which was omitted here for simplicity.

DC Initialization

Using the threshold function, you can implement the comparator model for transient simulation. However, for DC analysis there are further considerations. The purpose of the DC analysis is to determine the quiescent values of all of the system's internal variables and states. The interactions between these quantities can be complex. The algorithm used by the Saber simulator for a DC analysis is specifically designed to manage these complexities.

One of the major problems that arises is that it is not always possible to find a stable solution. A good example of a system that has no single stable operating point is a ring oscillator. When the DC algorithm detects an oscillation, it arbitrarily selects a state at a node and allows the unsatisfied scheduled events to occur during the transient simulation.

To understand these state interactions, it is necessary to look at the DC algorithm as it pertains both to the scheduling and detection of events and to the use of the threshold function. For a detailed description of the DC algorithm, refer to the MAST Reference Manual.

A simplified description of the DC algorithm is the following:

1. The simulator sets all analog system variables to zero, all event-driven analog states to the initial values defined by their initializers, and all digital states to the initial values defined by their unit declarations (e.g., logic_4 defines l4_x). Refer to topic titled "Initializing Connection Points - Digital MAST Modeling".
2. It executes when (dc_init) statements in all templates, propagating all events according to dependencies. It executes all when (dc_start) statements in all templates.

   By default, all uninitialized state logic_4 connection points are set to l4_x.
3. The effects of the analog subsystem on the discrete subsystem are found by observing all threshold conditions, as analog signals go from zero to their final DC values. For any threshold conditions that become satisfied, the statements portion of the corresponding when statement is executed. The simulator again propagates all events within the discrete subsystem according to dependencies.
4. Repeatedly, the simulator finds the solution of the analog subsystem solution and propagates all events within the discrete subsystem, until the system stabilizes.

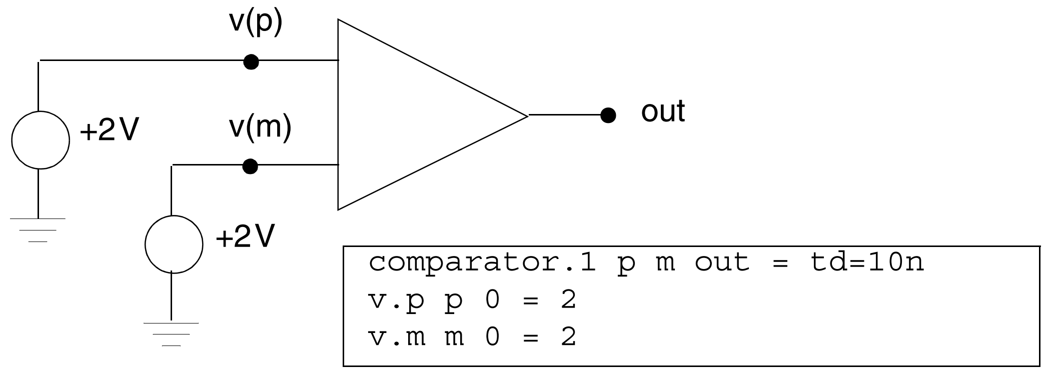
Initialization Example

Consider the DC initialization provided by the first when statement in the comparator template:

```
when (dc_init) {
  schedule_event(time, out, l4_0)
}
```

If this were not included in this template, the output could potentially be incorrect at the end of a DC analysis. Given the circuit shown in the figure below, if v(p) is 2 volts, v(m) is 2 volts, and the previous when (dc_init) statement is omitted, then the DC algorithm would proceed as follows:

1. Start off with 0 volts at p and m and with the state of the output pin out at its initial value of l4_x (logic X—unknown state).
2. Execute all when (dc_init) statements. (Assuming that the when (dc_init) statement is absent, there is no action here—out remains at l4_x.)
3. As v(m) and v(p) increase together from 0 (the initial values) to 2 volts (the source voltage), the threshold is not crossed, and the output remains at l4_x. This is the value at out at the end of the DC analysis, which is erroneous. The expected output would be logic 0 (l4_0), because v(m) is equal to v(p). However, by this algorithm, the output pin out was set to logic X and would remain there.



```
comparator.1 p m out = td=10n
v.p p 0 = 2
v.m m 0 = 2
```

Comparator DC analysis

Adding the DC initialization section shown above alleviates this problem by setting the output pin out to logic 0.

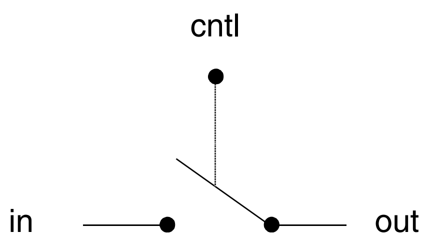
If v(p) is set to a DC value greater than v(m), then the threshold will take care of producing the correct DC value for the output, because the value of v(p) would cross the value of v(m) in step 3 of the algorithm (above).

The following paragraph is a generalization of the threshold function and dc_init statement:

> A when (threshold()) statement should be accompanied by a when (dc_init) statement that establishes the appropriate conditions, assuming that all node voltages and system variables are 0. If the threshold is crossed, then the conditions established by the when (dc_init) statement are overridden and produce the expected result.

Modeling A Digitally-Controlled, Ideal Switch with MAST

This model of an ideal SPST switch demonstrates another aspect of the mixing of analog and digital concepts. As shown in the following figure, a digital event comes to the template through the cntl input pin. If the cntl pin is logic 1, then the switch is closed. If the cntl pin is logic 0, logic X, or logic Z, then the switch is open. The resistance between the electrical pins p and m is equal to roff when the switch is open; the resistance equals ron when the switch is closed. The resistance changes instantaneously when the event occurs on the cntl pin.



Ideal switch

The following sw template describes the switch functionality:

```
template sw p m cntl = ron, roff  # template header
  electrical p, m                 # analog pins
  state logic_4 cntl              # digital connection
  number ron=1, roff=1meg         # parameter declaration
{
  state r res                     # internal state variable

  when (event_on(cntl)) {         # switch control
    if (cntl == l4_1) res = ron
    else              res = roff
    schedule_next_time(time)
    }
  equations {                     # switch analog equations
    i(p->m) += (v(p) - v(m))/res
  }
}
ASCII text of this example is located in:
install_home/example/MASTtemplates/structured/
sw.sin
```

Note:

For simplicity, this template does not include error-checking, specifically for a zero-valued denominator in the template equation. Therefore, the values of ron and roff must be non-zero.

The sw template description is divided into the following topics:

- Header Declarations
- Local Declarations
- When Statement -- shows the difference between using an assignment statement and the schedule_event intrinsic function, to change the value of a state
- Template Equation

- The schedule_next_time Function -- shows how to schedule analog time steps with schedule_next_time; shows the difference between schedule_next_time and the next_time simvar variable; and shows the independence between discrete event time points and analog simulation time steps
- DC Initialization -- shows how to assign a default value to a state (for DC analysis)
- Netlist Example for the Ideal Switch

Header Declarations

The header shows that this ideal switch template has two electrical pins, one digital connection point, and arguments for on and off resistances. These are declared as:

```
electrical p, m
state logic_4 cntl
number ron=1, roff=1meg
```

This is a mixed analog-digital template, because some connection points are analog pins (electrical) and others are digital states (state logic_4). The template arguments, ron and roff, are initialized to default values of 1 ohm and 1 Mohm, respectively. Thus, you do not have to assign values to ron and roff to use this template.

Local Declarations

Local declaration defines the analog state variable res, written as follows:

state r res

The variable named res has the units of resistance, r. This variable contains the present value of resistance, whose value changes discontinuously. It can be assigned the value of a passed-in argument. It cannot be assigned anywhere except in a when statement, but it can be referenced anywhere that state references are permitted. In this example, the value of res is referenced in the template equation.

The variable res must be of type state, because its value is set in a when statement. This is true for any variable whose value is set in a when statement, either by assignment or by a schedule_event function call.

When Statement

A when statement is used to implement the digital portion of the model as follows:

```
when (event_on(cntl)) {
  if (cntl == l4_1) res = ron
  else              res = roff
  schedule_next_time(time)
}
```

The event_on function is used in the when statement to determine at what time the state of the switch changes from open to closed, or vice-versa. When an event occurs at cntl, it is a simple matter to check and see whether it has changed to logic 1 (l4_1) or logic 0 (l4_0). If it is l4_1, then the local state variable for resistance, res, is set to the argument value for the on resistance, ron. Otherwise, the res is set to roff.

There is a difference between assigning a value to a state variable (first statement, below) and scheduling an event on a state variable (second statement, below). The value resulting from the assignment statement is not placed in the event queue.

```
1  if (cntl == l4_1) res = ron
2  if(cntl == l4_1)schedule_event(time, res, ron)
```

Either of these two constructs would work for the sw template. Because there is no need to detect an event (such as a change in the value of res), it is more efficient to use the assignment statement. If, for some reason, you wanted to detect the changing of the resistance value, you would have to use schedule_event. This would allow you to detect the change by using a when statement with the event_on function, as follows:

when (event_on(res)) {...

The general rule for scheduling an event is described in the following paragraph:

> If you need to detect a change (with a when statement and an event_on function), then you must use the schedule_event function to change the value of the state. This is especially true if the change affects a connection point. If you just need to use the value of a state in a another section of the template, use an assignment statement to change its value.

It is essential to understand the significance of the next portion of this particular when statement (as explained in the topic titled "The schedule_next_time Function”):

```
schedule_next_time(time)
```

Template Equation

The template equation for the sw template is straightforward, as shown below (remember that res must be non-zero):

```
equations {           # switch analog equations
  i(p->m) += (v(p) - v(m))/res
}
```

The current from p to m is the result of dividing the voltage across the nodes to which they are connected by res, the present resistance value. This resistance happens to be a state variable whose value is determined by events at the control pin cntl. As a result, the value of res is changing discontinuously in time. This does not generally cause a problem, because schedule_next_time restarts the integration algorithm with each event, using the value of res from the previous event as an initial point. However, it is best practice to provide continuity. Refer to the sw1_l4 template (in the Saber online documentation) for an example of how to provide this type of continuity in a template.

The schedule_next_time Function

Normally, analog time steps are determined by the analog simulation time-step algorithm, which uses variable time steps and an integration algorithm to ensure that the time-steps are as big as possible, without causing too much error in the solution at any given time.

It is important to note the following:

> The analog subsystem time points are completely independent of the discrete subsystem time (event) points. The threshold function lets the analog subsystems influence events in the discrete subsystems. On the other hand, the schedule_next_time(time) function forces an analog time step when a transient simulation reaches time.

Therefore, it is important to include schedule_next_time in the when statement, as follows:

```
when (event_on(cntl)) {
  if (cntl == l4_1) res = ron
  else              res = roff
  schedule_next_time(time)
}
```

The fact that an event occurred on the cntl pin does not mean that there will be an analog time step when res changes value. Instead, you must force an analog time step to occur at that time by scheduling it with schedule_next_time. This lets the new value of res affect the analog network at the right point in time.

Note that there are some subtle usage differences between the schedule_next_time function and the next_time simvar variable. They both have the same objective—to force a time step in the analog simulation, but they apply in different contexts:

- The schedule_next_time function is used in a when statement (generally, when modeling a mixed analog-digital system). It is invoked as a function call. Once scheduled, it stays in effect until either it is descheduled or the time point has been reached.
- The next_time simvar variable is used in an assignment statement when modeling an analog-only system. Its effect expires after each time step, regardless of whether the appropriate time point has been passed.

DC Initialization

This simplified template relies on detecting an event coming into the template during the DC analysis. The value of res is undefined until an event appears on cntl. To ensure that res has a value when the transient analysis begins, you could give it a default value:

```
state r res=roff
```

For simple use, a when (dc_init) statement is generally not required, although you could use one to initialize res. However, if you wanted to use the vary command, you would need to use a when (dc_init|tr_start) statement to initialize the value of res before each transient simulation (recall that the "|" operator means "or"). You would then insert the following statement to do this:

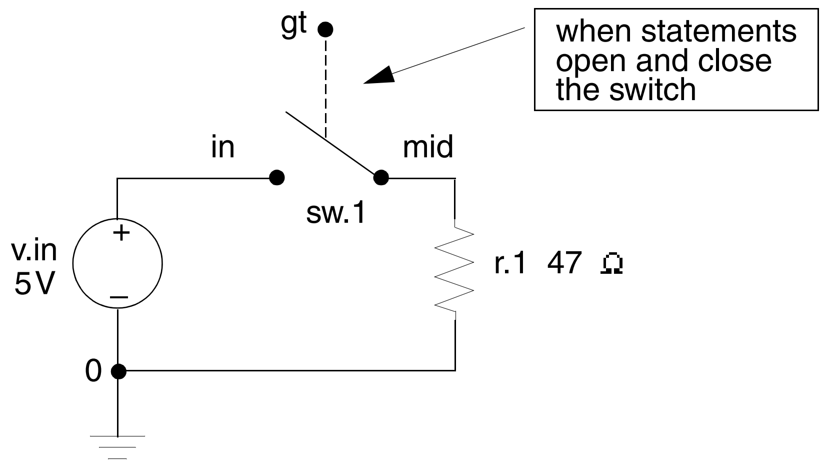
```
when (dc_init|tr_start) {
  res = roff
}
```

Refer to the MAST Reference Manual for more information on using these initialization simvar variables.

Netlist Example for the Ideal Switch

The following netlist example demonstrates the use of the sw template with a voltage source and a resistor as seen in the figure below. The when statements in the netlist provide an appropriate stimulus for controlling the switch at node gt.

```
sw.1 in mid gt
r.1 mid 0 = 47k
v.in in 0 = 5
when (dc_init) {
  schedule_event(time, gt, l4_0)
}
when (time_init) {
  schedule_event(1u, gt, l4_1)
  schedule_event(2u, gt, l4_0)
}
```



Digitally-controlled switch in an analog circuit

Using Interface Models in Mixed Analog-Digital Simulation

Mixed analog-digital simulation generally requires interface models to connect analog pins to digital connection points. Typically, an interface model has one input port and one output port. The objective of the interface model is to convert information from one form on the input port to another form on the output port. That is, it can convert a transition between digital states to a transition between analog voltage levels or vice versa.

A library of advanced interface models, called Hypermodel analog/digital interface templates, has been developed. These Hypermodel templates use the same basic modeling concepts presented in this section, but they incorporate additional features. For more information on Hypermodel, search the Saber online help index.

In many ways, the comparator and switch examples given in this chapter are supersets of what needs to be in an interface model. However, interface models can model very complicated effects. For example, a interface model might be used in the modeling of a TTL (transistor-transistor logic) device in the following way:

Most of the TTL devices could be modeled with strictly digital logic. An interface model could be inserted at any digital connection of the TTL device that connected to analog pins. All of the complex impedance characteristics and fan-out properties of the TTL totem-pole output stage would be present for the analog circuitry, while the convenience and speed of a digital description would be maintained for the greater portion of the model.

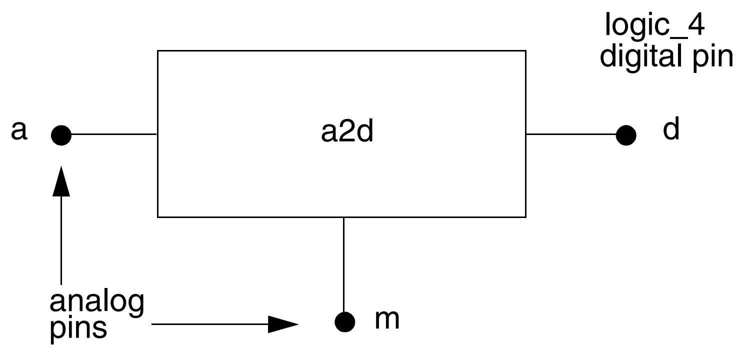
Two very simple analog-to-digital and digital-to-analog interface models are presented in the following examples:

- A MAST Analog-to-Digital (a2d) Interface Model
- A MAST Digital-to-Analog (d2a) Interface Model

A MAST Analog-to-Digital (a2d) Interface Model

This example of an analog-to-digital interface template (a2d) has two analog input pins, a and m, and a digital output pin, d. The digital output is defined to be HIGH (logic 1) when the analog input voltage (between a and m) is above the value specified for input high voltage, ih. The digital output is defined to be LOW (logic 0) when the analog input voltage is below the value specified for input low voltage il. The output state remains at its previous state while the input voltage is between ih and il.

The following figure shows a schematic representation of an analog-to-digital interface model.



Analog-to-digital interface model

The a2d template is as follows:

```
template a2d a m d = td, il, ih
  electrical a, m
  state logic_4 d
  number td=0,                # input/output time delay
         il=0.8,              # input low voltage
         ih=2.4               # input high voltage
{
  state nu before,            # variables for threshold
           after              #  direction

  when (dc_init) {            # dc initialization
    schedule_event(time, d, l4_0)
  }

   # threshold crossing low
  when (threshold(v(a)-v(m), il, before, after)) {
    if ((after<0) & (driven(d)~=l4_0)) {
      schedule_event(time+td, d, l4_0)
    }
  }

   # threshold crossing high
  when (threshold(v(a)-v(m), ih, before, after)) {
    if ((after>0) & (driven(d)~=l4_1)) {
      schedule_event(time+td, d, l4_1)
    }
  }
}
```

ASCII text of this example is located in:
install_home/example/MASTtemplates/structured/
a2d.sin

This a2d example description is divided into the following topics:

- Header Declarations
- Local Declarations
- When Statements
- Template Equations

Header Declarations

The a2d template has an analog input pin a, a digital output pin d, and an analog reference pin m as shown in the following code example. The input parameters are time delay (td) from input threshold crossing to output digital event, input logic low threshold (il), and input logic high threshold (ih). Input parameters are initialized to reasonable values for TTL.

```
template a2d a m d = td, il, ih
  electrical a, m
  state logic_4 d
  number td=0,
         il=0.8,
         ih=2.4
```

Local Declarations

The local states before and after are declared with no units (nu) to provide the variables needed by the threshold function.

```
state nu before,
         after
```

When Statements

The a2d template is very similar to the comparator example from the topic titled "Modeling a Voltage Comparator with MAST" — they both accept an analog input across two input pins, and they both provide a logic_4 value based on that input to a state connection point. The when (dc_init) statement initializes the output to a known value, l4_0, at the beginning of a DC analysis. The other two when statements detect the crossing of the analog input voltage with the specified logic thresholds. They schedule a logic event on the output if the appropriate threshold has been crossed and the output logic level changed accordingly.

```
when (dc_init) {           # dc initialization
  schedule_event(time, d, l4_0)
}

# threshold crossing low
when (threshold(v(a)-v(m), il, before, after)) {
  if ((after<0) & (driven(d)~=l4_0)) {
    schedule_event(time+td, d, l4_0)
  }
}

# threshold crossing high
when (threshold(v(a)-v(m), ih, before, after)) {
  if ((after>0) & (driven(d)~=l4_1)) {
    schedule_event(time+td, d, l4_1)
  }
}
```

Even though it may be tempting to put the following statement inside the when (threshold) statements above, you should resist this temptation:

```
schedule_next_time(time)
```

The implementation of the threshold function guarantees that an event will occur at the point of the threshold crossing. Because when (threshold) is independent of the time step algorithm, it is generally not necessary to use the schedule_next_time statement with it.

Template Equations

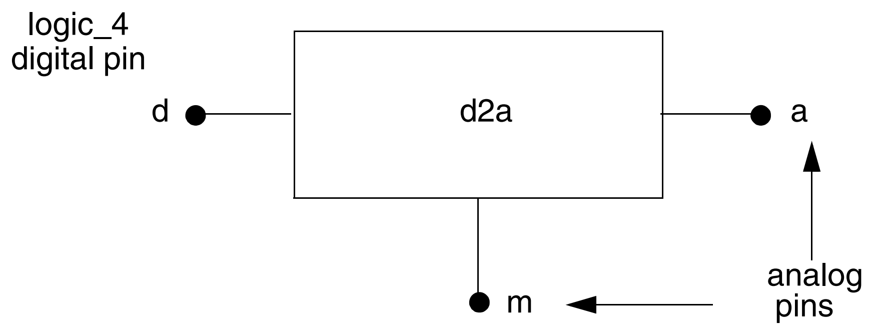
Although the a2d template has analog pins, it does not contain template equations. When there are no equations in a template with analog pins, the current contribution to each analog pin is assumed to be 0. In this example, this corresponds to infinite input impedance for the analog pins.

In other words, the digital output of the a2d template is not affected with input current—it is only concerned with input voltage levels that determine a logical output level.

The analog-to-digital interface template (a2d) has two analog input pins, a and m, and a digital output pin, d. The digital output is defined to be HIGH (logic 1) when the analog input voltage (between a and m) is above the value specified for input high voltage, ih. The digital output is defined to be LOW (logic 0) when the analog input voltage is below the value specified for input low voltage il. The output state remains at its previous state while the input voltage is between ih and il.

A MAST Digital-to-Analog (d2a) Interface Model

The following digital-to-analog interface template (d2a) has a digital input pin, d, and two analog output pins, a and m. When the state of the digital input pin changes, the analog output voltage (between a and m) changes to a corresponding value, as determined by the template input parameters.



Digital-to-analog interface model

When the input logic level is l4_1, the output voltage assumes the value specified for output high voltage, oh. When the input logic level is l4_0, the output assumes the value specified for output low voltage, ol. The delay from input logic event to output voltage change is specified by td. The initial output voltage is set to ol. Inputs of l4_x and l4_z are ignored.

Because the analog output changes only in response to the event-driven digital input, this interface model produces discontinuous analog quantities.

Be aware that this template is intended only to demonstrate certain MAST features. The library d2a Hypermodel templates (e.g., id_d2a) include a mechanism to provide a continuous analog output. However, including that level of detail here would obscure the more basic principles being illustrated.

The MAST template for this interface model (d2a) is listed below.

```
element template d2a d a m = td, ol, oh  # template header
  electrical a, m
  state logic_4 d
  number td=0,             # input to output time delay
    ol=0.5,                # output logic low voltage level
    oh=4.0                 # output logic high voltage level
{
  var i i                  # unknown branch current
  state v vout=ol          # output voltage
   # process input events
  when (event_on(d)) {
    if (d==l4_0) {                     # input low
    schedule_event(time+td, vout, ol) # change vout
    schedule_next_time(time+td)       # force analog step
    }
    else if (d==l4_1) {                 # input high
      schedule_event(time+td, vout, oh) # change vout
      schedule_next_time(time+td)       # force analog step
    }
  }
  equations {
    i(a->m) += i            # analog branch current
    i: v(a)-v(m) = vout      # equation for branch current
  }
}
ASCII text of this example is located in:
install_home/example/MASTtemplates/structured/
d2a.sin
```

The d2a template description is divided into the following topics:

- Header Declarations
- Local Declarations
- When Statement
- Template Equation

Header Declarations

The d2a template has a digital input pin, d, an analog output pin, a, and an analog reference pin, m. The input parameters are td, the time delay from input event to output voltage; ol, the output logic low level; and ol, the output logic high level. Input parameters are initialized to reasonable values.

```
element template d2a d a m = td, ol, oh  # template header
  electrical a, m
  state logic_4 d
  number td=0,            # input to output time delay
    ol=0.5,               # output logic low voltage level
    oh=4.0                # output logic high voltage level
```

Local Declarations

Local declarations include the variable i for branch current and the state variable, vout, which receives ol as its initial value.

```
var i i                    # unknown branch current
state v vout=ol            # output voltage
```

When Statement

The when statement evaluates the digital input events and performs two tasks:

1. Schedule a change in the output voltage after time delay td.
2. Schedule an analog time step at the exact time that the output voltage is to change.

The statement is as follows:

```
 # process input events
when (event_on(d)) {
  if (d==l4_0) {                        # input low
  schedule_event(time+td, vout, ol)     # change vout
  schedule_next_time(time+td)           # force analog step
  }
  else if (d==l4_1) {                   # input high
    schedule_event(time+td, vout, oh) # change vout
    schedule_next_time(time+td)       # force analog step
  }
}
```

It would be tempting to use an assignment statement to set the value of vout, as follows (note the two commented lines):

```
when (event_on(d)) {
  if (d==l4_0) {
    vout = ol    # vout assigned instead of scheduled
    schedule_next_time(time+td)
  }
  else if (d==l4_1) {
    vout = oh    # vout assigned instead of scheduled
    schedule_next_time(time+td)
  }
}
```

This would almost work, but the simulator would force the output branch voltage, v(a) - v(m), to become equal to vout at the very next analog time step. Although there will be a time step at time+td, there is no guarantee that some other template won't schedule a time step between time and time+td. Therefore, the output voltage might change before the time time+td. The following statement prevents this problem, so that the value of vout doesn't change until you want it to.

```
  schedule_event(time+td,vout,ol)
```

Template Equation

The template equation of the d2a template is very similar to that of a simple voltage source. In fact, this d2a template is nothing more than an event-driven voltage source.

```
equations {
  i(a->m) += i          # analog branch current
  i: v(a)-v(m) = vout   # equation for branch current
}
```

The vout variable is scheduled to have the discrete values ol and oh whenever the state of the input pin changes. This does not generally cause a problem because schedule_next_time restarts the integration algorithm each time it is used, using the values from the previous time point as an initial point. However, some problems may result from the discontinuous change. Refer to the id_d2a template for an example of how to provide this type of continuity in a template.

Analog-to-Digital and Digital-to-Analog Summary

The important points illustrated in the a2d and d2a interface model examples are as follows:

1. Digital information is transferred to analog using a combination of when, event_on, and schedule_next_time:

```
when (event_on (state)) {
  #set some values
}
schedule_next_time (time)
```

   Note that the analog output resulting from this approach is discontinuous in time—resolving this requires a more advanced technique (as implemented in the id_d2a ideal Hypermodel template).

2. Analog information is transferred to a digital value with the combination of when, threshold, and schedule_event:

```
when (threshold(input, condition, before, after)) {
  #set some values
}
schedule_event(time, state, level)
```

MAST Interface Models and Foreign Simulators

The Saber simulator simulates both analog systems and discrete systems, as well as mixed-signal systems. However, it is sometimes desirable to simulate the digital portion of the circuit on another simulator, which can be any one of several commercial or proprietary simulators. It is also possible to use two versions of the Saber simulator as partner simulators—one for simulating the analog portion of the design and one for simulating the digital portion.

The primary way to connect different portions of the design to be simulated by two separate simulators is with interface models. However, the a2d and d2a interface models will not work for a mixed-simulator application. One minor modification is needed—you must remove the digital pin from the header and declare it as an internal foreign state.

For example, the a2d_1 template (below) is created by modifying the a2d template.

```
template a2d_1 a m = td, il, ih
  electrical a, m
  number td=0,
         il=0.8,
         ih=2.4
{
 #***********************
  foreign state logic_4 d  #dig. pin declared as foreign state
 #***********************
  state nu before,
           after
  when (dc_init) {
    schedule_event(time, d, l4_0)
  }
  when (threshold(v(a)-v(m), il, before, after)) {
    if ((after<0) & (driven(d)~=l4_0)) {
    schedule_event(time+td, d, l4_0)
    }
  }
  when (threshold(v(a)-v(m), ih, before, after)) {
    if ((after>0) & (driven(d)~=l4_1)) {
    schedule_event(time+td, d, l4_1)
    }
  }
}
```

ASCII text of this example is located in:
install_home/example/MASTtemplates/structured/
a2d_1.sin

Pin d has been removed from the header and the header declaration state logic_4 d has been changed to the local declaration foreign state logic_4 d.

This minor change modifies the template to work with a foreign simulator. See the appropriate mixed-simulator product manuals for more information on simulation with such models. These manuals are provided by the respective partner simulator manufacturer.

9

Control System Modeling

A good modeling language lets modelers express their problems naturally. Previous chapters demonstrate the through and across variable approach for analog networks and the event-driven approach for digital networks. This topic introduces another approach: connecting functional blocks by using var and ref connections. This approach assumes that each functional block has distinct inputs and outputs and that the outputs are a function of the inputs. Because the inputs affect the outputs, but not vice versa, there is a notion of direction to the flow of information through the functional blocks.

For example, consider a functional block that takes two inputs and produces the sum of the two inputs as its output. The flow of information is from the summing inputs to the output. Because there is no physical quantity to be conserved, it isn't necessary to create an equivalent circuit network to model this block. The techniques described in this topic show how to model it directly.

In control system modeling, much of the information flows through functional blocks. Consequently, this topic uses control system examples to illustrate the concepts pertaining to information flow through functional blocks. However, the concepts apply to other fields as well.

Control system models are sometimes referred to as signal flow or data flow models.

The following topics introduce the modeling method of connecting functional blocks by using var and ref connections:

- Connection Points for Control System MAST Templates
- Creating Basic Control System MAST Templates
- S-Domain Modeling Using the MAST d_by_dt Operator
- Ideal Delay
- MAST Control System Example Results

The control system modeling topics show how to model control system blocks using the following:

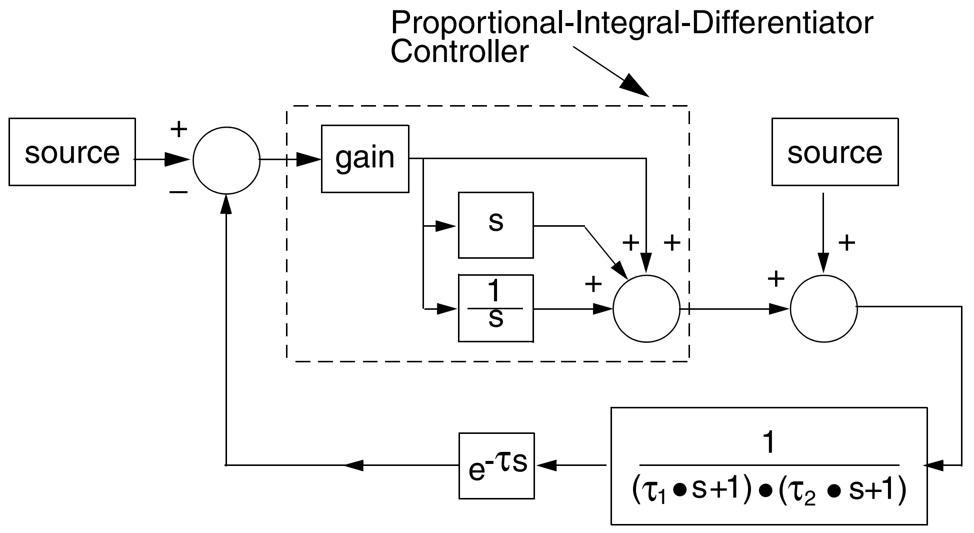
- input and output connection points (which are similar to ref variable and var variable connections)
- s-domain modeling using the d_by_dt operator (including integration with initial conditions)
- Ideal delay modeling

Connection Points for Control System MAST Templates

Many of the other examples used in this manual illustrate the use of connection points defined by through and across variables, principally electrical pins. A var variable is a system variable for which the simulator must solve; a ref variable is a reference to a variable declared as a var variable in another template.

This topic explains the advantages of using var variables and ref variables for describing the flow of information among various functional blocks. In modeling control system blocks, the output (a var variable) of one template is the input (a ref variable) to a second template. As a convenience, the MAST language provides two connection points that provide the same functionality as the var variable and the ref variable, called output and input, respectively. They are declared as connection points in the same way as var variables and ref variables are declared. Refer to the MAST Reference Manual for more information on ref variables and var variables as connection points.

Control system concepts are illustrated by developing element templates and using hierarchy to combine them into the system model shown in the following figure. The input to such a system comes from a “source” template, which is similar to voltage source templates described in previous chapters, but which has only a single connection, declared as output. The other templates are developed using output and input connection points, s-domain modeling, and ideal delay modeling.


Proportional-Integral-Differentiator Controller
source
+
–
gain
s
$\frac{1}{s}$
+
+
+
source
+
+
$\frac{1}{(\tau_1 \bullet s+1) \bullet (\tau_2 \bullet s+1)}$
$e^{-\tau s}$

Control system diagram

Creating Basic Control System MAST Templates

This topic describes the basic approach for writing control system templates using the following examples:

- A simple source (dcsrc)—although not used in the example circuit shown in the previous Control System diagram, this example illustrates the similarity to the simple voltage source.
- A full-featured source (multisrc)—this example extends the technique introduced for dcsrc to a control system version of the multiple-output voltage source.
- A two-input summer (sum2)—this example shows how to write a control system template whose output is the sum of its two inputs.
- A constant multiplier (mply)—this example shows how to write a control system template whose output is the product of a single input times a real-valued argument.

Simple DC Source

The DC source template (dcsrc) listed below is the simplest example of a control systems template, because it has no inputs and only one output. This template has one connection point, p, which is declared as an output.

```
element template dcsrc p = vs
  output nu p
  number vs
{
  equations {
    p: p = vs
  }
}
ASCII text of this example is located in:
saber_home/example/MASTtemplates/structured/dcsrc.sin
```

Compare this template to the simple voltage source template (vsource).

```
template vsource p m = vs
  electrical p,m
  number vs
{
  var i ivs
  equations {
    i(p->m) += ivs
    ivs: v(p) - v(m) = vs
  }
}
ASCII text of this example is located in:
saber_home/example/MASTtemplates/structured/
vsource.sin
```

Note the following differences between these two templates:

1. The dcsrc template is an element template, while vsource is not. One of the advantages of an element template is that any var variable that it contains becomes available for other templates.
2. The dcsrc template has only one pin, which is unitless and of type output, whereas the vsource template has two pins of type electrical. The vsource template needs at least two pins because it uses an across variable (voltage), which requires a circuit reference node. The dcsrc template needs only one pin because it is a simple number source with no units (nu).
3. The vsource template models the conservation of electrical quantities (current and voltage), and declares its pins (p, m) to be a branch for these quantities. The dcsrc template does not model conservation of any physical quantity; its output is a unitless value that requires no “path” to flow through or to be measured across.
4. In the vsource template, the through and across variables (current and voltage) at the output pins are referenced in the equations section as follows:

```
i(p->m) += ivs
ivs: v(p) - v(m) = vs
```

In the dcsrc template, the value at p is provided as a var variable with no units and can be referenced directly in the template equation. (The value of a var variable is always referenced directly.) It can also be used as a connection to other models.

Full-Featured Source - MAST multisrc Template

The multisrc template listed below is a control systems source template that uses the same basic principles as the dcsrc template shown previously. The only difference is that it is capable of producing transient and AC (small-signal) outputs, in addition to DC outputs. The transient capability includes sinusoidal, exponential, and step waveforms.

The principal difference between this source and those voltage sources is that multisrc has only one connection point (out), which is declared as an output with no units (nu). This provides the value at out as a var variable.

```
element template multisrc out = supply, tran, ac
  output nu out
  number supply = 0
  union {
    number                          off
    struc {number vo, va, f, td;}   sin
    struc {number v1,v2,tau;}       exp
    struc {number v1,v2,tstep,tr;}  step
  } tran = (off=1)
  struc {
    number mag=0,
           phase=0
  } ac = ()
```

```
{                         # Start template body
  number pi = 3.14159
  val nu vs
  number td,vo,va,w,ss,v1,v2,tau,tstep,tr,slew
    # define intermediate values,
    # depending on selected output
  parameters {
    if (union_type (tran,sin)) {
      td = tran->sin->td
      vo = tran->sin->vo
      va = tran->sin->va
      w  = 2*pi*tran->sin->f
      ss = 0.05/tran->sin->f
    }
    else if (union_type (tran,exp)) {
      v1  = tran->exp->v1
      v2  = tran->exp->v2
      tau = tran->exp->tau
    }
    else if (union_type (tran,step)) {
      tstep = tran->step->tstep
      v1    = tran->step->v1
      v2    = tran->step->v2
      tr    = tran->step->tr
      slew  = (v2-v1)/tr
    }
  }                         # End parameters section
    # determine vs, which is set
    # equal to vn in template equation
  values {
    if (dc_domain|time_domain) {
      if (union_type (tran,sin)) {
        if (time <= td) {
          vs = vo
          next_time = td
        }
        else {               # if (time > td)
          vs = vo + va*sin(w*(time-td))
          step_size = ss
        }
      }                      # end tran->sin
      else if (union_type (tran,exp)) {
        vs = v1 + (v2-v1)*(1-exp(-(time/tau)))
      }                        # end tran->exp
      else if (union_type (tran,step)) {
        if (dc_domain|(time < tstep)) {
          vs = v1
          next_time = tstep
        }
        else if ((time >= tstep) & (time < tstep+tr)){
          vs = v1 + (time-tstep)*slew
          next_time = tstep + tr
        }
        else {
          vs = v2
        }
      }                      # end tran->step
      else vs = supply
    }             # end dc_domain|time_domain
```

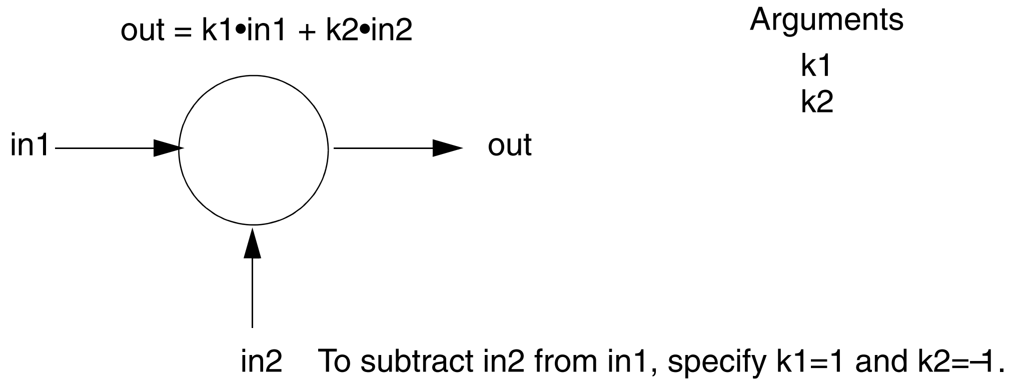
```
    else if (freq_mag) {     # begin small-signal (ac)
      vs = ac->mag           # use magnitude value
    }
    else if (freq_phase) {
      vs = ac->phase         # use phase value
    }                        # end small-signal
  }                          # end values section
  equations {
    out : out = vs
  }
}
ASCII text of this example is located in:
saber_home/example/MASTtemplates/structured/
multisrc.sin
```

Two-input Summer - MAST sum2 Template

This topic shows how to create a template that can take input stimuli, operate on them, and produce an output. The following example is a two-input summer, called sum2.

```
element template sum2 in1 in2 out = k1, k2
  input  nu in1, in2
  output nu out
  number k1=1, k2=1
{
  equations {
    out: out = k1*in1 + k2*in2
  }
}
ASCII text of this example is located in:
saber_home/example/MASTtemplates/structured/
sum2.sin
```

The schematic symbol and functionality are shown in the following figure.



Two-input summer

Note that the two inputs (in1 and in2) are declared as input connections with no units (nu). The output (out) is declared as an output connection with no units.

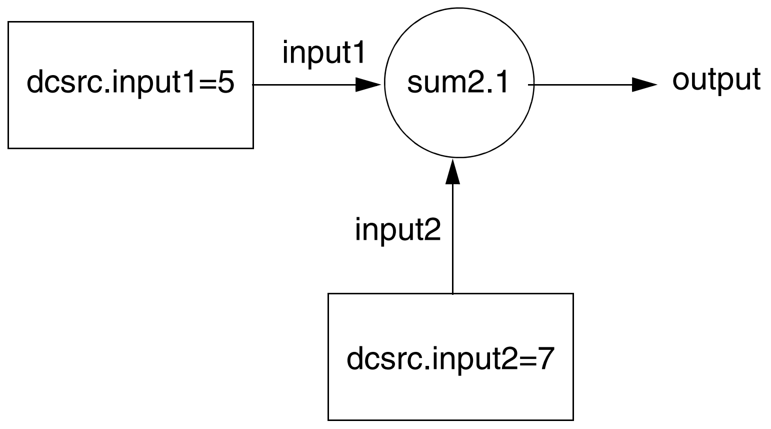
The output is defined as the sum of products, each the product of an input and its corresponding constant. Typically, k1 and k2 are either 1 or -1, depending on the desired function. If k1 is 1 then in1 is added, whereas if k1 is -1, then in1 is subtracted. A similar statement applies to k2 and in2. In general, k1 and k2 can be any desired real numbers.

Example

The following netlist shows how two instances of a source template (dcsrc) can be used as inputs to the sum2 template.

```
dcsrc.input1 input1 = vs = 5
dcsrc.input2 input2 = vs = 7
sum2.in input1 input2 output
```

The schematic for this system is shown in the figure below.



Connection diagram for sum2 and dcsrc templates

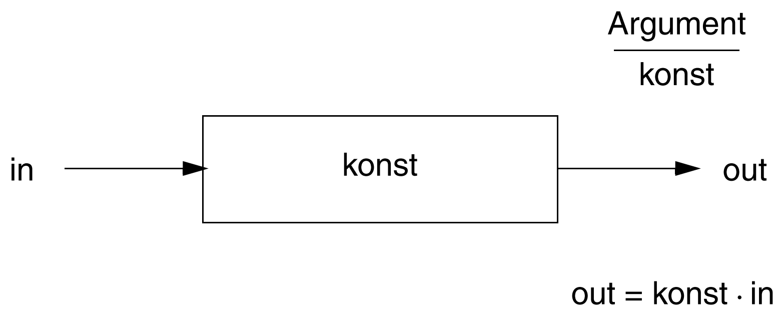
The completion of a DC or transient analysis using this netlist would produce the following node value results:

```
input1  5
input2  7
output 12
```

Multiplier - MAST mply Template

A multiplier template is also needed to complete the example shown in the figure below. The mply template shown below implements a constant gain function. The input to the template is declared as an input connection, and the output is declared as an output connection. The output (out) is defined such that out equals the product of the input (in) and the parameter constant (konst).

```
element template mply in out = konst
  input  nu in
  output nu out
  number konst
{
  equations {
    out: out = konst*in
  }
}
ASCII text of this example is located in:
saber_home/example/MASTtemplates/structured/
mply.sin
```



Constant gain

S-Domain Modeling Using the MAST d_by_dt Operator

In control systems modeling, the system description is often available in s-domain form. The control system diagram above shows several blocks where the block transfer function is defined as an s-domain function. The letter “s” is standard notation for the complex number quantity s+jw, where s is the real part of the quantity and jw is the imaginary part (j is the square root of -1 and w is frequency in radians/second). The “s” quantity is used when solving time-domain differential equations using Laplace transforms. The Laplace transforms and inverse Laplace transforms make it possible to perform calculations in either the time-domain or the s-domain.

The advantage of using the s-domain is that complicated operations in the time-domain are made easier in the s-domain. For example, differentiation in the time-domain is multiplication by s in the s-domain. Similarly, integration in the time-domain is simply division by s in the s-domain. Initial conditions must be established for integration.

The MAST modeling language does not directly implement s-domain modeling. However, a simple rule is given as follows for converting s-domain expressions into time-domain expressions and simplifying the operation:

> Whenever a quantity in the s-domain expression is multiplied by s, substitute the multiplication operation with a differentiation operation, using the MAST differentiation operator, d_by_dt.

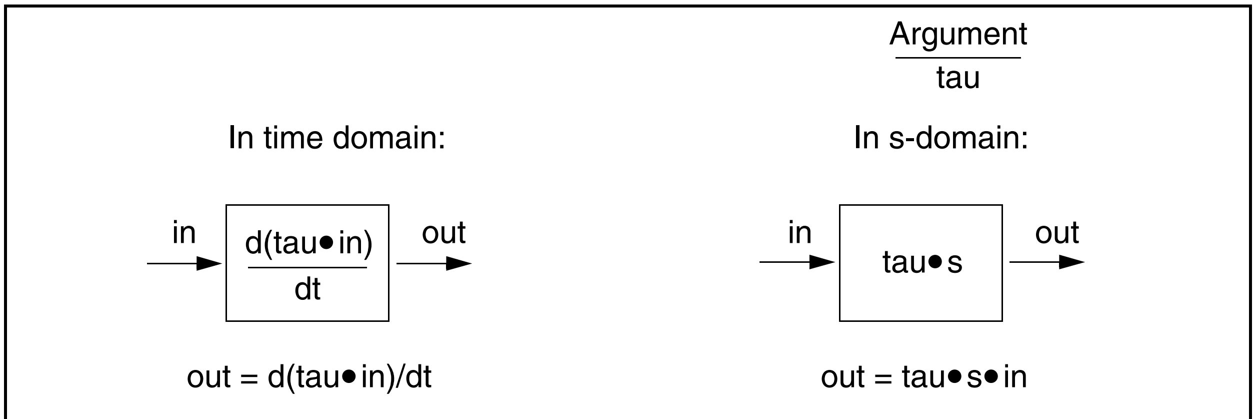
This process is illustrated using the following examples:

- A differentiator (using multiplication by s)
- An integrator (using division by s)
- A two-pole transfer function (using an s-domain polynomial in the denominator)

Note that integration can be performed without the explicit need for an integration operator—the differentiation operator (d_by_dt) is sufficient.

Differentiator

The following figure shows a diagram for a differentiator. This illustrates the equivalence of multiplication by s in the s-domain to differentiation in the time domain. The output is the time derivative of the input multiplied by a constant (parameter tau).



Differentiator model

The MAST template for this differentiator (deriv) is shown as follows:

```
element template deriv in out = tau
  input  nu in
  output nu out
  number tau
{
  equations {
    out: out = d_by_dt(tau*in)
  }
}
ASCII text of this example is located in:
saber_home/example/MASTtemplates/structured/
deriv.sin
```

Note that the constant multiplier, tau, must be inside the d_by_dt operation in the template equation. The simulator finds the value of out, such that out is equal to the time derivative of the product of tau and in.

Because the tau constant multiplier must be inside the d_by_dt operation, the following line:

out: out = tau*d_by_dt(in)

will produce a MAST syntax error. The MAST language requires that all multiplication take place inside the d_by_dt operator.

Integrator

Integration in the time domain is identical to division by s in the s-domain, which could be expressed as follows:

```
out = in/(tau*s)
```

To implement this as a valid template equation, you need to perform the following steps:

1. Revise the statement so that it has no denominator. To accomplish this, simply multiply both sides by tau*s, which yields the following:

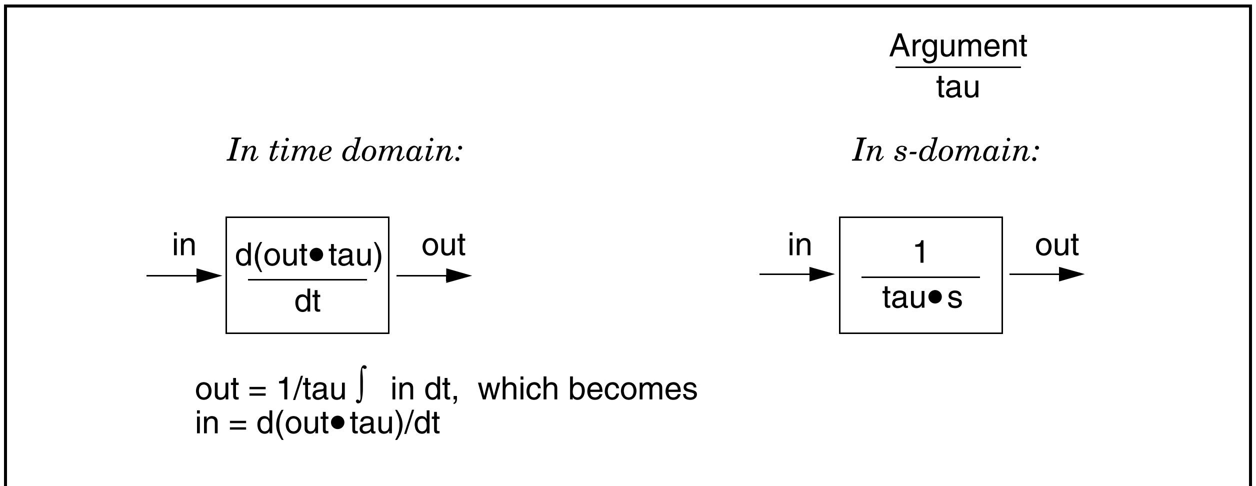
```
out: in = tau*s*out
```

2. Replace the s operator with the MAST d_by_dt operator, and move tau inside the d_by_dt operator. This sets the time derivative of the output multiplied by a constant (tau, the template argument) equal to the input as follows:

```
  out: in = d_by_dt(tau*out)
```

The MAST template for an integrator (intgr) is listed below and illustrated in the figure below.

```
element template intgr in out = tau
  input  nu in           # template input
  output nu out          # template output
  number tau             # constant multiplier
{
  equations {
    out: in = d_by_dt(tau*out)
  }
}
```



Integrator model

This integrator template operates as expected for transient and small signal AC analyses. However, it produces a singularity in the DC analysis. This is because the equation, expressed in integral form, is as follows:

```
out = 1/tau* Ú(in dt) + C
```

where C is a constant of integration. The MAST implementation effectively differentiates both sides of the integral equation. The constant of integration is therefore lost (the derivative of a constant is zero).

One way to eliminate the constant of integration is to change the integral from an indefinite integral to a definite integral, by specifying the initial condition for the out variable. The following template, intgr_1, shows this modification to intgr that includes the initial value (init) for the output.

```
element template intgr_1 in out = tau, init
  input  nu in
  output nu out
  number tau, init=undef
{
  control_section{
    initial_condition(out,init)
  }
  equations {
    out: in = d_by_dt(tau*out)
  }
}
ASCII text of this example is located in:
saber_home/example/MASTtemplates/structured/
intgr_1.sin
```

By definition, the DC solution to a system is the solution to the system when all time derivatives are set to 0. DC initialization is taken care of by using an initial_condition statement. Here, this is implemented in two steps:

1. Declare an argument (init) that allows the user to specify a value to be assigned to out at DC. Here it appears on the same line as tau and has been given a default of undef.

```
number tau, init=undef
```

2. Create a control section and insert an initial_condition statement within it:

```
control_section{
  initial_condition(out,init)
}
```

This technique allows you to specify the init parameter of the intgr_1 template as the output of the template during the DC analysis. If the analysis is in the time domain or frequency domain, the template equation (which is identical to the original template equation for intgr) is then in effect:

```
equations {
  out: in = d_by_dt(tau*out)
}
```

Two-Pole Transfer Function

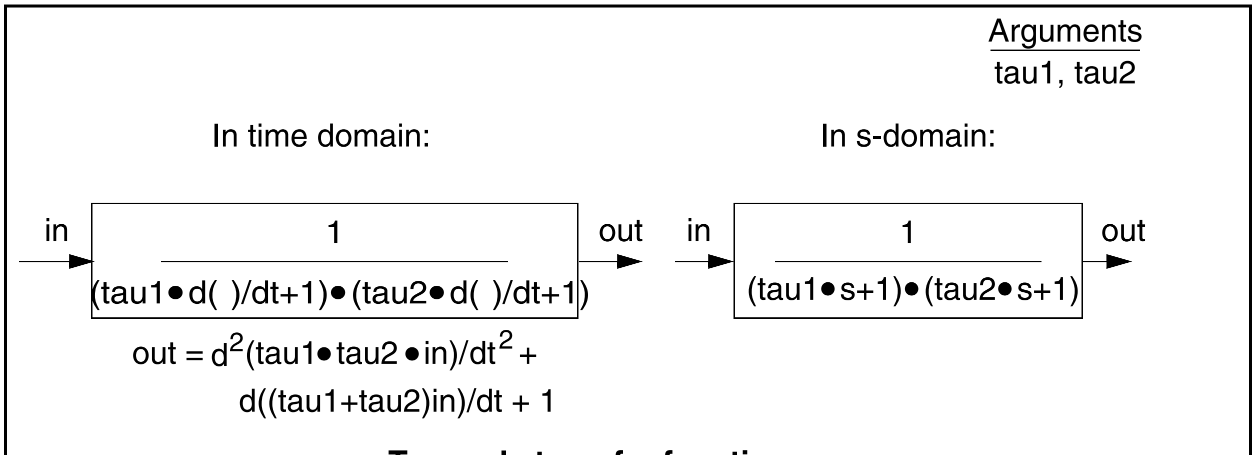
This example of a two-pole transfer function demonstrates how s-domain transfer functions that are of order greater than one are implemented in the MAST modeling language. The following figure compares the time domain and s-domain representations of this function.

In the s-domain, the two-pole transfer function (with poles represented by tau1 and tau2) is expressed as:

```
out/in = 1/((tau1*s +1)*(tau2*s + 1))
```

Isolating the output term on the left side, this transfer function can be expressed as follows:

```
out = 1/((tau1*s +1)*(tau2*s + 1)) * in
```



Two-pole transfer function

When you have a transfer function expressed in this general form:

$$output = \frac{numerator}{denominator} \bullet input\_expression$$

you can use the MAST transfer_function operator to implement it as a template equation, without having to rewrite each occurrence of s in terms of the d_by_dt operator.

The general form for using the transfer_function operator is as follows:

```
output = transfer_function(input_expression, \
   [numerator], [denominator])
```

where:

output	is the output variable (output, branch, or var)
input_expression	is an expression containing the input variable (input, branch, val, or ref)
numerator	a fixed-length array of coefficients for the numerator polynomial (highest order coefficients listed first)
denominator	a fixed-length array of coefficients for the denominator polynomial (highest order coefficients listed first)

Thus, you would expand the denominator of the following two-pole transfer function:

```
out = 1/((tau1*s +1)*(tau2*s + 1)) * in
```

to the following:

```
tau1*tau2*s**2 + (tau1+tau2)*s + 1
```

You can then take these denominator coefficients (tau1*tau2, tau1+tau2, and 1) and use them with the transfer_function operator to implement the following template equation (the numerator consists only of the constant, 1):

```
  equations {
    out: out = transfer_function(in, [1], \
                [tau1*tau2, tau1+tau2, 1])
  }
```

The template for this two-pole transfer function is shown as follows:

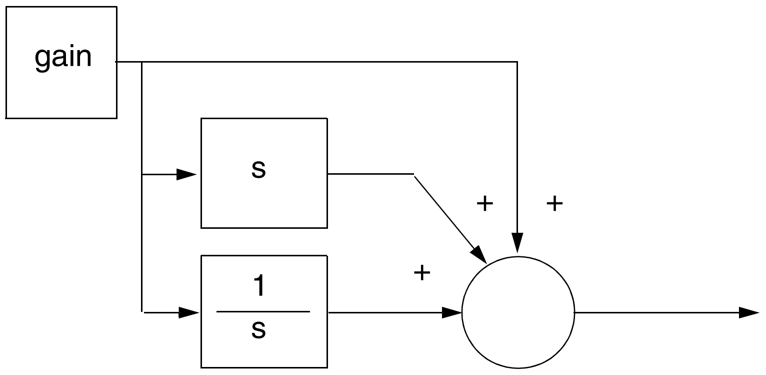
```
element template twopole in out = tau1, tau2
  input  nu in
  output nu out
  number tau1,  # time constant of first pole
         tau2   # time constant of second pole
{
  equations {
    out: out = transfer_function(in, [1], \
                [tau1*tau2, tau1+tau2, 1])
  }
}
ASCII text of this example is located in:
saber_home/example/MASTtemplates/structured/
twopole.sin
```

For contrast, the following example shows how you might have written this template if the transfer_function had not been available. You would rewrite each occurrence of s in terms of the d_by_dt operator as follows:

```
element template twopole_1 in out = tau1, tau2
  input  nu in        # template input
  output nu out       # template output
  number tau1,        # first pole time constant
         tau2         # second pole time constant
{
  var nu doutdt    # For finding first derivative
  equations {
    doutdt: doutdt = d_by_dt(out)
    out:  in = d_by_dt(tau1*tau2*doutdt) + \
          (tau1+tau2)*doutdt + out
  }
}
```

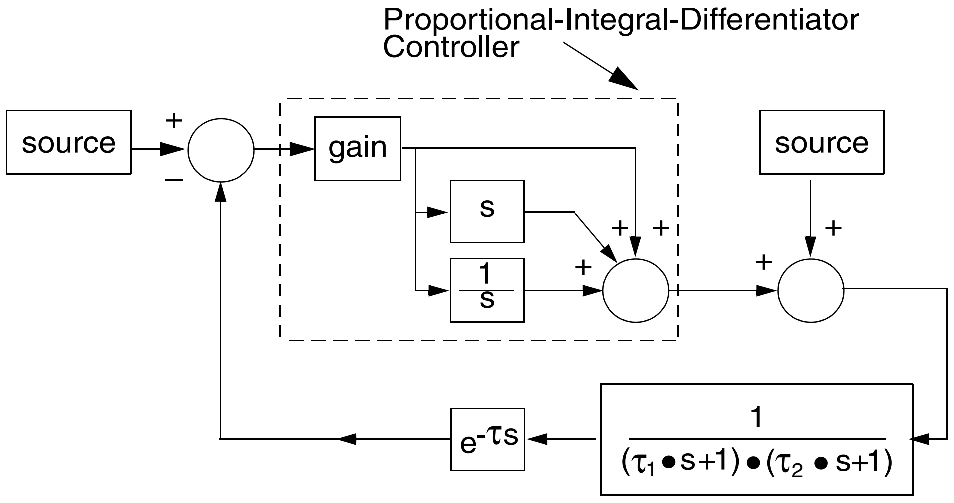
Combining Elements - MAST pid Template

Because the gain, derivative, and integral s-domain templates were declared as element templates, you can take advantage of MAST's hierarchical capabilities to use them as "building-blocks" to form the Proportional-Integral-Derivative (PID) model shown in the following figure:



Hierarchically constructed PID

The PID model functions as the controlling module of the system as shown in the following control system diagram figure:


Proportional-Integral-Differentiator Controller
source
+
–
gain
s
1
s
+
+
+
source
+
+
$e^{-\tau s}$
$\frac{1}{(\tau_1 \bullet s+1) \bullet (\tau_2 \bullet s+1)}$

Control system diagram

Note that the following PID template includes an embedded netlist along with parameters that are passed to the embedded templates. It also contains a template equation.

```
template pid in out = konst,dtau,itau,iinit
  input nu in
  output nu out
  number konst,dtau,itau,iinit=0
{
  mply.1   in in1    = konst
  deriv.1 in1 dout   = dtau
  intgr_1.1 in1 iout = itau,iinit
  equations {
    out: out = in1 + dout + iout
  }
}
ASCII text of this example is located in:
saber_home/example/MASTtemplates/structured/
pid.sin
```

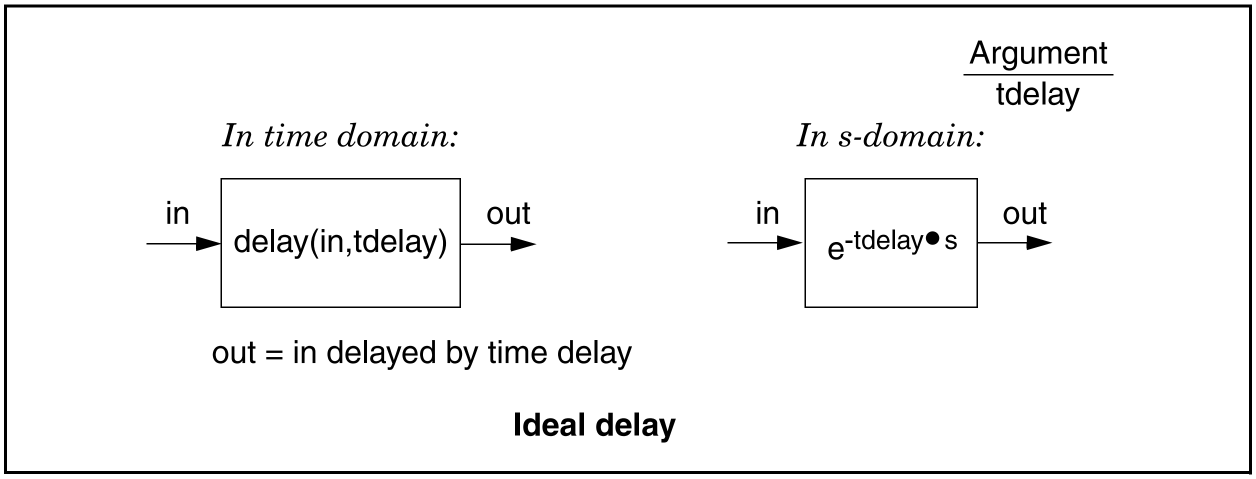
Ideal Delay

This topic describes the modeling of an ideal control system delay in the s-domain. The intrinsic delay function is used here.

The s-domain transfer function for an ideal delay is:

out/in = exp(-s*tdelay)

where the output is identical to the input, but delayed by an amount of time represented as tdelay. This function is implemented using the delay function, rather than replacing the quantity s with the operator d_by_dt, shown in the following figure.



The delay function makes it a simple matter to write a template for an ideal delay (dlay).

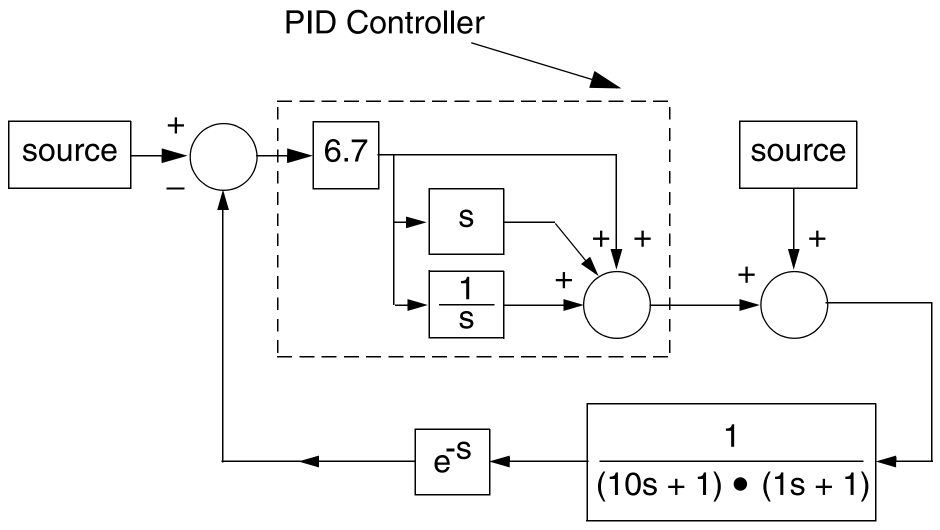
```
template dlay in out = tdelay
  input nu in    # template input
  output nu out  # template output
  number tdelay  # delay from input to output
{
  equations {
    out: out = delay(in,tdelay)
  }
}
```

```
ASCII text of this example is located in:
saber_home/example/MASTtemplates/structured/
dlay.sin
```

The quantity s can be replaced by the d_by_dt operator only in expressions involving linear combinations of s, which is not the case above.

MAST Control System Example Results

The templates described in this chapter provide all the pieces for the control system example shown in the following control system diagram figure. The figure adds numbers for the appropriate constants. Note that the PID model functions as the controlling module for the system.



System with PID controller

The following netlist (testpid.sin) relates to the system shown in the preceding figure:

```
multisrc.set     set         = 3, ac=(1,0)
multisrc.load    load        = tran=(step=(3,5,1,1u))
sum2.1   set dlyout error    = k2=-1
pid.1       error cntl       = konst=6.7,dtau=1,itau=3.9
sum2.2      cntl  load input
twopole.1   input out        = tau1=10,tau2=1
dlay.1      out    dlyout     = 1
```

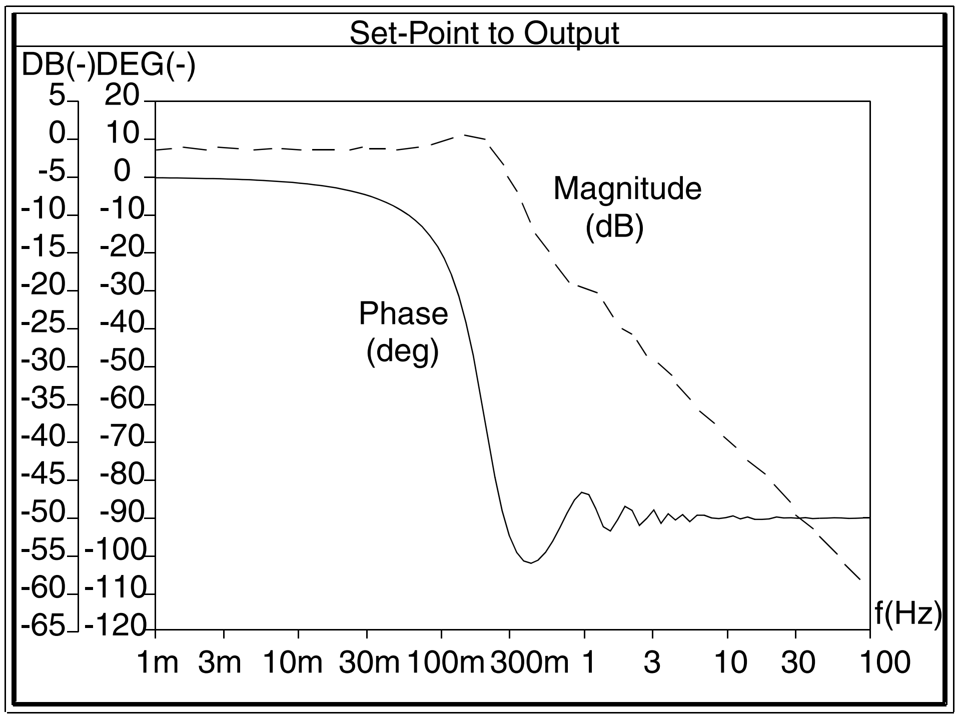
The stimulus to the system is provided by the two netlist entries named multisrc.set and multisrc.load. The multisrc.set entry models the control set point of the system. It specifies DC value 3 and AC value (1,0). This means that during a DC analysis the value of the set node will be 3.

During a transient analysis the value of the set node will remain constant at 3. During an AC analysis, the simulator linearizes the rest of the system, based upon value 3 at the set node. Then it applies a small signal to node set with magnitude 1 and a phase of 0 degrees.

Similarly, the multisrc.load source models the feed load into the system. During a DC analysis, the load node will have the initial value of the step, which is 3. During a transient analysis, the value of the load node will start out at 3 and then step from 3 to 5 at 1 second. The transition will take 1 microsecond. During an AC analysis, the simulator linearizes the system, using a load node value equal to the DC value, 3. The multisrc.load source has no AC source specification, so the load node has no effect during an AC analysis.

Small Signal AC Analysis Results

You can use the testpid.sin netlist to simulate the set point-to-output characteristics of the system. The following figure shows the magnitude (dashed line) and phase (solid line) of the changes in the output that would result from unity changes in the frequency of the control set point input signal. These results were obtained by invoking the Saber simulator on testpid.sin. From the simulator Transcript window, type <testpid.scs. Both testpid.sin and testpid.scs are files that are provided with the Saber simulator, along with the example templates shown in this manual.

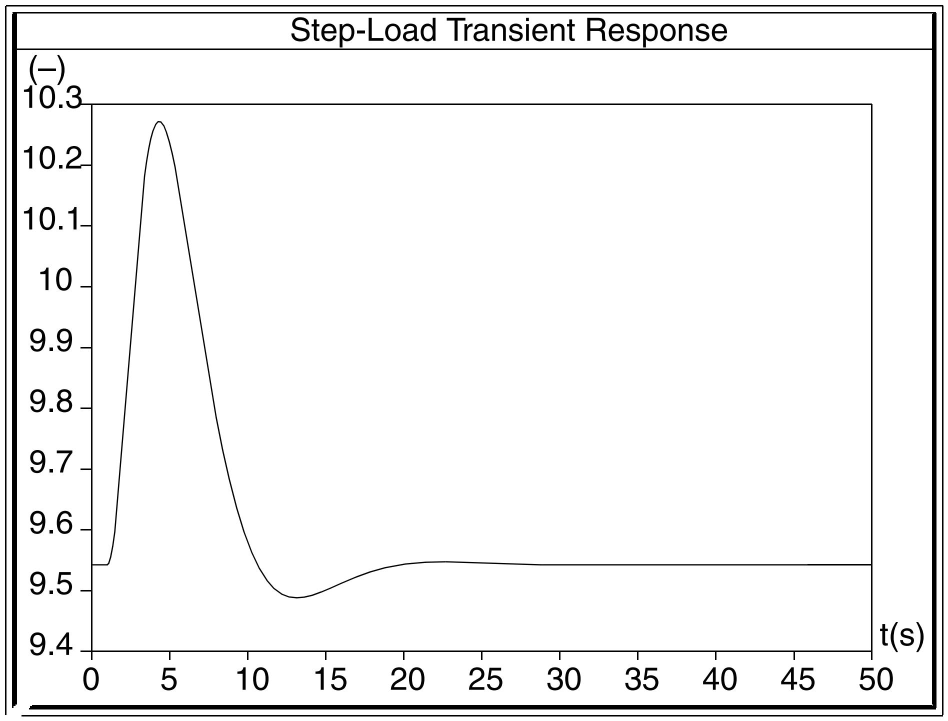


Set point-to-output small signal analysis

Step Load Transient Analysis Results

The testpid.sin netlist can also be used to simulate the step-load characteristics of the system. The following figure shows the system output that would result from a step in the input from 3 to 5. The system set point remains at 3, so the final system output should eventually approach this set point value. You can evaluate the effectiveness of the PID controller by observing the system response to the step change in input.

From the curve shown in the following figure, it is apparent that the system requires approximately 30 seconds to recover from a step change in the input from 3 to 5.



Step load transient result

10

Predefined MAST Declarations

In the examples used throughout the Guide to Writing MAST Templates topics, various variable types are used and assumed to be declared. One assumption includes the electrical pin type with i and v as the through and across variables. Another is the system temperature variable named temp. These are part of the default Saber simulation set of MAST declarations that are provided. They are not, however, part of the Saber simulator.

When you start the Saber simulator either from the Saber User Interface or with the saber command, without specifying a load option (-la; -ls; -ln; -lh; or -l filename), or if you start it with the -la option, the Saber simulator loads the analogy.sld file before reading your design. The analogy.sld file does several things, including setting up the simulation environment and declaring a set of templates. For a complete list of the templates loaded with analogy.sld, refer to the analogy.sin file located in the following directory:

```
install_home/template/support
```

The analogy.sld file is created from analogy.sin.

The included templates consists of the following major parts:

- Constant declarations
- Unit definitions (refer to the MAST Reference Manual)
- Pin definitions (refer to the MAST Reference Manual)
- External declarations (refer to the MAST Reference Manual)
- Set of frequently-used templates

Saber Simulator Include Files

An include file is a file that is specified to be used by the Saber simulator. This specification can be made either from another file, from a schematic through the use of the saber symbol, or as part of the saber command line. The general format for specifying an include file in a MAST template, netlist, or other include file is to enter a left angle bracket (<) followed by the name of the include file:

```
<filename
```

where filename can be the name of any file in your data search path. The left angle bracket (<) must appear in the first column of the line (i.e., it cannot be preceded by leading or embedded spaces).

When the simulator finds a statement for an include file, it replaces the statement with the complete contents of the named file. The Saber simulator reads in the contents of an include file directly, so you must position the include statement precisely where the information is required.

The following files are automatically specified as include files as part of the invocation of the Saber simulator:

consts.sin	declares some useful constants, such as pi, Boltzmann's constant, and electron charge.
header.sin	declares some commonly-used parameters (such as simulation temperature) and initializes them to default values.
units.sin	declares standard units and pin types used by all library templates.

These files are located in the following directory:

```
install_home/template/include
```

SPICE-Compatible Pre-Loaded Templates

The SPICE simulation environment is selected if you start the Saber simulator with the saber -ls load option. This environment differs from the default environment only in the collection of pre-loaded templates. Instead of loading MAST templates, the simulator includes SPICE-compatible templates. The SPICE simulation environment is stored in the spice.sld file.

For a list of the SPICE templates loaded with this environment, refer to the spice.set file located in the following directory:

```
install_home/template/spice
```

Note that you are not prevented from using SPICE-compatible templates if you do not use the -ls option—it just takes longer to initialize the Saber simulator. This is because it needs to find and read the appropriate templates, rather than have them pre-loaded.

Setting Up Your Own MAST Include Files

There may be cases when neither the provided pre-loaded include file set is sufficient for your problem. In such cases, you can either modify an existing include file or create a new one from scratch. Additionally, you can set up your include file set temporarily, or you can save it for later use. These options are described in the following topics:

- Adding Your Own Include File
- Creating Your Own Include Files
- Saving Your Include File Set

Adding Your Own Include File

The simplest way to modify the simulation environment is to add your own unit or pin-type declarations. One way to do this is put these declarations in a separate file, and include this file at the top of your Saber input file (netlist), before any other noncommented entries in this file. The way to include your own file from a schematic is to use the saber symbol and the associated SaberInclude property on your top-level schematic. This makes your declarations available everywhere in your design.

As an example, assume that you need a unit declaration for distance in meters and that you want to call the physical quantity dist. You must first check the units.sin file to see whether a declaration with that name already exists. If it does and is distance in meters, there is no need to modify the environment. If a unit declaration for dist already exists but means something else, you must select another name, because names of physical quantities must be unique. If no such declaration exists, put your declaration for distance into a file, say myunits.sin, as follows:

```
unit {"m", "meter", "distance"} dist
```

Then put the following line, without leading or embedded spaces, at the top of your Saber input file (design.sin):

```
<myunits.sin
```

This includes the myunits.sin file at the specified place in design.sin, which has the same effect as writing the contents of myunits.sin at that location.

In general, you can define units anywhere in your design, not just at the top level. In such a case, the unit will be defined only at that location in your design. For example, if you define a unit in the body of a template, that unit will be known only in that template and possibly its subordinate templates.

Creating Your Own Include Files

It is possible to create your own include file set, consisting of your units and pin types, without referring to the declarations in the units.sin and header.sin files. However, the templates provided depend on many of the units, pin types, and parameters in these files to be part of the environment.

Note:

It is recommended that your simulation environment always include the declarations provided.

To create your own include files, without any of the pre-declared templates, include the header.sin file at the top of your Saber input file, using the following statement:

```
<header.sin
```

without leading or embedded spaces. This automatically includes units.sin and also declares the parameters defined in the header.sin file. If you have your own unit declarations, include them immediately after header.sin:

```
<myunits.sin
```

The rest of your Saber input file will remain unchanged. When invoking the Saber simulator, be sure to specify the -ln option (load nothing) to tell the simulator not to load any other environment.

Saving Your Include File Set

It is sometimes useful to save your include file set, so that you can load it later without recreating it. Typically, in such cases you also want to include some frequently used templates in your saved file set, so that they don't have to be compiled each time they are needed. You can do so as described in the following example.

Assume you want to call your include file myenv. It should include the declarations from header.sin and your units from myunits.sin. Suppose, further, that you want the file to include the r, c, v, and q MAST templates, as well as your own opamp template. Create a file named myenv.sin that contains the following lines:

```
<header.sin
<myunits.sin
<r.sin
<c.sin
<v.sin
<q.sin
<opamp.sin
```

There must be no leading or embedded spaces on any of these lines. Each line instructs the simulator to include the specified file and to make their constants part of myenv.sin. Now, to create and save your environment, start the Saber simulator as follows:

```
saber -ln -p myenv
```

The -ln option tells the simulator not to load anything. The -p option instructs it to pre-compile myenv.sin, that is, to read it contents, together with all files it includes, find the declarations in those files, and put the result in a file named myenv.sld. This file now contains your environment as specified in myenv.sin.

To use your environment when simulating your design, start the simulator as follows:

```
saber -l myenv design
```

where design.sin is the full name of your Saber input file (netlist).

MAST Template Extraction Groups

All MAST library templates are designed to make it convenient for you to extract relevant information (such as all voltages or all currents), without having to know the name of each variable. Each template description (in the Saber online documentation) contains a post-processing section that lists all variables that are available for extraction.

You can use intuitive names for such things as voltages or currents and collect several of each type into extraction groups. The meaning, declaration, and use of extraction groups is described in the BJT example. The following list shows some of the groups used in MAST templates:

v	voltages
i	currents
q	charges
pwr	power dissipations
f	fluxes
noise	noises
dv8	deviations from nominal

11

Modeling Piecewise-Defined Behavior

Examples in earlier topics use models of familiar electrical circuit elements. This topic presents information about how to create models whose behavior is defined in piecewise segments. You can represent this type of nonlinear behavior with many of the same MAST constructs used for linear models in earlier topics.

This topic uses the following nonlinear example templates:

- Modeling a Simple Voltage Limiter with MAST
- Modeling a Voltage Divider with MAST

The two examples of nonlinear models introduce the following concepts:

- A method to approximate a discontinuous function by a continuous one, so that it can be modeled in the MAST language
- If expressions
- Newton steps to limit changes of the independent variable from one iteration to the next, which aids convergence
- Recommendations on when to specify newton steps
- Parameterized newton steps

Nonlinear Elements

A linear component is characterized by the fact that its template equations include only linear functions of system variables after substitution of all relevant expressions from val variable definitions. That is, there are no products or ratios of system variables in a template equation, and no system variable is an argument of a foreign or intrinsic function (except d_by_dt and delay).

If one or more of these requirements for a linear template is not met, the template is considered nonlinear. Note that a template can include nonlinear assignment statements yet still describe a linear element. The important question is whether the nonlinearity enters the template equation.

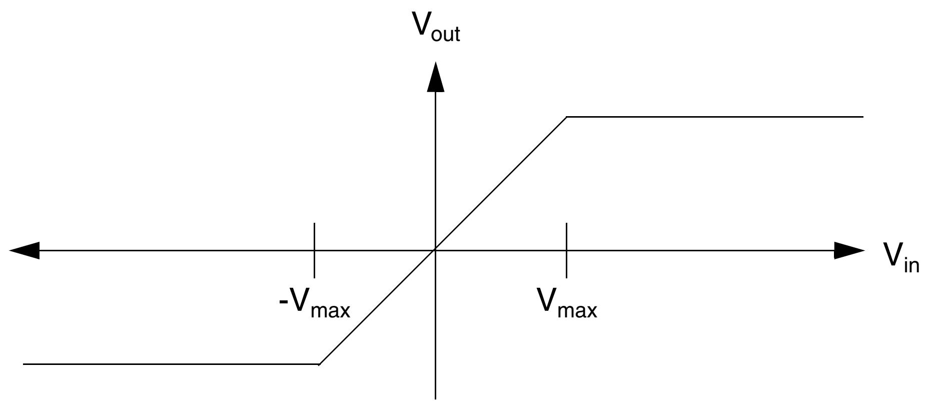
For example, the resistor_1 template defines power as the square of the voltage drop across the resistor divided by the resistance. Nevertheless, the template is linear because power does not enter the template equation. In other words, power is part of the resistor template, but it is not part of the resistor model. A template can also include nonlinear functions of time, frequency, or any parameter, without being a nonlinear template.

Nonlinear models are not confined to curvelinear functions—other types include those whose outputs have discontinuities or regions of piecewise linear behavior. The examples in this chapter illustrate these kinds of nonlinear characteristics.

There are issues that can arise when modeling a nonlinear element. Most of these are handled automatically by the Saber Simulator; however, there are MAST constructs, (sample points), that allow you to provide your own values for more efficient simulation. All such constructs require statements in the control section of the template.

Modeling a Simple Voltage Limiter with MAST

This topic shows how to use conditional expressions (if- else) in the template equation to define three regions of a symmetric voltage limiter shown in the following figure.



Ideal Voltage Limiter Characteristics

The structured template for this limiter is shown below.

```
element template vlim ip im op om = vmax
                                  # template header
  electrical ip, im, op, om       # header declarations
  number vmax
{                                 # start of template body
  val v vin, vout                 # local declarations
  var i iout
  number slope=1u, vmx
  struc {number bp, inc;} nvin[*]
  parameters {                    # start of parameters sect.
    vmx = abs(vmax)               # ensure use of positive
    nvin = [(-vmx,1.9*vmx),(vmx,0)]
  }                               # Newton step array for vin
  values {                        # start of values section
    vin = v(ip) - v(im)           # input voltage
    if (vin < -vmx)     vout = -vmx + slope * (vin + vmx)
    else if (vin > vmx) vout = vmx + slope * (vin - vmx)
    else                vout = vin  # voltage-limiting
  }                               # end of values section
  control_section {               # start of control section
    newton_step (vin, nvin)       # assign Newton steps
  }                               # end of control section
  equations {                     # start of equations section
    i(op -> om) += iout           # current contribution
    iout:  v(op) - v(om) = vout   # equation determining iout
  }                               # end of equations section
}                                 # end of template body
```

ASCII text of this example is located in:
install_home/example/MASTtemplates/structured/
vlim.sin

The description of the vlim template is divided into the following topics:

- Characteristic Equations
- Header and Header Declarations
- Values and Equations Sections -- This topic describes if expressions that include conditions. These conditions can use system variables, a branch variable, or a val variable that is a function of system variables to introduce nonlinear dependencies.

  Models described with if statements must satisfy various requirements for consistency, continuity, and non-zero slope.

  - Requirements For If Expressions
- Control Section—Newton Steps
  - Purpose of Newton Steps
  - Newton Step Example

Characteristic Equations

The characteristic equations of the voltage limiter are:

```
vout = -vmax   if vin < -vmax
vout = vin     if -vmax <= vin <= vmax
vout = vmax    if vin >= vmax
```

Header and Header Declarations

The vlim template is an element template with one argument, the limiting voltage (vmax). Its header and corresponding declarations are as follows:

```
element template vlim ip im op om = vmax
  electrical ip, im, op, om
  number vmax
```

There is no default value for vmax, which makes it mandatory for a user to specify an instance value. Note that this value may be specified as positive or negative—the template uses the absolute value of vmax.

Values and Equations Sections

The equations section for vlim follows the standard pattern for voltage-driven outputs as follows:

```
equations {
  i(op -> om) += iout
  iout:  v(op) - v(om) = vout
}
```

These equations closely reflect the limiting characteristics given in the topic titled "Characteristic Equations", except that they include a nonzero slope in the limiting regions (slope).

These equations require the following declarations in the local declarations section:

```
val v vout, vin
var i iout
number slope=1u, vmx
```

The limiting takes place in the following values section as follows:

```
values {
  vin = v(ip) - v(im)
  if (vin < -vmx) \
       vout = -vmx + slope * (vin + vmx)
  else if (vin > vmx) \
       vout = vmx + slope * (vin - vmx)
  else vout = vin
}
```

Although negative values for vmax are allowed, the equations for determining vout assume vmax is positive. That is, they use the absolute value of vmax, which is obtained by using abs, the intrinsic absolute value function. This absolute value of vmax is assigned to the local parameter vmx as follows:

```
parameters {
  vmx = abs(vmax)
  # other statements removed
}
```

The local declarations in conjunction with the values section enable the characteristic equations to express the output voltage (vout) as a function of the input voltage (vin), while finding the current contribution (iout) required for this to be true.

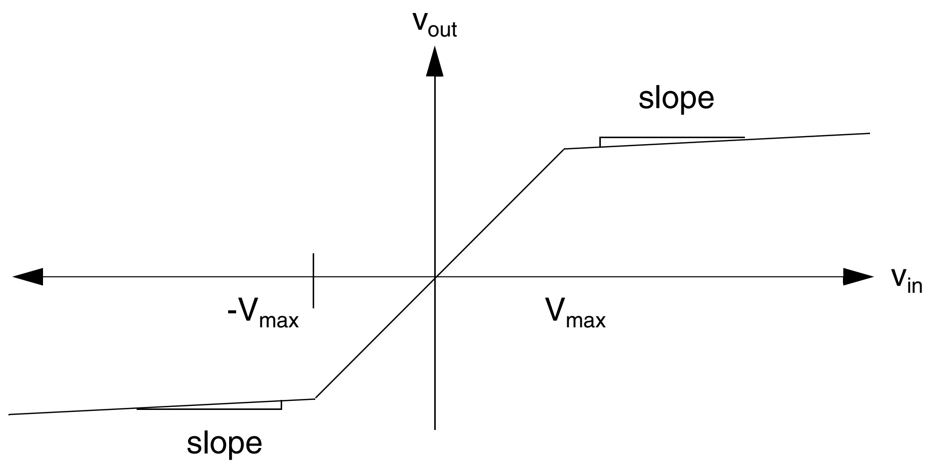
Requirements For If Expressions

There are several points worth noting about the conditional statements used in the values section:

- If a system variable, a branch variable, or a val variable that is a function of system variables appear in the condition of an if statement, the variables defined in the body of the if statement depend nonlinearly on the variable used in the condition. In this example, vout depends nonlinearly on vin.
- You must ensure that nonlinear models implemented with if statements or if expressions are continuous from one region to the next. In these template equations, it is necessary to force continuity at vin = ±vmax. Discontinuities can cause problems during simulations (e.g., small time steps (and long simulation time) or nonconvergence).

- You must ensure that variables in an if statement are always defined, regardless of the conditions of if statements or if expressions. One way to accomplish this is to make sure that any variable defined in any condition of an if statement is defined in every condition of the if statement or if expressions.
- An independent variable defined in the body of an if statement or if expression should never be set to a constant value. The reason is that, if the simulator, while iterating to find the solution of nonlinear equations, goes into a limiting region, it might not be able to get out of the region if the slope of the function is equal to 0—that is, the voltage limiter might latch.

  To prevent this problem, a small but non-zero multiple of vin named slope is added to vout (as shown in the following figure). In most cases, adding a very small slope yields a more realistic model than just a constant limit.



Voltage Limiter Template Characteristics

Purpose of Newton Steps

The previous figure shows that the dependence of vout on vin is piecewise linear, with -vmax and vmax defining crossover points for three separate regions of linear operation. For templates with this kind of input/output relationship, we recommend that you specify newton steps for the independent variable (here, vin). Newton steps are specified as pairs of numbers that specify a breakpoint and an increment, which is described in more detail below.

The purpose of newton steps is to place a limit on the change of the independent variable from one iteration to the next. The effect of this is to restrict the range of approximation the simulator performs around the crossover points, which helps improve simulation efficiency and is summarized as follows:

> When the variable is in a flat region, newton steps prevent the simulator from “guessing” a solution that grossly overshoots the actual solution. Such overshoots can cause slow convergence to a nonlinear solution or even numerical oscillation.
> Newton step increments are chosen to be large enough to let the independent variable move from one piecewise linear segment to another, but small enough to prevent it from moving too far and possibly skipping a segment altogether.

Newton steps are related to the iterative algorithm that the simulator uses to find the solution of nonlinear equations. If these equations include exponentials, convergence may be slow, because a small change in the independent variable of the exponential may cause a large change in the function value.

More specifically, the goal is for the value of the independent variable vin to move quickly into the intended region of operation and, once there, have its movement restricted so that it is unlikely to leave the region again.

Control Section—Newton Steps

Newton steps require three different statements to be included in the template as follows:

1. A declaration of a structure parameter (nvin) to specify values for breakpoints and increments, as pairs of numbers in an array (bp, inc). This parameter may be declared either as an argument in the header declarations or as a local parameter in the template body. Values for these pairs are specified as described in 2, below.

```
struc {
  number bp, inc;
} nvin[*]
```

2. An assignment statement in the parameter section that specifies values for nvin:

```
nvin = [(-vmx,1.9*vmx),(vmx,0)] #parameter sect.
```

3. A statement in the control section to associate the newton steps parameter (nvin) with the independent variable of the template (vin):

```
control_section {
   newton_step(vin,nvin)
}
```

It is possible for a template to have multiple independent variables requiring newton steps.

The meaning of the (breakpoint, increment) pairs is best defined by explaining the two pairs given for nvin in the assignment statement:

```
[(-vmx,1.9*vmx),(vmx,0)]
```

- Below the first breakpoint (-vmx), there is no restriction on how much vin can change from one iteration to the next.
- Between the first two consecutive breakpoints (-vmx and vmx), the change in vin is restricted to the first specified increment (1.9*vmx) per iteration.
- Above the last breakpoint (vmx), there is no restriction on how much vin can change.

To see why newton steps are used for this type of model, refer again to the figure above. Typically, the solution of the nonlinear equations should be in the nonlimiting (central) linear region. If, during iterations, there is limiting (say, on the left side), you do not want vin to “step over” the nonlimiting region and go directly to the limiting region on the right side. Instead, it is preferable to limit changes in vin such that it is in the nonlimiting region for at least one iteration.

Newton steps that have breakpoints (such as -vmx and vmx) that depend on the value given to an argument (vmax) are referred to as parameterized.

To accomplish this, newton steps are specified as shown for the nonlimiting region (between -vmx and vmx) but not for the upper and lower limiting regions. This has the effect of limiting the distance the simulator can step between ±vmx.—i.e., when it enters the nonlimiting region or is inside the region— to 1.9•(vmx). Because this region has width 2•(vmx), this newton step array prevents the simulator from stepping completely over the nonlimiting region.

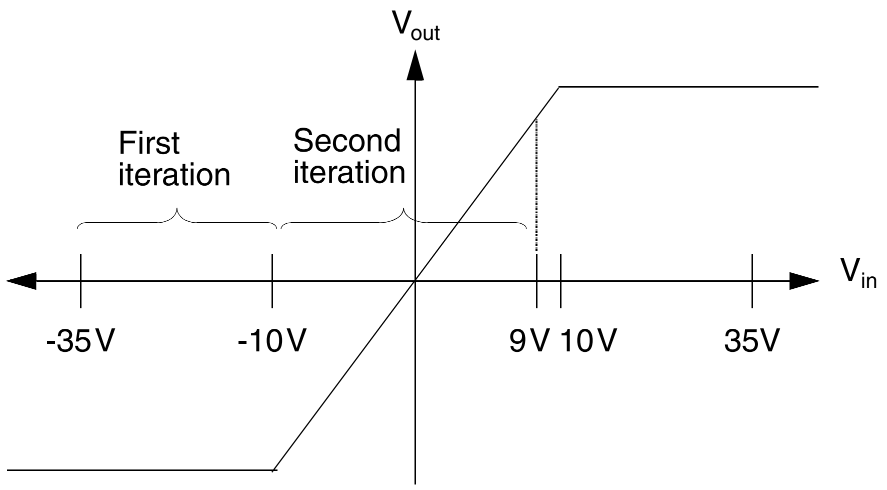
In general, the maximum allowable change should be less than the width of the critical region. In this example, there is no restriction on the size of an iteration step if vin remains in either the upper or lower limiting region. This is indicated by the fact that nvin does not specify limiting below -vmx or above +vmx. (The 0 increment means no limiting above +vmax.)

Newton Step Example

Assume that vmax has been specified by the user as 10V, which sets the lower limit of the output to -10V and the upper limit to +10V, as shown in the following figure. Further, assume that vin is in the lower limiting region at -35V and that the iterative algorithm intends to change it to +35V. This would result in the simulator stepping over the nonlimiting region between ±10V.

Inside the nonlimiting region, the amount of change is restricted to 1.9•vmax, which is 19V. However, this restriction does not affect the amount of change outside the nonlimiting region (i.e., vin can move from -35V to -10V in one iteration; the 19V limit does not apply until vin reaches -10V).

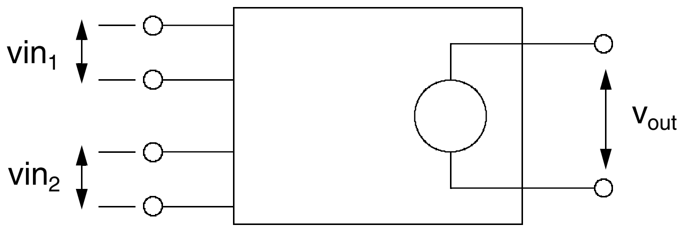
Therefore, for the next iteration, vin will have a value of +9V (-10 + 19), which is in the nonlimiting region.



How Newton Steps Limit the Change of vin

Modeling a Voltage Divider with MAST

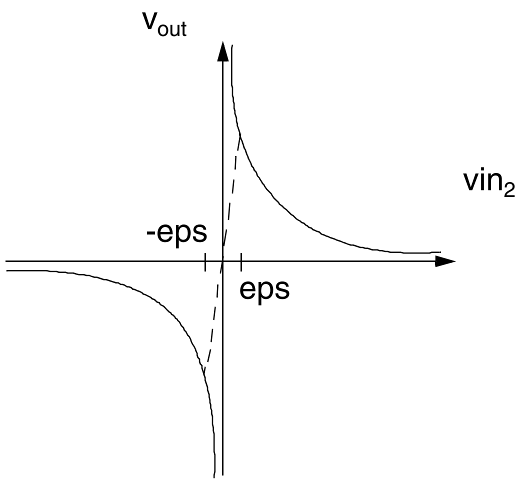
A voltage divider provides an output voltage as the ratio of two input voltages. The vdiv template models this relationship as a form of controlled voltage source as shown in the following figure:



Voltage Divider

Its characteristic equation needs to express the following: Determine the output current such that vout = vin1/vin2.

The vdiv template has a discontinuity. At vin2=0, vout “jumps” from -infinity to +infinity. Because the Saber Simulator requires models to be continuous, you have to modify the model to provide a “connection” across the discontinuity, as shown by the dashed line in the following figure:



Voltage Divider Output as a Function of vin2

The solid lines in the figure above show vout as a function of vin2 for a constant, positive value of vin1. There are various ways of connecting the two branches of the hyperbola so that vout is a continuous function of vin2. The dashed line shows the simplest way, using a straight line segment through the origin that intersects the hyperbolic branches of vout vs. vin2. The values of eps and -eps determine the points at which this line segment intersects the hyperbola.

By adding this connecting segment from -eps to +eps, the model for the voltage divider is expressed as:

Determine the output current such that:

```
vout = vin1/vin2  if vin2 < -eps or  if vin2 > eps
vout = vin1*vin2/eps2  if -eps <= vin2 <= eps
```

In general, if a model has a discontinuity, it must be converted to a continuous model (as in this example). Note that this procedure would be much more difficult if continuous derivatives were also required by the Saber Simulator.

```
element template vdiv ip1 im1 ip2 im2 op om
                                # template header
  electrical ip1, ip2, im1, im2, op, om
                                # header declarations
{                               # start of template body
  val v vin1, vin2, onev, vout   # local declarations
  var i iout
  number eps = 1e-6, eps2
  struc {number bp, inc;} nv2[*]
  parameters {                  # start of parameters section
    if (eps<=0) eps = 1e-15      # prevent negative eps values
    if (eps>.01) eps = .01
    eps2 = 1/(eps*eps)
    nv2 = [(-2*eps,eps), (2*eps,0)]
                                # newton steps for vin2
  }                             # end of parameters section
  values {                      # start of values section
    vin1 = v(ip1) - v(im1)       # input voltage vin1
    vin2 = v(ip2) - v(im2)       # input voltage vin2
    if (abs(vin2)<1e-50) onev = 0     # Prevent divide-by-zero
    else               onev = 1/vin2
    if (abs(vin2) > eps) vout = vin1*onev   # output voltage
      # Next line prevents output from growing without bounds
    else               vout = vin1*vin2*eps2
  }                             # end of values section
```

```
 control_section {           # start of control section
   newton_step (vin2, nv2)     # assign newton steps to vin2
 }                       # end of control section
 equations {                 # start of equations section
   i(op->om) += iout          # current contribution
   iout:  v(op) - v(om) = vout # equation to determine current
 }                       # end of equations section
}                          # end of template body
```

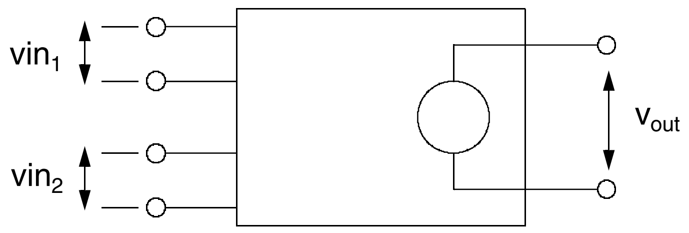
ASCII text of this example is located in:
install_home/example/MASTtemplates/structured/vdiv.sin

The description of the vdiv template is divided into the following topics:

- Header Declarations
- Parameters Section - MAST vdiv Template
- Newton Step Parameters -- shows unparameterized newton steps and requirements for newton steps
- Equation and Values Sections

Header Declarations

As shown in the following figure, the vdiv template has two input ports and one output port, each consisting of two connection points.



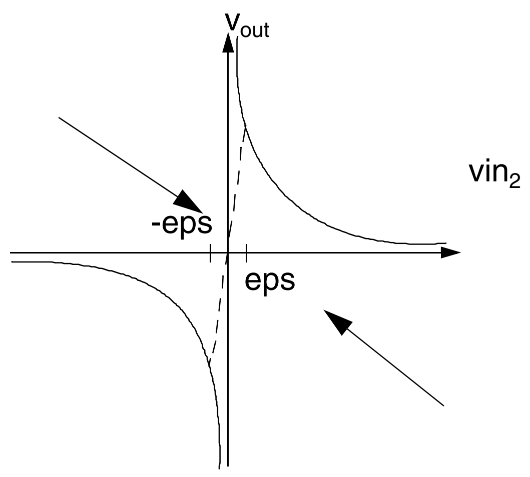
Voltage Divider

This template provides no arguments and the template header and header declaration is written as follows:

```
element template vdiv ip1 im1 ip2 im2 op om
  electrical ip1, ip2, im1, im2, op, om
```

Parameters Section - MAST vdiv Template

The eps parameter specifies half the horizontal distance between the end points of the line segment shown in the following figure:



eps Parameter

It is possible to make eps an argument of the template, but here it is declared locally and initialized to 10^{-6}. Although this approach does not allow the value of eps to be changed in a netlist, you can still change its value using the alter command.

```
number eps = 1e-6, eps2      # local declarations
```

Further, an error-checking statement is included that resets eps to 10^{-15} if a user tries to alter it to a negative value or zero:

```
if(eps <= 0) eps = 1e-15  #In parameters section
```

In addition, the following line keeps eps at a level no greater than.01:

```
if(eps>.01) eps = .01      #In parameters section
```

The eps2 parameter is defined as the reciprocal of eps squared; it is used in the equation that defines vout when vin2 lies in the region between -eps and +eps.

```
eps2 = 1/(eps*eps)         #In parameters section
```

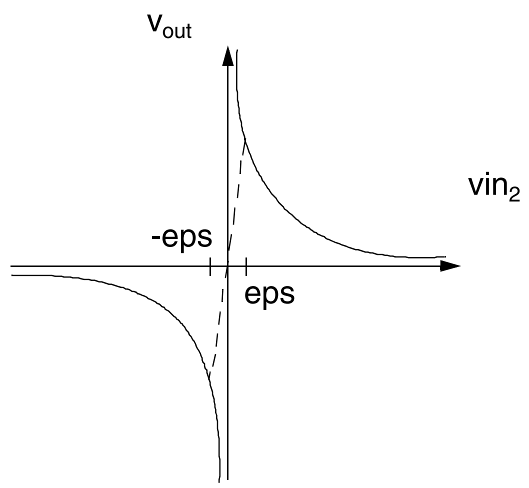
The last line in the parameters section relates to the nv2 parameter, which is an assignment statement that specifies (breakpoint, increment) values for nv2 as follows:

nv2 = [(-2*eps,eps), (2*eps,0)]

Refer to the topic titled "Newton Step Parameters".

Newton Step Parameters

The output voltage, vout, of the vdiv template depends nonlinearly upon the two input voltages vin1 and vin2. (The nonlinear dependence on vin1 is established by recognizing that
^{1}vout/^{1}vin1 = 1/vin2 is not constant, but a function of the circuit's operation.) As with the vlim template, the vdiv template provides different regions in which the output (vout) depends on the input (vin2).



Voltage Divider Output as a Function of vin2

The figure above shows that vout is a hyperbolic function of vin2, with -eps and eps defining crossover points for three separate regions of continuous operation. Because vout depends on vin2 differently in different regions of vin2, and because, when vin2 is near 0, vout changes considerably even for small changes in vin2, it is advisable to specify newton steps for the independent variable, vin2.

Newton steps require the inclusion of three different statements in the vdiv template:

- A declaration of a structure parameter (nv2) that specifies values for breakpoints and increments, as pairs of numbers (bp, inc) in an array of unspecified size. This parameter may be declared either as an argument in the header declarations or as a local parameter in the template body. Here, nv2 is declared as a local parameter:

```
struc { number bp, inc; } nv2[*]   #local decl.
```

- An assignment statement that specifies (breakpoint, increment) values for nv2:

```
nv2 = [(-2*eps,eps),(2*eps,0)] #parameters sect.
```

- A statement in the control section that associates the newton step variable (nv2) with the independent variable of the template (vin2):

```
control_section {
  newton_step (vin2, nv2)
}
```

The values for the (breakpoint, increment) pairs in the assignment statement (Item 2, above) enforce the following restrictions on iterations of the simulator:

- Below the first breakpoint (-2*eps), there is no restriction on how much vin2 can change from one iteration to the next.
- Between the first two consecutive breakpoints (-2*eps and 2*eps), the change in vin2 is restricted to the first specified increment (eps) per iteration.
- Above the last breakpoint (2*eps), there is no restriction on how much vin2 can change.

To see why newton steps are used for this type of model, refer back to the above figure. Typically, the solution of the nonlinear equations should be in one of the regions where vin2 ¼0 (i.e., on the hyperbola). If, during iterations, vin2 =0, you do not want vin2 to “jump over” from one branch of the hyperbola to the other. Instead, it is preferable to limit changes in vin2 such that it is in the connecting region for at least one iteration.

To accomplish this, newton steps are specified as shown for the connecting region (between ±2*eps) but not for the positive or negative regions of the hyperbola. Defining breakpoints with a factor of two means that the increment (eps) limits the distance the simulator can step to one-fourth of the distance between breakpoints. This ensures that at least one iteration is performed in this region. Limiting the distance the simulator can step as vin2 approaches 0 prevents the simulator from stepping completely over the connecting region.

Equation and Values Sections

In the vdiv template, the values section contains the statements that handle the output voltage. The equations section handles the current contribution.

The template equation for vdiv is similar to that of the voltage limiter template (vlim) as follows:

```
equations {
  i(op->om) += ioutn
  iout:  v(op) - v(om) = vout
}
```

This requires that iout is declared as a var variable as follows:

```
var i iout                    #local declaration
```

In addition, vout is declared as a val as part of the following local declaration:

```
val v vin1, vin2, onev, vout   #local declaration
```

The values section defines the output voltage as a function of the two input voltages, according to the modified model. Because vout depends nonlinearly on the input voltages, you must declare both vin1 and vin2 as val variables as in the previous statement.

The following values section contains comments that identify the function of each statement:

```
values {
  vin1 = v(ip1) - v(im1)     # input voltage vin1
  vin2 = v(ip2) - v(im2)     # input voltage vin2
    # Next line prevents divide-by-zero error
  if (vin2<1e-50) onev = 0
  else            onev = 1/vin2
    # Next lines prevent output from
    # growing without bounds
  if (abs(vin2) > eps) vout = vin1*onev
  else                 vout = vin1*vin2*eps2
}
```

12

Modeling Nonlinear Devices

This topic provides models of two common electrical devices—the junction diode and the bipolar junction transistor. For simplicity, the following example templates provide idealized models of these nonlinear devices:

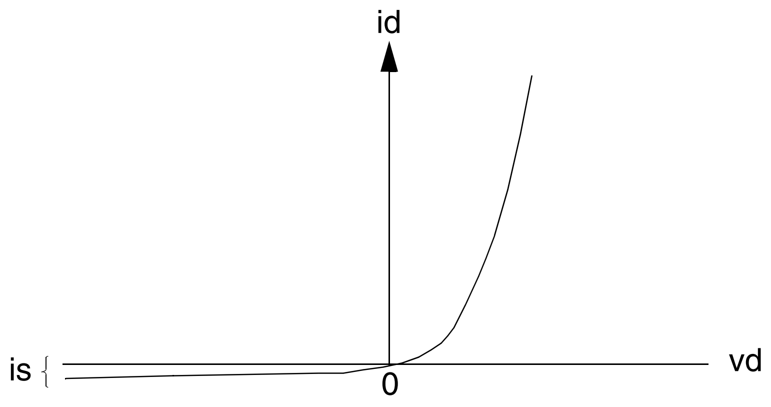
- Modeling an Ideal Diode with MAST
- Ebers-Moll MAST Model for the Bipolar Transistor, a bipolar junction transistor that allows the user to select as either NPN or PNP

Each of these templates uses a control section to include statements for newton steps and for initial conditions. In addition, these examples introduce the following concepts:

- Enumerated parameters
- Grouping of val variables or system variables for extraction
- Control section statements that specify small-signal parameters for use with the ssp command
- Collapsing nodes

Modeling an Ideal Diode with MAST

The ideal diode is a typical example of a nonlinear electrical device, because the diode current is proportional to the exponential of the voltage across the diode as shown in the following figure.



Ideal Diode Characteristics

The following shows the ideal diode example template:

```
element template diode p m = is, ic # template header
  electrical p, m                   # header declarations
  number is = 1e-16,
         ic = undef
  external number temp
```

```
{                                   # start of template body
  number k = 1.318e-23,             # local declarations
         qe = 1.602e-19,
         vt
  val v vd
  val i id
  struc {
    number bp, inc;        # Newton steps
  }  nvd[*] = [(0,.001),(2,0)]
  parameters {                  # start of parameters section
    vt = k * (temp+273.15) / qe # compute thermal voltage
  }                            # end of parameters section
  values {                      # start of values section
    vd = v(p) - v(m)            # diode voltage
    id = is * (limexp(vd/vt)-1) # diode current
  }                            # end of values section
  control_section {             # start of control section
    newton_step (vd,nvd)        # Newton steps assigned to vd
    initial_condition(vd,ic)
    start_value(vd,0.6)
    device_type("diode","example")
    small_signal(vd,voltage,"p-m voltage", vd)
  }                            # end of control section
  equations {                   # start of equations section
    i(p->m) += id               # current contribut. of diode
  }                            # end of equations section
}                               # end of template body
ASCII text of this example is located in:
install_home/example/MASTtemplates/structured/diode.sin
```

diode Template Topics

The description of the diode template is divided into the following topics:

- Characteristic Equation
- Header Declarations
- Modeling Temperature
- Newton Steps
  - Newton Steps Example - MAST diode Template
- Template Equation -- shows how to use the limexp function, which is a modified exponential operator that is limited for large exponents to prevent overflow.
- Initial Conditions
- Starting Value -- shows how to specify a starting value for the first iteration of a DC analysis, using a start_value statement in the control section (which is different from the initial_condition statement).
- Small-Signal Parameters -- shows how to specify small-signal parameters within a template, which are reported in response to a Saber ssp command.

Characteristic Equation

The characteristic equation for a diode is:

$id = is \bullet (e^{(vd \bullet q)/k \bullet T)} - 1)$

where:

id	is the current through the diode
vd	is the voltage across the diode
is	is the saturation current, typically in the order of 10^{-16} A
q	is the electron charge: $q = 1.602 \bullet 10^{-19}$ AÞs (1 AÞs = 1 coulomb)
k	is Boltzmann's constant: $k = 1.381 \bullet 10^{-23}$ J/K
T	is the absolute temperature (in kelvins)

The expression (k*T)/q is usually called thermal voltage. The template assigns this expression to the variable vt, which is then substituted into the template equation.

Header Declarations

The diode template is an element template with two electrical pins and arguments for the saturation current (is) and the initial voltage across the diode (ic). By initialization, is receives the default value 10^{-16} A and ic is undefined. The template header and header declarations are as follows:

```
element template diode p m = is, ic
  electrical p, m
  number is = 1e-16,
           ic = undef
```

In addition, the system's operating temperature, which is external to the diode template, must be made available—this is done by including temp as an external parameter in the header declarations as follows:

```
  external number temp # part of header declar.
```

Modeling Temperature

Because this simplified diode model does not include self-heating effects, it makes sense to compute the thermal voltage for use as a constant. The absolute temperature (T) used for calculating the thermal voltage is expressed in kelvins. However, the system temperature (temp) is expressed in ×C—this is converted to kelvins by adding 273.15 within the formula for thermal voltage in the parameters section as follows:

```
parameters {
  vt = k * (temp+273.15) / qe
}
```

In addition, you need to assign values to the Boltzmann's constant (k) and the electron charge (qe) parameters, which is done in the following local declarations:

```
number  k = 1.318e-23,
       qe = 1.602e-19,
       vt
```

Newton Steps

Newton steps place a limit on the change of the independent variable (vd) from one iteration to the next. The newton step parameter (nv) is declared as a structure that specifies values for breakpoint and increment pairs (bp, inc) in an array of unspecified size. This parameter may be declared either as an argument in the header declarations or as a local parameter in the template body. Here, nv is declared as a local parameter and initialized to the values indicated:

```
struc {
  number bp, inc;
}  nvd[*] = [(0,0.001),(2,0)]
```

This combines the declaration statement method and the assignment statement method of assigning newton steps. There is no functional difference between the two methods.

The statement for newton steps in the control section associates the newton steps parameter (nvd) with the independent variable of the template (vd) as follows:

```
newton_step(vd,nvd) # part of control_section
```

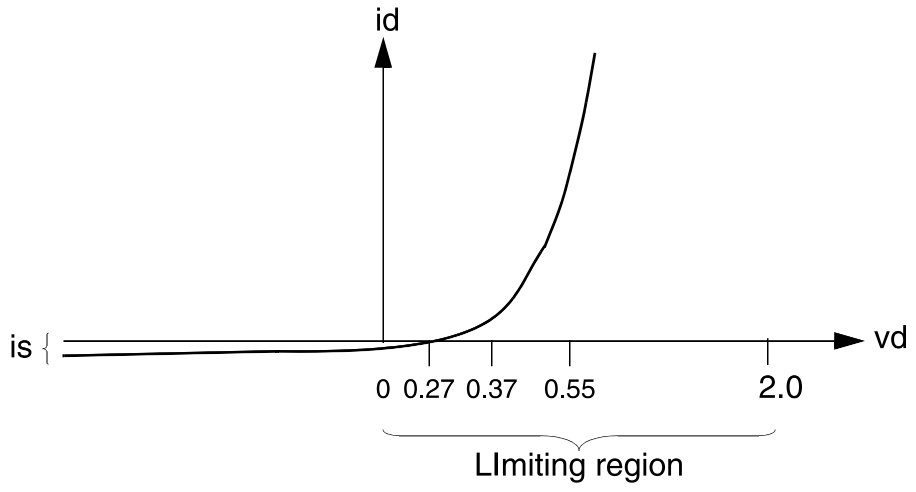
The breakpoint and increment values for nvd ((0,0.001),(2,0)) enforce the following restrictions on iterations of the simulator.

- Below the first breakpoint (0), there is no restriction on how much vd can change from one iteration to the next.
- Between the first two consecutive breakpoints (0 and 2), the change in vd is restricted to the first specified increment (0.001) per iteration.
- Above the last breakpoint (2), there is no restriction on how much vd can change.

Note that the value of nvd does not depend on the value of an argument to the template. Therefore, these newton steps are not parameterized.

Newton Steps Example - MAST diode Template

The effect of this newton step definition is best seen in the example figure shown below. Assume first that vd = 0.27V, and that the iterative algorithm intends to change it to 0.55V. However, between 0V and 2V, the change is restricted to 0.1V, so in the next iteration vd will have a value of 0.37V. Similarly, if the algorithm intends to change vd to 0, it will change only to 0.17V. However, if vd = -3V and the algorithm intends to change it to 0.67V, vd will change to 0.1V. This is because there is no limit to the amount vd can change below zero, whereas between 0V and 2V, vd can change by only 0.1V.



How Newton Steps Limit the Change of vd

Template Equation

The following equation expresses the branch current (id) as a function of the branch voltage (vd):

```
    id = is*(limexp(vd/vt)-1) # Part of values section
```

The assignment to id is handled in the values section. The equations section uses the computed value of id to assign the current contribution of the diode as follows:

```
equations {
  i(p->m) += id
}
```

Note the usage of the MAST limexp function rather than the exp function. The limexp function is a limited exponential function. Its value is identical to that of exp for arguments between -80 and 80, but for arguments outside this range, limexp limits the function value to prevent overflows. The exact definition of limexp is given in the MAST Reference Manual.

The diode voltage contribution is handled by the following statement in the values section:

```
    vd = v(p) - v(m) # Part of values section
```

Initial Conditions

Initial conditions allow you to specify the initial value for the voltage across the diode (vd) prior to a DC analysis. The initial_condition statement in the control section associates vd with the argument ic:

```
    initial_condition (vd,ic) #part of control_section
```

Starting Value

When finding a DC solution, the Saber simulator sets all system variables to their start values, which, by default, are 0. The start_value statement allows you to overwrite this default with a value that is closer to the solution you expect. For example, the forward bias value of a PN junction puts the junction into its conducting region and is somewhere around 0.6V. You can specify this with a start_value statement in the control section, as follows:

```
start_value (vd, 0.6)  # part of control_section
```

It is important to note the difference between start_value and initial_condition. The value of initial_condition is held throughout the DC analysis and is therefore the value at the end of DC. The value of start_value is used as an initial “guess” by the simulator for the first DC iteration only. After the first iteration, start_value is ignored for all subsequent iterations.

Small-Signal Parameters

The next two statements in the control section allow you to specify the small-signal characteristics of this model that will be reported in response to the Saber ssp command. See the topic titled "Small-Signal Parameters Report" below for more information on small-signal characteristics of the diode template.

```
device_type("diode","example")
small_signal(vd,voltage,"p-m voltage", vd)
```

Small-Signal Parameters Report

There are additional statements that you can insert into the control section of a MAST template that allow you to list the values of a set of small-signal parameters by using the ssp command. The simulator obtains these values by linearizing the model at a given operating point, usually by taking the partial derivative of a dependent variable with respect to an independent variable. The ssp command reports small-signal parameter values for the linearized model only at the operating point—you cannot plot these values. Note that the complete specification for small-signal parameters requires that you run a DC analysis, which gives additional DC operating point information such as node voltages and branch currents.

The report appears in the .out file following simulation and provides the following headings for small-signal parameters:

Parameter	Name	Classification	Value

For example, the report for the small-signal parameter of the diode template would look something like the following:

Parameter	Name	Classification	Value
p-m voltage	vd	voltage	0.46

Small-Signal Parameter Statements

You can specify a small-signal parameter (SSP) for a template by using three types of SSP statements in the control section, which are identified as follows:

- device_type - MAST Small Signal Parameter Statement
- small_signal - MAST Small Signal Parameter Statement
- Four Fields: - small_signal Statement
- Five Fields: - small_signal Statement
- ss_partial - MAST Small Signal Parameter Statement

Because of the simplicity of the diode model, there are not many small-signal dependencies that can take advantage of the SSP reporting feature.

device_type - MAST Small Signal Parameter Statement This statement is inserted into the control section to provide an identifier in the SSP report; it has no effect on determining the SSP values.

```
device_type("diode","example")
```

small_signal - MAST Small Signal Parameter Statement One small_signal statement is required to define each SSP. This statement can have either four or five fields that define the SSP characteristics. In either case, the first three fields are the same.

Four Fields: - small_signal Statement The following four-field small_signal statement appears in the diode template:

```
small_signal(vd,voltage,"p-m voltage", vd)
```

The four fields are specified as follows:

- parameter name (vd)—this is the name of the SSP that is reported under the Name heading by the ssp command.
- classification (voltage)—this is reported under the Classification heading by the ssp command.
- report identifier ("p-m voltage")—this is an identifier string that is reported under the Parameter heading by the ssp command.
- assigned variable (vd)—this is an internal variable whose value is assigned directly to the SSP. It must be either a val (an intermediate variable), a branch variable, a parameter, a value obtained from an ss_partial statement, or an expression of these. Here, the value of vd is assigned to vd.

Five Fields: - small_signal Statement The following five-field small_signal statement appears in the d template from the MAST Template Library:

```
small_signal(cd,capacitance,"p-is capacitance",qd,vdi)
```

The five fields are specified as follows:

1. parameter name (cd)—this is the name of the SSP that is reported under the Name heading by the ssp command.
2. classification (capacitance)—this is reported under the Classification heading by the ssp command.
3. report identifier ("p-is capacitance")—this is an identifier string that is reported under the Parameter heading by the ssp command.
4. dependent variable (qd)—this is differentiated with respect to the specified independent variable. It must be either a val (an intermediate variable), a branch variable, or an expression of these.
5. independent variable (vdi)—this is the variable with respect to which the dependent variable is differentiated.

The variable in field 4 must be directly dependent upon the independent variable in field 5. Otherwise, a value of 0 will be reported by the ssp command.

In other words, you cannot use the variable that should be in field 5 in an expression and then put the result of that expression in field 5.

For example, in the five-field statement above, qd must depend directly on the value of vdi; qd cannot depend on the result of an expression containing vdi.

ss_partial - MAST Small Signal Parameter Statement This is an alternate way of taking a partial derivative for use by the four-field form of a small_signal statement (above). It has three fields that define the differentiation. The following ss_partial statement appears in the d template from the MAST Template Library:

```
ss_partial(g_d,idi,vdi)
```

The line is composed of the following:

1. variable name—this is the name of the partial derivative of the next two fields. This partial derivative can be used in the fourth field of the 4-field form of the small_signal statement above.
2. dependent variable—this is differentiated with respect to the specified independent variable in the third field, below. It must be either a val (an intermediate variable), a branch variable, or an expression of these.
3. independent variable—this is the variable with respect to which the dependent variable is differentiated.

The variable in field 2 must be directly dependent upon the independent variable in field 3. Otherwise, a value of 0 will be reported by the ssp command. In other words, you cannot use the variable that should be in field 3 in an expression and then put the result of that expression in field 3.

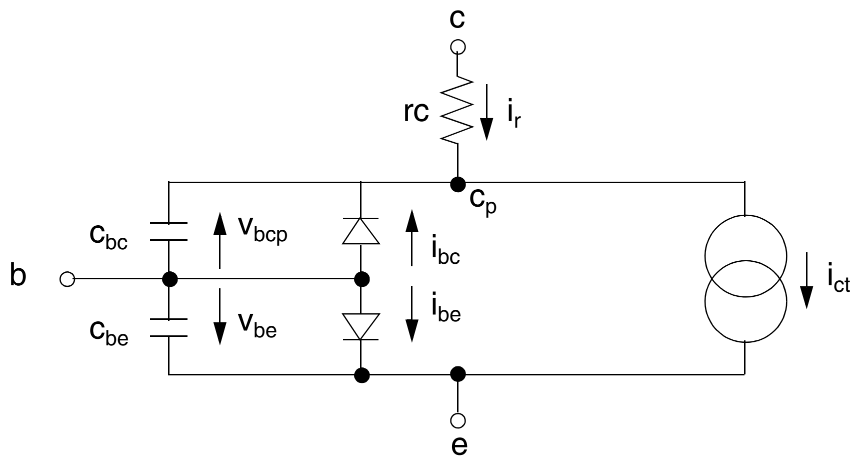
For example, in the ss_partial statement above, idi must depend directly on the value of vdi; idi cannot depend on the result of an expression containing vdi.

Ebers-Moll MAST Model for the Bipolar Transistor

A bipolar junction transistor (BJT) is a typical example of a device consisting of several nonlinear functions. However, implementing a complete transistor model in the MAST language is beyond the g c

of this manual. As a result, this topic describes a reduced implementation of an Ebers-Moll model that shows various important aspects of modeling a complex device. The model shown in this topic is a simplified version of the EM2 model described in the book titled Modeling the Bipolar Transistor, by Getreu, I. (Tektronix, Inc. 1976).

The transistor model presented in this section is shown in the following figure for an NPN transistor. It implements the Ebers-Moll DC model, non-zero collector resistance, and the junction capacitance of the base-emitter and base-collector diodes. The model has three external nodes (base, collector, and emitter) and one internal node (cp, the internal collector).



Ebers-Moll Model of an NPN transistor

The complete BJT template (bjt) is shown as follows (line numbers are added for reference):

```
element template bjt c b e = model, ic
  electrical c, b, e
  struc {                         # the transistor mode
    enum {_n, _p} type
    number  is=1e-16, bf=100, br=1, \
            cje=0, vje=.75, mje=.33, \
            cjc=0, vjc=.75, mjc=.33, rc=0
  } model = ()
  number ic[2]=[undef,undef]
  external number temp
{               # begin template body
   # declare local param., vals, and extraction groups
  number k = 1.381e-23,           # Boltzmann's constant
         qe = 1.602e-19,          # electron charge
         vt,
         qbe0, qbc0, vje0, vjc0
  struc {
    number bp, inc;
  } nv[*] = [(0,.1),(2,0)]
  val v vbc, vbe, vce             # declarations of vals
  val i iec, icc, iba, ico, ir
  val q qbc, qbe
  electrical cp                   # local node
  group {vbc,vbe} v               # extraction groups
  group {iba,ico,ir} i
  group {qbc,qbe} q
```

```
parameters {
 # calculate thermal volts and functions of model param.
  vt = k * (temp + 273.15) / qe
  qbe0 = model->cje * model->vje / (1 - model->mje)
  qbc0 = model->cjc * model->vjc / (1 - model->mjc)
  vje0 = 2 * model->vje / model->mje
  vjc0 = 2 * model->vjc / model->mjc
}           # end of parameters section
values {
   # calculate basic quantities of npn and pnp trans.
  vbc = v(b) - v(cp)
  vbe = v(b) - v(e)
  vce = v(cp) - v(e)
  if (model->type == _n) {
    iec = model->is * (limexp(vbc/vt) - 1)
    icc = model->is * (limexp(vbe/vt) - 1)
  }
  else {
    iec = -model->is * (limexp(-vbc/vt) - 1)
    icc = -model->is * (limexp(-vbe/vt) - 1)
  }
   # calculate base, collector, and resistor currents
  iba = iec/model->br + icc/model->bf
  ico = icc - iec - iec/model->br
  if (model->rc ~= 0) ir = (v(c) - v(cp)) / model->rc
  else              ir = 0
```

```
   # calculate charges
  if(model->type == _n) {
    if (vbc<0) {
      qbc = qbc0*(1-((1-vbc/model->vjc)**(1-model->mjc)))
    }
    else {
      qbc = model->cjc*vbc*(1 + vbc/vjc0)
    }
    if (vbe<0) {
      qbe = qbe0*(1-((1-vbe/model->vje)**(1-model->mje)))
    }
    else {
      qbe = model->cje*vbe*(1 + vbe/vje0)
    }
  }                 # end “if type _n” condition

  else { # if model is not of type _n
    if(vbc > 0) {
      qbc = -qbc0*(1-(1+vbc/model->vjc)**(1-model->mjc))
    }
    else {
      qbc = model->cjc*vbc*(1-vbc/vjc0)
    }
    if(vbe > 0) {
      qbe = -qbe0*(1-(1+vbe/model->vje)**(1-model->mje))
    }
    else {
      qbe = model->cje*vbe*(1-vbe/vje0)
    }
  }                 # end “if not type _n” condition
}                   # end values section
```

```
 control_section {
   # if no collector resistance, collapse nodes c and cp
  if (model->rc == 0) collapse(c,cp)
   # specify Newton steps
  newton_step((vbc,vbe),nv)
   # initial conditions and start value
  initial_condition(vbe, ic[1])
  initial_condition(vce, ic[2])
  start_value (vbe, 0.6)

   # small-signal parameters
  device_type("bjt", "example")
  small_signal(ibase,current,"base current",iba)
  small_signal(icoll,current,"collector current",ico)
  small_signal(vbe,voltage,"base-emitter voltage", vbe)
  small_signal(vbc,voltage,"base-collector voltage", vbc)
  small_signal(rc,resistance,"collector resistance",\
             model->rc)
 }                 # end control_section

 equations {
   # current at base, internal collector, and emitter
  i(b->e)  += iba + d_by_dt(qbe)
  i(b->cp) += d_by_dt(qbc)
  i(cp->e) += ico

   # current at collector resistor, if present
  if (model->rc ~= 0) i(c->cp) += ir
 }                  # end equations section
}                   # end template body
```

ASCII text of this example is located in:
install_home/example/MASTtemplates/structured/bjt.sin

The description of the bjt template is divided into the following topics. Some of the concepts highlighted by these topics follow the topic title:

- Basic Model Equations
- Preparing to Write the MAST bjt Template
- Header Declarations

  Transistor Type -- shows the enumerated parameter type (enum), which is useful if a parameter can take on only a limited set of values.
  Collector Resistance
  Initial Conditions
- Local Parameters

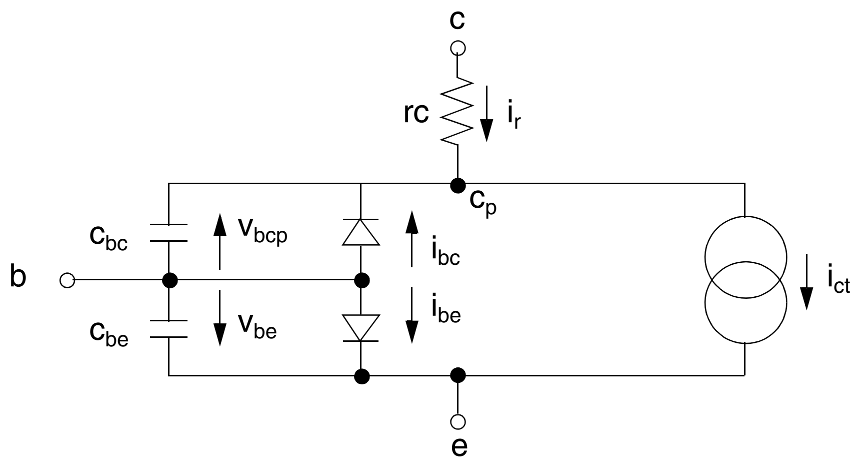
  Temperature
  Junction Capacitance
  Newton Steps Declaration - MAST bjt Template
  Local Node - MAST bjt Template
  Intermediate Current and Charge Variables
  Defining Groups For Extraction - MAST bjt Template -- shows that groups (specified using a group statement) are useful for grouping together several val variables or system variables, so that you can refer to them by a single name as when doing extraction.
- Thermal Voltage
- Junction Capacitance -- shows that model equations often need to be “transformed” into a MAST-compatible set of equations. In this example, you have to compute the charges stored in the nonlinear junction capacitances and account for their singularities.
- Intermediate Calculations

  Fundamental Quantities - MAST bjt Template
  Currents
  Charges
- Control Section
  - Collapse Node -- shows that collapsing nodes (using a collapse statement) is useful for reducing the size of a system, but it can place restrictions on parameter alteration.
  - Newton Steps -- shows association of the same set of newton steps with multiple variables.
  - Initial Conditions in Control Section

  - Starting Value
  - Small-Signal Parameters
- Equations Section

Basic Model Equations

The following equations describing the model in the following figure are taken from the book titled Modeling the Bipolar Transistor, by Getreu, I. (Tektronix, Inc. 1976).



Ebers-Moll Model of an NPN transistor

$i_{ct} = i_{cc} - i_{ec}$

$i_{bc} = i_{ec} / \beta_r$

$i_{be} = i_{cc} / \beta_f$

$i_{ec} = i_s * (\exp((qe * v_{bcp}) / (k * T)) - 1)$

$i_{cc} = i_s * (\exp((qe * v_{be}) / (k * T)) - 1)$

$c_{bc} = c_{jco} / (1 - v_{bcp} / v_{jc}) ** m_{jc}$

$c_{be} = c_{jeo} / (1 - v_{be} / v_{je}) ** m_{je}$

These equations use the following model parameters:

i_s	transistor saturation current (typically about 10^{-16} A)
β_f	forward current gain (typically about 100)
β_r	reverse current gain (typically about 1)
c_{jco}	collector-base junction capacitance (typically about 5pF)
c_{jeo}	emitter-base junction capacitance (typically about 5pF)
v_{jc}	collector-base barrier potential (typically about 0.75V)
v_{je}	emitter-base barrier potential (typically about 0.75V)
m_{jc}	gradient factor for collector-base capacitance (between 0.333 and 0.5)
m_{je}	gradient factor for emitter-base capacitance (between 0.333 and 0.5)

Preparing to Write the MAST bjt Template

Before starting to write a MAST template for this transistor model, you need to take two preliminary steps:

1. Take into account that the junction capacitances c_{bc} and c_{be} have singularities at vbcp = vjc and vbe = vje, respectively.
2. Compute the charge stored in the junction capacitances as a function of the junction voltage.

Both steps are contained in the context of expressing a generic junction capacitance:

$c_j = c_{jo} / (1 - v / v_j)^m$

According to the book titled Modeling the Bipolar Transistor, by Getreu, I. (Tektronix, Inc. 1976), this is an empirical equation that is not valid for forward bias (v > 0). Under forward bias conditions, the diffusion capacitances dominate, so that a simple approximation of the junction capacitances for v > 0 is usually sufficient. Although this transistor model does not include diffusion capacitances, it uses an approach similar to that in the Getreu book. That is, it assumes that, for v > 0, the junction capacitance depends linearly on the junction voltage. Thus, it has a slope that results in matching slopes at v = 0. This leads to the following equation set for junction capacitance:

$c_j = c_{jo} / (1 - v/v_j)^{m}$ for v < 0

$c_j = c_{jo} * (1 + m*v/v_j)$for v Š 0

The second step consists of computing the charge stored in the junction capacitor as a function of the junction voltage. This is given as the integral of the capacitance over junction voltage, v:

$q_j(v) = \int_0^v c_j(v)\, dv$

with the additional requirement that qj(0) = 0. The results, for reverse and forward bias of the junction, respectively, are:

$q_j = c_j*v_j*(1 - (1 - v/v_j)^{1-m})/(1-m)$for v < 0

$q_j = c_j*v*(1 + 0.5*m*v/v_j)$for v Š 0

Note that q_j is continuous at v = 0 in these equations.

Header Declarations

The bipolar junction transistor (BJT) is an electrical device with three terminals: collector (c), base (b), and emitter (e). Eleven parameters characterize the transistor model. Nine of them are the ones listed in the topic titled "Basic Model Equations". The other two are the transistor type (type) and the collector resistance (rc).

The following example shows the template header and header declarations for the bjt template:

```
1  element template bjt c b e = model, ic
2    electrical c, b, e
3    struc {
4      enum {_n, _p} type
5      number  is=1e-16, bf=100, br=1, \
6              cje=0, vje=.75, mje=.33, \
7              cjc=0, vjc=.75, mjc=.33, rc=0
8    } model = ()
9    number ic[2]=[undef,undef]
10   external number temp
```

The following topics describe the bjt template header declarations in more detail:

- Transistor Type
- Collector Resistance
- Initial Conditions

Transistor Type

There are only two possible values for transistor type—NPN or PNP. For parameters with a limited set of possible values, the MAST language provides the enumerated parameter type (enum), which has the following syntax:

```
enum {evalue [, evalue]} name[[=init_val], name[=init_val...]]
```

where evalue is a comma-separated list of values that the name parameter can assume. If present, init_val must be one of the values in evalue. For the transistor type, there are only two choices, which are represented as _n (for NPN) and _p (for PNP). Therefore, you can define an argument that allows selecting the transistor type as follows:

```
4      enum {_n, _p} type
```

Collector Resistance

Rather than having each model parameter as an individual argument of the bjt template, it is preferable to group them into a single structure argument and give this structure argument the name model. The parameter for collector resistance (rc) has been included in model as follows:

```
5       number  is=1e-16, bf=100, br=1, \
6               cje=0, vje=.75, mje=.33, \
7               cjc=0, vjc=.75, mjc=.33, rc=0
8     } model = ()
```

Using the model structure emphasizes the fact that the parameters it contains belong together. It is their combination that characterizes a particular transistor instance. It also allows you to refer to such a combination by a single name (model), which makes it easy to use the same group of parameters for several transistor instances.

In the declaration, initialize each structure member to a typical value. The initialized value is then that member's default value. The exceptions are the transistor type, for which there is no reasonable default, and the two junction capacitance values (cjc and cje), which are initialized to 0 so that, by default, the model does not include charge effects.

Each individual parameter of the structure model is referenced later using the format model->cje to reference the cje parameter or model->vje to reference the vje parameter and so on.

Initial Conditions

You can declare an argument that specifies the initial value for the voltage across the pn junctions of the transistor. However, a BJT has two junctions of interest: the base-emitter junction and the collector- emitter junction. Thus, the argument for initial conditions needs to specify initial values for base-emitter voltage (vbe) and collector-emitter voltage (vce). This requires that the ic argument be declared as a two-dimensional array as follows:

```
9   number ic[2]=[undef,undef]
```

This declaration is declared outside the model structure (lines 5 through 8), because it isn't really associated with the characterization of the BJT model (although it would work just as well if it were included within model).

In addition there needs to be initial_condition statements in the control section to associates vbe and vce with the ic argument.

The following external parameter, temp, makes the system temperature available in the template.

```
external number temp
```

Local Parameters

Because of the complexity of the BJT model, there are several intermediate calculations that must be performed prior to using the template equation as shown below:

```
  # declare local param., vals, and extraction groups
number k = 1.381e-23,             # Boltzmann's constant
       qe = 1.602e-19,            # electron charge
       vt,
       qbe0, qbc0, vje0, vjc0
struc {
  number bp, inc;
} nv[*] = [(0,.1),(2,0)]
val v vbc, vbe, vce               # declarations of vals
val i iec, icc, iba, ico, ir
val q qbc, qbe
electrical cp                     # local node
group {vbc,vbe} v                 # extraction groups
group {iba,ico,ir} i
group {qbc,qbe} q
parameters {
  # calculate thermal volts and functions of model param.
  vt = k * (temp + 273.15) / qe
  qbe0 = model->cje * model->vje / (1 - model->mje)
  qbc0 = model->cjc * model->vjc / (1 - model->mjc)
  vje0 = 2 * model->vje / model->mje
  vjc0 = 2 * model->vjc / model->mjc
}                      # end of parameters section
```

As a result, there are also several local parameters and variables that must be declared for use in these calculations, which are explained in the following topics:

- Temperature
- Junction Capacitance

- Newton Steps Declaration - MAST bjt Template
- Local Node - MAST bjt Template
- Intermediate Current and Charge Variables
- Defining Groups For Extraction - MAST bjt Template

Temperature

The following local declarations are for parameters used in calculations for thermal voltage:

```
13  number k = 1.381e-23,            # Boltzmann's constant
14         qe = 1.602e-19,          # electron charge
15         vt,
```

Junction Capacitance

The declarations below are for parameters used in calculations for junction charges:

```
16    qbe0, qbc0, vje0, vjc0    #Part of number declar.
```

where qbe0 and qbc0 are used to calculate junction charges under reverse bias conditions; vje0 and vjc0 are used for forward bias conditions.

Newton Steps Declaration - MAST bjt Template

The newton steps parameter (nv) is nearly identical to that of the diode template (see the topic titled "Modeling an Ideal Diode with MAST"):

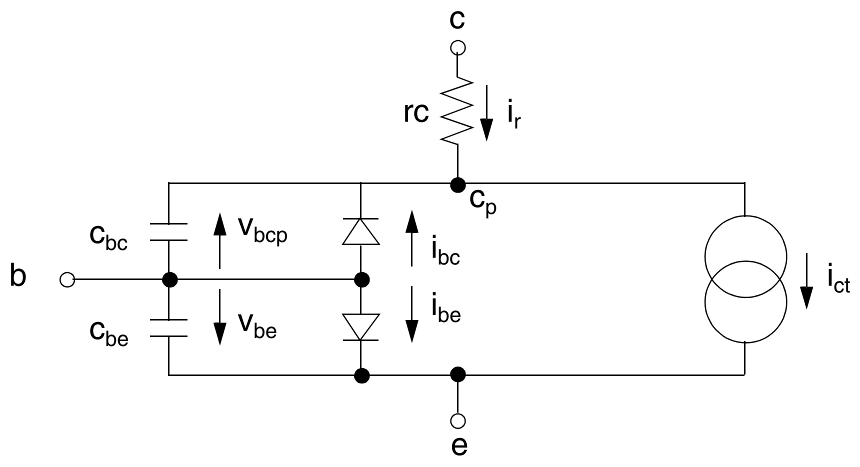
```
17  struc {number bp, inc;}\
18    nv[*] = [(0,0.1),(2,0)]
```

In the control section, line 88 assigns these newton step values to two independent variables (vbc and vbe) as follows:

```
88    newton_step((vbc,vbe), nv)
```

Local Node - MAST bjt Template

As shown in the following figure, this model allows you to specify collector series resistance (rc).



Ebers-Moll Model of an NPN transistor

For non-zero values of rc, the topology of the model changes, which requires an internal node called cp as follows:

```
23  electrical cp
```

This is the point at which this series resistance connects between the external collector (c) and the interior of the model.

You can specify that this node be collapsed to the external collector (meaning that it no longer exists) if rc=0 (see line 86).

Intermediate Current and Charge Variables

The variables listed below are declared as val variables and are used in calculating currents and charges.

```
21  val i iec, icc, iba, ico, ir
22  val q qbc, qbe
```

Defining Groups For Extraction - MAST bjt Template

Any val variable can be made available for post-processing using the extract command, along with any system variable (pin, var, and ref variables). It is sometimes useful to extract only currents or voltages or, in general, any convenient collection of val variables and system variables. You can do this by defining the collection as a group in the local declarations of the template. Once a group is defined, you can refer to all members of the group by the name of the group.

The following statement is the general form for defining a group of variables.

```
group {variable_list} name
```

where:

variable_list	is a comma-separated list of val variables and system variables.
name	is the name of the group and makes all variables in variable_list available by referring to name.

The bjt template contains three groups for holding voltages, currents, and charges:

```
group {vbc, vbe}   v
group {iba, ico, ir}   i
group {qbc, qbe}   q
```

For example, a netlist could contain the following netlist entries:

```
bjt.1 a b c = model=(type=_n)
bjt.2 d e f = model=(type=_p)
```

You could then use the Saber command shown below to extract all val variables and system variables in bjt.1, but only the currents defined in group i from bjt.2 (i.e., ib, ic, ir):

```
extract bjt.1/* bjt.2/i
```

Thermal Voltage

Thermal voltage (vt) is calculated by the following statement in the parameters section:

```
vt = k * (temp + 273.15) / qe
```

Junction Capacitance

The charges at the collector-base and the emitter-base junctions under reverse and forward bias conditions are calculated as follows in the parameters section:

```
qbe0 = model->cje * model->vje / (1 - model->mje)
qbc0 = model->cjc * model->vjc / (1 - model->mjc)
vje0 = 2 * model->vje / model->mje
vjc0 = 2 * model->vjc / model->mjc
```

Note:

Normally, some parameter checking would be provided at this point. To keep the example short, this has not been included in this example.

Intermediate Calculations

Because of the complexity of the BJT model, there are several intermediate calculations that must be performed prior to using the template equation. These calculations are located in the following values section:

```
35  values {
36     # calculate basic quantities of npn and pnp trans.
37    vbc = v(b) - v(cp)
38    vbe = v(b) - v(e)
39    vce = v(cp) - v(e)
40    if (model->type == _n) {
        ### lines 41 through 47
48     # calculate base, collector, and resistor currents
        ### lines 49 through 53
54     # calculate charges
55    if(model->type = = _n) {
        ### lines 56 through 82
83  }                             #end values section
```

The calculations used in the values section use the locally declared parameters and variables.

The following topics describe the different computational blocks shown in the previous code example:

- Fundamental Quantities - MAST bjt Template
- Currents
- Charges

Fundamental Quantities - MAST bjt Template

The fundamental quantities of the model, upon which all currents and charges are based, are the base-emitter and base-collector voltages and the currents iec and icc. Their definitions are straightforward, although dependent upon transistor type:

```
36     # calculate basic quantities of npn and pnp trans.
37    vbc = v(b) - v(cp)
38    vbe = v(b) - v(e)
40    if (model->type == _n) {     # If type = NPN
41      iec = model->is * (limexp(vbc/vt) - 1)
42      icc = model->is * (limexp(vbe/vt) - 1)
43    }
44    else {                       # If type = PNP
45      iec = -model->is * (limexp(-vbc/vt) - 1)
46      icc = -model->is * (limexp(-vbe/vt) - 1)
47    }
```

Lines 40 through 43 for an NPN type shows that in order to determine the base-collector voltage, you have to use the internal collector cp, rather than pin c. As with the diode template, you use limexp, a MAST language function, instead of exp, to compute the two currents. The reason is for protection against overflow.

Lines 44 through 47 provide the same function for the PNP type with negated values of voltage and current.

Currents

The currents in the bipolar transistor model are ict, ibc, ibe, and ir. These could easily be computed, but it is preferable to define currents that have a physical meaning and therefore are useful for extraction. The values of all val variables can be made available for post-processing using the extract command.

```
48     # calculate base, collector, and resistor currents
49    iba = iec / model->br + icc / model->bf
50    ico = icc - iec - iec / model->br
51    if (model->rc ~= 0) ir = (v(c) - v(cp)) / model->rc
52    else                ir = 0
```

Although you could define the emitter current, it is not necessary. Its value is given by ie = -iba - ico.

The definition of ir is meaningful only if the collector resistance (rc) is non-zero. For rc=0, it would be good to express ir as the sum of ic and d(qbc)/dt, but the MAST language allows usage of the d_by_dt operator only in a template equation. Therefore, set ir=0 and write the rest of the template such that the value of ir is not needed if rc=0.

You could make ir available independent of the value of rc. This would be done by declaring ir as a var variable and letting the simulator determine its value. For efficiency, this example does not do so.

Charges

The remaining intermediate calculations define the charges stored in the two junctions. This is a direct translation of the charge equations into the MAST language, except that it uses parameters as they are defined in the template as follows:

```
 # calculate charges
if(model->type == _n) {
  if (vbc<0) {
    qbc = qbc0*(1-(1-vbc/model->vjc)**(1-model->mjc)
  }
  else {
    qbc = model->cjc*vbc*(1 + vbc/vjc0)
  }
  if (vbe<0) {
    qbe = qbe0*(1-(1-vbe/model->vje)**(1-model->mje)
  }
  else {
    qbe = model->cje*vbe*(1 + vbe/vje0)
  }
}                  # end “if type _n” condition
else {             # if model is not of type _n
  if(vbc > 0) {
    qbc = -qbc0*(1-((1+vbc/model->vjc)**(1-model->mjc)))
  }
  else {
    qbc = model->cjc*vbc*(1-vbc/vjc0)
  }
  if(vbe > 0) {
    qbe = -qbe0*(1-((1+vbe/model->vje)**(1-model->mje)))
  }
  else {
    qbe = model->cje*vbe*(1-vbe/vje0)
  }
}                 # end “if not type _n” condition
```

Control Section

The following lines comprise the bjt template control section:

```
control_section {
  # if no collector resistance, collapse nodes c and cp
  if (model->rc == 0) collapse(c,cp)
   # specify Newton steps
  newton_step((vbc,vbe),nv)
   # initial conditions and start value
  initial_condition(vbe, ic[1])
  initial_condition(vce, ic[2])
  start_value (vbe, 0.6)

   # small-signal parameters
  device_type("bjt", "example")
  small_signal(ibase,current,"base current",iba)
  small_signal(icoll,current,"collector current",ico)
  small_signal(vbe,voltage,"base-emitter voltage", vbe)
  small_signal(vbc,voltage,"base-collector voltage", vbc)
  small_signal(rc,resistance,"collector resistance",\
               model->rc)
}                      # end control_section
```

These lines are further described in the following topics:

- Collapse Node
- Newton Steps
- Initial Conditions in Control Section
- Starting Value
- Small-Signal Parameters

Collapse Node

As described for local declarations, cp is declared as an internal node. When rc=0, there is no resistance between c and cp; thus, they actually refer to the same node. You can indicate this to the simulator by using a collapse statement in the control section as follows:

```
# if no collector resistance, collapse nodes c and cp
if (model->rc == 0) collapse(c,cp)
```

Collapsing nodes has the advantage that it reduces the size of the system, that is, the number of system variables. This is particularly valuable in systems where there are numerous instances of a template, because reducing system size typically increases simulation speed. However, a disadvantage of collapsing nodes in this example is that using the alter command to change the collector resistance from zero to nonzero (or vice-versa) would alter the topology of the system. This is not allowed after starting simulation.

Newton Steps

For the control section, you need to identify the independent variables of the nonlinear equations, in order to specify newton steps for them. Referring back to the topic titled "Basic Model Equations", the two voltages vbcp and vbe (vbc and vbe in the template) fit the requirements. All the nonlinear quantities ultimately depend upon vbc or vbe or both, and both vbc and vbe are expressed as the difference of two system variables. Therefore, both vbc and vbe require newton steps.

Because these voltages are used identically in the model equations, you can use the same newton step values for both. Further, because both the base-emitter and base-collector junctions are modeled as diode junctions, you can use the same newton step values as for the diode example.

The control section statement that associates the arrays of newton step values with the independent variables is as follows:

```
newton_step((vbc,vbe), nv)
```

Initial Conditions in Control Section

One initial_condition statement in the control section is required for each variable (vbe and vce) whose initial value is specified by the ic argument, which is declared as a two-dimensional array:

```
initial_condition (vbe,ic[1])
initial_condition (vce,ic[2])
```

Starting Value

You can use the following start_value statement to specify the forward bias value (vbe) of the base-emitter junction at the first iteration of the DC iteration. This puts the junction into its conducting region at a value that is closer to the solution you expect (around 0.6V).

```
start_value (vbe, 0.6)
```

Small-Signal Parameters

You can insert statements into the control section that allow you to list the values of a set of small-signal parameters (SSP) when using the ssp command. The simulator obtains these values by linearizing the model at a given operating point, usually by taking the partial derivative of a dependent variable with respect to an independent variable. The ssp command reports small-signal parameter values for the linearized model only at the operating point—you cannot plot these values. For the complete specification, you need to run a DC analysis, which gives additional DC operating point information such as node voltages and branch currents.

For example, some of the parameters that might be reported for the bjt template are listed below.

Parameter	Name	Classification	Value
collector resistance	rc	resistance	170
base-emitter voltage	vbe	voltage	0.672
base current	iba	current	0.176u

You can specify a small-signal parameter for a template by using the device_type and small_signal statements in the control section. These are explained below.

Note:

Because of the simplicity of this transistor model, there are not many small-signal dependencies that can take full advantage of these statements (for example, using an ss_partial statement).

device_type This statement is inserted into the template to provide an identifier in the SSP report; it has no effect on determining the SSP values:

```
device_type("bjt", "example")
```

small_signal One small_signal statement is required to define each SSP. This statement can have either four or five fields that define the SSP characteristics, as explained in the topic titled "Modeling an Ideal Diode with MAST". In either case, the first three fields are the same.

The following 4-field small_signal statements appear in the bjt template:

```
small_signal(ibase,current,"base current",iba)
small_signal(icoll,current,"collector current",ico)
small_signal(vbe,voltage,"base-emitter voltage", vbe)
small_signal(vbc,voltage,"base-collector voltage", vbc)
small_signal(rc,resistance,"collector resistance",\
             model->rc)
```

Equations Section

The template equations list the branch currents and express them as functions of the intermediate variables previously calculated:

```
equations {
   # current at base, internal collector, and emitter
  i(b->e)  += iba + d_by_dt(qbe)
  i(b->cp) += d_by_dt(qbc)
  i(cp->e) += ico

   # current at collector resistor, if present
  if (model->rc ~= 0) i(c->cp) += ir
}                         # end equations section
```

These equations show the following:

- How to use an if statement to set up equations differently for different parameter values. The condition part of such an if statement can include only parameters.
- It is preferable to write the template such that ir is needed only if the collector resistance (rc) is non-zero, as shown by the if statement. If rc=0, the internal and external collector nodes (cp and c) are collapsed to the same node.

13

Modeling Nonlinearities

This topic is divided into the following subtopics:

- Simulation Techniques for Evaluating Nonlinearities
- Modeling a Voltage Squarer - MAST vsqr Template -- describes the following concepts:
  - Using the control section of the template to pass certain information that is not part of the model to the simulator
  - Using sample points for the independent variables of nonlinear equations, including their selection and specification
  - The effect of the simulator's density variable on the sample points
  - Using arrays of structures and their initialization
  - Default sample points automatically provided by the Saber Simulator.

Simulation Techniques for Evaluating Nonlinearities

To illustrate the problems that arise in modeling a nonlinear element, consider the characteristic equation of a voltage squaring block:

vout = vin * vin

In finding the solution of nonlinear networks such as those containing squaring blocks, the simulator must solve a set of nonlinear, simultaneous equations. There are no techniques that do so directly, so the simulator uses the following method:

1. Guess a set of values for all unknowns.
2. Linearize each nonlinearity about these values, thereby obtaining a set of simultaneous linear equations.
3. Solve the linear equations (using well-known techniques).
4. Update the values of all unknowns using the solution of the linear equations.
5. Repeat steps 2 through 4 until the correct solution has been obtained.

This algorithm reveals two important concepts:

- Simultaneous nonlinear equations are solved iteratively
- The iterative method involves linearization

These are not independent of each other.

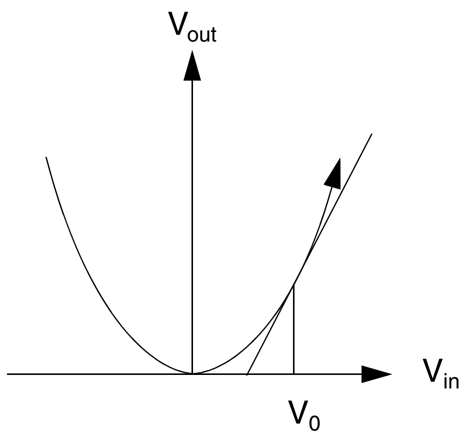
Simulation Linearization Techniques

There are several techniques for the linearization of nonlinear equations. Of these, three used by different simulators are described as follows:

1. Taking the slope of the characteristic equation at the present value
2. Using a piecewise linear approximation of the characteristic equation
3. Using a piecewise linear evaluation of the characteristic equation (used by the Saber Simulator)

Taking the Slope (Method 1)

A common technique for linearization is to take the slope of the characteristic equation (i.e., its first derivative) at the present value of the independent variable. Some simulators use this technique, which is shown in the following figure.

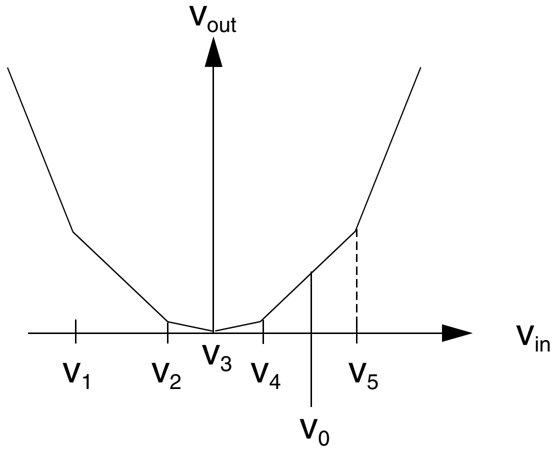


Linearization by taking the slope—Method 1

Simulators using this technique typically require a model to include both the characteristic equations of the element and their derivatives with respect to the independent variables. Moreover, both the model equations and their derivatives must be continuous functions of the independent variables. In particular, the requirement for continuous derivatives makes modeling of such characteristics very difficult, if they are described by different functions in different regions of the independent variables (as in the case of MOS devices). Simulators using this approach find, for the nonlinear equations, an approximate solution that is controlled by one or more convergence parameters.

Piecewise linear approximation (Method 2)

Another linearization technique is piecewise linear approximation. Rather than describing a nonlinear characteristic exactly, the model consists of a set of straight lines approximating the nonlinear equation, as shown in the following figure.



Piecewise linear approximation—Method 2

The piecewise linear approximation of the characteristic equations is obtained before the simulation begins, so all the simulator “sees” is the piecewise linear model, which must be continuous, but obviously has discontinuous derivatives. This model is solved exactly using special algorithms. However, the solution is only as accurate as the piecewise linear approximations, and you can change the accuracy only by changing the model.

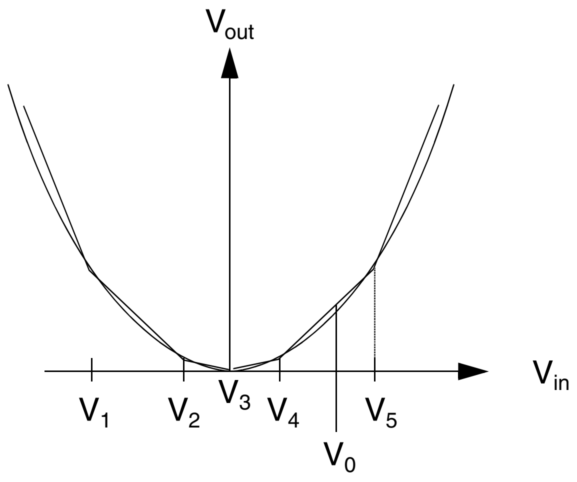
Piecewise Linear Evaluation (Method 3)

A third linearization technique, the one used by the Saber Simulator, is called piecewise linear evaluation. The model consists of the nonlinear equations plus a set of sample points for the independent variables. The simulator uses the sample points (which may be specified in the template) to find a piecewise linear approximation of the nonlinear equations, as shown in the figure below, where v1...v5 are the sample points.

Note:

The Saber Simulator automatically uses default sample points for any independent variables of a nonlinear template that require them. Consequently, all the information on sample points in this chapter is optional—it is necessary only if you want to change the values of sample points from the default. See the topic titled "Default Sample Points".

The simulator then solves the piecewise linear approximation of the model exactly, using specialized algorithms. Again the accuracy of the solution depends upon the piecewise linear approximation, but the main advantage of this approach is that the density of the sample points is easily changed. The density variable of any given analysis is a multiplier for the sample points of all templates that have them specified (default density is 1). This enables the user to choose (at run time) either increased accuracy or faster computation time. For more information on density, refer to the Calibrating DC Analysis topic in the Saber online documentation.



Piecewise linear evaluation—Method 3

Given that the slope technique cannot solve the equations exactly, this technique can produce results as accurate as those produced by the slope technique, if the density of the sample points is sufficiently high.

Comparison and Summary of Linearization Techniques

These three linearization techniques can be summarized as follows:

- slope technique requires a continuous model with continuous first-order derivatives. The simulator finds an approximate solution of the linearized model, where accuracy is controlled by convergence parameters.
- piecewise linear approximation technique requires a model consisting of continuous piecewise linear segments. The simulator finds an exact solution of the piecewise linear model, where accuracy can be changed only by changing the model.
- piecewise linear evaluation technique used by the Saber Simulator requires a continuous model and a set of sample points. The simulator finds an exact solution of the piecewise linear approximation specified by the sample points, where accuracy can be changed by changing the density of the sample points.

From this, then, a nonlinear model implemented in a MAST template must include the following information:

- The nonlinear equations describing the model
- A set of sample points for each independent variable in the model

Modeling a Voltage Squarer - MAST vsqr Template

This example shows a nonlinear model that is a simple voltage squarer template, vsqr, whose output voltage is the square of its input voltage. Note that this is implemented as an element template.

```
element template vsqr ip im op om
  electrical ip, im, op, om     # header declarations
{
  var i iout                    # local declarations
  val v vin, vout
                                # sample points defined
  struc {number bp, inc;} svin[*]=\
        [(-100k,1),(-1k,.1),(10,.01),\
         (0,.01),(10,.1),(1k,1),(100k,0)]
  values {
    vin  = v(ip) - v(im)        # input voltage
    vout = v(op) - v(om)        # output voltage
  }                            # end of values section
  control_section {
    sample_points(vin, svin)    # sample points associated
                                #  with input voltage
  }                            # end of control section
  equations {
    i(op->om) += iout           # current contribution
    iout:  vout = vin * vin
  }                            # end of equations section
}                              # end of template body
```

vsqr Template Topics

The following topics describe the vsqr template:

- Template Header
- Values Section

- Equations Section
- Control Section
- Understanding Sample Points
- Specifying Sample Points
- Density of Sample Points
- Default Sample Points

Template Header

The vsqr template has two input pins and two output pins, but no arguments. The header and its corresponding declarations are:

```
element template vsqr ip im op om
  electrical ip, im, op, om
```

Values Section

The values section declares vin and vout, which are used in the equations section as follows:

```
values {
  vin = v(ip) - v(im)       # input voltage
  vout = v(op) - v(om)      # output voltage
}                          # end of values section
```

Both vin and vout are declared as a val variable as follows:

```
val v vin, vout
```

Equations Section

The characteristic equation of the voltage squarer finds the voltage across the output pins (vout) in terms of the voltage across the input pins (vin). The equations section appears as follows:

```
equations {
  i(op->om) += iout    # current contribution
  iout:  vout = vin * vin
}
```

The equations section implements the voltage squarer as a nonlinear voltage-controlled voltage source as follows:

```
iout:  vout = vin * vin
```

The output current iout, contributes to the current at pins op and om as follows:

```
i(op->om) += iout
```

The simulator determines iout such that the output voltage is the square of the input voltage.

At this point, the voltage squarer template is complete, unless you want to specify sample point values for the independent variable (vin) that are different from the values specified in the template (see the topic titled "Specifying Sample Points").

The Saber Simulator automatically uses default sample points for any independent variables of a nonlinear template that require them. Consequently, all the information on sample points in this chapter is optional—it is necessary only if you want to specify values of sample points that differ from these automatically-supplied default values (see the topic titled "Default Sample Points").

Control Section

The control section of a template provides the simulator with information that is specific to the system being analyzed but is not directly a part of the model. An example of such information is the sample points required for the independent variables in nonlinear equations.

The control section consists of the keyword control_section, followed by a sequence of control section statements, enclosed between braces ({}). Such statements are special in the sense that they can occur only in the control section. A complete list of these statements is given in the MAST Reference Manual. Here, only a sample points statement for the independent variable of a nonlinear equation is of interest.

The sample points statement inserted in this section takes the following form:

```
sample_points (variable, sa_points)
```

where:

variable	is a branch variable, a var variable, or a val variable that is equal to either a system variable or a difference of two system variables (i.e., var1 - var2)
sa_points	is the name of an array of sample points —note that the actual sample point values are specified in an array declared as a local parameter

For the voltage squarer template, you have to specify sample points for the input voltage (vin), which is the independent variable in the characteristic equation.

With svin as the name of the array containing the sample points, the control section for this template is as follows:

```
control_section {
  sample_points(vin, svin) # sample points associated
                            #  with input voltage
}                          # end of control section
```

The actual values of the sample points (within svin) are specified as local parameters, as described in the topic titled "Specifying Sample Points".

Understanding Sample Points

The specification of sample points for an independent variable of a nonlinear equation consists of the following two parts:

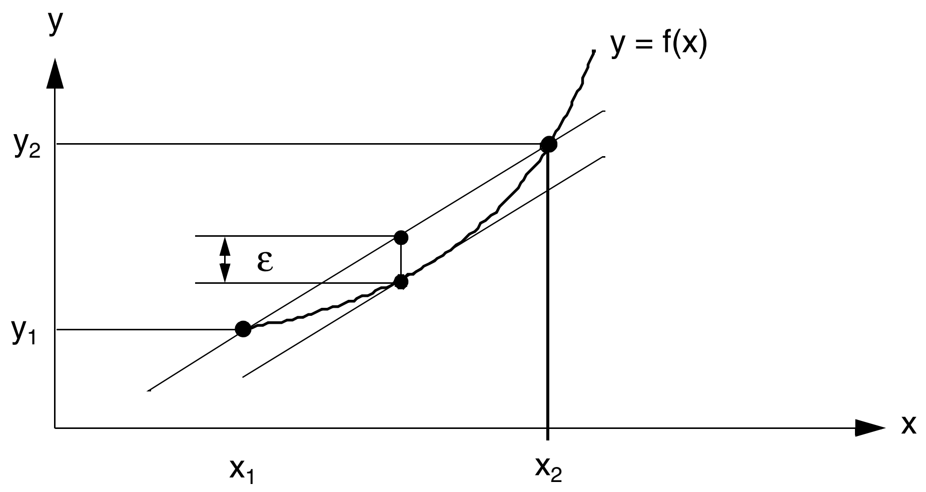
1. Considerations for selecting sample points (described in the next topic)
    - General Approach
    - Specific Approach (voltage squarer)
2. The actual specification of sample point values in a template, using MAST language constructs (described in the topic titled "Specifying Sample Points").

Considerations for Selecting Sample Points

There are several considerations for selecting sample points:

- Accuracy vs. speed. Denser sample points provide better accuracy of the piecewise linear approximation of the nonlinearity, but this is usually accompanied by slower simulation speed.
- Optimum combination of accuracy and speed. The Saber Simulator lets you change the density of the sample points at run time by means of the density variable of the analysis you are running (see the topic titled "Density of Sample Points"). You should specify values for sample points such that the accuracy of the piecewise linear approximation is sufficient for the default simulator density of 1 (which means density has no effect on sample points). Of course, the meaning of “sufficient accuracy” depends upon the application.
- Operation limits. These are the minimum and maximum values that the independent variable is not supposed to exceed—if it does, the Saber Simulator reports a warning and sets the variable to the limit value.
- Intended region of operation of the model. This is a region inside the operation limits where the model is intended to be used. Typically, you want better accuracy inside this region than outside.
- Numerical considerations. The independent variable may be restricted to a certain value range by the laws of physics, but during iterations it may assume values outside this range. (For example, absolute temperature may become negative during iterations.)
- Other requirements. The Saber Simulator requires that 0 (zero) be a sample point.

The accuracy of the piecewise linear evaluation is demonstrated in the following figure.



Accuracy of a piecewise linear approximation

This figure shows a nonlinear function y=f(x) and a linear approximation that intersects the function at x1 and x2. That is, x1 and x2 are sample points of y=f(x). Further, y1 and y2 are the function values at x1 and x2, respectively, and e is the maximum error of the linear approximation of f(x) between x1 and x2. In order to find approximate sample points for x, you need to express e as a function of Dx=x1-x2. If there were such a function, its inverse would yield the sample point spacing for a given maximum error. In general, such a function is difficult to derive, except in the very simplest cases, such as this voltage squarer. Therefore, a general approach is described first, then a specific approach as it is applied to the voltage squarer.

General Approach The preceding figure shows that e is always smaller than Dy=y2-y1 if the sign of the slope of f(x) does not change between x1 and x2.

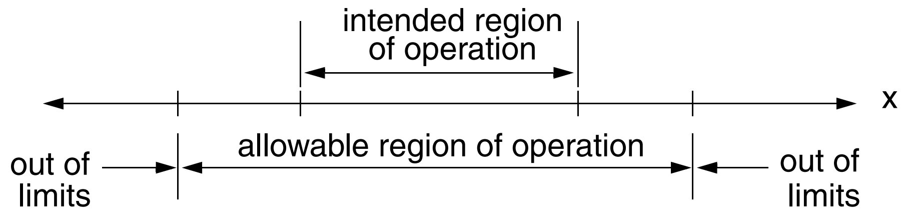
Therefore, if f(x) is monotonic between x1 and x2, Dy is an upper bound for the approximation error. Because simulation always involves a trade-off between accuracy and speed, you will expect a certain accuracy of simulation results (such as three digits of relative accuracy or an absolute accuracy of 10^{-3}). In both cases, you actually specify a minimum resolution or “granularity” for the simulator results, and you expect Dy to be smaller than this level of resolution.

Next, find Dx as a function of Dy. In many cases, it is possible to do this using the inverse relationship $x=f^{-1}$ (y). In more complicated cases (particularly with multi-dimensional nonlinearities), selecting the sample points may be a process of trial and error. Keep in mind the concept of resolution or “granularity” as an aid for quickly finding a reasonable set of sample points. Remember too, that Dy might be a pessimistic upper bound (i.e., too large) for the approximation error, depending upon the nonlinearity.

In the case where f(x) is non-monotonic between x1 and x2, divide Dx into smaller parts, such that f(x) is monotonic throughout each part. In practice, this further division is not critical, because the monotonicity of f(x) is required only to ensure that e is smaller than Dy. Therefore, if Dx is sufficiently small, Dy is a good upper bound for the approximation error.

Specific Approach (voltage squarer) For vsqr, use the direct approach. Note that $e=Dx^2$, which indicates that the (absolute) error of the piecewise linear approximation depends only on the distance between two sample points. Therefore, equal-spaced sample points yield constant approximation error. Similarly, if the relative error should be constant, Dx should be proportional to ÷x. However, as described earlier, the approximation error is not the only thing to consider when selecting sample points.

The value range of each independent variable consists of several parts, as shown in the figure below. You must specify sample points for the entire allowable region, but typically you want better accuracy (and therefore denser sample points) in the intended region of operation. Sometimes it is desirable to have different accuracy in several different regions in order to get better results when x is closer to the intended region of operation. For vsqr, limit the allowable region of operation to ±100 kV. Select sample points separated by 1 V near the outer limits of the allowable region and closer together near 0 V, which gives a relative approximation error that is fairly constant.

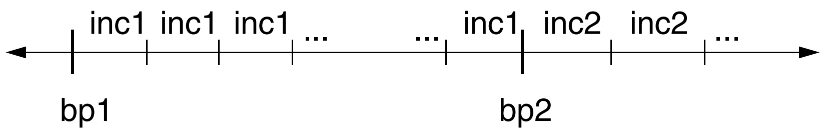


Value range of an independent variable

Taking all these points into account, it is clear that there is no universal algorithm for selecting sample points. Rather, the selection depends upon various trade-offs and often requires some experimentation.

Specifying Sample Points

The control section of a template associates sample points with an independent variable by a sample_points(variable, sa_points) statement. The sa_points variable is an array that describes the distribution of sample points. Rather than requiring you to specify each sample point individually, the sa_points array describes the sample point distribution by means of a collection of breakpoint (bp) and increment (inc) pairs, which the simulator interprets as shown in the following figure.



Specifying sample points using breakpoints and increments

Each breakpoint (bp) marks the left end of a region of x. The actual sample points throughout that region are separated by the distance inc.

Each (bp, inc) pair is kept in a structure, so sa_points becomes an array of structures. For vsqr, the local declarations becomes:

```
struc {number bp, inc;} svin[*]
```

which declares svin to be an array of structures, where the array is of undetermined length, and each structure holds the two values bp and inc. The semicolon (;) is required. It introduces a logical end-of-line so that the right brace is (syntactically) at the beginning of a new line, as required.

Sample Point Statement Syntax

The syntax for specifying a sample point statement in the control section is as follows:

```
sample_points (vin,svin)
```

where svin is an array of (bp, inc) pairs that are specified as one of the following:

- A local parameter
- An argument (in the header declarations)
- A control section statement (within the sample_points statement itself)

The values in svin can be defined either by means of an initializer in the declaration (used here) or separately, as described for newton steps in the topic titled “Modeling a Simple Voltage Limiter with MAST” in a previous chapter. As suggested in the topic titled “Understanding Sample Points”, svin has at least three entries:

- Two breakpoints, at 100kV and -100kV, to mark the allowable range of operation
- A breakpoint at 0, as required by the Saber Simulator

Thus, you can declare and initialize the sample points as shown below. Note that the backslash (\) indicates that the next line is to be a continuation of the current line.

```
struc {number bp, inc;} svin[*]=\
  [(-100k,1),(-1k,.1),(10,.01),(0,.01),\
   (10,.1),(1k,1),(100k,0)]
```

Sample Point Values

Select four additional breakpoints for vin, at ±1kV and ±10V, with increment values that start at 1 near the operational limits and decrease to 0.01 near 0. This provides a relative approximation error that is almost constant throughout the specified value range. Following the requirement that the breakpoints be listed in increasing order, define svin as follows:

```
[(-100k,1),(-1k,.1),(-10,.01),(0,.01),\
 (10,.1),(1k,1),(100k,0)]
```

Each pair of values between parentheses, (bp, inc), is one array element. These values have the following meanings:

- From -100k to -1k, and from 1k to 100k, the spacing between sample points is 1
- From -1k to -10, and from 10 to 1k, the spacing is 0.1
- From -10 to 10, the spacing is 0.01
- The first and last breakpoints determine the allowable region of operation
- The last breakpoint marks the beginning of the right-hand “off-limits” area, so the associated increment, while syntactically required, is not used (i.e., the number of breakpoints is always one greater than the number of intervals defined)

Note:

Increment values must be non-negative.

Density of Sample Points

The Saber Simulator lets you increase the density of the sample points, that is, reduce the size of the spaces separating them, in order to get better accuracy. Referring to the figure above, running a simulation with the density variable set to 1 (the default) will cause the simulator to sample the independent variable at the sample points specified in the template. Using a different density has no effect on the breakpoints, but the increments between breakpoints are divided by the specified density. Thus, with a density of 2, the spacing between sample points is half that specified in the template, and with a density of 0.2, the spacing is five times the default.

Specifying a density greater than 1 typically increases both simulation time and simulation accuracy. Similarly, specifying a smaller density typically decreases both simulation time and simulation accuracy.

The simulator automatically limits the number of sample points between any two consecutive breakpoints to 2^{31} -1, regardless of the selected density.

Default Sample Points

For any nonlinear template, the Saber Simulator uses the default breakpoints and increments listed below for the sample points of all independent variables that require them. There are two ways to express them, either with multiplier prefixes (shown first, below) or in scientific notation (shown second, below).

With multiplier prefixes:

```
[(-1t,1meg), (-1g,1k), (-1meg,1), (-1k,1m),\
  (1,1u),(1m,1n), (-1u,1p), (0,1p), (1u,1n),\
  (1m,1u), (1,1m),(1k,1), (1meg,1k), (1g,1meg),\
  (1t,0)]
```

In scientific notation:

```
[(-1d12,1d6), (-1d9,1d3), (-1d6,1d0), \
  (-1d3,1d-3), (-1d0,1d-6), (-1d-3,1d-9), \
  (-1d-6,1d-12), (0d0,1d-12), (1d-6,1d 9), \
  (1d-3,1d-6), (1d0,1d-3), (1d3,1d0),\
  (1d6,1d3), (1d9,1d6), (1d12,0d0)]
```

14

MAST Functions

It is sometimes useful to implement portions of a model as separate functions. These functions reside in separate files outside of the template, but they are called (invoked) from within the bjtm template.

There are three major reasons for using MAST functions:

1. You can modularize MAST code for readability and maintainability.
2. You can encapsulate MAST code for re-use (for example, use the same MAST function call in a diode template and in a bjt template).
3. You can easily convert code from a foreign routine to a MAST function (for example, convert a SPICE3 MOS model from a C routine to a MAST function).

Using a MAST Function Instead of a Foreign Routine

It is possible to call a foreign routine (such as one written in C or FORTRAN) from a template. However, it is recommended to use the MAST functions shown in the bjtm template instead whenever possible. Using a MAST function has the following advantages over using a foreign routine:

- The interface between function and template is substantially easier and less error-prone.
- A MAST function is more easily debugged. You can use message () statements, or you can pass signals back to the template for plotting.
- Porting to different computers is no longer required—the code is written in the MAST language, which is executed on every machine by the Saber Simulator.
- The Saber Simulator interprets a MAST function more readily than it does a foreign function, which generally results in greater simulation efficiency.

Modeling the Bipolar Transistor Using MAST Functions

The bjtm template shows how MAST functions can perform some of the calculations that are included within the bjt template. The MAST functions (residing in separate files) perform the intermediate calculations and return the results to the template. The combination of the bjtm template and the MAST functions have the same functionality as the original bjt template.

This topic is divided into the following subtopics:

- Guidelines for Splitting a MAST Template into Separate Functions
- The bjtm Template Architecture Using MAST Functions -- describes how the functions are placed in a file with the same name as the function. In addition, .sin is appended so that the Saber Simulator can access the functions.
- The bjtm Template -- shows calls to two different functions and a "companion" template.
- Function Call Overview - bjtm MAST Template
- bjtm_arg Declaration Template -- shows that creating a "companion" template is a more efficient way of providing argument and local parameter declarations for use by the original template and any functions it calls.
- Local Parameters Function bjtm_pars -- shows the essential parts of a MAST function.
- Calculated Values Function bjtm_values

Guidelines for Splitting a MAST Template into Separate Functions

Deciding how to split a model between a template and a MAST function can be summarized by the following general rule:

> When a model is split between a template and MAST functions, declarative parts must be in the template, while procedural parts (calculations and assignments) can be implemented in the MAST function.

Beyond this simple rule, the following guidelines can help decide how to split the model:

- The template header and header declarations must be specified within the template.
- All parameters, val and var variables, local nodes, and extraction groups must be declared in the template body.
- A “companion” template can be created that declares arguments and local parameters externally. Arguments are then referenced from this external template in the header declarations of the original template; local parameters are referenced from the external template in the body of the original template. This is demonstrated in the example in the topic titled "The bjtm Template".

  This is similar to an include file used in high-level languages, such as the C programming language.
- The control section, netlist section, and template equations are still specified in the template body, but they can include calls to MAST functions.
- All assignment statements and intermediate calculations using variables and parameters can be implemented in a MAST function.
- Template equations must be in the template.

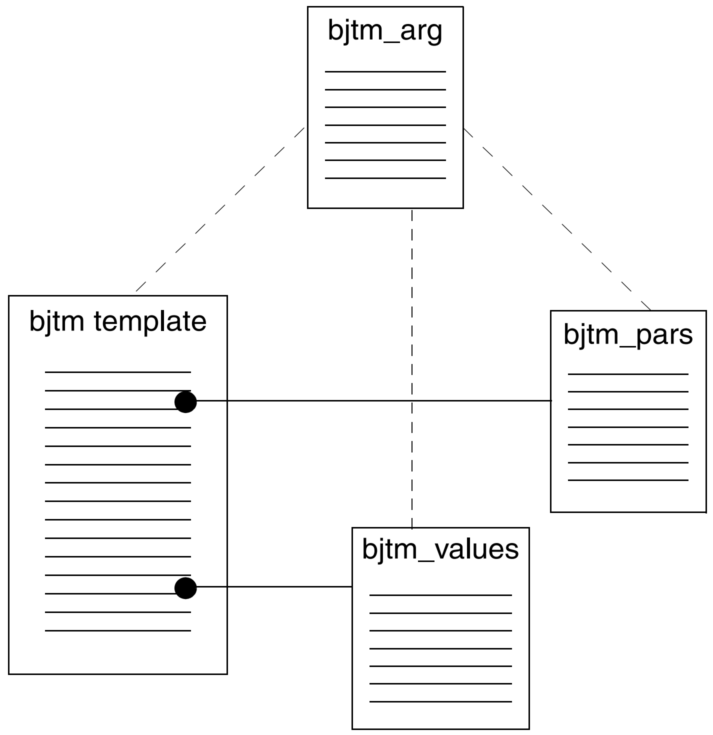
The bjtm Template Architecture Using MAST Functions

By implementing the guidelines described in the previous topic, the bjtm template is largely the same as the bjt template—the only parts that changed are the calls to the MAST functions (named bjtm_arg, bjtm_pars, and bjt_values), which are indicated by comments in the template.

Comparing the bjtm template with the bjt template, the calls from the bjtm template refer to the following external files:

- bjtm_arg.sin—a “companion” template
- bjtm_pars.sin—an external function
- bjtm_values.sin—an external function

Although these file names are arbitrary, each file must have a .sin extension. The following figure shows an overview of this relationship between functions and templates. The solid lines represent function calls; the dashed lines represent “centralized” declarations.



Calling external functions from the bjtm template

The bjtm Template

The bjtm template including MAST function calls is shown as follows:

```
element template bjtm c b e = model, ic
  electrical c, b, e
  bjtm_arg..model model = () # ... use arguments from
                             # ... "companion" template
  number ic[2]=[undef,undef]
  external number temp
{                            # begin template body
  bjtm_arg..work work     # ... use local parameters
                          # ... from “companion” template
  struc {
    number bp, inc;
  } nv[*] = [(0,.1),(2,0)]
  val v vbc, vbe, vce             # declare variables
  val i iec, icc, iba, ico, ir
  val q qbc, qbe
  electrical cp                   # local node
  group {vbc,vbe} v               # extraction groups
  group {iba,ico,ir} i
  group {qbc,qbe} q
  parameters {
  # calculate thermal voltage and 4 functions of model param.
                          # ... 1’st call to MAST function
    work = bjtm_pars(model,temp)
  }                              # end of parameters section
```

```
values {
   # calculate basic quantities of npn and pnp trans.
  vbc = v(b) - v(cp)
  vbe = v(b) - v(e)
  vce = v(cp) - v(e)
                    # ... 2'ond call to MAST function
  (iec,qbc,icc,qbe) = bjtm_values(model,work,vbc,vbe)

   # calculate base, collector, and resistor currents
  iba = iec/model->br + icc/model->bf
  ico = icc - iec - iec/model->br
  if (model->rc ~= 0) ir = (v(c) - v(cp)) / model->rc
  else                ir = 0
}                          # end values section
control_section {
   # if no collector resistance, collapse nodes c and cp
  if (model->rc == 0) collapse(c,cp)
   # specify Newton steps
  newton_step((vbc,vbe),nv)
   # initial conditions and start value
  initial_condition(vbe, ic[1])
  initial_condition(vce, ic[2])
  start_value (vbe, 0.6)
   # small-signal parameters
  device_type("bjtm", "example")
  small_signal(ibase,current,"base current",iba)
  small_signal(icoll,current,"collector current",ico)
  small_signal(vbe,voltage,"base-emitter voltage", vbe)
  small_signal(vbc,voltage,"base-collector voltage", vbc)
  small_signal(rc,resistance,"collector resistance",\
              model->rc)
}                          # end control_section
```

```
  equations {
     # current at base, internal collector, and emitter
    i(b->e)  += iba + d_by_dt(qbe)
    i(b->cp) += d_by_dt(qbc)
    i(cp->e) += ico

     # current at collector resistor, if present
    if (model->rc ~= 0) i(c->cp) += ir
  }                        # end equations section
}                          # end template body
ASCII text of this example is located in:
saber_home/example/MASTtemplates/structured/
bjtm.sin
```

Function Call Overview - bjtm MAST Template

The three types of function calls in the bjtm template are explained as follows:

1. The bjtm_arg template is not actually a MAST function (see the topic titled "bjtm_arg Declaration Template"). It is a template that serves as a central location for the declaration of arguments and local parameters of bjtm by declaring them in model and work, which are structure parameters. The bjtm template, the bjtm_pars function, and the bjtm_values function then use these parameters by calling them from bjtm_arg.

   This is done using the argdef operator (..), which references the model and work parameters from bjtm_arg as follows: (Refer to the MAST Reference Manual for information on the argdef operator.)

```
 bjtm_arg..model model = ()# use local parameters
                           # from “companion” template
# ...
 bjtm_arg..work  work      # use local parameters
                           # from “companion” template
```

Using bjtm_arg to provide these declarations illustrates the convenience of the modular approach. Although it is not necessary to use an additional template like bjtm_arg, not doing so means that you must declare variables and structures in both the calling template and in the function being called.

2. The bjtm_pars function provides values to the work parameter as the first call from the bjtm template:

```
                    # 1'st call to MAST function
  work = bjtm_pars(model,temp)
```

This shows the syntax for calling an external MAST function:

(variable_list) = name(argument_list)

where:

variable_list	is a comma-separated list of variables to receive the results of the function call. If a state is returned by the function, it must appear in a when statement. If only one value is returned, the parentheses enclosing variable_list must be omitted.
name	is the name of the MAST function being called (for example, bjtm_pars). The file containing this function (for example, bjtm_pars.sin) should be in the same directory as the calling template.
argument_list	is a comma-separated list of variables passed as arguments to the MAST function.

On the righthand side of the equals sign (=), the bjtm_pars function uses the arguments from model and the external parameter temp. It computes and returns the following values to work on the lefthand side of the equals sign:

```
vt, qbe0, qbc0, vje0, vjc0
```

These values are stored in the work structure rather than in individual numbers as in the bjt template in order to show how structures are passed between a template and an external MAST function. Note that model and temp are still declared as parameters in the bjtm template.

3. The bjtm_values function computes values for currents and charges based on whether the device is NPN or PNP. The transistor type checking is located in the bjtm_values function. This function call appears as follows:

```
                  # 2’ond call to MAST function
(iec,qbc,icc,qbe) = bjtm_values(model,work,vbc,vbe)
```

   Notice that these calls have identical variable lists and similar argument lists. The first argument is the model structure. The second argument is the work structure holding the values returned by bjtm_pars. The third and fourth arguments are the base-collector and base-emitter voltages.

   On the righthand side of the equals sign (=), the bjtm_values function uses the appropriate argument values to compute and return the following four values on the lefthand side of the equals sign:

```
iec,qbc,icc,qbe
```

   These variables are then used in the template equations. This function is described more completely in the topic titled "Calculated Values Function bjtm_values".

The remaining parts of the template are identical to the corresponding parts of the bjt example.

bjtm_arg Declaration Template

The bjtm_arg template performs the actual declaration of the model argument and local parameters used by the bjtm template, the bjtm_pars function, and the bjtm_values function.

```
template  bjtm_arg = model, work

  # the bjt model...
  struc {
    enum{_n, _p} type
    number is=1e-16, bf=100, br=1,
    cje=0, vje=.75, mje=.33,
    cjc=0, vjc=.75, mjc=.33, rc = 0
  } model

# working parameters for local bjt calculations...
  struc {
    number vt,
    qbe0, qbc0, vje0, vjc0
  } work

{   # empty template body...
}
```

There are three major points to note about this template:

- Although using bjtm_arg is not strictly required, it prevents having to declare model and work in both bjtm (the calling template) and in bjtm_pars and bjtm_values (the functions being called). This helps avoid errors associated with duplication and maintenance.
- It consists solely of a header and header declarations of two structure type parameters (model, work). These are arguments for this template that are used for other purposes in other templates and functions. This template has no connection points and an empty body, as shown by the empty braces at the bottom, { }.
- Local parameters vt, qbe0, qbc0, vje0, and vjc0 have been grouped under a structure named work for convenience. The bjtm template then declares all the parameters within work by referring to the declaration within bjtm_arg as follows:

```
bjtm_arg..work  work  # use local parameters
                      # from “companion” template
```

Local Parameters Function bjtm_pars

An external MAST function such as this one has some similarities to a MAST template:

- It has similar partitioning (header, header declarations, body).
- It uses the same referencing techniques.
- It resides in a file of the same name as the function and has the .sin extension (for example, bjtm_pars.sin). Although not required, it is good practice to make this file available to the Saber Simulator the same way that templates are.

Each section of this function is described in the following topics, using the following bjtm_pars function as an example:

- Function Header
- Header Declaration
- Function Body

```
function (work)= bjtm_pars(model,temp)

  bjtm_arg..work work      # output from this function
  bjtm_arg..model model    # input to this function
  number temp              # input to this function
{
  # The following include file declares math_boltz constant
  # (is k in bjt template), the math_charge constant
  # (is qe in bjt template), and the math_ctok constant
<consts.sin

  # Calculation of thermal voltage and 4 other quantities

  work->vt = math_boltz * (temp + math_ctok) / math_charge;
  work->qbe0 = model->cje * model->vje / (1.0 - model->mje);
  work->qbc0 = model->cjc * model->vjc / (1.0 - model->mjc);
  work->vje0 = 2.0 * model->vje / model->mje;
  work->vjc0 = 2.0 * model->vjc / model->mjc;
}
```

Function Header

As is the case for a template, the function header is the first noncommented line of the function. The header identifies the function, specifies the output from the function, assigns a name to the function, and specifies the input to the function:

```
1  function (work) = bjtm_pars(model,temp)
```

This line from bjtm_pars shows the general syntax for the header of a MAST function (note the similarity to the call from the template):

```
function (variable_list) = name(argument_list)
```

where:

function	a reserved word that identifies the contents of this file as a MAST function.
variable_list	a comma-separated list of parameters that receive the output of the function for passing to the calling template.
name	the name of the MAST function being called (bjtm_pars).
argument_list	a comma-separated list of parameters supplied as input to the function by the calling template.

The bjtm_pars function uses the arguments from model along with the external parameter temp to compute and return the following five values:

```
vt, qbe0, qbc0, vje0, vjc0
```

These values are stored in the work structure rather than as individual numbers. This structure is then passed to the calling template, bjtm.

Header Declaration

The following line in the bjtm_pars function declares the output parameter that appears in the variable_list of the function header:

```
3  bjtm_arg..work work
```

This is followed by declarations for the input parameters appearing in argument_list of the function header:

4 bjtm_arg..model model
5 number temp

Note that the bjtm_arg template is referenced again in lines 3 and 4 to obtain work and model, eliminating the need to enter all the parameters that they contain. The declaration for temp allows a simple numeric value to be passed into the function.

Function Body

The body begins with a left brace, {, and ends with a right brace, }. The first line within the body includes the consts.sin file (file inclusion is denoted by the <). This is a standard include file provided with the Saber Simulator that contains several commonly used constants for this function to perform calculations:

```
<consts.sin
```

The next five lines perform calculations for thermal voltage and junction characteristics. The results are assigned to the individual parameters in work, which is provided as a single output parameter from this function:

```
work->vt = math_boltz * (temp + math_ctok) / math_charge;
work->qbe0 = model->cje * model->vje / (1.0 - model->mje);
work->qbc0 = model->cjc * model->vjc / (1.0 - model->mjc);
work->vje0 = 2.0 * model->vje / model->mje;
work->vjc0 = 2.0 * model->vjc / model->mjc;
```

Calculated Values Function bjtm_values

The bjtm_values function takes the values in work (obtained from bjtm_pars) to compute currents and charges that will appear in the template equations. Although a little more elaborate than bjtm_pars, it has the same general characteristics as described in the topic titled "Local Parameters Function bjtm_pars".

```
function (iec,qbc,icc,qbe) =  bjtm_values(model,work,vbc,vbe)

  val i iec, icc          # output from function
  val q qbc, qbe          # output from function
  bjtm_arg..model model   # input to function
  bjtm_arg..work work     # input to function
  val v vbc, vbe          # input to function
{
   # calculate basic quantities of npn and pnp trans.
  if (model->type == _n) {
    iec = model->is * (limexp(vbc/work->vt) - 1)
    icc = model->is * (limexp(vbe/work->vt) - 1)
  }
  else {
    iec = -model->is * (limexp(-vbc/work->vt) - 1)
    icc = -model->is * (limexp(-vbe/work->vt) - 1)
  }
```

```
     # calculate charges
    if(model->type == _n) {
     if (vbc<0) {
      qbc = work->qbc0*(1-((1-vbc/model->vjc)**(1-model->mjc)))
     }
     else {
      qbc = model->cjc*vbc*(1 + vbc/work->vjc0)
     }
     if (vbe<0) {
      qbe = work->qbe0*(1-((1-vbe/model->vje)**(1-model->mje)))
     }
     else {
      qbe = model->cje*vbe*(1 + vbe/work->vje0)
     }
    }                  # end "if type _n" condition
    else { # if model is not of type _n
     if(vbc > 0) {
      qbc = -work->qbc0*(1-((1+vbc/model->vjc)**(1-model->mjc)))
     }
     else {
      qbc = model->cjc*vbc*(1-vbc/work->vjc0)
     }
     if(vbe > 0) {
      qbe = -work->qbe0*(1-((1+vbe/model->vje)**(1-model->mje)))
     }
     else {
      qbe = model->cje*vbe*(1-vbe/work->vje0)
     }
    }                   # end "if not type _n" condition
  }
```

The following topics describe each section of the bjtm_values function in more detail:

- Function Header
- Header Declaration
- Function Body

Function Header

The header for bjtm_values shows that the output consists of individual parameters (iec, qbc, icc, qbe), and the input contains both structures (model, work) and individual parameters (vbc, vbe):

```
function(iec,qbc,icc,qbe)=bjtm_values(model,work,vbc,vbe)
```

Header Declaration

The following five lines from the bjtm_values function declare the input and output parameters appearing in the header:

```
val i iec, icc          # output from function
val q qbc, qbe          # output from function
bjtm_arg..model model   # input to function
bjtm_arg..work work     # input to function
val v vbc, vbe          # input to function
```

Note that the bjtm_arg template is referenced once again to obtain work and model, eliminating the need to enter all the parameters that they contain. The declarations for the input and output vals duplicate their declarations in the calling template (bjtm).

Function Body

The body of the bjtm_values function template consists of the same equations for charges and currents as found in the bjt template. The only difference is that in the bjtm template, the following arguments are part of the work structure.

Therefore, these arguments are referenced in this function using the structure name work followed by -> and then the argument name (such as work->vt).

```
vt, qbe0, qbc0, vje0, vjc0
```

The body of the bjtm_values function is as follows:

```
{                  # start template body
  # calculate basic quantities of npn and pnp trans.
  if (model->type == _n) {
    iec = model->is * (limexp(vbc/work->vt) - 1)
    icc = model->is * (limexp(vbe/work->vt) - 1)
  }
  else {
    iec = -model->is * (limexp(-vbc/work->vt) - 1)
    icc = -model->is * (limexp(-vbe/work->vt) - 1)
  }
```

```
    # calculate charges
 if(model->type == _n) {
  if (vbc<0) {
   qbc = work->qbc0*(1-((1-vbc/model->vjc)**(1-model->mjc)))
  }
  else {
   qbc = model->cjc*vbc*(1 + vbc/work->vjc0)
  }
  if (vbe<0) {
   qbe = work->qbe0*(1-((1-vbe/model->vje)**(1-model->mje)))
  }
  else {
   qbe = model->cje*vbe*(1 + vbe/work->vje0)
  }
 }                   # end "if type _n" condition
 else { # if model is not of type _n
  if(vbc > 0) {
   qbc = -work->qbc0*(1-((1+vbc/model->vjc)**(1-model->mjc)))
  }
  else {
   qbc = model->cjc*vbc*(1-vbc/work->vjc0)
  }
  if(vbe > 0) {
   qbe = -work->qbe0*(1-((1+vbe/model->vje)**(1-model->mje)))
  }
  else {
   qbe = model->cje*vbe*(1-vbe/work->vje0)
  }
 }                 # end "if not type _n" condition
}
```

15

Foreign Routines in MAST

This topic, which shows how to use foreign routines as extensions of the MAST language, is divided into the following subtopics:

- Using a FORTRAN Function in a MAST Template -- shows a foreign routine that returns the factorial of its argument. Because it returns a single value, this routine can be used in a template wherever an intrinsic function can be used (if properly declared).
- Modeling the Bipolar Transistor Using Foreign Routines -- shows the bipolar transistor model, implemented here partly in the MAST language and partly in C. It demonstrates a more general use of foreign routines.
- Implementing a MAST Foreign Routine in C -- shows the calling interface for foreign routines written in C; shows the mechanism for passing structures, enumerated types, and arrays, both to and from the foreign routine; shows the implementation and usage of multi-purpose foreign routines that are called with a varying argument list and return different values for different calls; and shows guidelines for splitting a component model into a MAST template and a foreign routine.

Introduction

It is sometimes useful to implement part of a model in a routine that has been written in a general-purpose programming language, such as C or FORTRAN, rather than in the MAST language. One such case is when the model requires operations that are not supported in the MAST language, such as certain mathematical functions (e.g., Bessel functions). Another is when a model implemented for another simulator has to be adapted for use with the Saber Simulator. The MAST language includes a well-defined way of calling such foreign routines, which can be written in any programming language provided that they can be called from a FORTRAN environment.

A foreign routine requires an appropriate compiler (such as C) and a significant amount of interface between the MAST language and the language of the foreign routine. For these reasons, it is recommended to use a MAST function instead of a foreign routine whenever possible.

Using a FORTRAN Function in a MAST Template

The factorial, n!, of a positive integer, n, is defined as the product of all positive integers from 1 through n:

n! = 1X2X*3X...X(n-1)Xn

Therefore, a function computing the factorial (fact) has one argument and returns one value as shown in the following UNIX example:

```
subroutine fact(in,nin,ifl,nifl,out,nout,ofl,nofl,undef,ier)
  c..header declarations
     integer nin, ifl(*), nifl, nout, ofl(*), nofl, ier
     double precision in(*), out(*), undef
  c..local declarations
     integer n, i
  c..convert input value to integer (ignore fractions)
     n = in(1)
  c..compute factorial and store in out(1)
     out(1) = 1
     do 10 i=1,n
     10 out(1) = out(1)*i
  c..return to template
     return
end
```

The implementation of the FORTRAN routine for this is explained in the following topics:

- Writing the FORTRAN Routine -- shows the following concepts:
    - Defining the header of the factorial routine
    - Getting input in the first element of the in array
    - Returning results in the first element of the out array
    - Using c.. to indicate comments (which are ignored by the routine)

- Making the routine available to the Saber Simulator
- Declaring and calling the routine from a template

Writing the FORTRAN Routine

The way to call a foreign routine from the MAST language is through a unique calling interface that has two parts:

- The header of the foreign routine
- The mechanism for passing values between MAST templates and the foreign routine

The foreign routine header is identical for all foreign routines on a given platform and is independent of the way the routine is used in a MAST template. For a foreign routine, the UNIX header is as follows (an asterisk, *, indicates unlimited array size):

```
subroutine name(in,nin,ifl,nifl,out,nout,ofl,nofl,undef,ier)
  integer nin, ifl(*), nifl, nout, ofl(*), nofl, ier
  double precision in(*), out(*), undef
```

The header for Windows NT is shown as follows:

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

where:

name	is the user-selected name of the foreign routine. From MAST templates, the routine must be called by this name. The name must be unique and must be a valid name in both the MAST language and FORTRAN. For the factorial function, let name be fact.

in	is a double-precision array containing the arguments from the call to the foreign routine. These arguments, when received by the foreign routine, are packed. Arguments of certain types are encoded as well. The encoding scheme for arguments is further described in the MAST Reference Manual.
nin	is an integer containing the number of elements in the in array. It is often different from the number of arguments passed to the foreign routine in the MAST template.
out	is a double-precision array into which the foreign routine must place the values to be passed back to the MAST template. Depending upon the data type of the results (as declared in the calling template), the routine might have to encode certain values. The out array is guaranteed to be large enough to hold the properly-encoded results, except when the routine is returning a variable-length array to the template. Information about returning variable-length arrays is given in the MAST Reference Manual.
nout	is an integer containing the size of the out array.
undef	is a double-precision number indicating an undefined quantity. Its value is identical to the undef constant in the MAST language. Foreign routines can use it to interpret input values or to return undefined values.

The remaining arguments in the foreign routine header, namely ifl, nifl, ofl, nofl, and ier, are reserved for future use. Although currently unused, they must be present in the foreign routine header.

The mechanism used to pass the values of variables from a MAST template to a foreign routine depends upon the type of the variables, as declared in the template. Similarly, the routine must return its results in a format that depends on the type of the variables that receive those results. It is therefore important to understand that a foreign routine and the templates using it must agree in the number and the type of both the variables passed to the routine (its arguments) and the results of the foreign routine. Here, the mechanism is described only to the extent that it is used in the examples—refer to the MAST Reference Manual for more information on the argument-passing mechanism.

According to its intended use, the factorial function has a single numeric value as input and returns a single numeric value. Several types of MAST variables are represented by a single numeric value: number (both template arguments and local parameters), var variables, ref variables, val variables, and state variables. In the argument-passing mechanism all are handled identically: if a variable of any of the listed types is passed as the only argument to the foreign routine in a template, the foreign routine receives its value in the first element of the in array, and nin is 1. Similarly, if the foreign routine returns a single value of one of the types listed, nout has a value of 1, the out array has length 1, and the routine must return its result in the first element of the out array.

Declaring and Calling the Routine From a Template

Like any other user-defined quantity in the MAST language, a foreign routine must be declared in the calling template before being used.

You declare a foreign routine in the template body in either of the following two ways, depending upon how many values it returns:

A routine returning a single numeric value should be declared as:

```
foreign number (name)
```

You can use such a routine wherever you can use MAST intrinsic functions, even in expressions. This means you can use them anywhere in the template body including in the template equations, in the netlist section, in when statements, and even in a local declaration to initialize a number.

A routine returning multiple values must be declared as:

```
foreign (name)
```

An example of such a routine is described in the topic titled "Modeling the Bipolar Transistor Using Foreign Routines".

Because this factorial routine always returns a single value, it appears as a local declaration of any template that calls it, as follows:

```
foreign number fact()
```

Having done this, you can compute the factorial of a given positive integer anywhere in the template. For example, you could include the following statement in a template:

```
nfact = fact(n)
```

where nfact and n must be declared as numeric values (i.e., as number variables, val variables, or state variables). Note that n can also be a var variable or a ref variable. You could also use the fact function in an expression to define a 1.2k ohm resistor by writing the netlist entry as follows:

```
r.1 a b = 10 * fact(5)
```

Modeling the Bipolar Transistor Using Foreign Routines

This topic presents a more general way to use foreign routines with a MAST template. As an example, it uses the bipolar transistor model (bjt) as described in the topic “Ebers-Moll MAST Model for the Bipolar Transistor”. The goal of this example is to write a MAST template and a foreign routine that implement the model in such a way that the combination has the same functionality as the bjt template. Initial conditions, start values, and small-signal parameters are removed from the new template for brevity.

This new template (bjtf) and its foreign routine are such that the model contains only one-dimensional nonlinearities, which optimizes its speed.

This topic is divided into the following subtopics:

- Splitting Functionality Between a MAST Template and a Foreign Function
- Modifying the BJT Template to Use a Foreign Routine
- General Foreign Function Call Syntax
- Calling the Foreign Routines

Splitting Functionality Between a MAST Template and a Foreign Function

Unlike the example in the topic titled "Using a FORTRAN Function in a MAST Template" (which essentially added another mathematical function to the MAST language), deciding how to split a model between a MAST template and a foreign routine is not trivial.

Deciding how to split a model between a template and a MAST function can be summarized by the following general rule:

> When a model is split between a template and a foreign function, declarative parts must be in the template, while procedural parts (calculations and assignments) can be implemented in the foreign function.

Beyond this simple rule, the following guidelines can help decide how to split the model:

- The template header and header declarations must be specified entirely within the MAST template.
- The control section, values section, parameters section, netlist section, and template equations must be specified in the MAST template, but they can include calls to foreign routines that return a single numeric value. The factorial routine from the topic titled "Using a FORTRAN Function in a MAST Template" is an example of such a routine.
- The local declarations section of the template must include declarations of all variables used in the template. Specifically, it must include all parameters, val variables, var variables, local nodes, extraction groups, and foreign routine names.
- Messages and error handling should be done in the template, because the functionality of the MAST error(), warning(), and message() functions is not directly available in foreign routines.
- All assignment statements and intermediate calculations using variables and parameters can be implemented in a foreign routine.
- Any val variable defined by a foreign routine call is considered to be a nonlinear function of all the val variables or system variables passed as arguments to the foreign routine.

  This fact leads to the following rules:

  - val variables that are linear functions of system variables should be defined in the template.
  - val variables defined by foreign routine calls should, if possible, be grouped according to how they depend upon other val variables. It is typically more efficient to call a foreign routine several times with a small number of val variables as arguments (that is, with low dimensionality) than to call it once with all val variables (thereby creating a high-order nonlinear function). This rule is illustrated later in this chapter.
  - val variables defined only for extraction purposes can be defined either in the template or in foreign routines.

Modifying the BJT Template to Use a Foreign Routine

The bjtf template (listed below) is largely the same as bjt—the only parts that changed were the calls to the foreign routine (named bjt), which are indicated by comments.

Unlike the bjtm template, which uses calls to two different MAST functions plus an include file, all calls in the bjtf template are made to the same foreign routine (bjt).

```
element template bjtf c b e = model, ic
  electrical c, b, e
  struc {                    # the transistor model
    enum {_n, _p} type
    number  is=1e-16, bf=100, br=1, \
            cje=0, vje=.75, mje=.33, \
            cjc=0, vjc=.75, mjc=.33, rc=0
  } model = ()
  number ic[2]=[undef,undef]
  external number temp
{                            # begin template body
   # declare local param., vals, and extraction groups
  number work[5]
  struc {number bp,inc;} \
    nv[*] = [(0,.1),(2,0)]
  val v vbc, vbe, vce
  val i iec, icc, iba, ico, ir
  val q qbc, qbe
  electrical cp              # local node
  group {vbc,vbe} v #...extraction groups
  group {iba,ico, ir} i
  group {qbc,qbe} q
  foreign bjt                # ... foreign routine
```

```
control_section {
 # If no collector res., collapse nodes c and cp
if(model->rc == 0) collapse(c,cp)
 # specification of sample points and newton steps
newton_step((vbc,vbe),nv)
  # initial conditions, start values, and
  #  small-signal parameters removed for brevity
}
```

```
parameters {
  # Calculate thermal voltage and 4 functions of model
  #  parameters. They are stored in a work vector
  work = bjt(1,model,temp)             #...foreign call
}

values {
  vbc = v(b,cp)
  vbe = v(b,e)
  vce = v(cp,e)
   # calculate currents and charges
  if(model->type == _n) {
    (iec,qbc) = bjt(2,model,work,vbc)  #...foreign call
    (icc,qbe) = bjt(3,model,work,vbe)  #...foreign call
  }
  else {
    (iec,qbc) = bjt(2,model,work,-vbc) #...foreign call
    (icc,qbe) = bjt(3,model,work,-vbe) #...foreign call
  }
   #calculate base, collector, and resistor currents
  iba = iec/model->br + icc/model->bf
  ico = icc - iec - iec/model->br
  if (model->rc ~= 0) ir = (v(c,cp)) / model->rc
  else              ir = 0
}                          # end of values section
```

```
 equations {
   # current at base, internal collector, and emitter
  i(b->e)  += iba + d_by_dt(qbe)
  i(b->cp) += d_by_dt(qbc)
  i(cp->e) += ico

   # current at collector resistor, if present
  if (model->rc ~= 0) i(c->cp) += ir
 }                 # end of equations section
}                  # end of template body
```

ASCII text of this example is located in:
install_home/example/MASTtemplates/structured/
bjtf.sin

The foreign function calls are described in the following topics:

- General Foreign Function Call Syntax
- Calling the Foreign Routines

General Foreign Function Call Syntax

All of the foreign routine calls in the bjtf template use the syntax of the following general foreign routine call:

```
(variable_list) = name(argument_list)
```

where:

variable_list	is a comma-separated list of variables to receive the results of the routine call. If a state is returned by the routine, it must appear in a when statement. If only one value is returned by the foreign routine, the parentheses enclosing variable_list are optional.
name	is the name of the foreign routine to call.

argument_list	is a comma-separated list of variables passed as arguments to the foreign routine.

Calling the Foreign Routines

Note that the bjtf template contains the following declaration and calls to the foreign routine:

1. The bjt routine is declared locally as a foreign function:

```
foreign bjt
```

   This declaration indicates that the bjt routine may return more than one value, and that the type and number of returned values might differ for different calls. In fact, the bjt routine returns an array of length five for the first call, but it returns a pair of vals (that is, a pair of simple numeric values) for the second and third calls.

2. The first call to the bjt foreign routine is:

```
work = bjt(1, model, temp)
```

   The first argument of the call to bjt (1) identifies the first task to be performed. This is necessary because this same routine is called, with different arguments, further down in the template.

   The second and third arguments (model, temp) are the model argument and the system temperature (temp). Note that model and temp are still declared as parameters in the template.

   The values of model and temp are used to compute and return the following five values that are computed in bjt:

```
vt, qbe0, qbc0, vje0, vjc0
```

These values are stored in the work array, rather than in individual numbers, to show how arrays are passed between templates and foreign routines. This is described in the topic "Implementing a MAST Foreign Routine in C".

3. The calls to the bjt routine in the following lines are made to compute currents and charges:

```
if(model->type == _n) {
  (iec,qbc) = bjt(2,model,work,vbc)
  (icc,qbe) = bjt(3,model,work,vbe)
}
else {
  (iec,qbc) = bjt(2,model,work,-vbc)
  (icc,qbe) = bjt(3,model,work,-vbe)
}
```

Although all bjtf template currents and charges could be computed in one routine call, it is preferable to call the routine twice—first for the iec, qbc pair, second for the icc, qbe pair.

According to the above guidelines, all four of these val variable are interpreted as one-dimensional nonlinear functions of vbc or vbe. If all four val variables were computed in one foreign routine call, they would be considered two-dimensional nonlinear functions of vbc and vbe, which would be less efficient for simulation.

Notice that the second and third routine calls have similar arguments. The first tells the bjt routine whether the base-collector or the base-emitter characteristics have to be computed. The second argument is the model structure. The third argument is the work vector holding the values returned by the first call. The fourth argument is the base-collector or base-emitter voltage.

The remaining parts of the template are identical to the corresponding parts of the bjt template described in later topics and are therefore not discussed here.

Implementing a MAST Foreign Routine in C

This topic describes how to implement the bjt routine as a C language routine. The calling interface is functionally identical to the one described in the topic "Using a FORTRAN Function in a MAST Template" for FORTRAN routines, as are the meaning and usage of arguments. The following shows a UNIX routine format:

```
  name(in, nin, ifl, nifl, out, nout, ofl, nofl, undef, ier)
  int *nin, *ifl, *nifl, *nout, *ofl, *nofl, *ier;
  double *in, *out, *undef;
```

The following shows a Windows NT routine format:

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

In a Windows NT environment, the C routine name must be entered in uppercase characters.

Some systems require name to have a trailing underscore (_), in order for the routine to be callable from a FORTRAN environment, as the MAST language requires. This topic is divided into the following subtopics:

- Defining Template Arguments
- First Call—Setting Up Return Parameters
- Second and Third Calls—Performing Calculations
- Complete BJT C Routine

Defining Template Arguments

The factorial function example in the topic "Using a FORTRAN Function in a MAST Template" shows how a single numeric argument is passed to the foreign routine. If there are multiple arguments, the process is similar, except that arguments must appear in a particular order, and each argument occupies a specific number of places in the in or out array.

To improve the readability of the routine and reduce the possibility of errors, you should define a name for each entry in the in and out arrays. Choose names that suggest the purposes of the quantities passed to and from the routine.

The first argument is a single number telling the foreign routine what to do. It can have any of the values 1, 2, and 3. This argument is passed to the foreign routine as the first element of the in array, that is, in in[0]. (Unlike FORTRAN, arrays in C start with element 0.) The corresponding definitions are as follows:

```
#define JOB                    in[0]
#define PARAMETERS             1
#define BC_CHARACTERISTICS     2
#define BE_CHARACTERISTICS     3
```

The second argument is the model structure. A structure is passed to and from foreign routines by passing each member of the structure as a separate parameter, in the order in which they are declared in the structure. The first member of the model structure is the model type, which is of type enum. An enum is passed as a single number, which is 1 if the value of the variable is first in the list of possible values, 2 if the value is second in the list, etc. In the declaration of the model type, note that _n (for an npn transistor) is encoded as 1, and _p is encoded as 2. All other members of the model structure are numbers, which can be handled directly. Therefore, the definitions for the model structure are as follows:

```
#define MODEL_TYPE  in[1]
#define _N 1  /* npn */
#define _P 2  /* pnp */
#define MODEL_IS   in[2]
#define MODEL_BF   in[3]
#define MODEL_BR   in[4]
#define MODEL_CJE  in[5]
#define MODEL_VJE  in[6]
#define MODEL_MJE  in[7]
#define MODEL_CJC  in[8]
#define MODEL_VJC  in[9]
#define MODEL_MJC  in[10]
#define MODEL_RC   in[11]
```

In addition, the system temperature, which is a local parameter is defined as follows:

```
#define TEMP       in[12]
```

The rest of the definitions depend on the calls to the foreign routine, described in the next topics.

First Call—Setting Up Return Parameters

As its result, the new bjtf template returns an array, of length 5, called work. Arrays are passed to and from foreign routines in two parts, the array size and the array contents. The array size is a single number, and each array element takes as many places as a variable of the same type would. The work array consists of five numbers, so it needs six places in the out array.

Each array element has been given a name that indicates its use in the following lines:

```
#define WORK_P_SIZE  out[0]
#define WORK_P_VT    out[1]
#define WORK_P_QBE0  out[2]
#define WORK_P_QBC0  out[3]
#define WORK_P_VJE0  out[4]
#define WORK_P_VJC0  out[5]
```

The P in each name stands for parameters. You must distinguish these work values, which are returned from the bjt routine, from work values for the second and third calls (described in the next topic), which are passed to the bjt routine.

Note that, in order to compute thermal voltage, the values of Boltzmann's constant and the electron charge must also be included as follows:

```
#define K   1.381e-23  /*Boltzmann's constant */
#define QE  1.602e-19  /*electron charge      */
```

The first part of implementing the first call to the bjt routine is a straightforward translation of the intermediate calculations of vt, qbe0, qbc0, vje0, and vjc0 (from the original bjt template) into the C language:

```
if (JOB == PARAMETERS) {
 /* Calculate thermal voltage and four other quant. */
 WORK_P_VT = K * (TEMP + 273.15) / QE;
 WORK_P_QBE0 = MODEL_CJE * MODEL_VJE / (1.0 - MODEL_MJE);
 WORK_P_QBC0 = MODEL_CJC * MODEL_VJC / (1.0 - MODEL_MJC);
 WORK_P_VJE0 = 2.0 * MODEL_VJE / MODEL_MJE;
 WORK_P_VJC0 = 2.0 * MODEL_VJC / MODEL_MJC;
```

The second part consists of defining the size of the work array as follows:

```
 /* Define the array size */
 WORK_P_SIZE = 5;
}
```

It is important to define this size and to make it identical to the array size as declared in the template. If omitted, the size will be undefined, and the routine call will not have the expected effect.

Second and Third Calls—Performing Calculations

The second and third calls to this routine both have the work vector as their third argument. The work vector occupies the six positions following the model structure. (TEMP is not present in these calls):

```
#define WORK_V_SIZE   in[12]
#define WORK_V_VT     in[13]
#define WORK_V_QBE0   in[14]
#define WORK_V_QBC0   in[15]
#define WORK_V_VJE0   in[16]
#define WORK_V_VJC0   in[17]
```

Note that these definitions for the elements in the work array are similar to the ones given in the first call, except that the work array now appears in the in array. They are distinguished by the V in each name, which stands for values.

The second call computes the iec, qbc pair as a function of vbc. Therefore, vbc is an additional input, and iec and qbc are outputs of the routine.

```
#define VBC          in[18]
#define IEC          out[0]
#define QBC          out[1]
```

The code calculating both iec and qbc is again a straightforward translation into the C language of the corresponding calculations from the original bjt template as follows:

```
else if (JOB == BC_CHARACTERISTICS) {
  /* Calculation of iec and qbc from vbc */
  IEC = MODEL_IS * (exp(VBC / WORK_V_VT) - 1.0);
  if (VBC < 0.0) {
    QBC=WORK_V_QBC0 *
    (1.0 - pow(1.0 - VBC/MODEL_VJC, 1.0 - MODEL_MJC));
  }
  else {
    QBC = MODEL_CJC * VBC * (1.0 + VBC / WORK_V_VJC0);
  }
  if (MODEL_TYPE == _P) {
    IEC = -IEC;
    QBC = -QBC;
  }
}
```

The only difference is the exp() function of the C language is called instead of the limexp() function of the MAST language.

The third call is very similar to the second call, calculating icc and qbe:

```
#define VBE          in[18]
#define ICC          out[0]
#define QBE          out[1]

else if (JOB == BE_CHARACTERISTICS) {
  /* Calculation of icc and qbe from vbc */
  ICC = MODEL_IS * (exp(VBE / WORK_V_VT) - 1.0);
  if (VBE < 0.0) {
    QBE=WORK_V_QBE0 *
    (1.0 - pow(1.0 - VBE/MODEL_VJE, 1.0 - MODEL_MJE));
  }
  else {
    QBE = MODEL_CJE * VBE * (1.0 + VBE / WORK_V_VJE0);
  }
  if (MODEL_TYPE == _P) {
    ICC = -ICC;
    QBE = -QBE;
  }
}
```

This concludes the detailed description of the bjt routine and the argument-passing mechanism. Further information, including how foreign routines can return variable-length arrays, is given in the MAST Reference Manual.

Complete BJT C Routine

The following is a listing of the complete bjt routine, written in the C language.

```
/* Define names for the input and output of the foreign*/
/* routine. First the part that is common to all usage:*/
/* result = bjt(job, model...) */
#define JOB  in[0]
#define PARAMETERS 1
#define BC_CHARACTERISTICS 2
#define BE_CHARACTERISTICS 3
#define MODEL_TYPE  in[1]
#define _N  1                  /* npn */
#define _P  2                  /* pnp */
#define MODEL_IS    in[2]
#define MODEL_BF    in[3]
#define MODEL_BR    in[4]
#define MODEL_CJE   in[5]
#define MODEL_VJE   in[6]
#define MODEL_MJE   in[7]
#define MODEL_CJC   in[8]
#define MODEL_VJC   in[9]
#define MODEL_MJC   in[10]
#define MODEL_RC    in[11]
#define TEMP        in[12]   /* local parameter */
```

```
/* Define names for 1st call     */
/* work = bjt(1, model, temp)  */
#define WORK_P_SIZE   out[0]
#define WORK_P_VT     out[1]
#define WORK_P_QBE0   out[2]
#define WORK_P_QBC0   out[3]
#define WORK_P_VJE0   out[4]
#define WORK_P_VJC0   out[5]
/* Define names for 2nd and 3rd calls   */
/* result = bjt(job, model, work,...)       */
#define WORK_V_SIZE    in[12]
#define WORK_V_VT      in[13]
#define WORK_V_QBE0    in[14]
#define WORK_V_QBC0    in[15]
#define WORK_V_VJE0    in[16]
#define WORK_V_VJC0    in[17]

/* Define names for calculation of base/collector charac. */
/* (iec, qbc) = bjt(2, model, work, vbc) */
#define VBC            in[18]
#define IEC            out[0]
#define QBC            out[1]

/* Define names for calculation of base/emitter charac. */
/* (icc, qbe) = bjt(3, model, work, vbe) */
#define VBE            in[18]
#define ICC            out[0]
#define QBE            out[1]
```

```
/* Other defines */
#define K    1.381e-23   /* Boltzmann`s constant */
#define QE   1.602e-19   /* electron charge        */

/*---------------------------------------------------*/
/*The following two include statements are
  system-provided files used to declare input/output
  channels and the exp() and pow() mathematical functions*/
#include <stdio.h>
#include <math.h>
#include "saberApi.h" /* Specify the complete path here to
                      "<install_home>/include/saberApi.h" */

#if defined(_MSC_VER)
#define bjt BJT
#endif

 /*The following line works for SunOS 5.5.1 - 5.6 platforms
  void bjt_(double*in, long*nin, long*ifl, long*nifl, */

 /* The following line works for HP platforms */
void bjt(double*in, long*nin, long*ifl, long*nifl,
 double*out, long*nout, long*ofl, long*nolf, double*undef,
 long*ier)
{
```

```
if (JOB == PARAMETERS) {
 /* Calculate thermal voltage and four other quan. */
  WORK_P_VT = K * (TEMP + 273.15) / QE;
  WORK_P_QBE0 = MODEL_CJE * MODEL_VJE / (1.0 - MODEL_MJE);
  WORK_P_QBC0 = MODEL_CJC * MODEL_VJC / (1.0 - MODEL_MJC);
  WORK_P_VJE0 = 2.0 * MODEL_VJE / MODEL_MJE;
  WORK_P_VJC0 = 2.0 * MODEL_VJC / MODEL_MJC;

  /* Define the array size */
  WORK_P_SIZE = 5;
}
else if (JOB == BC_CHARACTERISTICS) {
  /* Calculation of iec and qbc from vbc */
  IEC = MODEL_IS * (exp(VBC / WORK_V_VT) - 1.0);
  if (VBC < 0.0) {
    QBC=WORK_V_QBC0 *
    (1.0 - pow(1.0 - VBC/MODEL_VJC, 1.0 - MODEL_MJC));
  }
  else {
    QBC = MODEL_CJC * VBC * (1.0 + VBC / WORK_V_VJC0);
  }
  if (MODEL_TYPE == _P) {
    IEC = -IEC;
    QBC = -QBC;
  }
}
```

```
  else if (JOB == BE_CHARACTERISTICS) {
    /* Calculation of icc and qbe from vbc */
    ICC = MODEL_IS * (exp(VBE / WORK_V_VT) - 1.0);
    if (VBE < 0.0) {
      QBE=WORK_V_QBE0 *
      (1.0 - pow(1.0 - VBE/MODEL_VJE, 1.0 - MODEL_MJE));
    }
    else {
      QBE = MODEL_CJE * VBE * (1.0 + VBE / WORK_V_VJE0);
    }
    if (MODEL_TYPE == _P) {
      ICC = -ICC;
      QBE = -QBE;
    }
  }

  else { /*If the bjt routine is called with an
           unrecognized first argument, do the following*/
    fprintf(stderr, "Bad job: %f\n", JOB);
  }
}
```

```
ASCII text of this example is located in:
install_home/example/MASTtemplates/structured/
bjt.c
```

16 Time-Domain Modeling

The following examples introduce the MAST capabilities that allow you to create models that depend on effects in the time domain:

- Using the MAST delay Function in an Ideal Delay Line -- shows how to represent a time delay using the intrinsic delay function and how to influence the time-step algorithm of the simulator from within a template (i.e., scheduling)
- Expanding the Multiple-Output Voltage Source -- shows how to define a val variable as a function of both time and other val variables

Using the MAST delay Function in an Ideal Delay Line

The MAST delay function can be used in template equations to model ideal delay. Using delay in an equation has the following general form:

```
delayed_value = delay (reference_value,time)
```

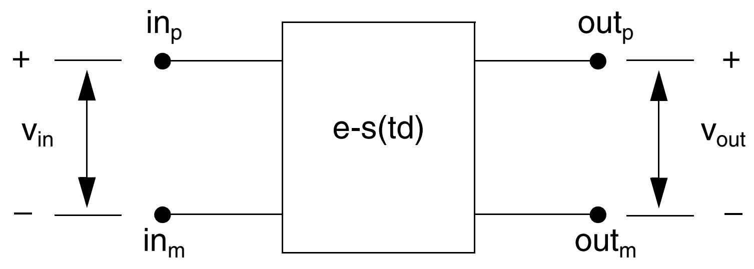
where delayed_value and reference_value are either a val variable, a branch variable, or a var variable; and time is a parameter that specifies the duration of the constant delay between delayed_value and reference_value. For example, suppose a template equation contains the following statement:

```
iout: vout = delay (vin, dtime)
```

This statement has the following meaning:

> Solve for a value of output voltage (vout) that has the same amplitude as the input voltage (vin), but delay the output from the input by the specified value of delay (dtime).

An example of a template using the delay function is the ideal delay line template, dline, shown in the figure below, which is equivalent to a voltage-controlled voltage source with a time delay and a gain



Ideal delay line

The dline template description is divided into the following topics:

- Ideal Delay Line (dline) MAST Template
- Delayed Sine Wave Transient Analysis
- Delayed Sine Wave AC Analysis
- MAST dline Template Summary

Ideal Delay Line (dline) MAST Template

The dline template is shown as follows:

```
template dline inp inm outp outm = td, a
  electrical inp, inm, outp, outm
  number td = 0.0,      # ideal delay, with default
          a = 1.0       # gain, with default
{
  var i iout      # current from outp to outm
  val v vout,     # voltage developed across outp and outm
      vin,        # controlling voltage
      vdl         # delayed voltage
  values  {
    vout = v(outp) - v(outm)
    vin = v(inp) - v(inm)
    vdl = vin * a
  }
  equations  {
    i(outp->outm) += iout
    iout: vout = delay(vdl, td)
  }
}
ASCII text of this example is located in:
install_home/example/MASTtemplates/structured/
dline.sin
```

The delay function in the template equation instructs the simulator to set the output voltage (across pins outp and outm) to the same value as vdl, but delay it by time td.

Note that vdl needs to be declared as a val variable because it is an intermediate variable that depends on the value of the input branch voltage, vin.

It is allowable to omit vdl altogether by using the following equation:

```
iout: vout=delay(vin*a, td)
```

Including vdl simplifies the template equation and permits the product of gain and input voltage to be extracted and plotted.

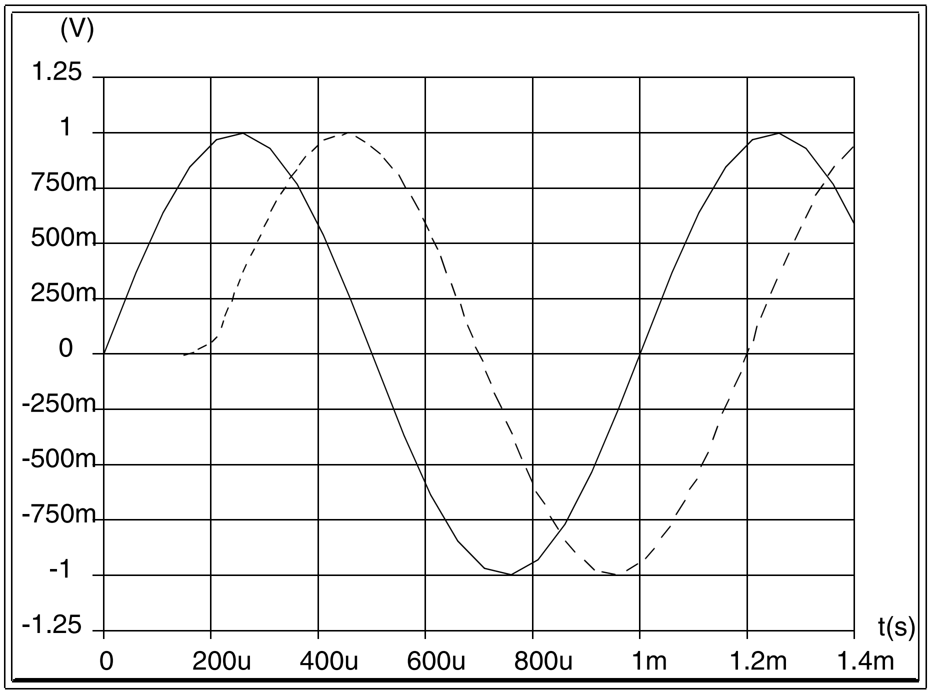
It is incorrect syntax to multiply the delay function by any quantity. For example, the following template equation would be incorrect:

```
# incorrect equation...
iout: vout = a*delay(vin, td)
```

The delay function cannot be multiplied by the constant a; instead, the constant a must multiply vin inside the parentheses, as shown in the preceding equation.

Delayed Sine Wave Transient Analysis

The following figure shows the result from a transient simulation of a sine wave input to the dline template.



Delayed sine wave-transient analysis result

The circuit netlist is shown below:

```
v.in    in 0       = tran=(sin=(0,1,1k))
dline.1 in 0 out 0 = td=200u
```

In this example, the voltage at pin out is the same as the voltage at pin in, except that it is delayed by 200 ms. During a DC analysis, the delay function has no effect. Consequently, the voltages at in and out are the same following a DC analysis.

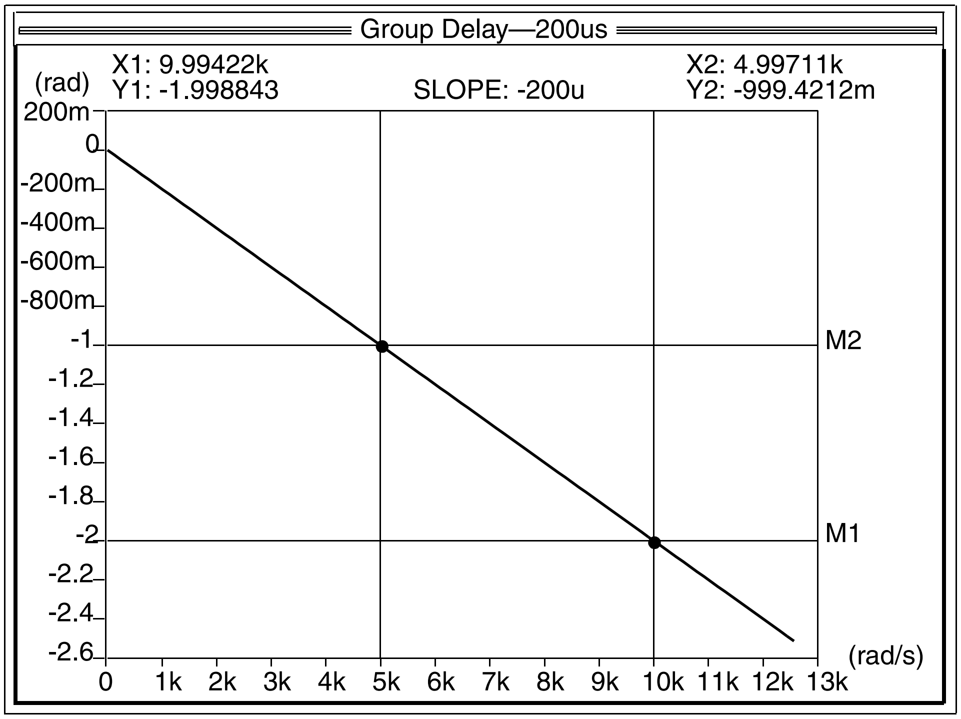
Delayed Sine Wave AC Analysis

The dline template is also effective for a small-signal AC analysis. If the multiplier, a, is left at the default value of 1, dline has no effect on the magnitude of the input signal. However, the phase is shifted as a function of frequency.

The voltage source (v.in) from the preceding circuit netlist can be modified to simulate an AC source, as follows:

```
v.in in 0 = ac=(1,0)
dline.1 in 0 out 0 = td=200u
```

The following figure shows the result of an AC analysis. The waveforms show the magnitude and phase at out, the output of dline.1.



Delayed sine wave-AC analysis (small-signal) result

The negative slope of the phase (in radians) with respect to frequency (in radians per second) is commonly called group delay. From the graph shown in the following figure, you can see that this slope for dline.1 is measured as -200 ms, which corresponds to the td=200u specified in the netlist.

MAST dline Template Summary

To summarize, you can conceptualize the MAST delay function as:

- A zero delay during DC analysis
- A time delay during transient analysis
- A constant group delay during a small signal (AC) analysis

Expanding the Multiple-Output Voltage Source

A simvar is a built-in variable that interacts with the simulator. Most simvar variables are set by the simulator, and templates can use them but not alter them. In effect, they are a "window" into what is happening in the simulator. Two simvar variables, named time and time_domain, are used in the multi-purpose voltage source (vsource_2) example, which is described in the following topics:

- Overview
- The vsource_2 MAST Template
- Header Declarations
- Union Type Parameters -- shows how to the intrinsic function union_type for indicating which member of the union has been selected
- Local Declarations
- Equations Section
- Determining Union Elements
- Assigning Internal Values
- Performing Calculations (Defining Signals) -- shows how to use the step_size and next_time simvar variables to communicate information to the simulator and how to use the sin and exp intrinsic functions
- Netlist Examples

Two other simvar variables, step_size and next_time, are special, in that templates can change their values. These simvar variables let templates communicate to the simulator. The vsource_2 example illustrates their use.

For a description of the simvar variables and their uses, see the MAST Reference Manual.

A point worth emphasizing is that vsource_2 is a linear template, even though the definition of tran is nonlinear. The linearity of a template is determined with respect to other variables, such as voltages or currents. The nonlinearity of tran in this voltage source example is with respect to time.

Overview

The voltage source (vsource_2) template provides three different, time-varying outputs. It is similar to vsource_1, but it uses a union parameter type to provide more flexibility. For more information on unions, refer to the MAST Reference Manual.

The vsource_2 template is used as a voltage supply or waveform source as follows:

- A constant output voltage for all large-signal analyses
- One of three output waveforms (a sine wave, an exponential signal, or a step function) for the transient (time-based) analysis

To specify one template for two separate purposes (constant supply or varying waveform), you must decide how to handle conflicting specifications, particularly regarding DC analysis. If the voltage source is specified as a supply, the supply value is obviously the DC value. If the voltage source is specified as a transient waveform, then the waveform value at time-equal-to-zero should be used as the DC value. However, if both the supply and transient specifications are given, only one can be chosen for the DC analysis. In this example, the transient specification overrides the supply specification by default. However, a provision is made to allow the template user to override this default.

Consequently, the template must have the following properties:

- The value of a transient waveform at time=0 must be able to override the constant supply voltage value.
- Although the transient specification must, by default, override the supply specification, the transient specification must also have an “off” setting that allows the supply specification to be in effect.

The vsource_2 MAST Template

The template for this voltage source (vsource_2) is listed below.

```
element template vsource_2 p m  = supply, tran
  electrical p, m
  number supply = 0
  union {
    number                         off
    struc {number vo, va, f, td;}  sin
    struc {number v1,v2,tau;}      exp
    struc {number v1,v2,tstep,tr;} step
  } tran = (off = 1)
{
  number pi = 3.14159
  val v vn, vs
  var i is
  number td,vo,va,w,ss,v1,v2,tau,tstep,tr,slew
  parameters {
   # define intermediate values, depends on selected output
    if (union_type (tran,sin))  {
      td = tran->sin->td
      vo = tran->sin->vo
      va = tran->sin->va
      w  = 2*pi*tran->sin->f
      ss = 0.05/tran->sin->f
    }
    else if (union_type (tran,exp)) {
      v1  = tran->exp->v1
      v2  = tran->exp->v2
      tau = tran->exp->tau
    }
```

```
      else if (union_type (tran,step)) {
        tstep = tran->step->tstep
        v1    = tran->step->v1
        v2    = tran->step->v2
        tr    = tran->step->tr
        slew  = (v2-v1)/tr
      }
    }                            # end parameters section
    values {
      vn = v(p) - v(m)
      if (dc_domain|time_domain) {
        if (union_type (tran,sin))  {
          if (time <= td) {
            vs = vo
            next_time = td
          }
          else {
            vs = vo + va*sin(w*(time-td))
            step_size = ss
          }
        }                              # end tran->sin
        else if (union_type (tran,exp)) {
          vs = v1 + (v2-v1)*(1-exp(-(time/tau)))
        }                              # end tran->exp
        else if (union_type (tran,step)) {
          if (dc_domain|(time < tstep)) {
            vs = v1
            next_time = tstep
          }
          else if ((time >= tstep) & (time < tstep+tr)){
            vs = v1 + (time-tstep)*slew
            next_time = tstep + tr
          }
          else {
            vs = v2
          }
        }
        else vs = supply
      }                       # end dc_domain|time_domain
      else vs = 0
    }                         # end values section
    equations {
      i(p) += is
      i(m) -= is
      is : vn = vs
    }                         # end equations section
  }
  ASCII text of this example is located in:
  install_home/example/MASTtemplates/structured/
  vsource_2.sin
```

Header Declarations

The template header has two arguments, supply (for constant output) and tran (for time-varying output) as follows:

```
1 element template vsource_2 p m = supply, tran
```

The tran parameter is not a simple type, so there is an example of how to use it in a netlist entry later in this section.

As always, header declarations declare the names used in the header. These are the names of the pins p and m, and the arguments supply and tran.

```
electrical p, m
number supply = 0
```

However, the tran argument, which must be able to represent any of three transient waveforms, cannot be defined simply as a number. It must be declared as a new type of parameter, the union parameter, which is described in the following section.

Union Type Parameters

You want to be able to specify any of the following kinds of signals when choosing the tran argument:

- Sine Wave Output (sin) Declaration
- Exponential Wave Output (exp) Declaration
- Step Function Output (step) Declaration

Each of these is complex enough to require its own list of parameters to define the signal function. When only one of a list of parameters can be used at a time, it is best to declare the list as a part of a union. A union is a parameter type that has multiple members, but each time the union is used, only one of the members is selected. The general form of a union declaration is:

```
union  {definition} name [ = ([initial values]) ]
```

Notice that this is different than the specification for an enumerated type (enum). An enum allows the selection of one out of a list of constant values. A union allows the selection and specification of one out of a group of possible members. Notice that the union specification is similar to the general form of the structure declaration. Initial values and defaults (optional) have the same meanings for unions as for structures. The following example is from the vsource_2 template,

```
union {
  number                          off
  struc {number vo, va, f, td;}   sin
  struc {number v1,v2,tau;}       exp
  struc {number v1,v2,tstep,tr;}  step
} tran = (off = 1)
```

The union definition contains a declaration for each member of the union. Each member may be of any type, even a structure or another union. In this example, the members are the three signal types defined for tran; sine, exponential, and step. In addition, there should be a parameter to turn the entire union (named tran) on and off. When on, this enables the tran parameter to override the supply argument. When off (tran=(off=1)), the supply argument is in effect.

As shown above, there are the following four members in this union:

off	disable tran (the only initialized parameter)
sin	sine wave output
exp	exponential wave output
step	step function output

The off parameter is declared a number while sin, exp, and step, each of which contains other parameters, are declared to be structures. Thus, this union consists of one number and three structures—selecting any one of these four in a netlist excludes the other three. The three structures are discussed in the following topics.

Although off is initialized in the template by setting off=1, this is not a Boolean function. In other words, the action of explicitly setting off to any value (even undef or 0) selects it and excludes the other three members of the tran union.

Sine Wave Output (sin) Declaration

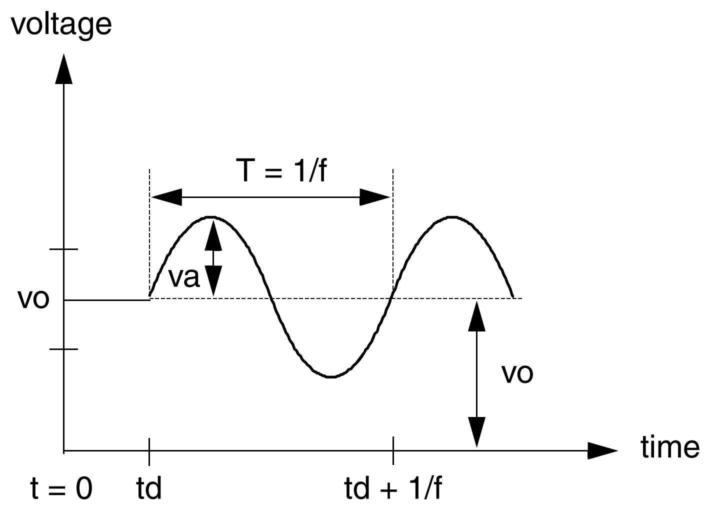
The equation for defining a sine wave, as shown in the following figure, is:

v = vo + va * sin (2πf (time - td))

where:

vo	the offset value in volts
va	the amplitude in volts
f	the frequency in hertz
td	the delay in seconds

All of these have numerical values and therefore, are declared as numbers within the sin structure. In addition, no initial values are assigned. Because time is a simvar, its value is provided by the simulator and does not need to be declared.



Describing a sine wave

Thus, the declaration for sin is as follows:

```
6   struc {number vo, va, f, td;} sin
```

Exponential Wave Output (exp) Declaration

The exponential signal, as shown in the following figure, is defined with the following equation:

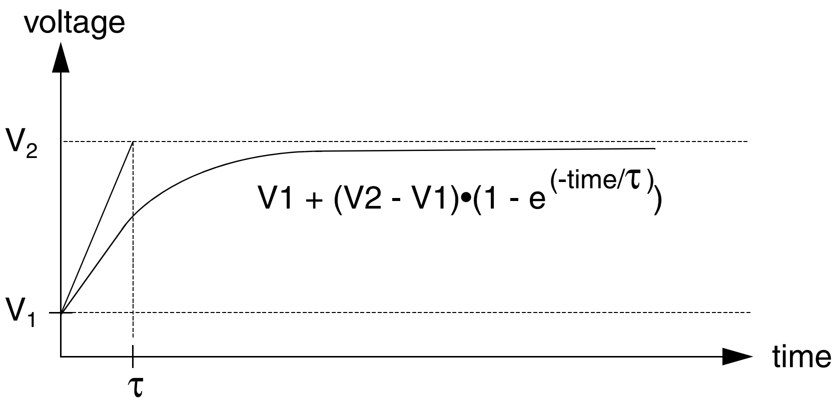
$v = v1 + (v2 - v1) * (1 - e^{-(time/tau)})$

where:

v1	the initial voltage
v2	the voltage at time=inf (infinite)
tau	the time constant

The structure declaration for exp is as follows:

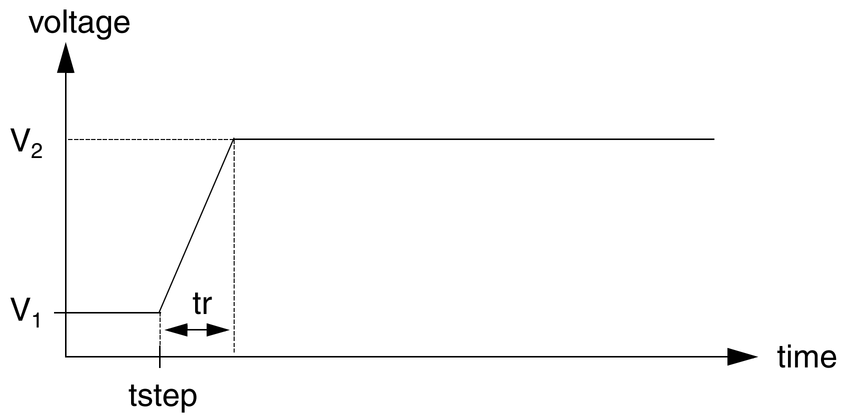
```
7   struc {number v1, v2, tau;} exp
```



Describing an exponential waveform

Step Function Output (step) Declaration

The step function, shown in the following figure, is defined as a stepped voltage from v1 to v2,



Describing a step function waveform

where:

v1	the initial voltage
v2	the step voltage level
tstep	the time at which the step begins
tr	the transition time from v1 to v2

The structure declaration for step is as follows:

```
8   struc {number v1, v2, tstep, tr;} step
```

Initial Values

The default value of off=1, sets the initial choice for tran. This default setting disables the transient waveforms unless tran is explicitly set to sin, exp, or step, thereby overriding off. When tran is set to one of these waveforms, it overrides the supply parameter, which satisfies one of the design considerations stated earlier.

The arguments of the sin, exp, and step structures are not initialized and have no default values. Thus, you must specify their values in a netlist entry whenever you set tran to sin, exp or step. Remember that this template does not include the parameter checking (out-of-range, divide-by-zero, etc.) that is incorporated into MAST library templates.

Also, recall that the declaration syntax of a structure requires that the closing brace (}) be on a separate line. However, inserting a semicolon (;) has the same effect as moving to a new line.

The complete set of parameter declarations for vsource_2 is:

```
number supply = 0
union {
  number off
  struc {number vo, va, f, td;}      sin
  struc {number v1, v2, tau;}        exp
  struc {number v1, v2, tstep, tr;}  step
} tran=(off=1)
```

The time simvar variable is part of the definition of each of the transient waveforms. Because it is a reserved word, it does not require a declaration (although you may define a variable named “time,” which will override the simvar time).

Netlist Example

Assume you want to specify a sine wave source (overriding any DC supply characteristics) with instance name input, connected to nodes named in and 0, and having the following sine wave output characteristics:

- 0 V offset (vo=0)
- 4.3 V amplitude (va=4.3)
- 1 kHz frequency (f=1k)
- 0 s delay (td=0)

The corresponding netlist entry for this would be as follows:

```
vsource_2.input in 0 = tran=(sin=(vo=0,va=4.3,f=1k,td=0))
```

or, specifying argument values without argument names, it would be the following:

```
vsource_2.input in 0 = tran=(sin=(0,4.3,1k,0))
```

Because none of the parameters are initialized in the template, all values of sin must be assigned in the netlist entry, even those specified as 0.

Local Declarations

The following declarations are required for use throughout the template:

- pi, the name used to represent the number p, defined here as 3.14159
- The branch current is
- The intermediate variables vn and vs, as a val
- Various numbers used in determining intermediate values

Therefore, the local declarations are:

```
number pi = 3.14159
val v vn, vs
var i is
number td,vo,va,w,ss,v1,v2,tau,tstep,tr,slew
```

Equations Section

The template equation is similar for this voltage source as it is for the voltage source template (vsource_1). The major difference is that here vs is defined with more options. These are based on which element of the tran union is specified, plus which simulation analysis is being performed.

```
equations {           ### FROM vsource_2
  i(p) += is
  i(m) -= is
  is : vn = vs
}                     # end equations section
```

```
equations {          ### FROM vsource_1
 i(p->m) += is
 is: v(p)-v(m)=vs # determine current contributed
                  #  by source
}                 # end of equations section
```

Determining Union Elements

As in the template vsource_1, several conditional statements are used to define the output term, vs. In this template, the definition of vs depends on the following conditions of the tran argument:

- If the tran argument is not selected (the default condition), then vs=supply
- If the tran argument is selected, then one of the sin, exp, or step parameters has been specified. The value of vs then depends on defining the corresponding waveform.

  In doing this, the template must use the values specified for sin, exp, or step, depending on which one has been specified for a given netlist entry.

However, you cannot use values from sin, exp, or step directly. This is because they are contained within a union and must be indirectly referenced, just as members of a structure must be indirectly referenced.

You need to use the structure reference operator (->) for indirectly referencing a variable inside a union of structures. The general syntax for using this operator is as follows:

```
union_name->structure_name->variable_name
```

For example,

```
td = tran->sin->td
```

This assigns the specified value of td (which is contained within sin, which is contained within tran) to the internal variable, td. (Refer to the MAST Reference Manual for more information on the structure reference operator, ->).

Assigning Internal Values

Structure referencing of values from the sin, exp, and step structure parameters is required so that they can be used to define vs for the appropriate output waveform. To do this, if-else statements are used along with an intrinsic MAST function, called union_type. which specifies a member of a union according to the following syntax:

```
union_type  (defined_union parameter, member)
```

This is a Boolean function whose value is true when a member of the union has been defined for a given instance of this template (i.e., whether it has values passed to it from a netlist). For example, the following statements provide a true/false indicator for each structure of tran:

```
union_type (tran, sin)
union_type (tran, exp)
union_type (tran, step)
```

As a result, union_type can be used with if-else statements and the structure reference operator (->) to make values nested within tran available for calculations of vs. These calculations are performed only for the signal (sin, exp, or step) that has been specified in a netlist.

```
if (union_type (tran,sin)) {
  td  = tran->sin->td
  vo  = tran->sin->vo
  va  = tran->sin->va
  w   = 2*pi*tran->sin->f
  ss  = 0.05/tran->sin->f
}
else if (union_type (tran,exp)) {
  v1  = tran->exp->v1
  v2  = tran->exp->v2
  tau = tran->exp->tau
}
else if (union_type (tran,step)) {
  tstep = tran->step->tstep
  v1    = tran->step->v1
  v2    = tran->step->v2
  tr    = tran->step->tr
  slew  = (v2-v1)/tr
}
```

Note that the sin portion calls an intrinsic function named sin, and the exp portion calls an intrinsic function named exp. These are functions included with the Saber Simulator to perform the sine and exponential (e) functions. Many mathematical functions are available as intrinsic functions (see the MAST Reference Manual for more information).

Refer also to the MAST Reference Manual for more information on the if-else statement and the structure reference operator (->).

Performing Calculations (Defining Signals)

The output of the voltage source (vs) is calculated using the values from either sin, exp, or step, as determined by the value of union_type. Because vs is conditional upon the tran argument, these calculations also use if-else statements and the union_type function.

These conditions can be defined in the template with the if statement, according to the following logic:

```
if (the large-signal analysis is selected) {
  define vs equal to supply, unless }

if (the sine wave is selected) {
  define vs as a sine function}

else if (the exponential is selected)  {
  define vs as an exponential function}

else if (the step is selected) {
  define vs as a step function}
```

Substituting from the template, use the union_type function with if statements to identify the members of the tran union. Thus, the logical statements above are converted to the following MAST statements:

```
if (dc_domain | time_domain) {
  if (union_type (tran, sin)) {
 # define vs as a sine wave}
  else if (union_type (tran, exp)) {
 # define vs as an exponential wave}
  else if (union_type (tran, step)) {
 # define vs as step function}
}

else vs = supply
```

Each of these logical steps uses the union_type function to select the appropriate tran choice (sin, exp, step) and then calculate their respective voltages.

Sine Wave Output

The sine wave voltage is defined with respect to time as:

v = vo + va * sin(2*pi*f*(time - td))

where:

vo	the offset value in volts
va	the amplitude in volts
f	the frequency in hertz
td	the delay in seconds

You cannot use the vo, va, f, and td values directly; you must reference them indirectly using the structure reference operator (->). For convenience, this was done previously in the template for all the tran structures, although it could have been done here.

If the time is less than or equal to the delay time, the output voltage (vs) is constrained to the value of the offset voltage, vo. This requires a subsidiary if statement, written as follows:

```
if (time <= td) {
  vs = vo
```

The complete sine wave definition then becomes:

```
if (union_type (tran,sin)) {
  if (time <= td) {
    vs = vo
    next_time = td
  }
  else {     # if (time > td)
    vs = vo + va*sin(w*(time-td))
    step_size = ss
  }
}
```

Also, notice the use of the assignment to the step_size variable as follows:

```
step_size = ss
```

The use of the step_size simvar variable allows the template to influence the size of the time step allowed during the transient simulation. The value of step_size places an upper bound on the variable step size performed by the Saber Simulator. In this instance, this is necessary to ensure an adequate resolution of the output, making it look like a sine wave. Here, step_size is assigned the value of ss, which was defined previously by the following statement:

```
ss = 0.05/tran->sin->f
```

This limits the step size for this example to 5% of the period of the sine wave. The step_size simvar is one of only two simvar variables that can be assigned a value in a template. The other is next_time (see the topic titled "Step Function Output"). Like all other simvar variables, they need not be declared.

Exponential Waveform Output

The exponential waveform uses a similar approach. The equation for the output voltage with respect to time is:

$v = v1 + (v2 - v1) * 1 - e^{-(time/tau)}$ where:

v1	the initial voltage
v2	the voltage at time=inf (infinity)
tau	the time constant

As with the sine wave, you cannot use values for v1, v2, and tau directly; you must reference them indirectly using the structure reference operator (->). The complete section for the exponential is as follows:

```
else if (union_type (tran,exp)) {
  vs = v1 + (v2-v1)*(1-exp(-(time/tau)))
}
```

Note that exp, the exponential function, is another intrinsic function. Note also that step_size is not used in the exponential example. Typically, step_size is not necessary. This is especially true of complex systems, where the complexity of the system forces the time steps to be small enough to ensure the desired effect. However, the step_size construct is supplied to give template writers as much control as they might need.

Step Function Output

A similar approach applies to the step function output. The step function is defined as a stepped voltage from v1 to v2, where:

v1	the initial voltage
v2	the stepped voltage
tstep	the start time of the step
tr	the transition time from v1 to v2

The complete step function section is as follows:

```
else if (union_type (tran,step)) {
  if (dc_domain|(time < tstep)) {
    vs = v1
    next_time = tstep
  }
  else if ((time >= tstep) & (time < tstep+tr)){
    vs = v1 + (time-tstep)*slew
    next_time = tstep + tr
  }
  else {
  vs = v2
  }
}
```

Using if statements (if-else) allows you to specify each region of the step function and determine its voltage appropriately. Before time=tstep, the voltage should be v1. After time tstep + tr, the voltage should be v2. During the transition from v1 to v2, the voltage should be determined by linear interpolation between v1 and v2.

It is very important that there be a simulation time step at the exact points where the transition starts (tstep) and ends (tstep + tr). This prevents the Saber Simulator from skipping over abrupt changes as functions of time in the step function—it is forced to go through the transition and use the correct points between steps.

Consequently, a MAST construct is required to let the model tell the simulator the time points that must have corresponding simulation time steps. This construct is the next_time simvar variable, which corresponds to a time at which the simulator must perform a time step. The next_time simvar variable is invoked only in an assignment statement. Its effect expires after each time step, regardless of whether the appropriate time point has been passed. Although this simvar variable is effective only for the selection of the very next time step, it works here because this portion of the template will be evaluated at every time step.

Therefore, the time region that precedes the start of the step transition has a statement assigning the value of tstep to next_time. Also, the transition region assigns the value of tstep + tr to next_time (the end of the transition). This guarantees time steps at the necessary locations.

No tran Output

Although it is possible to specify what occurs if off is selected (in a manner similar to that for sin, exp, and step), it is not necessary to do so. This is because the desired effect for off=1 is to leave the value of vs at the voltage specified by the supply parameter. That automatically occurs when none of the three if-else conditions are true.

All definitions for vs are as follows:

```
if (dc_domain|time_domain) {
  if (union_type (tran,sin)) {
    if (time <= td) {
      vs = vo
      next_time = td
    }
    else {     # if (time > td)
      vs = vo + va*sin(w*(time-td))
      step_size = ss
    }
  }
  else if (union_type (tran,exp)) {
    vs = v1 + (v2-v1)*(1-exp(-(time/tau)))
  }

  else if (union_type (tran,step)) {
    if (dc_domain|(time < tstep)) {
      vs = v1
      next_time = tstep
    }
    else if ((time >= tstep) & (time < tstep+tr)){
      vs = v1 + (time-tstep)*slew
      next_time = tstep + tr
    }
    else {
      vs = v2
    }
  }
  else vs = supply
}
else vs = 0
```

Netlist Examples

The following examples show how this template could be used in a netlist entry. For the sake of simplicity, the instance name in all the examples is src, and connection points are declared to be connected to nodes 1 and 2.

1. To designate a DC source src with a value of 5 volts:

```
vsource_2.a 1 2 = supply=5
```

or

```
vsource_2.b 1 2 = 5
```

2. To assign the following characteristics to the sin structure of the tran union:

offset voltage	vo	0 V
amplitude	va	4.3 V
frequency	f	1 kHz
delay time	td	0 s

```
vsource_2.c 1 2 = tran=(sin=(0,4.3,1k,0))
```

Notice that because the argument values in this example are assigned in the same order in which they are declared in the template, it is not necessary to specify the name of each argument. If you do not know the order or wish to write the names for clarity, simply specify the name of the field and an equals sign (=) to the left of the value, as follows:

```
vsource_2.d 1 2 = tran=(sin=(vo=0,va=4.3,f=1k,td=0))
```

17

Modeling Noise

The Saber simulator can perform a noise analysis to include the noise contributions of a circuit or system element. To do this, the template must contain information that defines its noise contribution.

The following topics show how noise information is added to the simple resistor and vsource (voltage source) templates:

- Adding Noise to a Resistor MAST Template -- shows adding noise information to a template, and the use of the noise_source statement in the control section.
- Adding Noise to a Voltage Source MAST Template
- Adding Noise to the MAST diode Template

Introduction

In general, a noise source for an electrical element is defined either as a current or voltage source between two nodes of the element. For a simple element, such as a resistor, there is a single noise source. For a more complex element, such as a transistor, there may be several noise sources, as well as several types of noise: thermal noise, shot noise (due to DC current), and flicker noise (a frequency-related noise).

Adding noise information to a template is not difficult, and the procedure is the same for all types of noise. In general, it consists of the following:

- Define the name of the noise source as a val variable (a local variable).
- Provide the defining expression for the noise variable (the val variable).
- In the control section, insert a noise_source statement that supplies one of the following kinds of information:
    - If the noise source is a current source, the statement describes the location of the noise source in terms of the connection points or internal nodes.

- If the noise source is a voltage source, the statement associates the name of the noise source with a var variable (a system variable).

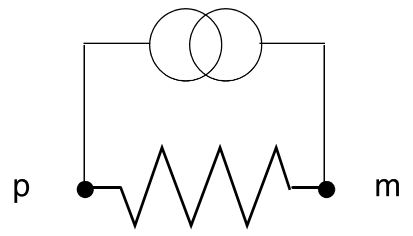
If there is more than one noise source, the control section must contain a separate variable, definition, and statement for each.

Adding Noise to a Resistor MAST Template

For reference, the resistor template without the noise functionality is shown as follows:

```
template resistor p m = res
  electrical p, m
  number res
{
  equations  {
    i(p->m) += (v(p)-v(m))/res
  }
}
```

For this example, only the thermal noise through the resistor will be added. This noise source is defined as a current source in parallel with the resistor, as shown in the figure below. Note that there is no direction associated with the current source.



Defining a noise generator

To include thermal noise effects in the template, the following expression is used to define them:

```
noise = (abs(4kT/r))
```
$^{1/2}$

where:

k	is Boltzmann's constant ($1.38 * 10^{-23}$ joules/K)
T	is the temperature in K
r	is the specified resistance

Note that it is necessary to declare variables for Boltzmann's constant and temperature, in addition to the noise source variable. The following resistor_2 template includes the noise functionality:

```
template resistor_2 p m = res
  electrical p, m
  number res
  external number temp            #noise-related
{
  val ni nsr                      #noise-related
  number k = 1.38e-23             #noise-related
  number t                        #noise-related
  parameters  {
    t = temp + 273.15             #noise-related
  }
  values  {
    nsr = sqrt(abs(4.0*k*t/res))  #noise-related
  }
  control_section  {
    noise_source (nsr, p, m)      #noise-related
  }
  equations {
    i(p->m) += (v(p)-v(m))/res
  }
}
```

```
ASCII text of this example is located in:
install_home/example/MASTtemplates/structured/
resistor_2.sin
```

The following topics describe the resistor_2 template:

- Header Declarations
- Local Declarations
- Expression for Noise
- Control Section -- shows how to use the noise_source statement in the control section.

Header Declarations

The variable for simulation temperature (temp, in ×C) is declared in the header declarations section:

```
external number temp            #noise-related
```

Local Declarations

A unit is provided for thermal noise (ni) generated by a current source, which is defined in A/÷Hz. By declaring a noise variable as a val variable, you can assign the unit ni to it. For example, a noise variable named nsr would be declared as follows:

```
val ni nsr
```

The constants used to calculate noise must also have local declarations:

```
number k = 1.38e-23
number t
```

Because the value defined externally for temp is in ×C, a statement is required to convert the temperature (t) to kelvins:

```
t = temp + 273.15
```

Expression for Noise

Using these variable names for the noise generator, insert a statement to perform the noise calculation as follows:

```
values  {
  nsr = sqrt(abs(4.0*k*t/res))  #noise-related
}
```

The statement in the values section uses two intrinsic functions: sqrt, the square root function, and abs, the absolute value function.

Control Section

A noise_source statement is used in the control section. It identifies the noise source in relation to the rest of the template. If the noise source is a current source, as in this template, the statement contains the name of the pins (or internal nodes) to which the noise generator is connected. If one side of it is connected to ground, only the other need be listed, in which case the simulator assumes that the other side is grounded.

In this example, the noise source nsr is connected between pins p and m. Therefore, the complete control section is as follows:

```
control_section {
  noise_source (nsr, p, m)      #noise-related
}
```

The noise_source statement adds the noise to p and subtracts it from m. Alternately, because the noise analysis ignores the sign of the noise source, the following statement would be an equivalent statement (swapping positions of m and p):

```
noise_source (nsr, m, p)
```

For a noise current source, the general form of the noise_source statement in the control section is as follows:

```
noise_source (val_name, pin [, pin])
```

For a noise voltage source, the form of the statement would be as shown below, where var_name is the name of a var variable defining the current through the voltage source.

```
noise_source (val_name, var_name)
```

Adding Noise to a Voltage Source MAST Template

Adding a voltage noise source requires the following three types of statements:

- Define the name of the noise source as a val variable.
- Provide the defining expression for the noise variable (the val variable).
- In the control section, insert a noise_source statement that associates the name of the noise source with a var variable.

However, because a var variable is required in the noise_source statement for this template, the var variable must appear in a template equation (as implemented for the opamp template.

Adding the necessary noise source statements to the vsource template makes it the vsource_3 template shown below. These statements are indicated with comments.

Note that the noise voltage has been made available as an argument (noise), which is then assigned to the noise val variable, nsv.

```
template vsource_3 p m = vs, noise
  electrical p, m
  number vs, noise           # add argument for noise
{
  var i i
  val nv nsv                 # (1) declare noise val
  values {
    nsv = noise              # (2) set value of noise val
  }
  control_section {
    noise_source(nsv, i)     # (3) associate noise val
  }                          #     with var i
  equations {
    i(p->m) += i
    i: v(p)-v(m) = vs
  }
}
ASCII text of this example is located in:
install_home/example/MASTtemplates/structured/
vsource_3.sin
```

Adding Noise to the MAST diode Template

The diode template defines a simplified diode model. In a more fully-defined diode model, such as the d template in the Standard Template Library, all three types of noise are defined. However, this template incorporates only the shot noise from the DC current.

The shot noise is defined as a current source, and it is connected between pins p and m. The defining equation for shot noise is as follows:

```
nsi = sqrt(2*qe*abs(id))
```

where qe is the charge on the electron and id is the current through the diode (both previously defined in the template).

Therefore, only three statements need be added to this template to define shot noise:

- Define the name of the noise source as a val:

```
val ni nsi
```

- Provide the defining expression for the noise variable:

```
nsi = sqrt(2*qe*abs(id))
```

- In the control section, insert a noise_source statement that associates the name of the noise source with a var variable:

```
noise_source(nsi, p, n)
```

From this example, it should be clear that adding noise information is a very straightforward process, regardless of the complexity of the template. Adding the other noise information (thermal noise and/or flicker noise) is simply a matter of defining each necessary variable, adding its defining expression, and inserting its noise_source statement in the control section of the template.

18

Statistical Modeling

Statistical modeling (also known as a Monte Carlo analysis) includes the features that are described in the following topics:

These features enable you to define a model with built-in variability. Then the simulator, running an mc (Monte Carlo) command, uses the model to run a series of simulations, where each simulation uses a new set of values for the variable parameters.

Thus, it is possible to use the Saber Simulator in a statistical or non-statistical (deterministic) environment.

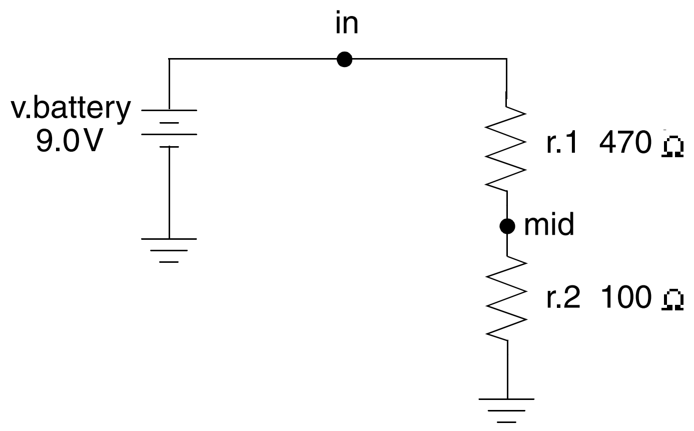
Introduction

Models for a particular component or design are described with equations and accompanying coefficients (model parameters). In some models the model parameters are assumed to be constants that characterize the object. This is a good assumption when the model represents a single sample of a component or circuit. However, another sample of the same component or circuit might be better characterized by a slightly different set of model parameters due to the tolerances of the components.

The topic of statistical modeling introduces the notion that a model parameter may be best described as a collection of possible values, with each value having its own likelihood of occurring. Statistical modeling describes the process of varying model parameters in a precise, yet random, way. This method defines parameters statistically.

Varying Values in a Simple Voltage Divider

Assume you are designing the simple voltage divider circuit shown in the following figure, consisting of two resistors and a voltage source.



Simple voltage divider

For simulation, this circuit has the following netlist:

```
v.battery in 0   = 9
r.1       in mid  = 470k
r.2       mid 0   = 100k
```

The voltage source is a battery whose voltage varies slightly, depending both on the lot from which it was produced and its age. Assume the voltage varies uniformly from 8.9 to 9.1 volts.

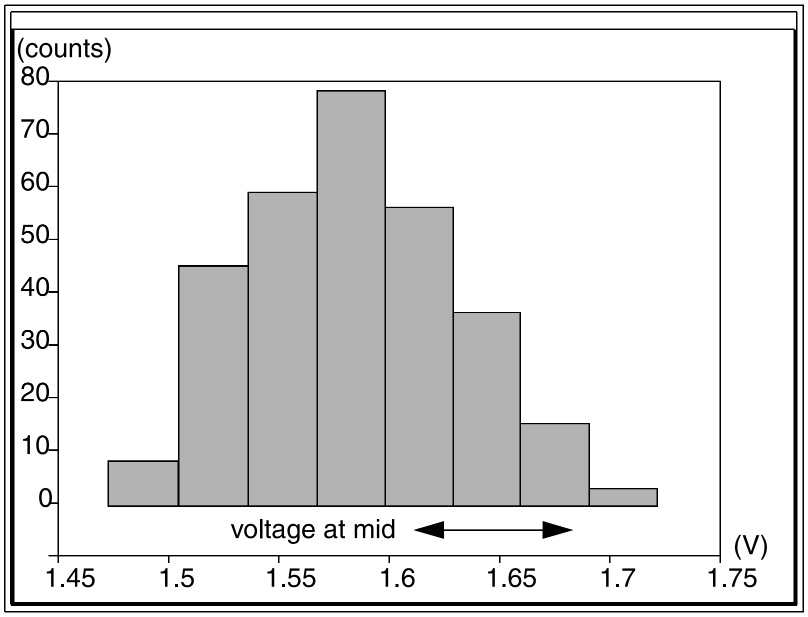
The resistors available are 100k resistors with gold outer bands (5% tolerance) and 470k resistors with silver outer bands (10% tolerance). You do some measurements and discover that the resistor variation has a normal distribution around the nominal value. The standard deviation is approximately one third of the tolerance, so that, for example, ±0.10*470000, would be the tolerance for the 470k resistor. Statistically, 99.7% of the 470k resistors will actually have resistances between 423k (470k - 47k) and 517k (470k + 47k).

The goal is to determine the nominal value of the voltage at the mid node, along with the expected distribution of this voltage based on the distributions of the battery voltage and resistor values.

A simple DC analysis using this circuit produces the result that the voltage at mid is 1.5789 volts. To add the variations to this model, include the uniform and normal intrinsic distribution functions in the netlist:

```
v.battery in 0    = uniform(9, 8.9, 9.1)
r.1        in mid  = normal(470k, 0.1)
r.2       mid 0    = normal(100k, 0.05)
```

You can then perform a Monte Carlo DC analysis with this circuit, which produces many DC analysis results—each one randomly varying all distributed parameters. If you simulate enough times (the number of simulations is a Monte Carlo analysis parameter), you can clearly see the distribution of the voltage at mid (see the figure below). The normal distribution in the figure below was produced with 300 simulations; the average is near the expected 1.5789 volts.



Histogram of voltages at `mid`

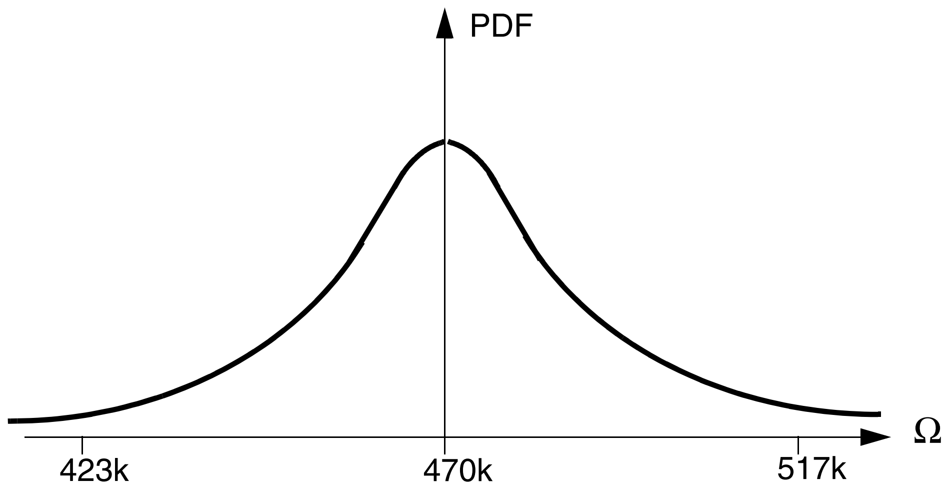
From this analysis, observe that the voltage at mid is likely to vary between 1.45 and 1.7 volts. This information lets you decide either to accept this variation as being within your design specification or to adjust your design to compensate for the variation.

Probability Density Functions (PDFs)

The primary tool for describing model parameter variations is the probability density function (PDF). Among the types of PDFs are the following:

- Intrinsic Probability Density Functions
- Uniform Probability Density Function
- Normal Probability Density Function
- Piecewise Linear Probability Density Function

The PDF is a continuous function of an independent variable, say x, such that, for real numbers a and b, the probability that a random value of x will be between a and b is the area under the PDF curve between a and b. For example, the following figure shows the distribution of the resistor r.1, which is used in the voltage divider example in the preceding topic. The uniform and normal intrinsic functions, used in the voltage divider example, are nothing more than predefined probability density functions.



Normal PDF for resistor `r.1`

The X-axis is the value of resistance; the Y-axis is the probability density. The distribution is normal in this example. The normal distribution has the familiar bell-shaped curve. The average (also called expected) value of the normal distribution corresponds to the peak of the curve. This means that the values of resistance (X-axis values) with the highest probability of occurring (Y-axis value) are those near the average value. The total area under the curve, from -infinity to infinity, must equal 1. This means that, for r.1 in the example, the probability is 1 that the resistance value of a resistor taken from a bin full of 470k resistors will be between-infinity and infinity. This obviously must be true.

Intrinsic Probability Density Functions

The topic titled "Probability Density Functions (PDFs)" on the previous page introduces two examples of intrinsic PDFs: normal and uniform. Another intrinsic PDF is the piecewise linear (pwl). All three are discussed in more detail below.

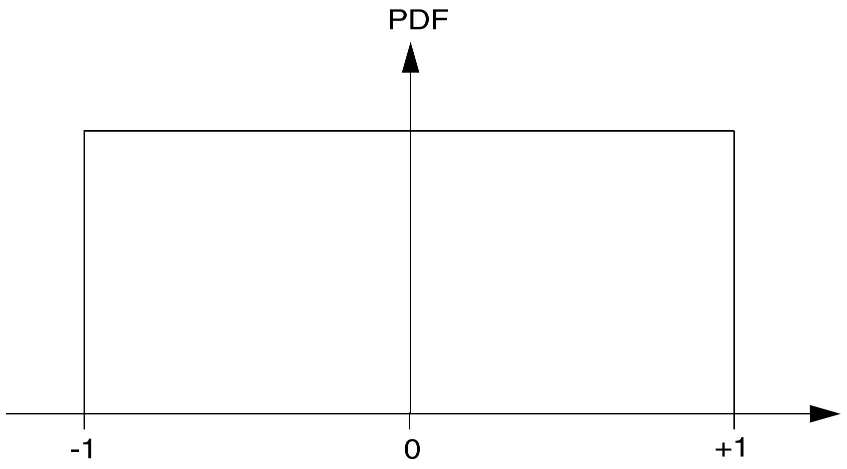
It is useful to think of a PDF as follows:

- Begin with the basic, normalized, distribution, called the prototype distribution (MAST provides three intrinsic functions: normal, uniform, and pwl).
- Use a scaling multiplier (that stretches or compresses the distribution) and an offset factor (that shifts it right or left) to customize the prototype distribution for the application. The resulting distribution is called the actual value distribution.

In other words, you first describe the prototype distribution, then you define the parameters that modify the prototype into the corresponding actual distribution.

Uniform Probability Density Function

The prototype distribution for the uniform probability density function is illustrated in the figure below. The default nominal value is 0 and the limits are 1 and -1.



Uniform prototype distribution

By passing parameters to the uniform function, you can modify the prototype uniform distribution to shift the nominal value and scale the limits. The uniform function has several forms, including the following two:

```
uniform(nominal_value, lower_limit, upper_limit)
uniform (nominal_value, tolerance)
```

where the arguments have the following meanings:

nominal_value	the listed, or stated value; often, the intended value
lower_limit	the smallest value of the distribution
upper_limit	the largest value of the distribution
tolerance	a value greater than -1 and less than 1, such that the limits of the distribution are nominal_value + tolerance*nominal_value and nominal_value - tolerance*nominal_value

This second form of the uniform function shown above is used if the uniform distribution is symmetrical with respect to nominal_value.

Regardless of how you specify an actual value uniform distribution from its corresponding prototype distribution, it must satisfy the following requirements:

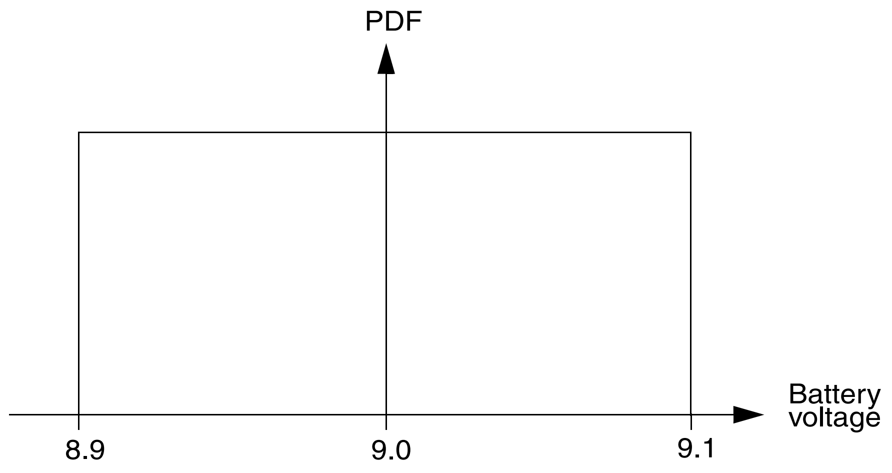
- The nominal_value of the actual value distribution corresponds to the 0 value of the prototype distribution.
- The lower_limit of the actual value distribution corresponds to the -1 value of the prototype distribution.
- The upper_limit of the actual value distribution corresponds to the 1 value of the prototype distribution.

Note that these correspondences are maintained even if the prototype distribution is changed.

The battery voltage source example from the topic titled "Varying Values in a Simple Voltage Divider" illustrates using the uniform distribution:

```
v.battery in 0 = uniform(9, 8.9, 9.1)
```

This means that all values between 8.9 volts and 9.1 volts are equally likely. The figure below shows this actual value uniform distribution for battery voltage.



Uniform actual value distribution

The same effect could have been achieved by specifying a tolerance value rather than limits as follows:

```
v.battery in 0 = uniform(9, 0.01111)
```

The uniform function can detect that if two arguments are specified, they indicate the nominal value and the tolerance; whereas three arguments indicate the nominal, minimum, and maximum values, respectively.

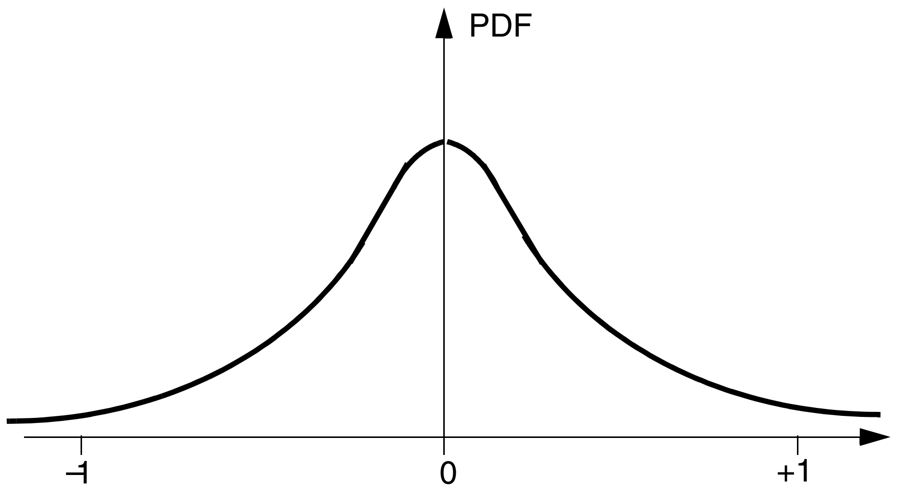
The uniform function can have other arguments as well as described in the topic titled "Modifying a Uniform Prototype Distribution".

Normal Probability Density Function

The prototype distribution for the normal probability density function is illustrated in the figure below. The default nominal value is 0 and the limits are 1 and -1. The limits correspond to the 3s and -3s points, respectively, where s is the normal distribution's standard deviation.

By passing parameters to the normal function, you can modify the prototype normal distribution to shift the nominal value and scale the limits. The normal function has several forms, including the following:

```
normal(nominal_value, lower_limit, upper_limit)
normal(nominal_value, tolerance)
```



Normal prototype distribution

where the arguments have the following meanings:

nominal_value	The listed or stated value; often, the intended value.
lower_limit	The -3s value (when using the default) of the distribution, where s is the standard deviation of the normal distribution. You may change the multiplier of s from its default of -3.
upper_limit	The +3s value of the distribution, where s is the standard deviation of the normal distribution. You may change the multiplier of s from its default of 3.
tolerance	A value greater than -1 and less than 1, such that the limits of the distribution are nominal_value + tolerance*nominal_value and nominal_value - tolerance*nominal_value. The tolerance specifies the ±3s limit.

The second form of the normal distribution shown above applies if the normal distribution is symmetrical with respect to nominal_value.

The normal function automatically detects that if two arguments are specified, they indicate nominal value and tolerance, whereas three arguments indicate nominal_value, nominal_value - 3s, and nominal_value + 3s, where the two 3s values need not be equal. If they are unequal, then the left side of the distribution is normal with one standard deviation, while the right side is normal with a different standard deviation.

Regardless of how you specify an actual value normal distribution from the normal prototype, it must satisfy the following requirements:

- The nominal_value of the actual value distribution corresponds to the 0 value of the prototype distribution.
- The lower_limit of the actual value distribution corresponds to the -1 value of the prototype distribution, which corresponds to the -3s.
- The upper_limit of the actual value distribution corresponds to the +1 value of the prototype distribution.

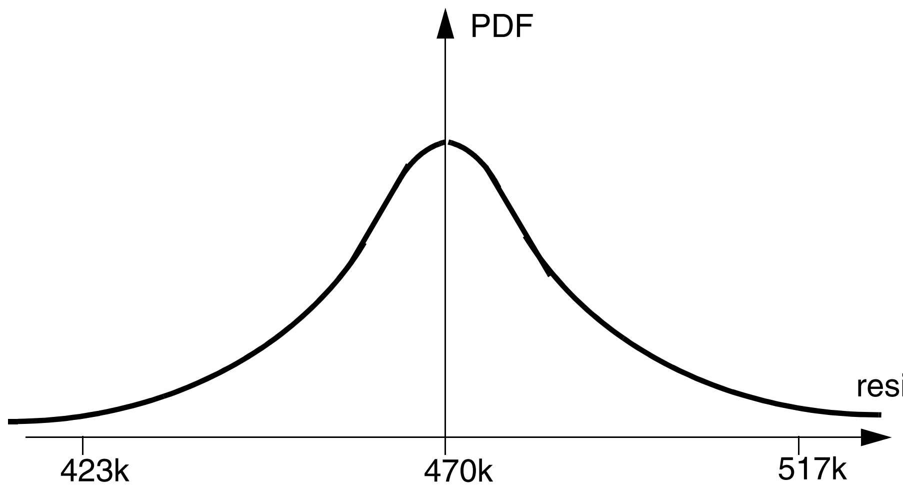
Note that these correspondences are maintained even if the prototype distribution is changed.

The resistor r.1 from the example in the topic titled "Varying Values in a Simple Voltage Divider" illustrates the use of the normal distribution:

```
r.1 in mid = normal(470k, 0.1)
```

This means that the resistance values for resistor r.1 follow the normal distribution, with a nominal value of 470 kOhm and a tolerance of 0.1. Thus, the -3s limit is 470kOhm - (470kOhm•0.1) = 423kOhm, and the +3s limit is 470kOhm + (470kOhm•0.1) = 517kOhm. Because the 3s value is 470kOhm•0.1 or 47kOhm, then the value of one standard deviation (s) would be 47kOhm/3 or 15.67kOhm.

The figure below shows the actual value normal distribution. The tolerance specifies the symmetrical ±3s limits. Using 3s as the tolerance means that, given a randomly selected resistor from the batch, the probability that its resistance lies inside the tolerance (i.e., between ±3s) is approximately 0.997.



Normal actual value distribution

You could have achieved exactly the same effect by specifying limits rather than a tolerance:

```
r.1 in mid = normal(470k, 423k, 517k)
```

The normal function can have other arguments as well, see the topic titled "Modifying a Uniform Prototype Distribution".

Piecewise Linear Probability Density Function

The piecewise linear probability density function has no default prototype PDF. The use of a piecewise linear PDF provides a great amount of flexibility. Accordingly, its use requires more complex constructs. The following steps are required to create a piecewise linear PDF:

1. Create a prototype PDF. Because the piecewise linear prototype can be anything, it must first be defined (unlike the uniform or normal prototype PDFs, which are uniquely defined).
2. Map actual values to the prototype PDF.
3. Use the resulting actual value PDF in a netlist.

These steps are expanded in the following topics to change the distribution for r.1 in the example to use a piecewise linear PDF.

1. Creating a Piecewise Linear Prototype PDF

Creating a piecewise linear prototype PDF requires a structure parameter similar to the following:

```
struc p_pwl {
  enum {_pdf,_cdf} type
  struc {number x, y;} pwl[*]
}
```

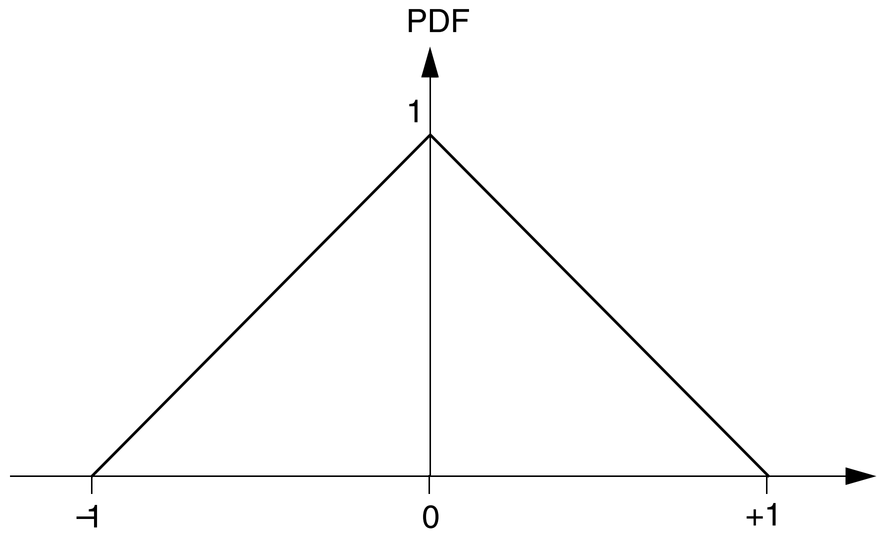
This structure (named p_pwl) declares the local variable that is to hold the prototype piecewise linear PDF. The pwl intrinsic function, to be used in the netlist specification, looks for this structure. You can use this structure in a netlist, as described in the topic titled "3. Using a Piecewise Linear Prototype PDF in a Netlist", or you can include it in a template using the MAST include construct (<) and the predefined file named distrib.sin as follows:

```
<distrib.sin
```

The distrib.sin file contains the definition of the p_pwl structure shown above (as well as other definitions of statistical distributions). Once you make this definition of p_pwl available, you can use it as a type to declare local variables of the same type, which you can then modify. A convenient way of doing this is to use the standard template, which is explained in the following topic.

Using the Standard Template You can declare a variable of type p_pwl by calling the provided template standard, which already includes the distrib.sin file (as described above). When referenced, the standard template declares an argument named p_pwl as the correct type. This allows you to use an argdef (..) declaration to declare a local variable of this type from standard. Refer to the MAST Reference Manual for information on the argdef operator. An example of doing this using a local parameter named ppwl1 is shown below.

Example Consider the triangle-shaped distribution shown in the following figure:



Example piecewise linear prototype PDF

This prototype PDF can be specified with the following declaration in a template:

```
standard..p_pwl ppwl1=(type=_pdf,pwl=[(1,0),(0,1),(1,0)])
```

The declaration of the p_pwl structure is called from the standard template and given the local name of ppwl1 for this particular template (i.e., p_pwl and ppwl1 are the same type of parameter).

The type field is initialized to _pdf, indicating that it is a PDF (as opposed to a cumulative density function, CDF, see the topic titled "Cumulative Density Functions (CDFs)"). The pwl field is an array of coordinate pairs that correspond to the points shown on the PDF. This declaration and initialization of the local variable named ppwl1 completes the creation of the prototype piecewise linear PDF called ppwl1.

If the type field is _pdf, the ordered pairs (x, y) in the pwl field must satisfy the following requirements:

- There must be at least two (x, y) pairs.
- The x values must be monotonically non-decreasing.
- The y values must be Š 0.

- The first x value (x1) must be <0, and the last (xn) must be > 0. (The simulator uses these values as truncation bounds when assigning random values to the distribution.)
- The integral of the PDF, from x1 to xn, must be positive (not necessarily 1). Note that this implies that at least one y value must be > 0.

2. Correspondence Between Actual Values and Prototype PDF Values

Once the prototype piecewise linear PDF is defined, you can use it to create an actual value piecewise linear PDF. You do this by passing parameters to the pwl function, which has the following format:

```
pwl(nominal_value, tolerance, lower_limit, upper_limit,
    prototype)
```

where:

nominal_value	the value that the distribution is to have in a deterministic (non-statistical) environment.
tolerance	either undef or a numeric value between -1 and 1—if it is a numeric value, then the upper and lower limits of the distribution are nominal_value + tolerance*nominal_value and nominal_value - tolerance*nominal_value
lower_limit	either undef or a numeric value less than nominal_value
upper_limit	either undef or a numeric value greater than nominal_value
prototype	the name of the prototype piecewise linear PDF

You can specify either nominal_value and tolerance or nominal_value and both lower_limit and upper_limit. If you specify the tolerance, then you must set lower_limit and upper_limit to undef. On the other hand, if you specify the lower_limit and upper_limit, you must set the tolerance equal to undef. You must identify prototype.

Regardless of how you specify an actual value piecewise linear distribution, it must satisfy the following requirements:

- The nominal_value of the actual value distribution corresponds to value 0 of the prototype distribution.
- The lower_limit of the actual value distribution corresponds to value -1 of the prototype distribution.
- The upper_limit of the actual value distribution corresponds to value 1 of the prototype distribution.

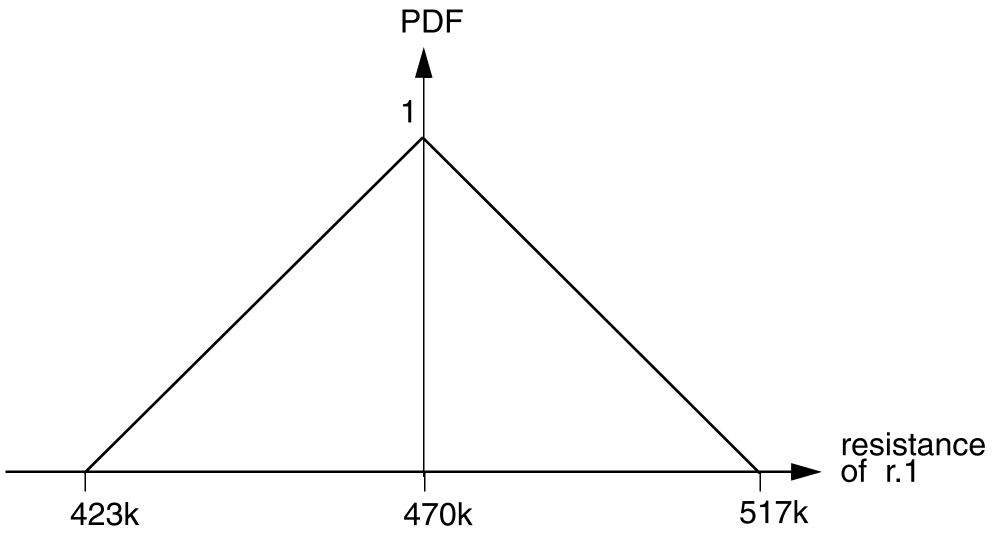
Note that these correspondences are maintained even if the prototype distribution is changed.

3. Using a Piecewise Linear Prototype PDF in a Netlist

The following example duplicates the example given at the beginning of the Statistical Modeling topic with one exception: the resistor r.1 has a piecewise linear distribution. This is implemented in the netlist as follows (note the comments):

```
# creates prototype pwl PDF
standard..p_pwl ppwl1=(type=_pdf,pwl=[(-1,0),(0,1),(1,0)])

v.battery in 0   = uniform(9, 8.9, 9.1)

# map values from pwl PDF to r.1
r.1       in mid = pwl(470k, 0.1, undef, undef, ppwl1)
r.2       mid 0  = normal(100k, 0.05)
```

The above example specifies for the resistor r.1 the actual value PDF by giving the nominal value (470k, the value used in non-statistical analyses), the tolerance value (0.1), two undefined values (for the upper and lower bound), and the name of the prototype distribution (ppwl1). The figure below shows the resulting actual value PDF for the r.1 resistor.



Actual value PDF for resistor `r.1`

The following is an alternate way of specifying the same distribution:

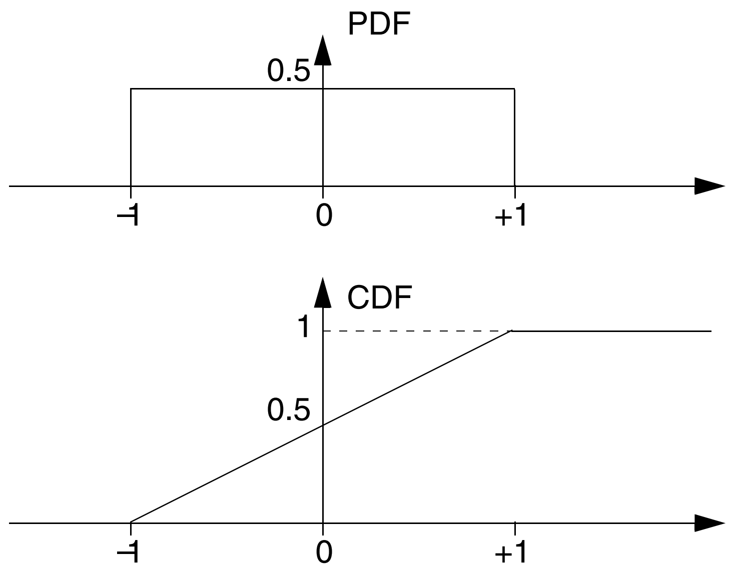
```
standard..p_pwl ppwl1=(type=_pdf, pwl=[(1,0),(0,1),(1,0)])
  # same as above

v.battery in 0   = uniform(9, 8.9, 9.1)
r.1       in mid = pwl(470k, undef, 423k, 517k, ppwl1)
  # alternate method
r.2     mid 0 = normal(100k, 0.05)
```

This example specifies, for the resistor r.1, the actual value PDF by giving the nominal value (470k), the tolerance (undef), the upper and lower bounds (423k and 517k, respectively), and the name of the prototype PDF (ppwl1).

Note that in each example either the tolerance or both upper and lower limits must be undef. It is an error to specify numeric values for all three, even though all three must have values. The pwl intrinsic function, unlike the uniform and normal functions, cannot infer the tolerance or the limits from the context of the calling sequence.

Cumulative Density Functions (CDFs)

The probability density function (PDF) is a common way of specifying the statistical variations of a design parameter. However, sometimes it is more convenient to specify a function of cumulative probability, the cumulative density function (CDF). Both the PDF and the corresponding CDF describe the same distribution, but they do so in slightly different ways as shown in the following figure:



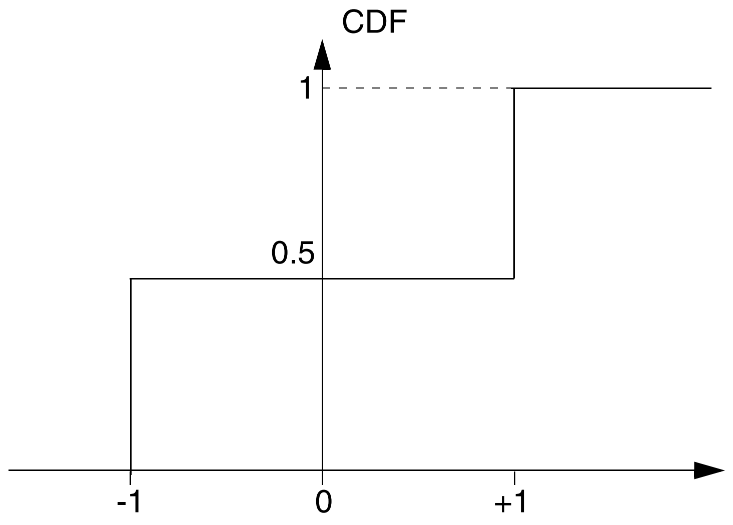
PDF and corresponding CDF

The CDF, like the PDF, is a function of x, where x ranges from -infinity to infinity. The value of a CDF at x is the integral of the PDF, evaluated between -infinity and x.

In other words, the CDF at any point x is the probability that a sample from the distribution has a value less than x. Obviously, at x equals infinity the value of the CDF function must equal 1, because the probability that a sample from any distribution will be less than infinity is 1. Correspondingly, the CDF function must equal 0 at x equals -infinity, because the probability that a sample from any distribution will be less than -infinity is 0. The figure below shows an example of a uniform PDF and its corresponding CDF.

Some distributions, such as those with only discrete values, cannot be described using a PDF. Consider, for example, an experiment that consists of flipping a coin and assigning value 1 if the coin lands with heads showing and -1 if it lands with tails showing. This experiment has a binary distribution, with heads and tails each having probability 0.5. It is not possible to describe this distribution using a PDF, because the area under the points at 1 and -1 would each have to be 0.5, but the area under any other point cannot exceed 0.

The CDF for this binary distribution, however, is easily demonstrated in the following figure:



CDF for a binary distribution

The cumulative probability that a sampled value will be less than -1 is zero. The cumulative probability that a sampled value will be less than 1 is 0.5. The cumulative probability that a sampled value will be less than any number greater than 1 is 1.

Intrinsic Piecewise Linear Cumulative Density Function

The only intrinsic CDFs provided are those that correspond to piecewise linear PDFs. The following steps are required to create a piecewise linear CDF:

- Create a prototype CDF.
- Map actual values to the prototype CDF.
- Use the resulting actual value CDF in a netlist.

These steps are expanded in the following topics to change the distribution for v.battery in the example to use a piecewise linear CDF. The result is a specification that produces simulation results identical to those of the uniform PDF specification used in the topic titled "Uniform Probability Density Function".

1. Creating a Piecewise Linear Prototype CDF

Creating a piecewise linear prototype CDF requires a structure parameter similar to the following (the same as for a prototype PDF, explained in the topic titled "1. Creating a Piecewise Linear Prototype PDF"):

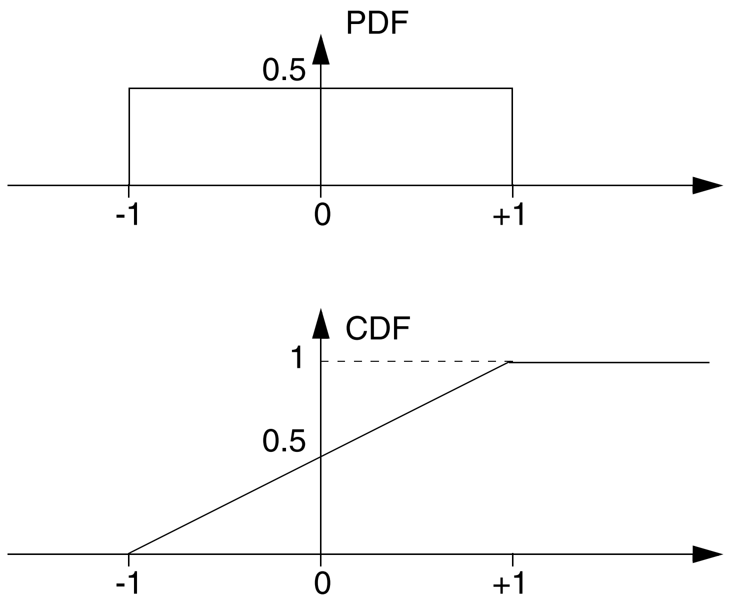
```
struc p_pwl {
  enum {_pdf,_cdf} type
  struc {number x,y;} pwl[*]
}
```

This structure (named p_pwl) declares the local variable that is to hold the prototype piecewise linear CDF. The pwl function used in the netlist specification searches for this structure. You can use this structure in a netlist, as described in the topic titled "3. Using a Piecewise Linear Prototype CDF in a Netlist", or you can include it in a template using the MAST include construct (<) and the pre-defined file named distrib.sin:

```
<distrib.sin
```

As explained in the topic titled “Using the Standard Template” for a prototype PDF, you can use the standard template, which already includes the distrib.sin file as described above. The standard template declares an argument named p_pwl as the correct type. You can then use an argdef (..) declaration to declare a local parameter of this type referenced from standard. An example of doing this using a local parameter named cpwl1 is shown below.

Example Consider the uniform prototype distribution shown in the upper portion of the following figure. It corresponds to the PDF used in the original v.battery example. The corresponding CDF is shown in the lower portion of the figure.



Prototype uniform PDF (above), corresponding CDF (below)

You can specify this prototype CDF in a template with the following declaration:

```
standard..p_pwl cpwl1 = (type=_cdf, pwl=[(-1,0),(1,1)])
```

The declaration of the p_pwl structure is called from the standard template and given the local name of cpwl1 for this particular template (p_pwl and cpwl1 are the same type of parameter). The type field is initialized to _cdf, indicating that it is a CDF (as opposed to a PDF). The pwl field is an array of coordinate pairs that correspond to the points on the CDF shown in the lower portion of the previous figure. This declaration and initialization of the variable named cpwl1 completes the creation of the prototype piecewise linear CDF called cpwl1.

If the type field is _cdf, the ordered pairs (x, y) in the pwl field must satisfy the following requirements:

- There must be at least two (x, y) pairs.
- The x and y values must be monotonically non-decreasing.
- The y values must be Š 0.
- The first x value (x1) must be < 0, and the last (xn) must be > 0. The simulator uses these as truncation bounds when assigning random values to the distribution.
- The first y value must equal 0, the last y value must be greater than 0.

2. Correspondence Between Actual Values and Prototype CDF Values

Once the piecewise linear prototype CDF is defined, you can use it to create an actual value CDF. You do this by passing parameters to the pwl function, which has the following format:

```
pwl(nominal_value, tolerance, lower_limit, upper_limit,
    prototype)
```

where:

nominal_value	the value the distribution has in a deterministic environment.
tolerance	either undef or a numeric value between -1 and 1—if it is a numeric value, then the upper and lower limits of the distribution are nominal_value + tolerance*nominal_value and nominal_value - tolerance * nominal_value
lower_limit	either undef or a numeric value less than nominal_value
upper_limit	either undef or a numeric value greater than nominal_value
prototype	the name of the piecewise linear prototype CDF

You can specify either nominal_value and tolerance or nominal_value and both lower_limit and upper_limit. If you specify the tolerance, then you must set lower_limit and upper_limit to undef. On the other hand, if you specify the lower_limit and upper_limit, you must set the tolerance equal to undef. You must identify prototype.

Regardless of how you specify an actual value piecewise linear distribution, it must satisfy the following requirements:

- The nominal_value of the actual value distribution corresponds to value 0 of the prototype distribution.
- The lower_limit of the actual value distribution corresponds to value -1 of the prototype distribution.
- The upper_limit of the actual value distribution corresponds to value 1 of the prototype distribution.

Note that these correspondences are maintained even if the prototype distribution is changed.

3. Using a Piecewise Linear Prototype CDF in a Netlist

The following example duplicates the example given in the topic titled "Probability Density Functions (PDFs)", with one exception: the v.battery voltage source has a piecewise linear cumulative distribution that is equivalent to the original uniform distribution.

```
# create prototype pwl CDF
standard..p_pwl cpwl1 = (type=_cdf,pwl=[(-1,0),(1,1)])

# map values from pwl CDF to v.battery
v.battery in 0   = pwl(9, undef, 8.9, 9.1, cpwl1)
r.1       in mid = normal(470k, 0.1)
r.2       mid 0   = normal(100k,0.05)
```

This example specifies the actual value CDF by giving the nominal value (9, which is the value used in non-statistical analyses), an undefined tolerance value, the lower and upper values (8.9 and 9.1, respectively), and the name of the prototype distribution (cpwl1).

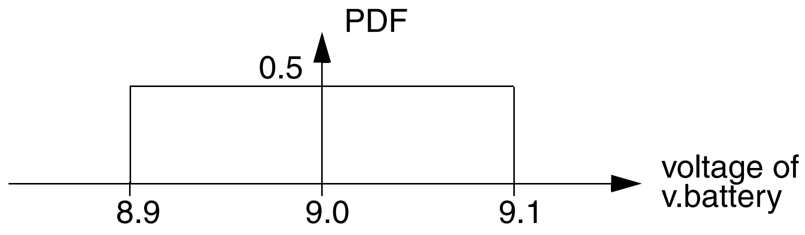
The following is an alternate way of specifying the same distribution:

```
standard..p_pwl cpwl1 = (type=_cdf,pwl=[(-1,0),(1,1)])
  # same as above

# alternate method of specifying v.battery
v.battery in 0 = pwl(9, 0.01111, undef, undef,
                     cpwl1)
r.1       in mid = normal(470k, 0.1)
r.2       mid 0   = normal(100k, 0.05)
```

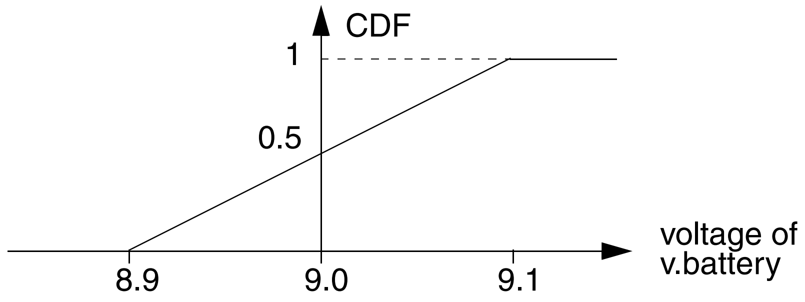
This method specifies the actual value CDF by giving the nominal value (9), the tolerance (0.01111), the lower and upper bounds (both undef), and the name of the prototype CDF (cpwl1).

Note that in each example either tolerance OR both upper and lower limits must be undef. It is an error to specify numeric values for all three, even though all three must have values. The pwl intrinsic function, unlike the uniform and normal functions, cannot infer the tolerance or limits from the context of the calling sequence.

The following figure shows the resulting actual value PDF and CDF for the voltage of v.battery.



Actual value PDF for v.battery voltage



Actual value CDF for v.battery voltage

Correlating Distributions

Sometimes it is desirable to model two or more quantities that tend to vary together. For example, two resistors may be manufactured on an integrated circuit. The resistor values may vary a great deal from wafer to wafer, or even from die to die. However, the resistors on the same die may tend to vary together. That is, if one resistor is at the high end of its range, others from that die tend to be at the high end of their ranges as well. When this occurs, the resistor values are said to be correlated.

Correlation occurs in numerous actual applications. The objective of this topic is to explain how this kind of variation can be modeled with constructs already described.

The voltage divider example (in the topic titled "Varying Values in a Simple Voltage Divider") can be used to show how to correlate two parameters. Assume that the two resistor values in the voltage divider are uniformly distributed with a tolerance of 10%, and that they correlate with each other within 0.5%. The following netlist implements these relationships:

```
number common=uniform(1,0.1)

v.battery in 0  = uniform(9, 8.9, 9.1)
r.1      in mid = normal(common*470k, 0.005)
r.2       mid 0 = normal(common*100k, 0.005)
```

The first line declares an arbitrarily named variable called common and uses an initializer (by invoking the uniform function) to assign a value from a uniform distribution. This uniform distribution has a nominal value of 1 and a 10% tolerance. The resistor netlist entries (r.1, r.2) use the common variable as a multiplier, providing the desired correlation. These resistor netlist entries also provide a 0.5% correlated variation.

Therefore, each resistor will have values that vary with a 10% tolerance, but they will vary (relatively, in the ratio 47:10) from each other with only a 0.5% tolerance.

Modifying Uniform and Normal Default Distributions

If necessary, you can modify the default prototype distributions provided for the uniform and normal distributions. Modifying these default distributions is similar to defining piecewise linear distributions.

There are several reasons for changing the default prototype distributions:

- To create uniform distributions that are asymmetrical about the nominal value
- To create uniform distributions with limits that are a multiple of the limits of a piecewise linear distribution
- To truncate either side of a normal distribution
- To change the standard deviation of a normal distribution

Next you will learn the following topics:

- Modifying a Uniform Prototype Distribution
- Modifying a Normal Prototype Distribution

Modifying a Uniform Prototype Distribution

To modify a uniform prototype PDF, use a MAST structure parameter such as the following:

```
struc p_uniform {
  number min=-1
  number max=1
}
```

You can implement this in any of the following ways:

- Use this structure in a netlist
- Include it in a template preceded by <distrib.sin as described in the topic “Creating a Piecewise Linear Prototype PDF”
- Include it in a template as by calling it from the standard template by using an argdef (..) declaration as described in the topic titled “Using the Standard Template”

The p_uniform structure defines the uniform prototype distribution. The default values for min and max are -1 and 1, respectively. Note that the min and max values are not the values that become associated with the limits named lower and upper in the following use of the uniform function:

```
uniform(nominal, lower, upper)
```

The limits of the function call always map to -1 and 1 in the prototype distribution. Therefore, if min and max are specified to be other than -1 and 1, the actual value PDF will have values defined above or below the specified lower and upper limits.

You can modify the prototype distribution in either of the following ways:

1. Using a variable initializer
2. Modifying, in the template body, variables of the structure defined using the p_uniform prototype PDF

1. Modifying a Uniform Prototype PDF Using Initializers

It is not necessary to declare a prototype variable when using the default. However, modifying the default prototype PDF requires a variable declaration.

The most direct way is to use initializers when defining the prototype variable patterned after the p_uniform prototype PDF by including the following in the template:

standard..p_uniform punif =(min=-2, max=0.5)
v.battery in 0 = uniform(9, undef, 8.95, 9.2, punif)

The above example produces the same distribution for v.battery as in the original example (shown below), which was based on the values min=-1 and max=1:

```
v.battery in 0 = uniform(9, 8.9, 9.1)
```

Note that, in the modified version, the uniform function call requires specification of all the possible arguments: nominal value (9), tolerance (undef), lower limit (8.95), upper limit (9.2), and prototype name (punif). The uniform function has the following general syntax:

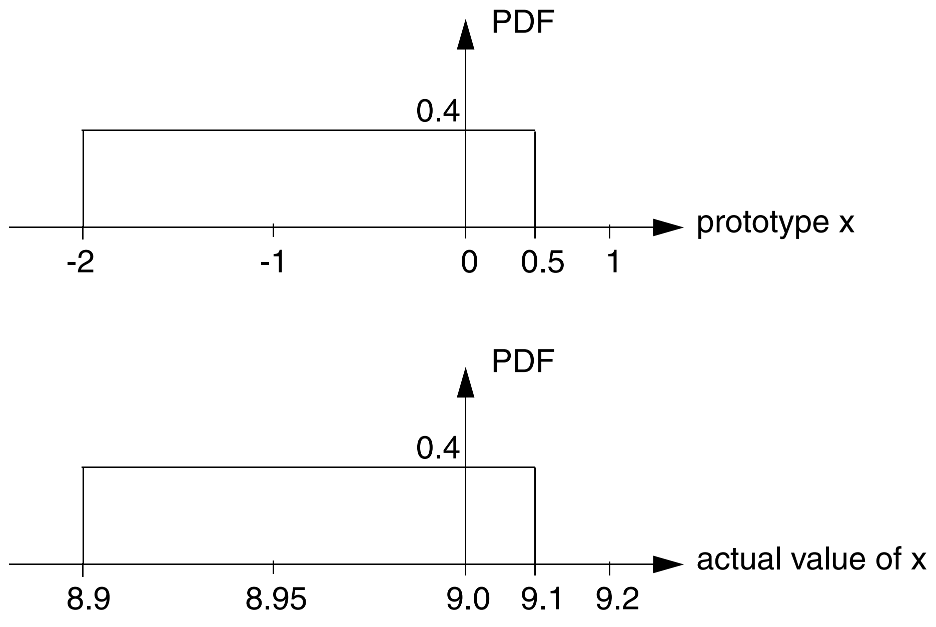
```
uniform(nominal_value, tolerance, lower_limit,
        upper_limit,prototype)
```

where:

nominal_value	the value that the distribution is to have in a deterministic (non-statistical) environment.
tolerance	either undef or a numeric value between -1 and 1—if it is a numeric value, then the upper and lower limits of the distribution are nominal_value + tolerance*nominal_value and nominal_value - tolerance * nominal_value
lower_limit	either undef or a numeric value less than nominal_value
upper_limit	either undef or a numeric value greater than nominal_value
prototype	the name of the prototype uniform PDF

When using a non-default prototype function, you must specify all parameters, with either tolerance set to undef or both lower_limit and upper_limit being set to undef.

Note that producing the symmetry of the original distribution required specifying asymmetrical limits in the function call. This is because the scaling on the two sides of the nominal value are different, as shown in the following figure:



Uniform PDF using non-default prototype

The scaling is different because of the mapping between the prototype distribution and the parameters passed to the uniform function:

- The parameter value 8.95 maps to the prototype value -1
- The parameter value 0 maps to the prototype value 0
- The parameter value 9.2 maps to the prototype value +1

The result is a uniform distribution between 8.9 and 9.1.

2. Modifying a Uniform Prototype PDF in a Template

You can obtain the same results as above (using an initializer) by modifying the prototype PDF in the template body, as follows:

```
standard..p_uniform punif
punif->min = -2
punif->max = 0.5

v.battery in 0 = uniform(9, undef, 8.95, 9.2,
                         punif)
```

The example just given produces the same distribution for v.battery as the previous example (reproduced below):

```
v.battery in 0 = uniform(9, 8.9, 9.1)
```

Modifying a Normal Prototype Distribution

Modifying a normal prototype PDF requires the use of a MAST structure similar to the following:

```
struc p_normal {
  number mean=0
  number std_dev=0.33333333333333
  number min=undef
  number max=undef
}
```

As shown above for a uniform distribution, you can implement this in any of the following ways:

- Use this structure in a netlist
- Include it in a template, preceded by <distrib.sin, as described in the topic titled "Creating a Piecewise Linear Prototype PDF"
- Include it in a template, calling it from the standard template by using an argdef (..) declaration, as described in the topic titled “Using the Standard Template”

The p_normal structure defines the normal prototype distribution. The default value for the mean is 0. The default value for the standard deviation is 1/3. The default values for min and max are both undef, which defines the distribution from -infinity to infinity. Otherwise, the distribution is truncated at the specified value.

Note that the min and max values are not the values mapped to the lower and upper limits in the following call to the normal function:

```
normal(nominal, lower, upper)
```

The limits of the function call always map to -1 and 1 in the prototype distribution.

The values of min and max are used only to specify the point at which the distribution becomes truncated. If the values of min and max are left at undef (the default) then the actual value PDF continues from -infinity to infinity. If the values of min and max are specified as -1 and 1, then the actual value PDF will be truncated at the specified limits (lower and upper). If the values of min and max are specified to be other than -1 and 1, the actual value PDF will be truncated accordingly. This is shown in the example that follows.

You can modify the prototype distribution in either of the following ways:

1. Using a variable initializer
2. Modifying, in the template body, variables of the structure defined using the p_normal prototype PDF

1. Modifying a Normal Prototype PDF using Initializers

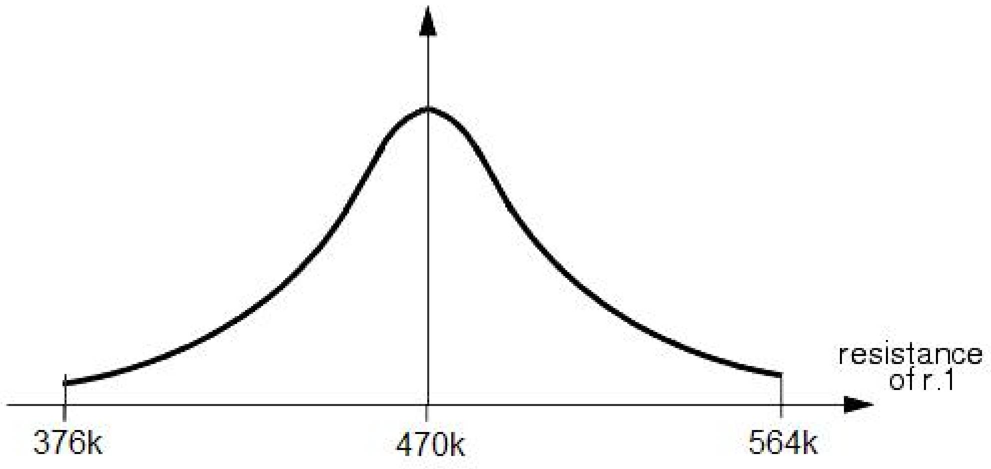
It is not necessary to declare a prototype variable when using the default. However, modifying the default prototype PDF requires a variable declaration. The most direct way is to use initializers when defining the prototype variable patterned after the p_normal prototype PDF by including the following in the template:

```
standard..p_normal pnorm = (std_dev=1/4, min=-2, max=2)
r.1 p m = normal(470k, 0.1, undef, undef, pnorm)
```

The above example produces a distribution somewhat like that for r.1 in the original example (shown below):

```
r.1 p m = normal(470k, 0.1)
```

However, the modified distribution differs by having an actual value distribution whose standard deviation is (470k•0.1)/4, rather than (470k•0.1)/3. Also, because the limit of the modified distribution was apecified as min=-2 and max=2, the upper end is truncated at 564k and the lower end is truncated to 376k. The following figure illustrates this modified distribution.



Normal PDF using non-default prototype

Note that, in the modified version, the call to the normal function requires specification of all the possible arguments: nominal value (470k), tolerance (0.1), lower limit (-2), upper limit (2), and prototype name (pnorm). The normal function has the following general syntax:

```
normal(nominal_value, tolerance, lower_limit,
        upper_limit, prototype)
```

where:

nominal_value	the value that the distribution is to have in a deterministic (non-statistical) environment.

tolerance	either undef or a numeric value between -1 and 1—if it is a numeric value, then the upper and lower limits of the distribution are nominal_value + tolerance*nominal_value and nominal_value - tolerance * nominal_value
lower_limit	either undef or a numeric value less than nominal_value
upper_limit	either undef or a numeric value greater than nominal_value
prototype	the name of the prototype normal PDF

When using a non-default prototype function, you must specify all parameters, and either the tolerance or both limits must be undef.

2. Modifying a Normal Prototype PDF in a Template

You can obtain the same results as above (using an initializer) by modifying the prototype PDF in the template body, rather than in the initialization of the variable. This modification (shown below) produces the same distribution for r.1 as does the preceding example.

```
standard..p_normal pnorm=()
pnorm->std_dev=1/4
pnorm->min = -2
pnorm->max = 2
r.1 p m = normal(470k, 0.1, undef, undef, pnorm)
```

Parameterized PDF and CDF Specifications

Occasionally, it is useful to be able to change the statistical properties of a parameter. For example, you might want to turn some parameters on statistically or turn some off, or both. There are two intrinsic functions for this purpose, one for PDFs, the other for CDFs. You can call them as follows:

```
parameter = pdf(nominal, tol, bounds,
                prototype_pdf)
parameter = cdf(nominal, tol, bounds,
                prototype_cdf)
```

The prototype distributions are defined in distrib.sin as follows:

```
union p_pdf {
  number off=1
  struc p_uniform uniform=()
  struc p_normal normal=()
  struc {number x,y;} pwl[*]
}
union p_cdf {
  number off=1
  struc {number x,y;} pwl[*]
}
```

The following example shows how to use a parameterized PDF—using a parameterized CDF is similar.

- In the local declarations section of the template, include the declarations to specify a normal distribution:

```
standard..p_pdf pdf1 = (normal=())
struc {number min, max;} bounds =(undef,
                                  undef)
```

- In the netlist section, place the following entry:

```
r.1 a b =pdf(1k, 0.1, bounds, pdf1)
```

Later, when running the simulator, you could change to a uniform distribution by entering the following command:

```
alter pdf1 =(uniform=())
```

Note that the structure bounds was used to simplify the specification of the two undefined quantities required by the pdf function call. The parameter pdf1 specifies a normal distribution. Later, when running the simulator, you could change to a uniform distribution by entering the following command:

```
alter pdf1 =(off=1)
```

The random MAST Function

The MAST language includes the random() function, which has no arguments. The random function returns a pseudo-random number in the interval that includes 0 and goes up to, but does not include, 1. The pseudo-random sequence can be seeded when the statistical environment is activated. Seeding is associated with the mc command.

The random() function is useful if you want to do something with a certain probability. For example, assume you want to flip a coin (i.e., have a variable that takes on two discrete values with certain probabilities). Although this could be described with the pwl distribution, it is simpler to use the random() function.

The following example shows how to set up a parameter that has the value 1K with probability 0.4, and 2K with probability 0.6:

```
number r, value

r = random()
if (r<.4) {
  value = 1K
}
else {
  value = 2K
}
```

Use of the statistical MAST Simvar Variable

One of the intrinsic simulation variables is statistical, whose value is 0 if the simulation environment is deterministic. On the other hand, it is non-zero if the environment is statistical, such as when you execute the mc command.

When a statistical environment has been established, statistically defined parameters take on random values according to the distribution functions defined for them. On the other hand, such parameters take on their nominal values if the environment is deterministic.

You can use the statistical simvar variable if you want to do different things in the deterministic and statistical environments. For example, you could use it when the nominal value of the statistical distribution is different from the deterministic value as follows:

```
if (statistical) {
  resistance = normal(10k, 200)
}
else {
  resistance = 8k
}
```

Worst-Case Statistical MAST Modeling

It is sometimes useful to perform what is called a worst-case analysis (WCA) on a design using Monte Carlo techniques. The MAST language supports this through the worst_case simvar variable. This simvar variable has a value of 0 except during a Monte Carlo analysis in which the worst_case variable is set to yes, when it assumes a non-zero value (such as 1).

The worst_case simvar variable interacts with the statistical simvar as follows:

	value of statistical	value of worst_case
Deterministic environment	0	0
Statistical environment (Monte Carlo analysis)	1	0
Worst-case analysis in statistical environment (Monte Carlo analysis)	1	1

You can use the worst_case simvar variable to do different things in standard and worst-case Monte Carlo analyses. For example, the statistical distributions provided (such as uniform, normal, pwl) change their behavior depending on the value of the worst_case simvar as follows:

- If worst_case is 0, these distributions are implemented.
- If worst_case is 1 (or any non-zero value), these distributions are implemented as discrete distributions—they return, with equal probability, only the upper and lower limit values that result after applying the appropriate prototype distribution.

19

Adding Stress Measures to a MAST Template

A stress measure is a definition of an operating condition for which a safe operating limit can be specified. The operating condition will typically correspond to a rating (or SOA) specification for a device in a manufacturer’s data sheet.

The following steps describe a way to add stress measures to a template for which stress ratios can then be calculated by a stress analysis.

1. Add stress_measure Statements to Template
2. Determine if Specified Variables are Accessible
3. Add Stress Ratings
4. Add Thermal Resistances (Optional)
5. Add a Way to Disable Stress (Optional)
6. Add a Way to Specify Device Type and Class (Optional)

Add stress_measure Statements to Template

To add the required stress_measure statements to the control section of the template:

- Place a stress_measure statement for each stress measure you want to implement in the control section of your template.

The following examples show stress_measure statements for a resistor template:

```
stress_measure (pdmax,power,"Max Power Diss.",
  pwrd,winmax,pdmax)
stress_measure (pdavg,power,"Avg Power Diss.",
  pwrd,average,pdmax)
stress_measure (tjmax,temperature,"Max Temperature",
  tempj,winmax,xtjmax,tempj_tnom)
stress_measure (tjavg,temperature,"Avg temperature",
  tempj,average,xtjmax,tempj_tnom)
stress_measure (tjmin,temperature,"Min Temperature",
  tempj,min,xtjmin,tempj_tnom)
stress_measure (vmax,voltage,"Max Voltage",
  abs(v),max,xvmax)
```

A stress_measure statement takes the following form:

```
stress_measure (uid, gid, "name", val,
          measure, rating[, ref_rating])
```

where

uid	stress report formatting - (unique identification) identifies the stress measure (for example, pdmax). The uid is used as the value for the smeasurelist variable of the stress command.
gid	stress report formatting - (group identification) identifies a type or grouping of stress measures to which this stress measure belongs (for example, power)
name	stress report formatting - specifies the text to be used to describe this stress measure in a stress report (for example, "Max Power Diss."). Limited to 18 characters.

val	is the name of a variable in the template from which the value of the stress measure is to be extracted using the measurement specified in the measure field (e.g., pwrd). See the topic titled "Determine if Specified Variables are Accessible" for more information about template variables.
measure	specifies the measurement to be made on the template variable specified in the val field. This measurement provides the “actual” or “measured” value for the stress ratio calculation. Possible measurements are one of: peak, max, winmax, min, winmin, rms, and average. When winmax or winmin are used (rather than max or min), a sliding average filter is applied to the waveform before the maximum or minimum value is determined. The stress command variable xwindow is used to specify the time constant of the filter to be applied. Note that if the value for rating is positive, the peak measurement is equivalent to the max measurement. If the value for rating is negative, the peak measurement is equivalent to the min measurement.
rating	is the manufacturer’s rating for the stress measure (e.g., 40, pdmax). You can enter the actual value of the rating as a constant in this field, or you can enter a variable. If a variable is entered, the value can be provided as an argument in the header of the template (see the topic titled "Add Stress Ratings"). The rating is a parameter passed into the template and should not be confused with the uid which may have a similar name.
ref_rating	is an optional single value reference rating. When a reference rating is specified, the measured and derated values are referenced to this value rather than to 0 when the stress ratio is calculated. For example, you may want to use 25××C rather than 0 as a reference for a temperature-related stress measure (for example, for tempj_tnom). The stress measures in templates in the MAST libraries are all referenced to 0 and do not use the ref_rating variable

Determine if Specified Variables are Accessible

Determine if the variable specified in the val field of each stress measure is available in the template and add if needed.

Somewhere in your template, a value must be provided from which the stress measure can be extracted. The stress measures shown in the topic titled "Add stress_measure Statements to Template" are obtained by making measurements on the three variables pwrd, tempj, and v. These values (val variables) are calculated in the body of the resistor template as shown below:

```
v = v(p) - v(m)
.
.
.
power = v*i
pwrd = power
tempj = temp + pwrd*rth_eff
```

For examples on using val variables in a template, see Variables and Arguments on page 97.

Add Stress Ratings

If a rating is not provided directly in a stress statement, it must be passed into the template as an argument. By convention, ratings are passed in using a structure parameter. An example of this method is shown below. However, if only one or two stress measures are included in the template, you may prefer to specify the ratings arguments individually rather than in a structure.

1. In the header declarations section of the template, declare a structure for the ratings. An example is shown as follows:

```
struc {
 number pdmax_ja=undef,  # Max. Pwr, no htsnk
        pdmax_jc=undef,  # Max. Pwr, with htsnk
        tjmax=undef,     # Max. temperature
        tjmin=undef,     # Min. temperature
        vmax=undef       # Max. voltage
} ratings=()
```

2. Add the name of the structure to the template header. In this example, ratings is the name of the structure. The header is shown as follows:

```
element template r  p m = rnom, tc, tnom, nons,
  model, l, w, ratings, rth_ja, rth_jc, rth_hs,
  part_type, part_class
```

3. In the body of the template, define local variables in which to store ratings after error checking is complete. By convention, a local variable that corresponds to a rating variable takes the same name as the rating variable preceded by the character x.

```
number   r,  #Final resistance.
         g,  #Final conductance.
        nx,  #Noise flag.
        xl,  #Effective resister length.
        xw,  #Effective resister width.
        dl,  #Final value of geometry reduction.
   rth_eff,  #Final value of thermal resistance.
     pdmax,  #Final value of power diss. rating.
     xvmax,  #Final value of max. voltage rating.
    xtjmax,  #Final value of max temp rating.
    xtjmin   #Final value of min temp rating.
number tempj_tnom=25 #Ref_rating for temp stress.
```

4. Add error checking for ratings. A function called ratingbp is provided that returns the absolute value specified for the rating and provides standard error checking and appropriate warning and error messages. The ratingbp function can be found in install_home/template/function/ratingbp.sin.

   In the following example, this function is used to assign the absolute value of the rating vmax to the variable xvmax and to check the value for errors.

```
xvmax = ratingbp(ratings->vmax,"vmax")
```

   The second part of this section of the example handles the special case of tjmax and tjmin where a negative value is allowed. It checks for values of undef (no value provided) and inf (non-applicable value) and determines if xtjmax is greater than xtjmin. If tjmax is not greater then tjmin, the message TMPL_S_REL_VALUE is displayed that states that “the maximum value should be greater than the minimum value.”

```
xtjmax = ratings->tjmax
xtjmin = ratings->tjmin
if(xtjmin ~= undef & xtjmin ~= inf) {
  if(xtjmax ~= undef & xtjmax~=inf &
  xtjmax < xtjmin) {
    saber_message("TMPL_S_REL_VALUE",instance(),
     "tjmax","tjmin")
     xtjmin = undef
  }
}
```

Add Thermal Resistances (Optional)

Thermal resistances are typically passed in as arguments to the template. An intrinsic function called thermpar can be used to determine the effective thermal resistance and maximum power dissipation from the values that are passed in.

1. In the header declarations section of the template, add thermal resistance variables as shown in the example below:

```
number  rnom=undef, # Nominal resistance.
           tnom=27, # Nominal temperature.
       tc[2]=[0,0], # Temperature coefficients.
          nons=0.0, # Resistor will be noiseless
                    # if non-zero value.
             l=0.0, # Optional length of resistor.
             w=0.0, # Optional width of resistor.
      rth_ja=undef, # Junction-Ambient Thermal
                    # resistance deg C/W).
      rth_jc=undef, # Junction-Case Thermal
                    # resistance (deg C/W).
      rth_hs=undef  # Heatsink Thermal resistance
                    # (deg C/W).
```

2. Add local variables to the body of the template for thermal resistance and power dissipation:

```
number    r,  # Final resistance.
          g,  # Final conductance.
         nx,  # Noise flag.
         xl,  # Effective resister length.
         xw,  # Effective resister width.
         dl,  # Final value of geometry reduction.
    rth_eff,  # Final value of thermal resistance.
      pdmax,  # Final value of power diss. rating.
      xvmax,  # Final value of max. voltage rating.
     xtjmax,  # Final value of max temp rating.
     xtjmin   # Final value of min temp rating.
number tempj_tnom=25  # Ref_rating for temp stress.
```

3. Add a statement to the template that determines the effective thermal resistance rth_eff and the power dissipation pdmax to be used. The following example makes use of the thermpar function.

```
(rth_eff,pdmax) = thermpar(rth_ja,rth_jc,rth_hs,
  if(ratings->pdmax_ja==undef) then r_pdmax
    else ratings->pdmax_ja,
  ratings->pdmax_jc)
```

   Note that the thermpar function has five arguments separated by commas. The fourth argument contains an if then else condition. The thermpar function can be found in install_home/template/function/thermpar.sin.

   The thermpar function implements the truth table shown as follows:

INPUTS			**OUTPUTS**	
rth_ja	**rth_jc**	**rth_hs**	**rth_eff**	**pdmax**
undef	undef	undef	0	pdmax_ja
undef	undef	val_hs	0	pdmax_ja
undef	val_jc	undef	val_jc	pdmax_jc
undef	val_jc	val_hs	val_jc+val_hs	pdmax_jc
val_ja	undef	undef	val_ja	pdmax_ja
val_ja	undef	val_hs	val_ja	pdmax_ja
val_ja	val_jc	undef	val_ja	pdmax_ja
val_ja	val_jc	val_hs	val_jc+val_hs	pdmax_jc

Add a Way to Disable Stress (Optional)

A mechanism can be implemented in a template to allow you to inactivate the stress statements in a template when they are not needed. This feature is useful if a template is to be used as a macromodel building block. Typically, in a macromodel, an overall value for an operating condition such as power is calculated by combining values from the building blocks and then stress measures for the macromodel itself are extracted. In this case, the stress measures in the individual templates are typically of less use in a stress report.

1. In the header declarations section of the template, add include_stress to the list of parameters declared as external numbers as shown in the following example:

```
external number temp, include_stress,
  r_tol, r_pdmax
```

   The parameter name include_stress is not a MAST reserved word. It is the name used by convention in MAST templates

   Values declared in external statements are found at a higher level in the hierarchy of a design. The include_stress parameter, for example, is set to a default value of 1 in the header.sin file.

   The header.sin file contains declarations for various global variables such as temp and include_stress. It is included in the Saber input file (netlist) for the design when the Saber Simulator is invoked.

2. Place a conditional statement around the stress_measure statements in the control section of the template to detect if they are to be included in the simulation. See the following example:

```
if(include_stress) {
  stress_measure(pdmax,power,"Max Power Diss.",
    pwrd,winmax,pdmax)
  stress_measure(pdavg,power,"Avg Power Diss.",
    pwrd,average,pdmax)
  stress_measure(tjmax,temperature,
    "Max Temperature",tempj,winmax,
    xtjmax,tempj_tnom)
  stress_measure(tjavg,temperature,
    "Avg Temperature",tempj,average,
    xtjmax, tempj_tnom)
  stress_measure(tjmin,temperature,
    "Min Temperature",tempj,min,
    xtjmin, tempj_tnom)
  stress_measure(vmax,voltage,"Max Voltage",
    abs(v),max,xvmax)
}
```

Add a Way to Specify Device Type and Class (Optional)

Information about the type and class of a device provided in the template are used as sorting criteria for the stress report.

1. Add a device_type statement to the control section of the template as shown in the following example:

```
device_type(part_type,part_class)
```

   In the previous statement, part_type and part_class could be replaced by the actual part type and part class of the device. Alternatively, these values can be passed in as arguments to the template as shown next.

2. In the header declarations section, declare part_type and part_class as strings and give them default values as shown in the following example:

```
string part_type="resistor",# type of the device
       part_class="generic" # class of the device
```

   The part_type string must be limited to 9 characters and the part_class string to 18 characters to fit into the format of the stress report.

If you are modifying an existing template to add stress measures, device_type, part_type, and part_class statements may have already been defined in the template. However, you can alter them to provide part type or part class names that may be more useful as sorting criteria in your stress reports.

MAST Example Including Stress Statements

A complete listing of the example resistor template is shown below. Stress-related statements are shown in bold.

```
#************************************************************
*****
# Constant resistor (called by: r )
# Zero value is not allowed and will generate an error message.
# Geometric description is allowed.
#************************************************************
*****

#************************************************************
*****
# This template created by Synopsys, Inc. for exclusive use
with
# the Saber Simulator.
# Copyright 1987,1988,1989,1993,2005 Synopsys, Inc.
# This template may not be reproduced in any way (physically or
# electronically) without permission from Synopsys, Inc.
# The content of this template is subject to change without
# notice. Synopsys does not assume liability for the use of this
# template or the results obtained from using it.
#************************************************************
****

element template r  p m = rnom,tc,tnom,nons,model,l,w,
                          ratings,rth_ja,rth_jc,
                          rth_hs,part_type,part_class

  #...declaration of connections:
  electrical  p,m
  process..imodel model = ()# Process model for resistor.
```

```
#...declaration of arguments (tnom is in degrees celsius)

number rnom=undef,   # Nominal resistance.
       tnom=27,      # Nominal temperature.
       tc[2]=[0,0],  # Temperature coefficients.
       nons=0.0,     # Resistor will be noiseless if
                     # non-zero value.
       l=0.0,        # Optional length of resistor.
       w=0.0,        # Optional width of resistor.
       rth_ja=undef, # Junction-Ambient Thermal
                     # resistance (deg C/W)
       rth_jc=undef, # Junction-Case Thermal
                     # resistance (deg C/W)
       rth_hs=undef  # Heatsink Thermal
                     # resistance (deg C/W)
#...Bring in external numbers
external number temp, include_stress, r_tol, r_pdmax
external standard..pdist pdist

struc {
  number pdmax_ja= undef,# Max. power diss. w/out heatsink
  pdmax_jc=undef,        # Max. power diss. w/ heatsink
  tjmax=undef,           # Max. temperature
  tjmin=undef,           # Min. temperature
  vmax=undef             # Max. voltage
} ratings=()

string part_type="resistor",# type of the device
       part_class="generic" # class of the device
  export val tc tempj  # instantaneous junction temperature
  export val p  pwrd   # instantaneous power dissipation
  export val i  i      # instantaneous current

#+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
  # Start the definition
{
  #...Quantities useful for output:

  val v v
  val p power
  val ni nsr
  val tc temp_case
  val rth rth_hs_tjmax

  #...Define a group for extraction purposes
  group {nsr}        noise
  group {power,pwrd} pwr
  #...Define quantities used later

  number r,       # Final resistance.
         g,       # Final Conductance.
         nx,      # Noise flag.
         xl,      # Effective resister length.
         xw,      # Effective resister width.
         dl,      # Final value of geometry reduction.
         rth_eff, # Final value of thermal resistance.
         pdmax,   #Final value of power dissipation rating.
         xvmax,   # Final value of max. voltage rating.
         xtjmax,  # Final value of max temp rating.
         xtjmin   # Final value of min temp rating
  number  tempj_tnom=25  # Ref_rating for temp stress.

  #...Bring in mathematical constants
<consts.sin
```

```
#+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

  parameters {
    #...Check input parameters.
    if ( (rnom == undef) & ((model->rsh == 0)|(l == 0)|
     ((w == 0)&(model->wdf == 0))) ) {
      # Resistance is not specified.
      saber_message("TMPL_S_ALT_SPEC",  instance(),
        "resistance","rnom","model->rsh, l, and w")
    }

    #...Include temperature effects
    if (rnom ~= undef) {
      #...Resistor specification.
      if (rnom == inf) {
       r = inf
      }
      else {
       #...Call function to apply distribution
       #...to resistor value
       r = distfunc(rnom,r_tol,pdist)
       r = r*(1 + tc[1]*(temp-tnom) + tc[2]*((temp-tnom)**2))
      }
    }
```

```
else {
  #...Process specification.
  #...Check input parameters.
  if ((model->dl == undef)|(model->dl < 0)) {
   dl = 0
  }
  else {
   dl = model->dl
  }
  if ( ((w == 0)|(w == undef)) &
   ((model->wdf == 0)|(model->wdf == undef)) ) {
    saber_message("TMPL_S_ALT_SPEC",  instance(),
     "resistor width","w","model->wdf")
  }
  #...Take into account the geometry
  #...reduction of the length.
  xl = l - dl

  if (xl <= 0) {
    saber_message("TMPL_S_POS",instance(),
     "effective resistor length")
  }

  #...Take into account the geometry
  #...reduction of the width.
  if ((w > 0)&(w ~= undef)) {
    xw = w - dl
  }
  else {
    xw = model->wdf
  }
  #...Calculate the resistance from
  #...the sheet resistance.
  if (xw > 0) {
    #...Call function to apply distribution
    #...to resistor value
    r = distfunc(model->rsh*(xl/xw),r_tol,pdist)
    r = r*(1 + tc[1]*(temp-tnom) + tc[2]*((temp-tnom)**2))
  }
  else {
    saber_message("TMPL_S_POS",instance(),
     "effective resistor width")
  }
}

#...Calculate conductance and print message if r=0
if (r == 0) {
  saber_message("TMPL_S_RANGE_NE_0",instance(),
   "resistance value")
  g = 0
}
else if (r < 0) {
  #...negative resistance
  saber_message("TMPL_W_GE_REL_VALUE",instance(),
   "resistance value","zero")
  g = 1/r
}
```

```
    else if (r == inf) {
      g = 0
    }
    else {
      g = 1/r
    }
    #...Bulletproofing on ratings
    (rth_eff,pdmax) = thermpar(rth_ja,rth_jc,rth_hs,
      if(ratings->pdmax_ja==undef) then r_pdmax
       else ratings->pdmax_ja,
      ratings->pdmax_jc)

    xvmax = ratingbp(ratings->vmax,"vmax")

    xtjmax = ratings->tjmax
    xtjmin = ratings->tjmin
    if(xtjmin ~= undef & xtjmin ~= inf) {
      if(xtjmax ~= undef & xtjmax~=inf & xtjmax < xtjmin) {
        saber_message("TMPL_S_REL_VALUE",instance(),"tjmax",
          "tjmin")
        xtjmin = undef
      }
    }
    #...Determine the noise "switch" multiplier
    if ((nons == 0) & (r > 0)) {
      nx = 1
    }
    else {
      nx = 0
    }
  }
```

```
#++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
  values {
    #...Definition of output quantities.
    v = v(p) - v(m)
    i = g*v
    # If r=0, i=0. The value for i is wrong but a
    # message has been printed out indicating that r=0.

    #...Calculate noise generator
    if (freq_domain & nx) {
      nsr = sqrt(abs(4.0*math_boltz*(temp + math_ctok)*g))
    }
    else {
      nsr = 0.0
    }
    #...Determine power term for extraction

    power = v*i
    pwrd = power
    tempj = temp + pwrd*rth_eff

    if(rth_jc ~= undef & rth_jc ~= inf & rth_jc > 0) {
      if (pwrd ~= 0) {
        rth_hs_tjmax = (xtjmax - temp)/pwrd - rth_jc
      }
      else rth_hs_tjmax = inf
    temp_case = tempj - pwrd*rth_jc
    }
  }
```

```
#++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
  control_section {
    #...device type and class
    device_type(part_type,part_class)

    #...Specify noise source
    noise_source(nsr,p,m)

    #...Specify the stress measures
    if(include_stress) {
      stress_measure(pdmax,power,"Max Power Diss.",
        pwrd,winmax,pdmax)
      stress_measure(pdavg,power,"Avg Power Diss.",
        pwrd,average,pdmax)
      stress_measure(tjmax,temperature,"Max Temperature",
        tempj,winmax,xtjmax, tempj_tnom)
      stress_measure(tjavg,temperature,"Avg Temperature",
        tempj,average,xtjmax,tempj_tnom)
      stress_measure(tjmin,temperature,"Min Temperature",
        tempj,min,xtjmin,tempj_tnom)
      stress_measure(vmax,voltage,"Max Voltage",
        abs(v),max,xvmax)
    }
  }
#++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
  equations {
    i(p->m) += i
  }
}
```

A

Unstructured Modeling Approach — Examples

This appendix presents examples that use an unstructured approach to modeling.

Constant Current Source

The contents of the constant current source template are shown below modeled in an unstructured format.

```
template isource p m = is
  electrical p,m
  number is
{
  branch i = i(p->m)     # branch declaration
  i=is                   # template equation
}
ASCII text of this example is located in:
install_home/example/MASTtemplates/unstructured/
isource.sin
```

Branch Declarations

The following branch declaration identifies the through variable flowing between pins p and m as a branch current and assigns it to a variable named i:

```
branch i = i(p->m)
```

Note:

Although you could omit the branch declaration shown above and just write the equation as i(p->m) = is, it is not recommended.

Template Equation

The template equation is written to describe the effect of the continuous analog portion of a template at its connection points.

The source current is enters at pin p and leaves at pin m. That is, if isource is used in a netlist, the current flowing between the nodes to which pins p and m are connected is modified by the source current from isource. In the MAST language, the simplest way to express this is:

```
i(p->m) = is
```

The terms in this equation include:

i(p->m)	The branch current flowing from the node connected to pin p to the node connected to pin m. (This current is assigned to the variable i in the branch declaration, so that i is used in the actual template equation).
is	The user-specified current contributed by the source.

For a constant current source, this means that a current of the amount is provided between the node connected to pin p and the node connected to pin m. Because an independent current source is not affected by its branch voltage, no equations need to be provided for voltage across the source (that is, the branch voltage becomes whatever is required to maintain the specified level of branch current, is).

Therefore, the following equation is a complete description of a constant current source:

```
i = is
```

where i has been assigned to represent the branch current:

```
i(p->m)
```

Linear Resistor MAST Template

The un-structured approach to modeling a simple resistor template with MAST is as follows:

```
template resistor p m = res
  electrical p,m
  number res
{
  branch cur=i(p->m)    # Branch current
  branch vlt=v(p,m)     # Branch voltage
  cur=vlt/res           # Equation for current
                        # through the resistor
}
ASCII text of this example is located in:
install_home/example/MASTtemplates/unstructured/
resistor.sin
```

Branch Declarations

This template provides a single branch from p to m. This means that, when p and m are connected to system nodes, the template models a path for the through variable (current). Similarly, an across variable (voltage) can be calculated between nodes to which p and m are connected. The branch variables for this template are named and declared as:

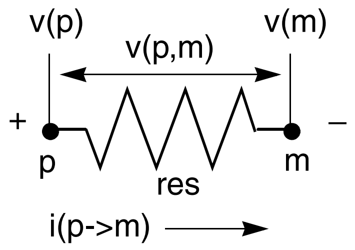
```
branch cur = i(p->m)
branch vlt = v(p,m)
```

Branches can be declared in any order within the template body.

Template Equation

Referring to the ideal resistor figure, the voltage across the resistor is given as the difference between the voltages at its pins:

```
v(p,m) = v(p) - v(m)
```



Ideal resistor

Because both p and m are electrical, v(p) and v(m) are implicitly declared as system variables (meaning, the simulator supplies their values). To be written as a MAST equation, the characteristic equation of the resistor must use the branch variables as they have been declared (above). Therefore, the current through the resistor is given by:

```
cur = vlt/res
```

This is the equation for the resistor template; it defines the amount of current that the resistor contributes to the current flowing from pin p to pin m.

Linear Capacitor MAST Template

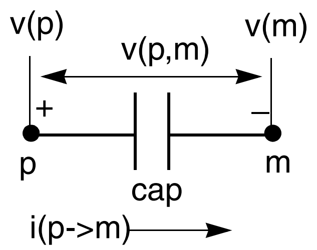
The un-structured approach to modeling a linear capacitor appears as follows:

```
template capacitor p m = cap     # header
  electrical p, m          # header declarations
  number cap
{              # Start template body
  branch i = i(p->m)
  branch v = v(p,m)
  i = d_by_dt(v*cap)
}              # End template body
ASCII text of this example is located in:
install_home/example/MASTtemplates/unstructured/
capacitor.sin
```

The branch declarations are very similar to those for the isource and the resistor templates.

Template Equation

The simulator solves for the values of v(p) and v(m) as system variables, so they are known to the template. Refer to the following figure.



Linear capacitor

To write the template equation, you need to express the voltage across the capacitor in terms of these system variables, which is done by the branch declaration:

```
branch v = v(p,m)
```

Referring to the characteristic equation, the branch current (i) of the capacitor is given as the time derivative of its charge. In the MAST language, taking the time derivative of an expression is represented by applying the d_by_dt operator to the expression.

In this example, this becomes:

```
i = d_by_dt(v * cap)
```

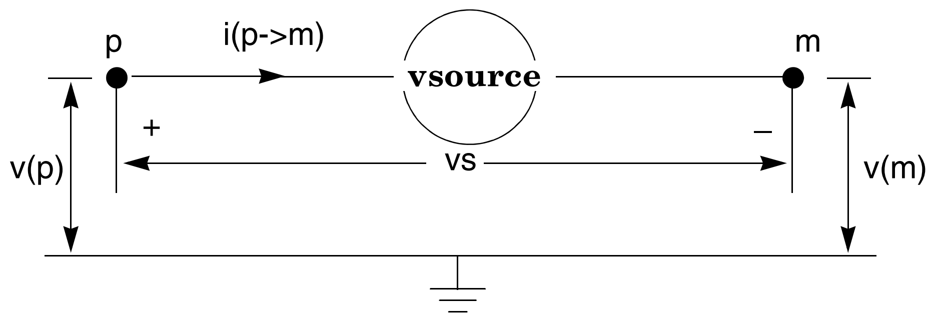
The d_by_dt operator can operate on any expression that does not include a delay operator or another d_by_dt operator. It can appear only in a template equation, and you can only add terms to it and subtract terms from it. Therefore, the expression cap*d_by_dt(v), while valid in conventional calculus, is not valid in the MAST language. Higher order derivatives are implemented with multiple d_by_dt statements.

Constant Voltage Source MAST Template

The un-structured approach to modeling a constant voltage source is as follows:

```
template vsource p m = vs
  electrical p,m
  number vs
{
  branch i=i(p->m), v=v(p,m)
  v = vs
}
ASCII text of this example is located in:
install_home/example/MASTtemplates/unstructured/
vsource.sin
```

The following figure illustrates the voltage source; it provides a constant voltage (vs) across pins p and m. The value of vs is user-specified for an instance of this template in a netlist; it is therefore an argument for the source model.



Voltage source

This topic introduces the need for an additional branch declaration to provide the through variable (current) as a system variable

Branch Declarations

The value of current through the constant voltage source, i(p->m), cannot be determined by the source alone. Instead, this current depends upon the system to which the voltage source is connected (recall that the simulator solves for system variables, such as the across variables at the pins). The branch current needs to be declared, as explained below.

Notice that vsource uses branch voltage much the same way as isource uses branch current—it defines the appropriate branch variable and then sets it equal to the value of the template argument:

vsource	isource
v=v(p,m)	i=i(p->m)
v=vs	i=is

However, there is one important difference: vsource also needs to declare its branch current as a system variable for use by the simulator, whereas isource does not need to declare its branch voltage (because the simulator already finds it as a system variable). Thus, vsource needs to declare two branch variables, i and v:

```
branch i = i(p->m)
branch v = v(p,m)
```

However, MAST allows you to combine branch declarations on one line:

```
branch  i=i(p->m), v=v(p,m)
```

Template equation

After the branch declarations described above have been made, the template equation for this voltage source can be written (similar to that for the current source):

```
v=vs
```

For a constant voltage source, this means that the branch voltage from p to m is maintained at a value equal to vs. Because an independent voltage source is not affected by its branch current, no equations have to be provided for current through the source, as long as that current appears in a branch declaration (i.e., the branch current becomes whatever is required to maintain the specified level of branch voltage, vs).

Linear Inductor MAST Template

The un-structured approach to modeling a template describing a linear inductor is as follows:

```
element template inductor p m = ind
  electrical p,m
  number ind
{
  branch i=i(p->m), v=v(p,m)
  v = d_by_dt(ind*i)
}
ASCII text of this example is located in:
install_home/example/MASTtemplates/unstructured/
inductor.sin
```

Characteristic Equations

The voltage, V_L, across an inductor is defined as the derivative of magnetic flux (f) with respect to time:

$V_L = df/dt$

For a linear inductor, the flux is defined as the product of the inductance, L, of the inductor and the current (IL) through it:

$f = L \bullet I_L$

Therefore, inductor voltage can be expressed in terms of inductance and current:

$V_L = d(LI_L)/dt$

The inductance (L) characterizes each instance of an inductor and is provided as an argument of the template (ind). The following figure shows the symbol and the relevant characteristics of a linear inductor, including the branch variables to be contained in the ideal inductor template.

Setting Up the Template Equation

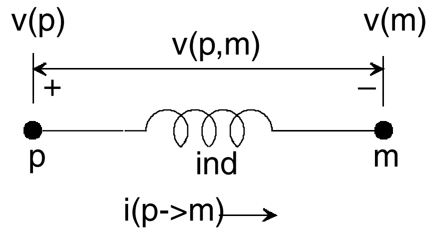
As with the resistor template, you should try to express the inductor current (the through variable contribution) as a function of the voltage across the inductor, which would be:

$I_L = 1/L \int V_L \, dt$

Although this equation serves as a compact implementation of the model, it requires the use of an integral, which the MAST language does not support. Therefore, it is necessary to use a new approach that differs from the one used to implement the resistor and capacitor templates.

Note:

When you encounter an integral expression as in Equation 4, you need to differentiate both sides of the equation to eliminate the integral.



Ideal inductor

As with the constant voltage source (vsource), you need to declare a branch variable for current through inductor, which can be done on the same line as the branch voltage:

```
branch i = i(p->m), v=v(p,m)
```

This allows you to use current in an equation such that you can express voltage in terms of current. Now you can replace the integration formula for the inductor current with a differentiation formula for the inductor voltage.

Taking the derivative of both sides of Equation 4 and multiplying both sides by L, it becomes:

$d(LI_L)/dt = V_L$

which is implemented in the template as (ind is the value of inductance, L, provided as an argument):

```
v = d_by_dt(ind*i)
```

Although this violates a guideline of expressing current as a function of voltage, it is the only method available.

Current-Controlled Voltage Source MAST Template

The un-structured approach to modeling a template describing a current-controlled voltage source (CCVS) is as follows:

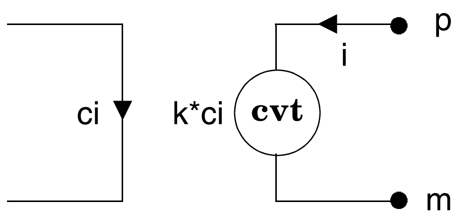
```
template cvt ci p m = k
  ref i ci
  electrical p, m
  number k
{
  branch i=i(p->m), v=v(p,m)
  v = k*ci
}
ASCII text of this example is located in:
install_home/example/MASTtemplates/unstructured/
cvt.sin
```

Characteristic Equation

The characteristic equation for a CCVS is similar to that of a constant voltage source, except that the output voltage is determined differently.

v = k*ci
choose i such that KVL is satisfied.

In this equation, ci is the controlling current, k is the transimpedance characterizing an instance of the cvt template, v and i are the branch voltage and current, respectively.



Current-controlled voltage source

Template Equation and Local Declarations

As with the vsource template, you must declare the branch current i along with the branch voltage, v, to be able to solve for voltage:

```
branch i=i(p->m), v=v(p,m)
```

This makes the current through the source (i) available as a system variable, for which the simulator can now solve. The template equation is as follows:

```
v= k*ci
```

shows that this voltage source depends on a controlling input current (ci), which is from another template. This input current is obtained from a different kind of connection point, the ref.

Mutual Inductance MAST Template

The un-structured approach to modeling a template describing mutual inductance is as follows:

```
element template mutind i1 i2 = m
  ref i i1, i2
  number m
{
  i1 -= d_by_dt(m*i2)
  i2 -= d_by_dt(m*i1)
}
ASCII text of this example is located in:
install_home/example/MASTtemplates/unstructured/
mutind.sin
```

Setting Up the Template Equations

The template equations for mutind use the equation in the inductor template. There, the associated equation was essentially the first term shown in Equations 1 and 2 shown in the topic titled "Characteristic Equations for Modeling Mutual Inductance" as follows:

$$V_1 = d(L_1I_1)/dt + d(MI_2)/dt$$
$$V_2 = d(L_2I_2)/dt + d(MI_1)/dt$$

The mutind template modifies the equation in the inductor template that is associated with the branch current, i, of the inductor. That is, when used in a netlist, the mutual inductance template “searches out” the equations of the two inductors and couples them as specified in the mutind model.

The following MAST construct in a template equation modifies an equation associated with an explicitly declared system variable such as a branch current:

system_variable   operator   expression

where:

system_variable	is the name of a branch variable, var, or a ref (which is a var from another template)

operator	is either += or -=, indicating that expression is added to or subtracted from the left side of the equation defining system_variable
expression	is a MAST expression formed from variables of different types; mathematical functions; the algebraic operators +, -, *, /, and **; parentheses; and the special operators d_by_dt and delay

Using this construct, the characteristic equations of mutual inductance yield the following template equations for mutind:

```
i1 -= d_by_dt(m*i2)
i2 -= d_by_dt(m*i1)
```

It is important to understand that d_by_dt(m * i2) is not subtracted from i1, but rather from the left side of the equation that defines i1 (and, because i1 is a ref in this template, its defining equation is in the inductor template).

This is illustrated using the following example netlist:

```
inductor.l1 p1 m1 = ind=1
inductor.l2 p2 m2 = ind=2
mutind.1 i(inductor.l1) i(inductor.l2) = \
m=0.98*sqrt(ind(inductor.l1)*ind(inductor.l2))
```

Because the inductor template declares i as a branch current (which makes it a system variable), its associated equation is:

```
v = d_by_dt(ind*i)
```

This corresponds to the first term of either Equation 1 or Equation 2. Considering inductor.l1, the current from inductor.l2 is passed as ref i2 into the mutind template, which, in turn, modifies the equation in inductor.l1, using its branch current, i, and inductance, ind. The effective equation in inductor.l1 would be:

```
v-d_by_dt(m*i2) = d_by_dt(ind*i)
```

which, after algebraic rearrangement, corresponds to Equation 1. A similar modification is applied to the equation associated with the branch current i and inductance, ind in inductor.l2. to yield Equation 2. Thus, the mutind template, together with two instances of the inductor template, provide the inductance coupling shown in the “Coupled inductors” figure in the topic titled "Characteristic Equations for Modeling Mutual Inductance" in a previous chapter.

Flattened Hierarchy MAST Template

The un-structured approach to modeling an RLC template with flattened hierarchy is as follows:

```
template rlc2 p m = r,l,c
  electrical p,m
  number r = 10k,    # resistance arg. w/default
         l = 1m,     # inductance arg. w/default
         c = 1u      # capacitance arg. w/default
{
 electrical x                   # internal node
 branch ipm=i(p->m), vpm=v(p,m) # resistor branch
 branch ipx=i(p->x), vpx=v(p,x) # inductor branch
 branch ixm=i(x->m), vxm=v(x,m) # cap. branch

                             # template equations
 ipm = vpm/r                  # resistor current
 vpx = d_by_dt(ipx*l)         # inductor voltage
 ixm = d_by_dt(vxm*c)         # capacitor current
}
ASCII text of this example is located in:
install_home/example/MASTtemplates/unstructured/
rlc2.sin
```

Template Equations

The template equations for the rlc2 template consist of the equations from the resistor, inductor, and capacitor templates using the variable names from the rlc2 template. Note that these statements can appear in any order.

```
ipm = vpm/r
vpx = d_by_dt(ipx*l)
ixm = d_by_dt(vxm*c)
```

Template Body - Local Declarations

The internal node x, because it is a local internal node, must be declared following the opening brace before it can be used:

```
{
electrical x
```

Note that there are three branches in this model, one for each element—resistor, inductor, capacitor. Here, the current and voltage for each branch are declared and given variable names that are unique within this template:

```
branch ipm=i(p->m), vpm=v(p,m)
branch ipx=i(p->x), vpx=v(p,x)
branch ixm=i(x->m), vxm=v(x,m)
```

Mixed Hierarchy MAST Template

The un-structured approach to modeling an RLC template with mixed hierarchy is as follows:

```
template rlc3 p m = r,l,c
  electrical p,m
  number r = 10k,     # resistance arg. w/default
         l = 1m,     # inductance arg. w/default
         c = 1u      # capacitance arg. w/default
{
  electrical x           # internal node

  inductor.l1 p x = l    # use inductor template

  i(p->m) = v(p,m)/r            # res. equation
  i(x->m) = d_by_dt(v(x,m)*c)   # cap. equation
}
```

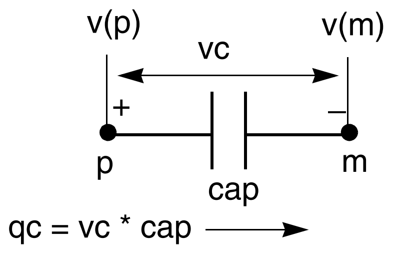
ASCII text of this example is located in:
install_home/example/MASTtemplates/unstructured/
rlc3.sin

MAST capacitor_1 Template

The following example shows the un-structured approach to modeling capacitor_1 such that the voltage across the capacitor and its charge become available for extraction:

```
element template capacitor_1 p m = cap, ic
  electrical p,m
  number cap, ic=undef
{
  branch i=i(p->m), vc=v(p,m)
  val q qc
  qc = vc*cap                 # charge assignment

  control_section {
    initial_condition(vc,ic)  # initial voltage
  }                          # across capacitor

  i = d_by_dt(qc)
}
```

```
ASCII text of this example is located in:
install_home/example/MASTtemplates/unstructured/
capacitor_1.sin
```



Linear capacitor

Local Declarations - Assignment Statements

The syntax of an assignment statement is as follows:

variable = expression

where:

variable	is a val, var, branch variable, local parameter, or simvar (simulator variable). If you declare variable as any of these (except as a simvar), you must do so in the body of the template.
expression	is a MAST expression defining variable—the value of expression is assigned to variable (this is not a mathematical equality). An expression can include variables of different types (except for through variables); mathematical functions; the +, -, *, /, **, d_by_dt( ), and delay( ) operators; and parentheses (()).

Statements in the template body are both procedural and declarative. Procedural statements are performed in sequence, so you must order the statements such that each val is defined before it is used in a statement. Declarative statements are performed only if the simulator requires the declared variable (for example, if variable is a val that is to be extracted or is needed for solution of the system variables).

If variable is not needed, the simulator skips the statement during evaluation, although it might use certain information provided by a statement during initial setup. Therefore, including statements that declare vals in a template does not impose any performance penalties if no variables are extracted.

Note:

Assignment statements can appear anywhere in the template body, but should not be confused with template equations—the expression on the right is evaluated and its result is assigned to the variable on the left. They are evaluated in sequential order, as in a computer program.

Template equations express mathematical equality and are evaluated simultaneously.

Local Declarations - Declaring a val

One of the stated objectives for this example was to make the voltage across the capacitor and the charge of the capacitor available for post-processing. The relationship of these quantities, which are incorporated in the capacitor_1 template are as follows:

```
branch vc = v(p,m)
val q qc
qc = vc*cap
```

Note that the assignment statement defining charge (qc) uses variables that have been previously declared—vc (branch voltage) in a branch declaration and cap (template argument) in a header declaration.

Further, the quantity qc is declared as a charge (val q) and is then assigned the value of the product of vc and cap (the capacitance). As required, the declaration of qc as a val precedes its assignment statement. The letter q in this declaration establishes qc as an electrical charge, as defined in the units.sin file, which is provided with the Saber simulator.

The simulator, by default, includes the units.sin file with each input file (netlist). The unit for charge (coulomb) is then automatically used when qc is displayed after simulation in the Scope Waveform Analyzer.

Note:

> As with vars and refs, the unit of the val declaration is used only to label plots. Thus, it is possible to exclude the val declaration (val q qc), and the rest of the template will function properly as a capacitor model. However, there will be no units of charge (coulombs) assigned to qc when it is displayed in Scope.

The declaration of a val appears in the following general form (optional declarations are enclosed by italicized square brackets, [ ]):

```
val unit name [,name, name, ...]
```

This declares one or more variables to be vals having the specified unit.

You can assign a value to a val only in an assignment statement, as described above. However, you can use a val variable in the template equation, the control section, and in when statements.

Template Equation

The template equation for capacitor_1 is very similar to the one for capacitor, except that now the capacitor charge (qc) replaces the quantity of voltage times capacitance:

```
i = d_by_dt (qc)
```

The Saber simulator uses the template equation and assignment statements jointly to set up the system equations. It uses these assignment statements only to the extent necessary to determine the through variable contribution at the capacitor pins as a function of the across variables at the pins. It does not, however, evaluate the assignment statements, except when a user specifies that certain information is to be extracted—and even then it evaluates only the equations needed to provide the requested information.

As described earlier, this does not mean that assignment statements are evaluated to solve the system variables. Rather, the simulator uses certain information when it sets up the simultaneous equations (from all templates in the netlist) describing the system. Consequently, the equation that the simulator solves for this capacitor is identical to the one for the capacitor template.

Multiple-Output Voltage Source

The following example shows the un-structured approach to modeling vsource_1, which models a source that provides a constant voltage output, independent of time and frequency:

```
element template vsource_1 p m = supply, tran, ac
  electrical p, m      # header declarations
  number supply=0

  struc {              # start of tran structure
  number v1=0,         # initial voltage
         v2=0,         # voltage at time=inf
        tau=0          # time constant
  } tran=()            # end of tran structure

  struc {              # start of ac structure
    number   mag=0,    # AC magnitude
           phase=0     # AC phase
  } ac=()              # end of ac structure

{                              # start template body
 branch i=i(p->m), v=v(p,m)    # branch declarations

 # determine output based on analysis being run...

 # begin large signal (supply or tran)
 if (dc_domain | time_domain) {
  # if tran specified, use tran value
  if ((tran->v1~=0 | tran->v2~=0) & tran->tau>0) {
   vs = tran->v1 + (tran->v2-tran->v1) * (1-exp(-time/tran-
>tau))
  }
  else {                  # if tran not specified,
   vs = supply            # use supply value
  }
 }                        # end large signal

 else if (freq_mag) {     # begin small-signal (ac)
  vs = ac->mag            # use magnitude value
 }
 else if (freq_phase) {
  vs = ac->phase          # use phase value
 }                        # end small-signal  (ac)

 v = vs                   # assign value of vs to output, v
}
ASCII text of this example is located in:
install_home/example/MASTtemplates/unstructured/
vsource_1.sin
```

Template Body

The branch current needs to be declared because the value of current through the voltage source, i(p->m), cannot be determined by the source alone. Instead, this current depends on the system to which the voltage source is connected (because the simulator solves for system variables, such as the across variables at the pins). The branch current is declared on the same line as the branch voltage:

```
branch i=i(p->m), v=v(p,m)
```

The remainder of this unstructured example is very similar to the structured vsource_1 example.

Linear Transformer

The following example shows the un-structured approach to modeling xformer, which consists of two inductors and their mutual coupling:

```
element template xformer p1 m1 p2 m2 = l1, l2, k
  electrical p1, m1, p2, m2
  number l1, l2, k = 1
{                           # start body of template
  if (k < -1 | k > 1) {
  # if arg. value invalid, display message and end simulation
    error("%:coupling factor must be between -1 and 1:k=%",
      instance(), k)
  }
  # Use following netlist to make a transformer from
  # two mutually-coupled inductors
  inductor.1 p1 m1 = l1     # inductor netlist entry
  inductor.2 p2 m2 = l2     # other inductor netlist entry
  mutind.12 i(inductor.1) i(inductor.2) = k * sqrt(abs(l1 *
l2))
                            # mutual inductance netlist entry
}
```

ASCII text of this example is located in:
`install_home/example/MASTtemplates/unstructured/`
`xformer.sin`

This unstructured template is similar to the structured version. The major changes are that this unstructured template has no local declaration or use of m and the parameters section in this template is eliminated.

Temperature-Dependent Resistor

The following example shows the un-structured approach to modeling resistor_1. This template can operate at various ambient temperatures, checks for a value of zero resistance and allows power to be extracted

```
element template resistor_1 p m = res, tc, tnom
  electrical p, m
  number res, tc[2]=[0, 0], tnom=27
  external number temp          # use simulation temperature
  export val p power            # make power available
{
  branch i=i(p->m), v=v(p,m)

  power = v*i                   # calculate power

  # local parameter (calculated from arguments)
  r = res * (1 + tc[1]*(temp-tnom) + tc[2]*(temp-tnom)**2)

  # check for zero resistance, display message, end simulation:
  if (r==0) error("%: resistance value is zero",instance())

  # template equation using calculated r (not res)
  i = v/r
}# end of template body
```

```
ASCII text of this example is located in:
install_home/example/MASTtemplates/unstructured/
resistor_1.sin
```

This unstructured template is similar to the structured version. The major changes are that this unstructured template has no explicit local declaration of r, v and i. Also, this template does not use a parameters section, a values section, or an equations section.

Simple Idealized Op Amp

The following example shows the un-structured approach to modeling an opamp. The opamp example introduces a modeling technique that lets you combine different model equations such that a single equation satisfies all values of the template parameters. This template allows specifying either finite or infinite gain.

```
element template opamp ip im out = a
  electrical ip, im, out # header declarations
  number a = inf
{                         # start of template body
  var i iout              # local declarations
  branch vin=v(ip,im)     # input branch voltage
  number x1, x2
  # checking value of gain argument, a
  if (a==inf | a==undef) {    # if gain is infinite
      x1=1; x2=0              # or undefined, then input
  }                           # voltage is 0;
  else {                      # otherwise, output voltage
    x1=a; x2=1                # is gain times input voltage
  }
  # equations for output current at one connection point
  i(out) += iout           # current contribution at output
  iout: x1 * vin = x2 * v(out)
                           # equation associated with iout
}
ASCII text of this example is located in:
install_home/example/MASTtemplates/unstructured/
opamp.sin
```

Unstructured MAST clock Template

The following example shows how to construct a simple clock template using internal events in the unstructured modeling approach. It also describes initialization of states.

```
template  clock ckout = freq, duty
  state logic_4 ckout
  number freq=0,           # clock frequency
         duty=0.5          # duty cycle (pulse time/period)
{
  state nu tick            # internal "wake-up" state

  number ton=0,            # clock on-time
         toff=0            # clock off-time

    # calculate off and on time
  if (freq > 0) {
    ton = duty/freq
    toff = 1/freq - ton
  }

  when (dc_init) {
    schedule_event(time,ckout,l4_0)
  }
    # start clock ticking after delay time
  when (time_init) {
    if (freq > 0) schedule_event(time,tick,1)
  }
  when (event_on(tick)) {
    if (driven (ckout)==l4_0) {
       # turn clock on (set to 1)
      if (ton > 0) {
        schedule_event(time,ckout,l4_1)
        schedule_event(time+ton,tick,1)
      }
    }

    else {
       # turn clock off (set to 0)
      if (toff > 0) {
        schedule_event(time,ckout,l4_0)
        schedule_event(time+toff,tick,1)
      }
    }
  }
}
```

Modeling a Simple Voltage Limiter

The unstructured template for the limiter, vlim, is shown as follows:

```
element template vlim ip im op om = vmax
                                  # template header
  electrical ip, im, op, om       # header declarations
  number vmax
{                                 # start of template body
  branch  vin=v(ip,im)
  branch vout=v(op,om)
  branch iout=i(op->om)
  struc {
    number bp, inc;
  } nvin[*]
  number slope = 1u

  vmx = abs(vmax)
  nvin = [(-vmx,1.9*vmx),(vmx,0)]

  control_section {
    newton_step(vin,nvin)
  }

# template equations using conditional expressions
  vout = if vin < -vmx then  -vmx + slope*(vin + vmx)
    else if vin > vmx then    vmx + slope*(vin - vmx)
    else vin
}
```

```
ASCII text of this example is located in:
install_home/example/MASTtemplates/unstructured/
vlim.sin
```

Local Declarations

The branch declarations enable the characteristic equations to express the output voltage (vout) as a function of the input voltage (vin), while finding the current contribution (iout) required for this to be true.

```
branch vin=v(ip,im),
branch vout=v(op,om)
branch iout=i(op->om)
```

Although negative values for vmax are allowed, the equations for determining vout assume vmax is positive. That is, they use the absolute value of vmax, which is obtained by using abs, the intrinsic absolute value function. This absolute value of vmax is assigned to the local parameter vmx as follows:

```
vmx = abs(vmax)
```

A variable named slope is declared as a local parameter and initialized to 10^{-6}. This is used in the template equations, as explained below.

```
number slope=1u
```

Template Equations

In the template equations, note that this expression defines three conditions for the value of vout, although vout is explicitly listed only once. That is, in the first two lines, the value of vout evaluates to the expression following then. In the last line, the value of vout evaluates to the value of vin. Also note the use of the backslash (\) as a line continuation character.

Modeling a Voltage Divider

The unstructured template for the voltage divider, vdiv, is shown as follows:

```
element template vdiv ip1 im1 ip2 im2 op om
                                 # template header
  electrical ip1, ip2, im1, im2, op, om
 {                                # start of template body
  branch vin1=v(ip1,im1)
  branch vin2=v(ip2,im2)
  branch iout = i(op->om), vout=v(op,om)

  eps = 1e-6
  val v onev, vout
  onev=if (abs(vin2)<1e-50) then 0 else 1/vin2
  struc {
    number bp,inc;
  } nv2[*]

  if(eps <= 0) eps = 1e-15
  eps2 = 1 / (eps * eps)
  nv2 = [(-2*eps,eps),(2*eps,0)]

  control_section {
    newton_step(vin2,nv2)
  }
  # template equations
  vout = if(abs(vin2) > eps) then vin1*onev \
    else vin1 * vin2 * eps2
}
```

`ASCII text of this example is located in:`
`install_home/example/MASTtemplates/unstructured/`
`vdiv.sin`

Modeling an Ideal Diode

The following example combines the structured and unstructured modeling techniques to model the ideal diode:

```
element template diode p m = is, ic  # template header
  electrical p, m                # header declarations
  number is = 1e-16,
         ic = undef              # initial branch voltage
  external number temp
{                             # start of template body
  branch id=i(p->m), vd=v(p,m)
  number k = 1.318e-23,       # local declarations
         qe = 1.602e-19

  struc {
    number bp, inc;           # Newton steps
  }  nvd[*] = [(0,.001),(2,0)]

  control_section {           # start of control section
    newton_step (vd,nvd)      # Newton steps assigned to vd
    initial_condition(vd,ic)
    device_type("diode","example")
    small_signal(vd,voltage,"p-m voltage", vd)
  }                           # end of control section

  vt = k * (temp+273.15) / qe # computation of thermal volt.
  id = is * (limexp(vd/vt)-1) # diode current
}                             # end of template body
```

```
ASCII text of this example is located in:
install_home/example/MASTtemplates/unstructured/
diode.sin
```

Ebers-Moll Model for a BJT

The BJT template (bjt) is shown below using a combination of the structured and unstructured modeling approach:

```
element template bjt c b e = model, ic
  electrical c, b, e
  struc {
    enum {_n, _p} type
    number  is=1e-16, bf=100, br=1, \
            cje=0, vje=.75, mje=.33, \
            cjc=0, vjc=.75, mjc=.33, rc=0
  } model = ()
  number ic[2]=[undef,undef]
  external number temp
{
  # declare local param., vals, and extraction groups
  number k = 1.381e-23,              # Boltzmann's constant
        qe = 1.602e-19,              # electron charge
        vt, qbe0, qbc0, vje0, vjc0

  struc {
    number bp, inc;
  } nv[*] = [(0,.1),(2,0)]

  electrical cp                      # local node
```

```
branch ibe=i(b->e), vbe=v(b,e)
branch ibc=i(b->cp), vbc=v(b,cp)
branch ice=i(cp->e), vce=v(cp,e)
val i iec, icc, iba, ico
val q qbc, qbe
group {vbc,vbe} v                  # extraction groups
group {iba,ico} i
group {qbc,qbe} q

control_section {
   # If no collector resistance, collapse nodes c and cp
  if(model->rc == 0) collapse(c,cp)
   # specification of newton steps
  newton_step((vbc,vbe),nv)
   # initial conditions and start value
  initial_condition(vbe, ic[1])
  initial_condition(vce, ic[2])
  start_value (vbe, 0.6)
   # small-signal parameters
  device_type("bjt", "example")
  small_signal(ibase,current,"base current",iba)
  small_signal(icoll,current,"collector current",ico)
  small_signal(vbe,voltage,"base-emitter voltage", vbe)
  small_signal(vbc,voltage,"base-collector voltage", vbc)
  small_signal(rc,resistance,"collector resistance",\
               model->rc)
}
```

```
 #calculate thermal voltage and 4 functions of model param.
vt   = k * (temp + 273.15) / qe
qbe0 = model->cje * model->vje / (1 - model->mje)
qbc0 = model->cjc * model->vjc / (1 - model->mjc)
vje0 = 2 * model->vje / model->mje
vjc0 = 2 * model->vjc / model->mjc

 #calculate fundamental quant. for npn and pnp trans.
if(model->type == _n) {
  iec = model->is * (limexp(vbc/vt) - 1)
  icc = model->is * (limexp(vbe/vt) - 1)
}
else {
  iec = -model->is * (limexp(-vbc/vt) - 1)
  icc = -model->is * (limexp(-vbe/vt) - 1)
}
 # calculate base and collector currents
iba = iec/model->br + icc/model->bf
ico = icc - iec - iec/model->br
```

```
 # calculate charges
if(model->type == _n) {
  if(vbc < 0) {
   qbc = qbc0 * (1 - ((1 - vbc/model->vjc)**(1-model->mjc)))
  }
  else {
   qbc = model->cjc * vbc * (1 + vbc/vjc0)
  }
  if(vbe < 0) {
   qbe = qbe0 * (1 - ((1 - vbe/model->vje)**(1-model->mje)))
  }
  else {
    qbe = model->cje * vbe * (1 + vbe/vje0)
  }
}
else {
  if(vbc > 0) {
   qbc = -qbc0 * (1 - ((1 + vbc/model->vjc)**(1-model->mjc)))
  }
  else {
    qbc = model->cjc * vbc * (1 - vbc/vjc0)
  }
  if(vbe > 0) {
   qbe = -qbe0 * (1 - ((1 + vbe/model->vje)**(1-model->mje)))
  }
  else {
   qbe = model->cje * vbe * (1 - vbe/vje0)
  }
}
```

```
   # template equations (calculate branch currents)
  ibe = iba + d_by_dt(qbe)
  ibc = d_by_dt(qbc)
  ice = ico

   # current through collector resistor if present
  if(model->rc ~= 0) {
    i(c->cp) = v(c,cp)/model->rc
  }
}              # end template body
```

ASCII text of this example is located in:
install_home/example/MASTtemplates/unstructured/
bjt.sin

Digitally-Controlled, Ideal Switch

This example shows the unstructured version of an ideal, single-pole, single-throw switch (SPST) controlled by a digital input.

```
template sw p m cntl = ron, roff
  electrical p, m                   # analog pins
  state logic_4 cntl                # digital connection
  number ron=1, roff=1meg           # parameter declaration
{
  branch cur=i(p->m), vlt=v(p,m)
  state r res                       # internal state variable

  when (event_on(cntl)) {           # switch control
    if (cntl == l4_1) res = ron
    else              res = roff
    schedule_next_time(time)
  }
    # template equation for analog switch
  cur = vlt/res
}
```

Digital-to-Analog (d2a) Interface Model

The following template shows the unstructured version of the d2a template:

```
element template d2a d a m = td, ol, oh
  electrical a, m
  state logic_4 d
  number td=0,             # input to output time delay
         ol=0.5,           # output logic low voltage level
         oh=4.0            # output logic high voltage level
{
  state v vout=ol          # output voltage
  branch cur=i(a->m), vlt=v(a,m)

   # process input events
  when (event_on(d)) {
    if (d==l4_0) {                        # input low
      schedule_event(time+td, vout, ol)   # change vout
      schedule_next_time(time+td)         # force analog step
    }
    else if (d==l4_1) {                   # input high
      schedule_event(time+td, vout, oh)   # change vout
      schedule_next_time(time+td)         # force analog step
    }
  }
  vlt = vout
}
```

Integrator (intgr)

Integration in the time domain is identical to division by s in the s-domain, which could be expressed as follows:

```
out = in/(tau*s)
```

To implement this as a valid template equation, you need to perform the following steps:

1. Revise the statement so that it has no denominator. To accomplish this, simply multiply both sides by tau*s, which yields the following:

```
tau*s*out = in
```

2. Replace the s operator with the MAST d_by_dt operator, and move tau inside the d_by_dt operator. This sets the time derivative of the output multiplied by a constant (tau, the template argument) equal to the input as follows:

```
d_by_dt(out*tau) = in
```

3. For a template that does not use the equations section (an unstructured model), insert the keyword make at the beginning of the statement as follows:

```
make d_by_dt(out*tau) = in
```

This is required because the lefthand side of #2 above is not a simple output, but an expression, which means this relation cannot be expressed directly. Statements such as this must be prefixed with the keyword make so it will be recognized as a template equation. This kind of a statement is called a constraint equation.

In an unstructured template, the keyword make is required anytime the output is an expression. It can also be used (although it is not required) if the lefthand side is an output (such as for the deriv template in Differentiator on page 219).

The unstructured MAST template for an integrator (intgr) is listed as follows:

```
element template intgr in out = tau
  input  nu in      # template input
  output nu out     # template output
  number tau        # constant multiplier
{            # solve for the input variable
  make in = d_by_dt(out*tau)
}
```

Using MAST Functions - Unstructured bjtm Template

The BJT template (bjtm) is shown below using function calls and a combination of the structured and unstructured modeling approach:

```
element template bjtm c b e = model, ic
  electrical c,b,e
  bjtm_arg..model model = () # use arguments from
                             # "companion" template
  number ic[2]=[undef,undef]
  external number temp
{                            # begin template body
  # use local parameters from "companion" template
  bjtm_arg..work work

  struc {
    number bp,inc;
  } nv[*] = [(0,.1),(2,0)]
  electrical cp                   #...local node
  branch ibe=i(b->e),  vbe=v(b,e)
  branch ibc=i(b->cp), vbc=v(b,cp)
  branch ice=i(cp->e), vce=v(cp,e)
  val i iec,icc,iba,ico           #...declare variables

  val q qbc,qbe
  group {vbc,vbe} v               #...extraction groups
  group {iba,ico} i
  group {qbc,qbe} q   control_section {
   #...If no collector resistance, collapse nodes c and cp
    if(model->rc == 0) collapse(c,cp)

    #...specification of sample points and newton steps
    newton_step((vbc,vbe),nv)
  }
  # calculate thermal voltage and 4 funct. of model param.
       #... 1’st call to MAST function
  work = bjtm_pars(model,temp)

  #...calculation of currents and charges
       #...2’nd call to MAST function
    (iec,qbc,icc,qbe) = bjtm_values(model,work,vbc,vbe)



  #...calculate base and collector currents for extraction
  iba = iec/model->br + icc/model->bf
  ico = icc - iec - iec/model->br

  #...calculation of branch currents
  ibe = iec/model->br + icc/model->bf + d_by_dt(qbe)
  ibc = d_by_dt(qbc)
  ice = icc - iec - iec/model->br

  #...current through collector resistor if present
  if(model->rc ~= 0) i(c->cp) = v(c,cp)/model->rc
}
```

The function calls and functions are identical to the bjtm structural model described in the following topics found in Chapter 4:

- Function Call Overview - bjtm MAST Template
- bjtm_arg Declaration Template
- Local Parameters Function bjtm_pars
- Calculated Values Function bjtm_values

Ideal Delay Line - Unstructured dline Template

The ideal delay line template (dline) is shown below using the delay function and the unstructured modeling approach:

```
template dline inp inm outp outm = td, a
  electrical inp, inm, outp, outm
  number td=0.0,                           # time delay
             a=1.0                         # gain

{
  branch iout=i(outp->outm) # output current
  branch vin=v(inp,inm), vout= v(outp,outm) # input & output
                                           # voltages
  val v vdl                                # delayed voltage

  vdl = vin*a
  vout=delay(vdl, td)
}
```

Multiple-Output Voltage Source - Unstructured vsource_2 Template

The vsource_2 template is a voltage source that provides three different, time-varying outputs. The following template is modeled using the unstructured approach:

```
element template  vsource_2 p m = supply, tran
  electrical p, m
  number supply = 0
  union {
    number                          off
    struc {number vo, va, f, td;}   sin
    struc {number v1,v2,tau;}       exp
    struc {number v1,v2,tstep,tr;}  step
  } tran = (off=1)
{
  number pi = 3.14159
  branch is=i(p->m), vn=v(p,m)
  val v vs

  # define intermediate values depending on selected output
  if (union_type (tran,sin)) {
    td = tran->sin->td
    vo = tran->sin->vo
    va = tran->sin->va
    w  = 2*pi*tran->sin->f
    ss = 0.05/tran->sin->f
  }
  else if (union_type (tran,exp)) {
    v1  = tran->exp->v1
    v2  = tran->exp->v2
    tau = tran->exp->tau
  }
```

```
else if (union_type (tran,step)) {
  tstep = tran->step->tstep
  v1    = tran->step->v1
  v2    = tran->step->v2
  tr    = tran->step->tr
  slew  = (v2-v1)/tr
}
# determine vs, which is set equal to vn in temp. equ.

if (dc_domain|time_domain) {
  if (union_type (tran,sin)) {
    if (time <= td) {
      vs = vo
      next_time = td
    }
    else {     # if (time > td)
      vs = vo + va*sin(w*(time-td))
      step_size = ss
    }
  }                                # end tran->sin
  else if (union_type (tran,exp)) {
    vs = v1 + (v2-v1)*(1-exp(-(time/tau)))
  }                                # end tran->exp
```

```
    else if (union_type (tran,step)) {
      if (dc_domain|(time < tstep)) {
        vs = v1
        next_time = tstep
      }
      else if ((time >= tstep) & (time < tstep+tr)){
        vs = v1 + (time-tstep)*slew
        next_time = tstep + tr
      }
      else {
        vs = v2
      }
    }                              # end tran->step
    else vs = supply
  }                                # end
dc_domain|time_domain

  else vs = 0

  # template equation...find branch voltage, vn
  vn = vs
}
```

B

Making User Templates Visible for Unix and NT

This chapter describes how to make user templates visible in both Unix and NT environments.

Making User Templates Visible for Unix

This topic describes the following:

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

How the Applications Find Files

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

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

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

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

Data Search Path

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

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

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

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

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

The topic titled Manually Creating Template Information Files, in the Managing Symbols and Models Manual, describes how to update custom templates that do not have the proper permissions for a user. You must be a site manager with read and write permissions to use this feature.

Never point SABER_DATA_PATH to install_home.

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

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

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

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

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

Using Templates Written in MAST

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

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

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

1. Define or modify the SABER_DATA_PATH environment variable

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

Shell & File	SABER_DATA_PATH Definition

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

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

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

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

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

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

2. Re-initialize your startup file

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

Using Custom Models From Your Capture Tool

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

Making Symbols Available in Saber Sketch

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

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

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

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

Using C or FORTRAN Routines Called by Templates

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

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

- Compile each foreign routine

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

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

Command to Compile a FORTRAN Foreign Routine

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

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

To compile a C routine, complete the following steps:

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

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

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

   For more information, refer to Foreign Routines in MAST on page 337.

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

Command to Compile a C Foreign Routine

System	Command	Trailing Underscore?

Solaris	`cc -c -K PIC -cg89 \`
	`      -dalign -ftrap=%none \`
	`        -xlibmil filename.c`
		yes

HP-UX	`cc -c +Z  filename.c`	no

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

How to Make a Single Routine Available to the Saber Simulator

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

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

   Complete one of the following:

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

2. Invoke the Saber simulator

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

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

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

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

Making a Library of Routines Available to the Saber Simulator

To make a library of routines available to the simulator:

1. Compile the subroutines using the appropriate compiler.

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

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

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

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

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

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

3. Declare the shared library file as global

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

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

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

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

5. Re-initialize your startup environment

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

```
login login_name
```

Making User Templates Visible for NT

This topic describes the following:

How Applications Find Files
Making Symbols Available in Saber Sketch
Using Templates Written in MAST
Using C or FORTRAN Routines Called by Templates (NT)

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

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

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

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

Data Search Path

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

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

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

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

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

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

   Note:

   Never point SABER_DATA_PATH to saber_home.

   Saber Sketch searches the value of the AI_SCH_PATH environment variable to search for directories containing symbols. The AI_SCH_PATH variable is a semicolon-separated list of directories. Any custom symbols intended for use by others at your site should be stored in a directory that is part of AI_SCH_PATH. If such a directory does not exist, you should create one and add its path to AI_SCH_PATH. If AI_SCH_PATH does not exist, you should create it.
3. The saber_home\config directory holds configuration information specific to a site. Do not place any custom libraries in this directory.
4. The last place(s) an application will search are the additional directory(s) that are appended by the application. These are the homes for specific-supplied data. For the Saber simulator these directories are

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

Do not place any custom libraries in this directory.

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

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

To check the Path variable setting, do the following:

- Navigate to, and start the System program:

  Start > Settings > Control Panel > System > Environment tab
- Look at the System Environment Variable list for the Path variable.
- Add the appropriate directory(s) to the value.

Making Symbols Available in Saber Sketch

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

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

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

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

Using Templates Written in MAST

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

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

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

1. Define or modify the SABER_DATA_PATH environment variable

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

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

   To check the SABER_DATA_PATH variable setting, do the following:

   - Navigate to, and start the System program:

     Start > Settings > Control Panel > System > Environment tab
   - Look at the System Environment Variable list for the SABER_DATA_PATH variable.
   - If it does not exist, create it and add the appropriate directory(s) to the value.

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

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

2. Re-initialize your startup environment

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

Using C or FORTRAN Routines Called by Templates

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

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

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

The C Language Header

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

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

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

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

The FORTRAN Language Header

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

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

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

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

How to Make a Single Routine Available to the Saber Simulator

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

One-Step Dynamic Library Linking

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

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

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

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

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

One-Step C Language Compiling and Linking

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

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

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

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

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

One-Step FORTRAN Language Compiling and Linking

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

```
fl32 /LD filename.f
saber_home\lib\libai_saber.lib
saber_home\lib\libai_analogy.lib
```

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

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

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

How to Compile and Link Libraries of Routines

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

1. Compile the subroutines using the appropriate compiler.

   Compiling subroutines is a language-dependent operation.

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

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

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

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

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

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

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

3. Declare the DLL file as global.

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

Set the SABER_GLOBAL variable as follows:

Navigate to, and start the System program:
Start > Settings > Control Panel > System > Environment tab

- Set the variable as follows:

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

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

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

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

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

   Check or edit the SABER_DATA_PATH variable as follows:

   Navigate to, and start the System program:
   Start > Settings > Control Panel > System > Environment tab

   - In either the System or User environment variable list box, an entry for SABER_DATA_PATH may appear. If it does not appear, create it. Enter the path(s) to the directory(s). If there is more than one path, list them and separate by colons.

5. Re-initialize your startup environment

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

Index