Saber®
MAST Language, Book 2,
User Guide

Version W-2004.12, December 2004

SYNOPSYS®

Copyright Notice and Proprietary Information

Copyright © 2004 Synopsys, Inc. All rights reserved. This software and documentation contain confidential and proprietary information that is the property of Synopsys, Inc. The software and documentation are furnished under a license agreement and may be used or copied only in accordance with the terms of the license agreement. No part of the software and documentation may be reproduced, transmitted, or translated, in any form or by any means, electronic, mechanical, manual, optical, or otherwise, without prior written permission of Synopsys, Inc., or as expressly provided by the license agreement.

Right to Copy Documentation

The license agreement with Synopsys permits licensee to make copies of the documentation for its internal use only. Each copy shall include all copyrights, trademarks, service marks, and proprietary rights notices, if any. Licensee must assign sequential numbers to all copies. These copies shall contain the following legend on the cover page:

> "This document is duplicated with the permission of Synopsys, Inc., for the exclusive use of ____ and its employees. This is copy number __________."

Destination Control Statement

All technical data contained in this publication is subject to the export control laws of the United States of America. Disclosure to nationals of other countries contrary to United States law is prohibited. It is the reader's responsibility to determine the applicable regulations and to comply with them.

Disclaimer

SYNOPSYS, INC., AND ITS LICENSORS MAKE NO WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, WITH REGARD TO THIS MATERIAL, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE.

Registered Trademarks (®)

Synopsys, AMPS, Arcadia, C Level Design, C2HDL, C2V, C2VHDL, Cadabra, Calaveras Algorithm, CATS, CSim, Design Compiler, DesignPower, DesignWare, EPIC, Formality, HSPICE, Hypermodel, I, iN-Phase, in-Sync, Leda, MAST, Meta, Meta-Software, ModelAccess, ModelTools, NanoSim, OpenVera, PathMill, Photolynx, Physical Compiler, PowerMill, PrimeTime, RailMill, Raphael, RapidScript, Saber, SiVL, SNUG, SolvNet, Stream Driven Simulator, Superlog, System Compiler, Testify, TetraMAX, TimeMill, TMA, VCS, Vera, and Virtual Stepper are registered trademarks of Synopsys, Inc.

Trademarks (™)

abraCAD, abraMAP, Active Parasitics, AFGen, Apollo, Apollo II, Apollo-DPII, Apollo-GA, ApolloGAII, Astro, Astro-Rail, Astro-Xtalk, Aurora, AvanTestchip, AvanWaves, BCView, Behavioral Compiler, BOA, BRT, Cedar, ChipPlanner, Circuit Analysis, Columbia, Columbia-CE, Comet 3D, Cosmos, CosmosEnterprise, CosmosLE, CosmosScope, CosmosSE, Cyclelink, Davinci, DC Expert, DC Expert *Plus*, DC Professional, DC Ultra, DC Ultra Plus, Design Advisor, Design Analyzer, Design Vision, DesignerHDL, DesignTime, DFM-Workbench, DFT Compiler, Direct RTL, Direct Silicon Access, Discovery, DW8051, DWPCI, Dynamic-Macromodeling, Dynamic Model Switcher, ECL Compiler, ECO Compiler, EDAnavigator, Encore, Encore PQ, Evaccess, ExpressModel, Floorplan Manager, Formal Model Checker, FoundryModel, FPGA Compiler II, FPGA *Express*, Frame Compiler, Galaxy, Gatran, HDL Advisor, HDL Compiler, Hercules, Hercules-Explorer, Hercules-II, Hierarchical Optimization Technology, High Performance Option, HotPlace, HSPICE-Link, iN-Tandem, Integrator, Interactive Waveform Viewer, i-Virtual Stepper, Jupiter, Jupiter-DP, JupiterXT, JupiterXT-ASIC, JVXtreme, Liberty, Libra-Passport, Library Compiler, Libra-Visa, Magellan, Mars, Mars-Rail, Mars-Xtalk, Medici, Metacapture, Metacircuit, Metamanager, Metamixsim, Milkyway, ModelSource, Module Compiler, MS-3200, MS-3400, Nova Product Family, Nova-ExploreRTL, Nova-Trans, Nova-VeriLint, Nova-VHDLlint, Optimum Silicon, Orion_ec, Parasitic View, Passport, Planet, Planet-PL, Planet-RTL, Polaris, Polaris-CBS, Polaris-MT, Power Compiler, PowerCODE, PowerGate, ProFPGA, ProGen, Prospector, Protocol Compiler, PSMGen, Raphael-NES, RoadRunner, RTL Analyzer, Saturn, ScanBand, Schematic Compiler, Scirocco, Scirocco-i, Shadow Debugger, Silicon Blueprint, Silicon Early Access, SinglePass-SoC, Smart Extraction, SmartLicense, SmartModel Library, Softwire, Source-Level Design, Star, Star-DC, Star-MS, Star-MTB, Star-Power, Star-Rail, Star-RC, Star-RCXT, Star-Sim, Star-SimXT, Star-Time, Star-XP, SWIFT, Taurus, Taurus-Device, Taurus-Layout, Taurus-Lithography, Taurus-Process, Taurus-Topography, Taurus-Visual, Taurus-Workbench, TimeSlice, TimeTracker, Timing Annotator, TopoPlace, TopoRoute, Trace-On-Demand, True-Hspice, TSUPREM-4, TymeWare, VCS Express, VCSi, Venus, Verification Portal, VFormal, VHDL Compiler, VHDL System Simulator, VirSim, and VMC are trademarks of Synopsys, Inc.

Service Marks (SM)

MAP-in, SVP Café, and TAP-in are service marks of Synopsys, Inc.

SystemC is a trademark of the Open SystemC Initiative and is used under license.
ARM and AMBA are registered trademarks of ARM Limited.
All other product or company names may be trademarks of their respective owners.

Printed in the U.S.A.

Document Order Number: 00000-000 WA
Family Name Product Name Manual Type, version W-2004.12

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.

Copyright © 1985-2003 Synopsys, Inc.

What This Manual is About

This manual, along with the companion Book 1, 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 Reference Manual* also explains what, but in greater detail—it provides more detailed information on topics that are introduced in this manual.

Copyright © 1985-2003 Synopsys, Inc.

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 Avant! libraries. Be aware that, for simplicity, these example templates do not include the error-checking features contained in the library templates.

Copyright © 1985-2003 Synopsys, Inc.

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 SaberSketch schematic and specifying its properties, from which a netlist is created. There is no difference in simulation.

Copyright © 1985-2003 Synopsys, Inc.

Table Of Contents

Preface i
What You Need to Know to Use This Manual i
What This Manual is About ii
Basic Versus Advanced Modeling iii
Schematic Entry Versus Netlist iv
Conventions v
Revision History v

Chapter 1. Modeling Piecewise-Defined Behavior 1-1
Nonlinear Elements 1-2
Modeling a Simple Voltage Limiter with MAST 1-3
Characteristic Equations 1-5
Header and Header Declarations 1-5
Values and Equations Sections 1-5
Requirements For If Expressions 1-7
Purpose of Newton Steps 1-8
Control Section—Newton Steps 1-9
Newton Step Example 1-10
Modeling a Voltage Divider with MAST 1-11
Header Declarations 1-14
Parameters Section - MAST vdiv Template 1-15
Newton Step Parameters 1-16
Equation and Values Sections 1-18

Chapter 2. Modeling Nonlinear Devices 2-1
Modeling Nonlinear Devices with MAST 2-1
Modeling an Ideal Diode with MAST 2-2

Copyright © 1985-2004 Synopsys, Inc.

Characteristic Equation 2-4
Header Declarations 2-4
Modeling Temperature 2-5
Newton Steps 2-5
Newton Steps Example - MAST diode Template 2-6
Template Equation 2-7
Initial Conditions 2-8
Starting Value 2-8
Small-Signal Parameters 2-8
Small-Signal Parameters Report 2-8
Small-Signal Parameter Statements 2-9
Ebers-Moll MAST Model for the Bipolar Transistor 2-13
Basic Model Equations 2-19
Preparing to Write the MAST bjt Template 2-20
Header Declarations 2-21
Transistor Type 2-22
Collector Resistance 2-22
Initial Conditions 2-23
Local Parameters 2-24
Temperature 2-25
Junction Capacitance 2-25
Newton Steps Declaration - MAST bjt Template 2-25
Local Node - MAST bjt Template 2-26
Intermediate Current and Charge Variables 2-27
Defining Groups For Extraction - MAST bjt Template 2-27
Thermal Voltage 2-28
Junction Capacitance 2-28
Intermediate Calculations 2-29
Fundamental Quantities - MAST bjt Template 2-29
Currents 2-30
Charges 2-31
Control Section 2-32
Collapse Node 2-33
Newton Steps 2-33

Copyright © 1985-2004 Synopsys, Inc.

Initial Conditions in Control Section....2-33
Starting Value....2-34
Small-Signal Parameters....2-34
Equations Section....2-36

Chapter 3. Modeling Nonlinearities....3-1
Simulation Techniques for Evaluating Nonlinearities....3-2
Simulation Linearization Techniques....3-2
Taking the Slope (Method 1)....3-3
Piecewise linear approximation (Method 2)....3-4
Piecewise Linear Evaluation (Method 3)....3-4
Comparison and Summary of Linearization Techniques....3-6
Modeling a Voltage Squarer - MAST vsqr Template....3-7
Template Header....3-8
Values Section....3-8
Equations Section....3-8
Control Section....3-9
Understanding Sample Points....3-10
Considerations for Selecting Sample Points....3-11
Specifying Sample Points....3-14
Sample Point Statement Syntax....3-14
Sample Point Values....3-15
Density of Sample Points....3-16
Default Sample Points....3-16

Chapter 4. MAST Functions....4-1
Overview....4-1
Using a MAST Function Instead of a Foreign Routine....4-1
Modeling the Bipolar Transistor Using MAST Functions....4-2
Guidelines for Splitting a MAST Template into Separate Functions....4-2
The bjtm Template Architecture Using MAST Functions....4-3
The bjtm Template....4-5
Function Call Overview - bjtm MAST Template....4-7
bjtm_arg Declaration Template....4-9

Copyright © 1985-2004 Synopsys, Inc.

Local Parameters Function bjtm_pars 4-10
Function Header 4-11
Header Declaration 4-12
Function Body 4-12
Calculated Values Function bjtm_values 4-13
Function Header 4-14
Header Declaration 4-15
Function Body 4-15

Chapter 5. Foreign Routines in MAST 5-1
Introduction 5-2
Using a FORTRAN Function in a MAST Template 5-2
Writing the FORTRAN Routine 5-3
Declaring and Calling the Routine From a Template 5-5
Modeling the Bipolar Transistor Using Foreign Routines 5-6
Splitting Functionality Between a MAST Template and a Foreign Function 5-7
Modifying the BJT Template to Use a Foreign Routine 5-8
General Foreign Function Call Syntax 5-10
Calling the Foreign Routines 5-11
Implementing a MAST Foreign Routine in C 5-13
Defining Template Arguments 5-14
First Call—Setting Up Return Parameters 5-15
Second and Third Calls—Performing Calculations 5-17
Complete BJT C Routine 5-19

Chapter 6. Time-Domain Modeling 6-1
Using the MAST delay Function in an Ideal Delay Line 6-2
Ideal Delay Line (dline) MAST Template 6-3
Delayed Sine Wave Transient Analysis 6-4
Delayed Sine Wave AC Analysis 6-5
MAST dline Template Summary 6-6
Expanding the Multiple-Output Voltage Source 6-7
Overview 6-8
The vsource_2 MAST Template 6-9

Copyright © 1985-2004 Synopsys, Inc.

Header Declarations 6-12
Union Type Parameters 6-12
Sine Wave Output (sin) Declaration 6-14
Exponential Wave Output (exp) Declaration 6-15
Step Function Output (step) Declaration 6-16
Initial Values 6-16
Netlist Example 6-17
Local Declarations 6-18
Equations Section 6-18
Determining Union Elements 6-19
Assigning Internal Values 6-19
Performing Calculations (Defining Signals) 6-21
Sine Wave Output 6-22
Exponential Waveform Output 6-23
Step Function Output 6-24
No tran Output 6-25
Netlist Examples 6-27

Chapter 7. Modeling Noise 7-1
Introduction 7-1
Adding Noise to a Resistor MAST Template 7-2
Header Declarations 7-4
Local Declarations 7-4
Expression for Noise 7-5
Control Section 7-5
Adding Noise to a Voltage Source MAST Template 7-6
Adding Noise to the MAST diode Template 7-7

Chapter 8. Statistical Modeling 8-1
Introduction 8-2
Varying Values in a Simple Voltage Divider 8-2
Probability Density Functions (PDFs) 8-4
Intrinsic Probability Density Functions 8-5
Uniform Probability Density Function 8-6
Normal Probability Density Function 8-8

Copyright © 1985-2004 Synopsys, Inc.

Piecewise Linear Probability Density Function 8-11
1. Creating a Piecewise Linear Prototype PDF 8-12
2. Correspondence Between Actual Values and Prototype PDF Values 8-14
3. Using a Piecewise Linear Prototype PDF in a Netlist 8-15
Cumulative Density Functions (CDFs) 8-16
Intrinsic Piecewise Linear Cumulative Density Function 8-18
1. Creating a Piecewise Linear Prototype CDF 8-19
2. Correspondence Between Actual Values and Prototype CDF Values 8-21
3. Using a Piecewise Linear Prototype CDF in a Netlist 8-22
Correlating Distributions 8-23
Modifying Uniform and Normal Default Distributions 8-24
Modifying a Uniform Prototype Distribution 8-25
1. Modifying a Uniform Prototype PDF Using Initializers 8-25
2. Modifying a Uniform Prototype PDF in a Template 8-27
Modifying a Normal Prototype Distribution 8-28
1. Modifying a Normal Prototype PDF using Initializers 8-29
2. Modifying a Normal Prototype PDF in a Template 8-31
Parameterized PDF and CDF Specifications 8-31
The random MAST Function 8-32
Use of the statistical MAST Simvar Variable 8-33
Worst-Case Statistical MAST Modeling 8-34

Chapter 9. Adding Stress Measures to a MAST Template 9-1
Add stress_measure Statements to Template 9-1
Determine if Specified Variables are Accessible 9-4
Add Stress Ratings 9-4
Add Thermal Resistances (Optional) 9-6
Add a Way to Disable Stress (Optional) 9-8
Add a Way to Specify Device Type and Class (Optional) 9-9
MAST Example Including Stress Statements 9-11

Appendix A. Un-Structured Modeling Approach - Examples A-1
Using MAST Functions - Unstructured bjtm Template A-1

Copyright © 1985-2004 Synopsys, Inc.

Ideal Delay Line - Unstructured dline Template A-3
Multiple-Output Voltage Source - Unstructured vsource_2 Template A-4

Appendix B. Making User Templates Visible for Unix and NT B-1
Making User Templates Visible for Unix.. B-1
How the Applications Find Files B-1
Using Templates Written in MAST... B-3
Using Custom Models From Your Capture Tool B-5
Making Symbols Available in SaberSketch.. B-5
Using C or FORTRAN Routines Called by Templates B-6
How to Make a Single Routine Available to the Saber Simulator B-7
How to Make a Library of Routines Available to the Saber Simulator B-9
Making User Templates Visible for NT B-11
Making Symbols Available in SaberSketch B-13
Using Templates Written in MAST... B-14
Using C or FORTRAN Routines Called by Templates B-15
The C Language Header.. B-15
The FORTRAN Language Header B-16
How to Make a Single Routine Available to the Saber Simulator ... B-16
One-Step Dynamic Library Linking B-16
One-Step C Language Compiling and Linking B-17
One-Step FORTRAN Language Compiling and Linking B-17
How to Compile and Link Libraries of Routines.............................. B-18

Index Index1

Bookshelf Bookshelf-1

Copyright © 1985-2004 Synopsys, Inc.

Copyright © 1985-2004 Synopsys, Inc.

chapter 1

Modeling Piecewise-Defined Behavior

Examples in Book 1 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 Book 1.

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

Copyright © 1985-2003 Synopsys, Inc.

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.

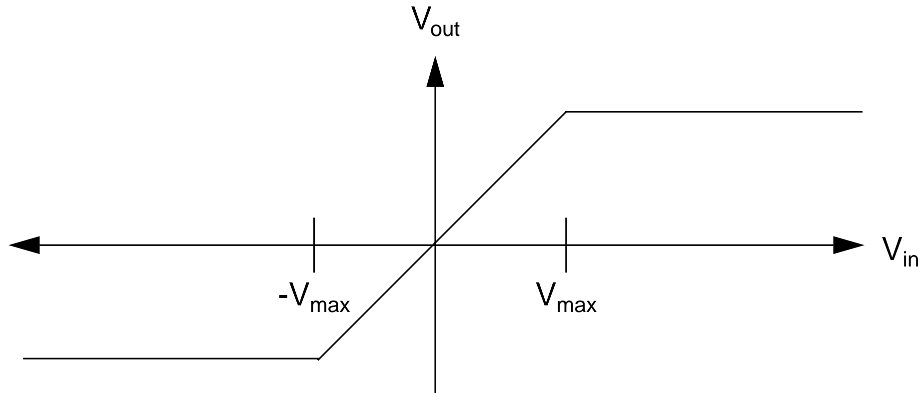
Nonlinear models are not confined to curvelinear functions—other types include those whose outputs have discontinuities or regions of piecewise linear behavior. The examples in this chapter illustrate these kinds of nonlinear characteristics.

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.

Copyright © 1985-2003 Synopsys, Inc.

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

Copyright © 1985-2003 Synopsys, Inc.

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

vlim Template Topics

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.

Copyright © 1985-2003 Synopsys, Inc.

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
}
```


Copyright © 1985-2003 Synopsys, Inc.

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.

Copyright © 1985-2003 Synopsys, Inc.

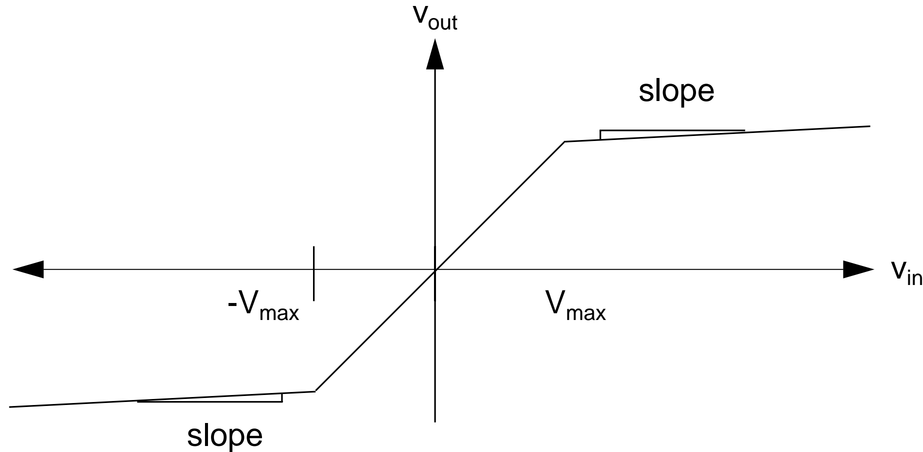
Requirements For If Expressions

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.

Copyright © 1985-2003 Synopsys, Inc.



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.

Copyright © 1985-2003 Synopsys, Inc.

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)
   }
   ```

NOTE

It is possible for a template to have multiple independent variables requiring newton steps.

Copyright © 1985-2003 Synopsys, Inc.

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`.)

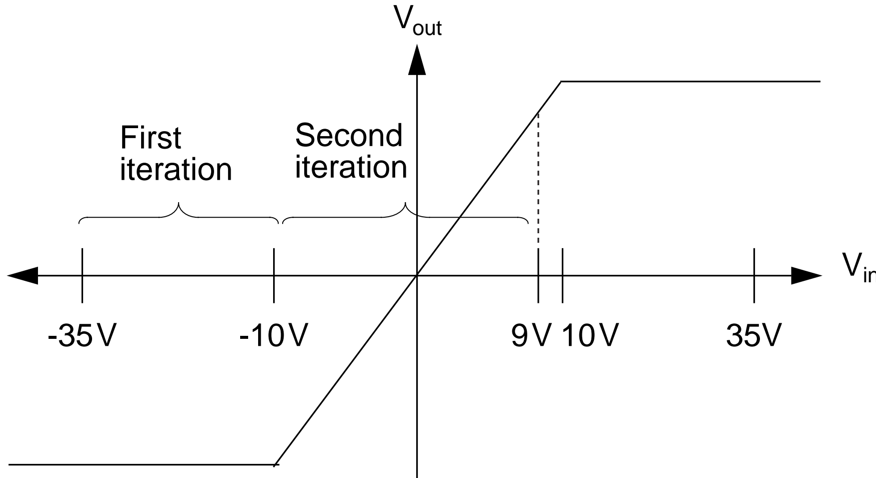
Newton Step Example

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.

Copyright © 1985-2003 Synopsys, Inc.

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).

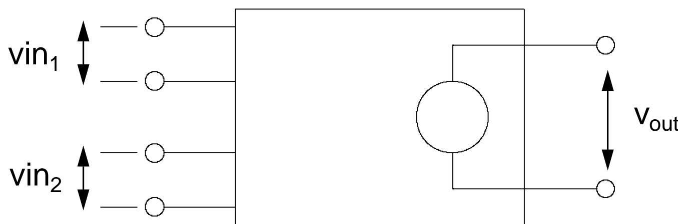
Therefore, for the next iteration, `vin` will have a value of +9V (−10 + 19), which is in the nonlimiting region.



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:



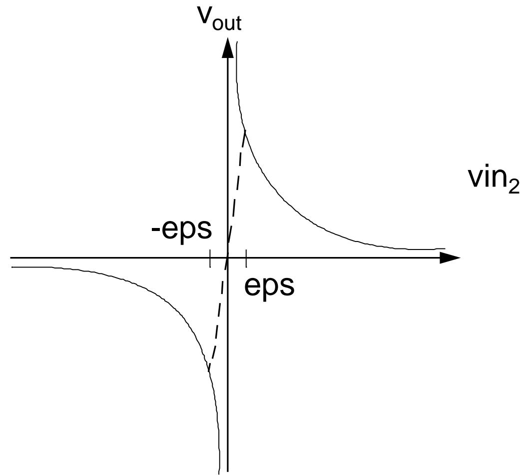
Voltage Divider

Its characteristic equation needs to express the following:

> Determine the output current such that `vout = vin1/vin2`.

Copyright © 1985-2003 Synopsys, Inc.

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.

Copyright © 1985-2003 Synopsys, Inc.

```
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`

Copyright © 1985-2003 Synopsys, Inc.

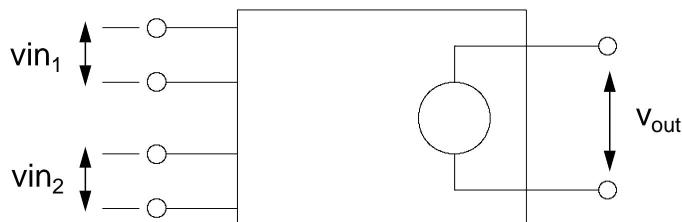
vdiv Template Topics

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

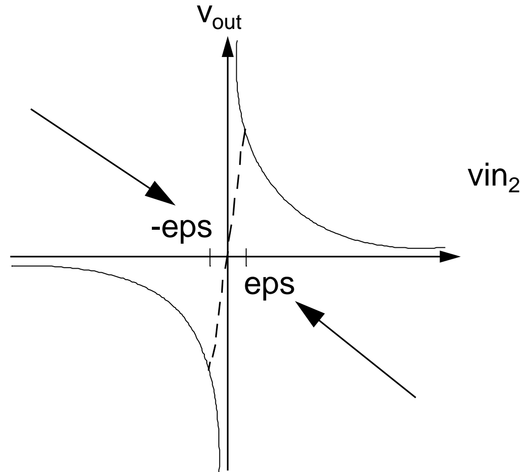
This template provides no arguments and the template header and header declaration is written as follows:

```
element template vdiv ip1 im1 ip2 im2 op om
  electrical ip1, ip2, im1, im2, op, om
```


Copyright © 1985-2003 Synopsys, Inc.

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
```


Copyright © 1985-2003 Synopsys, Inc.

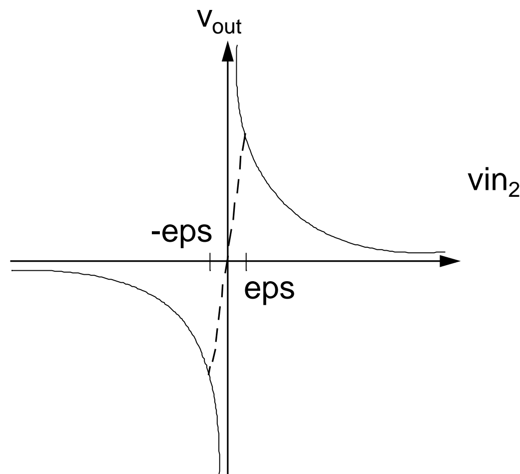
The last line in the parameters section relates to the `nv2` parameter, which is an assignment statement that specifies (breakpoint, increment) values for `nv2` as follows:

```
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
$\partial vout/\partial vin1 = 1/vin2$ 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`.

Copyright © 1985-2003 Synopsys, Inc.

Newton steps require the inclusion of three different statements in the **vdiv** template:

1. 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.
   ```

2. An assignment statement that specifies (breakpoint, increment) values for `nv2`:

   ```
   nv2 = [(-2*eps,eps),(2*eps,0)] #parameters sect.
   ```

3. 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` ≠`0` (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.

Copyright © 1985-2003 Synopsys, Inc.

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.

Copyright © 1985-2003 Synopsys, Inc.

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
}
```


Copyright © 1985-2003 Synopsys, Inc.

Copyright © 1985-2003 Synopsys, Inc.

chapter 2

Modeling Nonlinear Devices

Modeling Nonlinear Devices with MAST

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

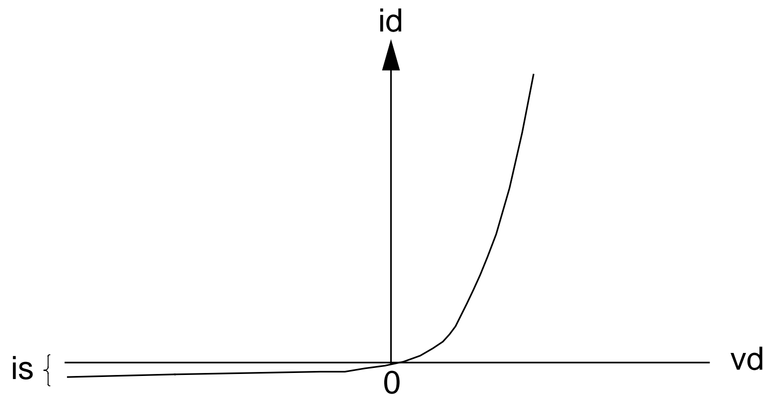
Each of these templates uses a control section to include statements for newton steps and for initial conditions. In addition, these examples introduce the following concepts:

- 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

Copyright © 1985-2004 Synopsys, Inc.

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:

```
1  element template diode p m = is, ic # template header
2    electrical p, m                   # header declarations
3    number is = 1e-16,
4           ic = undef
5    external number temp
```


Copyright © 1985-2004 Synopsys, Inc.

```
6  {                                    # start of template body
7    number k = 1.318e-23,              # local declarations
8           qe = 1.602e-19,
9           vt
10   val v vd
11   val i id
12   struc {
13     number bp, inc;        # Newton steps
14   }  nvd[*] = [(0,.001),(2,0)]
15   parameters {                   # start of parameters section
16     vt = k * (temp+273.15) / qe # compute thermal voltage
17   }                             # end of parameters section
18   values {                       # start of values section
19     vd = v(p) - v(m)             # diode voltage
20     id = is * (limexp(vd/vt)-1) # diode current
21   }                             # end of values section
22   control_section {              # start of control section
23     newton_step (vd,nvd)         # Newton steps assigned to vd
24     initial_condition(vd,ic)
25     start_value(vd,0.6)
26     device_type("diode","example")
27     small_signal(vd,voltage,"p-m voltage", vd)
28   }                             # end of control section
29   equations {                    # start of equations section
30     i(p->m) += id                # current contribut. of diode
31   }                             # end of equations section
32 }                                # 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

Copyright © 1985-2004 Synopsys, Inc.

- 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:

$$id = is \bullet (e^{(vd \bullet q)/k \bullet T} - 1) \qquad (1)$$

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: $q = 1.602 \bullet 10^{-19}$ A s (1 A s = 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:

```
1  element template diode p m = is, ic
2    electrical p, m
3    number is = 1e-16,
4           ic = undef
```


Copyright © 1985-2004 Synopsys, Inc.

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:

```
5    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:

```
15    parameters {
16      vt = k * (temp+273.15) / qe
17    }
```

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:

```
7    number  k = 1.318e-23,
8           qe = 1.602e-19,
9           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:

```
12    struc {
13      number bp, inc;
14    }  nvd[*] = [(0,0.001),(2,0)]
```


Copyright © 1985-2004 Synopsys, Inc.

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:

```
23    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.

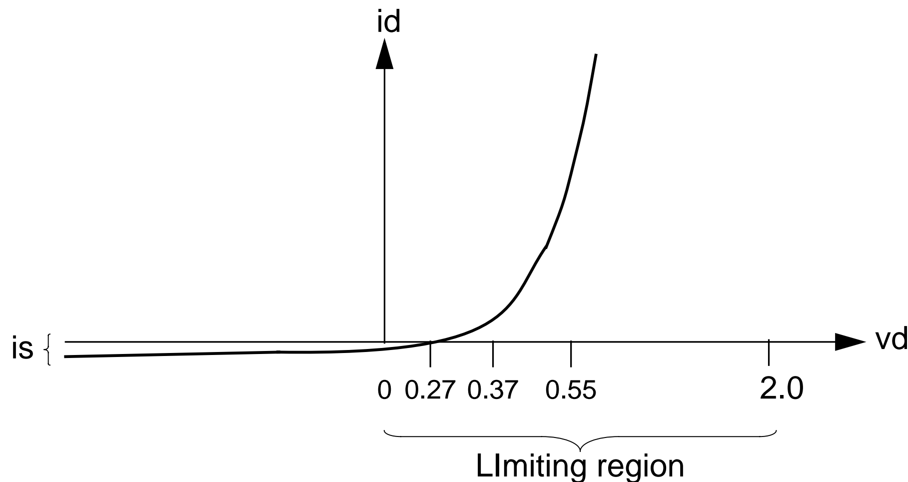
- Below the first breakpoint (`0`), there is no restriction on how much `vd` can change from one iteration to the next.
- Between the first two consecutive breakpoints (`0` and `2`), the change in `vd` is restricted to the first specified increment (`0.001`) per iteration.
- Above the last breakpoint (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.

Copyright © 1985-2004 Synopsys, Inc.



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`):

```
20      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:

```
29    equations {
30      i(p->m) += id
31    }
```

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:

```
19      vd = v(p) - v(m) # Part of values section
```


Copyright © 1985-2004 Synopsys, Inc.

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`:

```
24      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:

```
25      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.

```
26      device_type("diode","example")
27      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

Copyright © 1985-2004 Synopsys, Inc.

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.

```
26      device_type("diode","example")
```


Copyright © 1985-2004 Synopsys, Inc.

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:

```
27      small_signal(vd,voltage,"p-m voltage", vd)
```

The four fields are specified as follows:

1. parameter name (vd)—this is the name of the SSP that is reported under the Name heading by the ssp command.
2. classification (voltage)—this is reported under the Classification heading by the ssp command.
3. report identifier ("p-m voltage")—this is an identifier string that is reported under the Parameter heading by the ssp command.
4. 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)
```


Copyright © 1985-2004 Synopsys, Inc.

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)
```


Copyright © 1985-2004 Synopsys, Inc.

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.

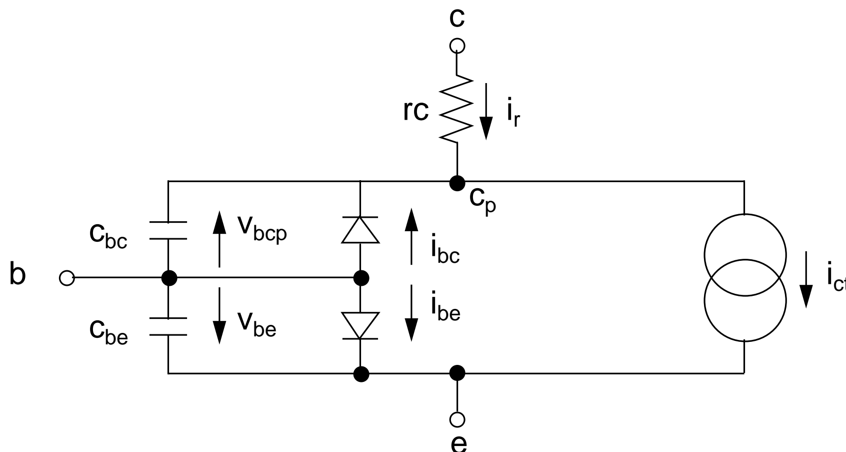
For example, in the `ss_partial` statement above, `idi` must depend directly on the value of `vdi`; `idi` cannot depend on the result of an expression containing `vdi`.

Copyright © 1985-2004 Synopsys, Inc.

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 scope 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):

Copyright © 1985-2004 Synopsys, Inc.

```
1  element template bjt c b e = model, ic
2    electrical c, b, e
3    struc {                          # the transistor mode
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
11 {                # begin template body
12    # declare local param., vals, and extraction groups
13   number k = 1.381e-23,            # Boltzmann's constant
14          qe = 1.602e-19,           # electron charge
15          vt,
16          qbe0, qbc0, vje0, vjc0
17   struc {
18     number bp, inc;
19   } nv[*] = [(0,.1),(2,0)]
20   val v vbc, vbe, vce              # declarations of vals
21   val i iec, icc, iba, ico, ir
22   val q qbc, qbe
23   electrical cp                    # local node
24   group {vbc,vbe} v                # extraction groups
25   group {iba,ico,ir} i
26   group {qbc,qbe} q
```


Copyright © 1985-2004 Synopsys, Inc.

```
27   parameters {
28    # calculate thermal volts and functions of model param.
29     vt = k * (temp + 273.15) / qe
30     qbe0 = model->cje * model->vje / (1 - model->mje)
31     qbc0 = model->cjc * model->vjc / (1 - model->mjc)
32     vje0 = 2 * model->vje / model->mje
33     vjc0 = 2 * model->vjc / model->mjc
34   }               # end of parameters 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) {
41       iec = model->is * (limexp(vbc/vt) - 1)
42       icc = model->is * (limexp(vbe/vt) - 1)
43     }
44     else {
45       iec = -model->is * (limexp(-vbc/vt) - 1)
46       icc = -model->is * (limexp(-vbe/vt) - 1)
47     }
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
53
54      # calculate charges
55     if(model->type == _n) {
56       if (vbc<0) {
57         qbc = qbc0*(1-((1-vbc/model->vjc)**(1-model->mjc)))
58       }
59       else {
60         qbc = model->cjc*vbc*(1 + vbc/vjc0)
61       }
62       if (vbe<0) {
63         qbe = qbe0*(1-((1-vbe/model->vje)**(1-model->mje)))
64       }
65       else {
66         qbe = model->cje*vbe*(1 + vbe/vje0)
67       }
68     }                   # end "if type _n" condition
```


Copyright © 1985-2004 Synopsys, Inc.

```
69     else { # if model is not of type _n
70       if(vbc > 0) {
71         qbc = -qbc0*(1-(1+vbc/model->vjc)**(1-model->mjc))
72       }
73       else {
74         qbc = model->cjc*vbc*(1-vbc/vjc0)
75       }
76       if(vbe > 0) {
77         qbe = -qbe0*(1-(1+vbe/model->vje)**(1-model->mje))
78       }
79       else {
80         qbe = model->cje*vbe*(1-vbe/vje0)
81       }
82     }                   # end "if not type _n" condition
83   }                     # end values section


84   control_section {
85      # if no collector resistance, collapse nodes c and cp
86     if (model->rc == 0) collapse(c,cp)
87      # specify Newton steps
88     newton_step((vbc,vbe),nv)
89      # initial conditions and start value
90     initial_condition(vbe, ic[1])
91     initial_condition(vce, ic[2])
92     start_value (vbe, 0.6)
93
94      # small-signal parameters
95     device_type("bjt", "example")
96     small_signal(ibase,current,"base current",iba)
97     small_signal(icoll,current,"collector current",ico)
98     small_signal(vbe,voltage,"base-emitter voltage", vbe)
99     small_signal(vbc,voltage,"base-collector voltage", vbc)
100    small_signal(rc,resistance,"collector resistance",\
101                model->rc)
102  }                     # end control_section
103  equations {
104     # current at base, internal collector, and emitter
105    i(b->e)  += iba + d_by_dt(qbe)
106    i(b->cp) += d_by_dt(qbc)
107    i(cp->e) += ico
108
109     # current at collector resistor, if present
110    if (model->rc ~= 0) i(c->cp) += ir
111  }                       # end equations section
112}                        # end template body
```

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

Copyright © 1985-2004 Synopsys, Inc.

bjt Template Topics

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

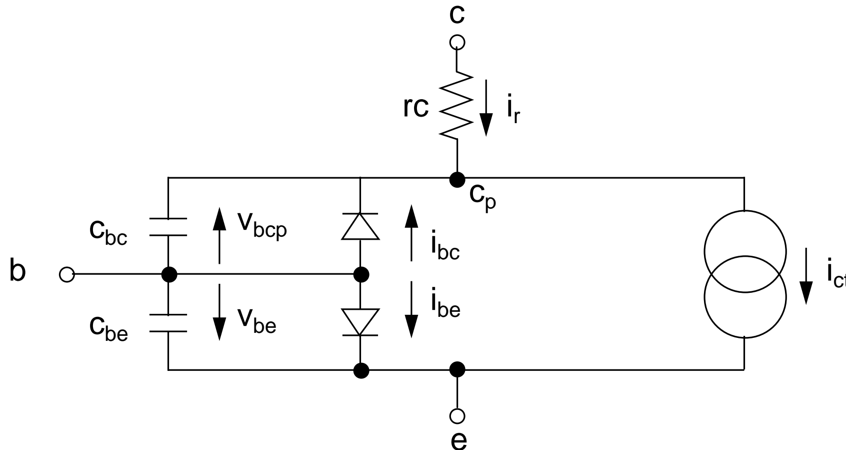
Copyright © 1985-2004 Synopsys, Inc.

    - 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

Copyright © 1985-2004 Synopsys, Inc.

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 5 pF)

c_{jeo} emitter-base junction capacitance (typically about 5 pF)

Copyright © 1985-2004 Synopsys, Inc.

v_{jc} collector-base barrier potential (typically about 0.75 V)

v_{je} emitter-base barrier potential (typically about 0.75 V)

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:

$$c_j = c_{jo} / (1 - v / v_j)^m \qquad (1)$$

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:

$$c_j = c_{jo} / (1 - v/v_j)^m \text{ for } v < 0 \qquad (2)$$

$$c_j = c_{jo} * (1 + m*v/v_j) \text{for } v \geq 0 \qquad (3)$$

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:

$$q_j(v) = \int_0^v c_j(v)\, dv \qquad (4)$$

Copyright © 1985-2004 Synopsys, Inc.

with the additional requirement that $q_j(0) = 0$. The results, for reverse and forward bias of the junction, respectively, are:

$$q_j = c_j*v_j*(1 - (1 - v/v_j)^{1-m})/(1-m) \text{ for } v < 0 \quad (5)$$

$$q_j = c_j*v*(1 + 0.5*m*v/v_j) \text{ for } v \geq 0 \quad (6)$$

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

Copyright © 1985-2004 Synopsys, Inc.

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.

Copyright © 1985-2004 Synopsys, Inc.

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.

```
10   external number temp
```


Copyright © 1985-2004 Synopsys, Inc.

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:

```
12     # declare local param., vals, and extraction groups
13   number k = 1.381e-23,            # Boltzmann's constant
14          qe = 1.602e-19,           # electron charge
15          vt,
16          qbe0, qbc0, vje0, vjc0
17   struc {
18     number bp, inc;
19   } nv[*] = [(0,.1),(2,0)]
20   val v vbc, vbe, vce              # declarations of vals
21   val i iec, icc, iba, ico, ir
22   val q qbc, qbe
23   electrical cp                    # local node
24   group {vbc,vbe} v                # extraction groups
25   group {iba,ico,ir} i
26   group {qbc,qbe} q
27   parameters {
28     # calculate thermal volts and functions of model param.
29     vt = k * (temp + 273.15) / qe
30     qbe0 = model->cje * model->vje / (1 - model->mje)
31     qbc0 = model->cjc * model->vjc / (1 - model->mjc)
32     vje0 = 2 * model->vje / model->mje
33     vjc0 = 2 * model->vjc / model->mjc
34   }                      # 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

Copyright © 1985-2004 Synopsys, Inc.

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)]
```

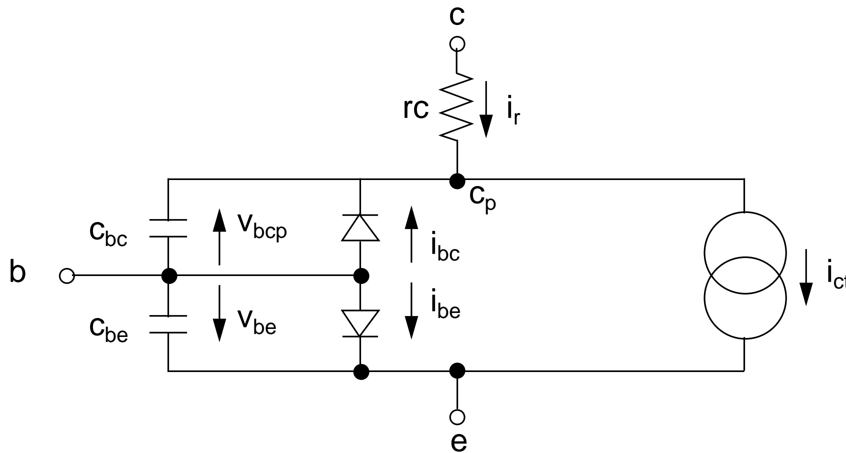
In the control section, line 88 assigns these newton step values to two independent variables (`vbc` and `vbe`) as follows:

```
88    newton_step((vbc,vbe), nv)
```


Copyright © 1985-2004 Synopsys, Inc.

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).

Copyright © 1985-2004 Synopsys, Inc.

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:

```
24  group {vbc, vbe}   v
25  group {iba, ico, ir}   i
26  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)
```


Copyright © 1985-2004 Synopsys, Inc.

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:

```
29    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:

```
30    qbe0 = model->cje * model->vje / (1 - model->mje)
31    qbc0 = model->cjc * model->vjc / (1 - model->mjc)
32    vje0 = 2 * model->vje / model->mje
33    vjc0 = 2 * model->vjc / model->mjc
```


Copyright © 1985-2004 Synopsys, Inc.

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 i_{ec}

Copyright © 1985-2004 Synopsys, Inc.

and i_{cc}. 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 i_{ct}, i_{bc}, i_{be}, and i_r. 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 i_e = `-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`.

Copyright © 1985-2004 Synopsys, Inc.

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:

```
54     # calculate charges
55    if(model->type == _n) {
56      if (vbc<0) {
57        qbc = qbc0*(1-(1-vbc/model->vjc)**(1-model->mjc)
58      }
59      else {
60        qbc = model->cjc*vbc*(1 + vbc/vjc0)
61      }
62      if (vbe<0) {
63        qbe = qbe0*(1-(1-vbe/model->vje)**(1-model->mje)
64      }
65      else {
66        qbe = model->cje*vbe*(1 + vbe/vje0)
67      }
68    }                   # end "if type _n" condition
69    else {              # if model is not of type _n
70      if(vbc > 0) {
71        qbc = -qbc0*(1-((1+vbc/model->vjc)**(1-model->mjc)))
72      }
73      else {
74        qbc = model->cjc*vbc*(1-vbc/vjc0)
75      }
76      if(vbe > 0) {
77        qbe = -qbe0*(1-((1+vbe/model->vje)**(1-model->mje)))
78      }
79      else {
80        qbe = model->cje*vbe*(1-vbe/vje0)
81      }
82    }                  # end "if not type _n" condition
```


Copyright © 1985-2004 Synopsys, Inc.

Control Section

The following lines comprise the **bjt** template control section:

```
84  control_section {
85    # if no collector resistance, collapse nodes c and cp
86    if (model->rc == 0) collapse(c,cp)
87     # specify Newton steps
88    newton_step((vbc,vbe),nv)
89     # initial conditions and start value
90    initial_condition(vbe, ic[1])
91    initial_condition(vce, ic[2])
92    start_value (vbe, 0.6)
93
94     # small-signal parameters
95    device_type("bjt", "example")
96    small_signal(ibase,current,"base current",iba)
97    small_signal(icoll,current,"collector current",ico)
98    small_signal(vbe,voltage,"base-emitter voltage", vbe)
99    small_signal(vbc,voltage,"base-collector voltage", vbc)
100   small_signal(rc,resistance,"collector resistance",\
101                model->rc)
102 }                      # 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

Copyright © 1985-2004 Synopsys, Inc.

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:

```
85    # if no collector resistance, collapse nodes c and cp
86    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 v_{bcp} and v_{be} (`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:

```
88    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:

```
90    initial_condition (vbe,ic[1])
91    initial_condition (vce,ic[2])
```


Copyright © 1985-2004 Synopsys, Inc.

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 0.6 V).

```
92    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). Refer to the* **q** *template in SaberBook for a more elaborate model of the bipolar transistor; its control section illustrates the many uses of SSP statements.*

Copyright © 1985-2004 Synopsys, Inc.

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:

```
95    device_type("bjt", "example")
```


Copyright © 1985-2004 Synopsys, Inc.

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:

```
96    small_signal(ibase,current,"base current",iba)
97    small_signal(icoll,current,"collector current",ico)
98    small_signal(vbe,voltage,"base-emitter voltage", vbe)
99    small_signal(vbc,voltage,"base-collector voltage", vbc)
100   small_signal(rc,resistance,"collector resistance",\
101                model->rc)
```

Equations Section

The template equations list the branch currents and express them as functions of the intermediate variables previously calculated:

```
103  equations {
104     # current at base, internal collector, and emitter
105    i(b->e)  += iba + d_by_dt(qbe)
106    i(b->cp) += d_by_dt(qbc)
107    i(cp->e) += ico
108
109     # current at collector resistor, if present
110    if (model->rc ~= 0) i(c->cp) += ir
110  }                         # end equations section
```


Copyright © 1985-2004 Synopsys, Inc.

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.

Copyright © 1985-2004 Synopsys, Inc.

Copyright © 1985-2004 Synopsys, Inc.

chapter **3**

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.

Copyright © 1985-2004 Synopsys, Inc.

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                                                    (1)
```

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

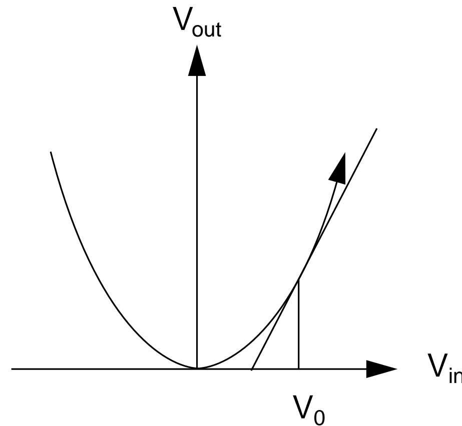
There are several techniques for the linearization of nonlinear equations. Of these, three used by different simulators are described as follows:

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)

Copyright © 1985-2004 Synopsys, Inc.

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.



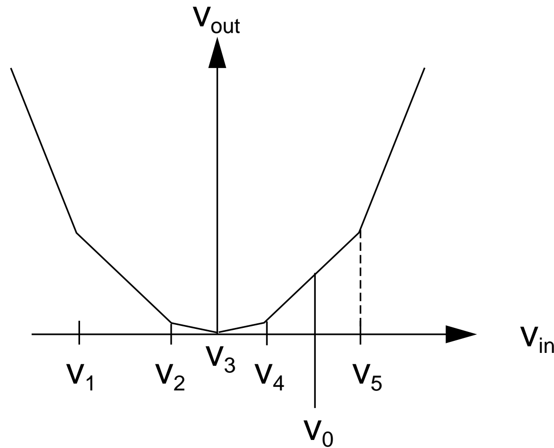
Linearization by taking the slope—Method 1

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.

Copyright © 1985-2004 Synopsys, Inc.

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)

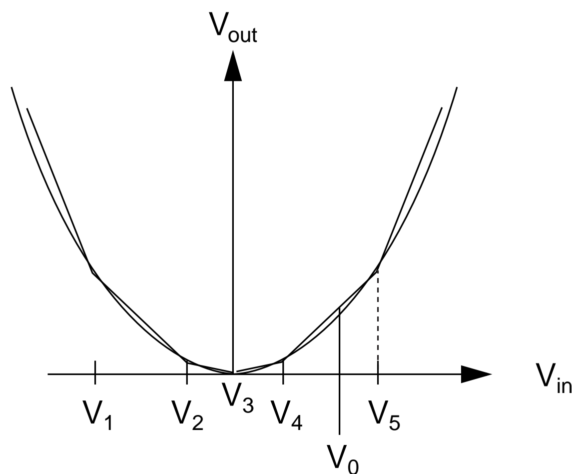
A third linearization technique, the one used by the Saber simulator, is called *piecewise linear evaluation*. The model consists of the nonlinear equations plus a set of sample points for the independent variables. The simulator uses the sample points (which may be specified in the template) to find a piecewise linear approximation of the nonlinear equations, as shown in the figure below, where v1...v5 are the sample points.

Copyright © 1985-2004 Synopsys, Inc.

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). Refer to the Calibrating DC Analysis topic in SaberBook for more information on density. This enables the user to choose (at run time) either increased accuracy or faster computation time.



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.

Copyright © 1985-2004 Synopsys, Inc.

Comparison and Summary of Linearization Techniques

These three linearization techniques can be summarized as follows:

1. *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.
2. *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.
3. *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

Copyright © 1985-2004 Synopsys, Inc.

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

Copyright © 1985-2004 Synopsys, Inc.

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
```


Copyright © 1985-2004 Synopsys, Inc.

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)
```


Copyright © 1985-2004 Synopsys, Inc.

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").

Copyright © 1985-2004 Synopsys, Inc.

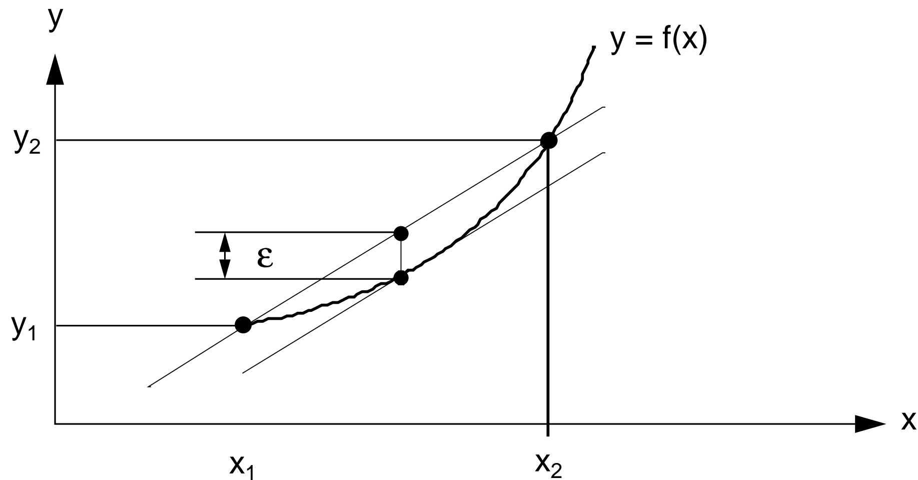
Considerations for Selecting Sample Points

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.

Copyright © 1985-2004 Synopsys, Inc.

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 x_1 and x_2. That is, x_1 and x_2 are sample points of y=f(x). Further, y_1 and y_2 are the function values at x_1 and x_2, respectively, and ε is the maximum error of the linear approximation of f(x) between x_1 and x_2. In order to find approximate sample points for x, you need to express ε as a function of $\Delta x = x_1 - x_2$. 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 ε is always smaller than $\Delta y = y_2 - y_1$ if the sign of the slope of f(x) does not change between x_1 and x_2.

Therefore, if f(x) is monotonic between x_1 and x_2, Δy 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 Δy to be smaller than this level of resolution.

Copyright © 1985-2004 Synopsys, Inc.

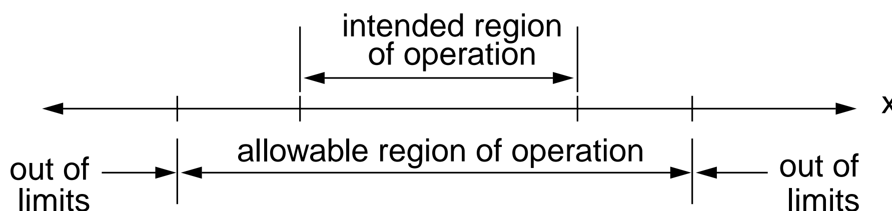
Next, find Δx as a function of Δy. In many cases, it is possible to do this using the inverse relationship $x=f^{-1}(y)$. In more complicated cases (particularly with multi-dimensional nonlinearities), selecting the sample points may be a process of trial and error. Keep in mind the concept of resolution or “granularity” as an aid for quickly finding a reasonable set of sample points. Remember too, that Δy 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 x_1 and x_2, divide Δx 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 ε is smaller than Δy. Therefore, if Δx is sufficiently small, Δy is a good upper bound for the approximation error.

Specific Approach (voltage squarer)

For **vsqr**, use the direct approach. Note that $\varepsilon=\Delta x^2$, 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, Δx should be proportional to $\sqrt{x}$. 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 ±100kV. Select sample points separated by 1V near the outer limits of the allowable region and closer together near 0V, which gives a relative approximation error that is fairly constant.



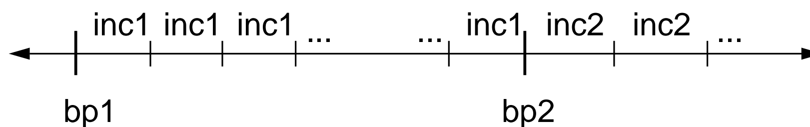
Value range of an independent variable

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.

Copyright © 1985-2004 Synopsys, Inc.

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)

Copyright © 1985-2004 Synopsys, Inc.

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)

Copyright © 1985-2004 Synopsys, Inc.

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 `den`sity 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)]
```


Copyright © 1985-2004 Synopsys, Inc.

chapter 4

MAST Functions

Overview

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.

Copyright © 1985-2004 Synopsys, Inc.

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.

Copyright © 1985-2004 Synopsys, Inc.

- 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

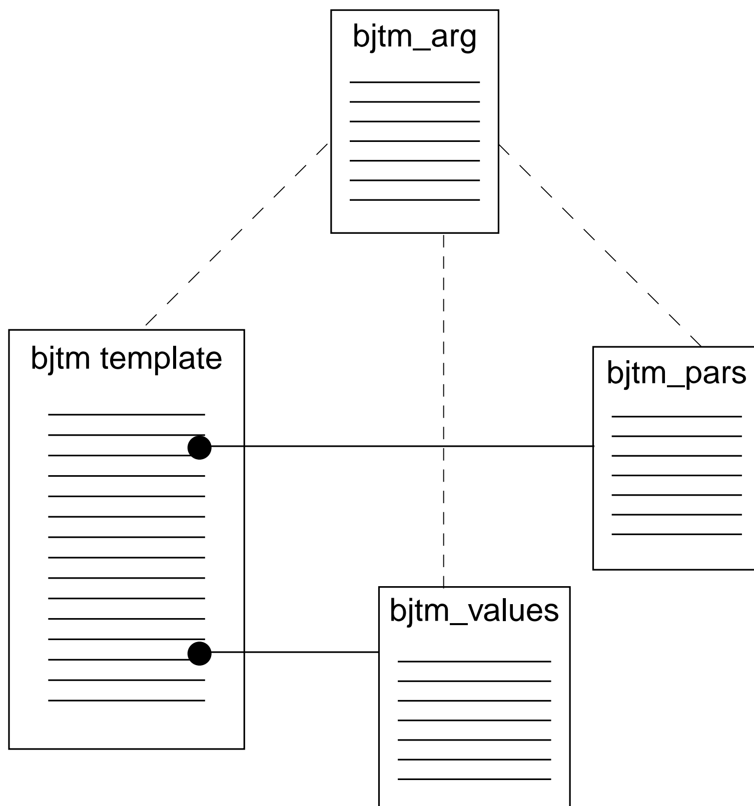
By implementing the guidelines described in the previous topic, the **bjtm** template is largely the same as the **bjt** template—the only parts that changed are the calls to the MAST functions (named **bjtm_arg**, **bjtm_pars**, and **bjt_values**), which are indicated by comments in the template.

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

Copyright © 1985-2004 Synopsys, Inc.

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

Copyright © 1985-2004 Synopsys, Inc.

The bjtm Template

The **bjtm** template including MAST function calls is shown as follows:

```
1  element template bjtm c b e = model, ic
2    electrical c, b, e
3    bjtm_arg..model model = () # ... use arguments from
4                               # ... "companion" template
5    number ic[2]=[undef,undef]
6    external number temp
7  {                            # begin template body
8    bjtm_arg..work work     # ... use local parameters
9                            # ... from "companion" template
10   struc {
11     number bp, inc;
12   } nv[*] = [(0,.1),(2,0)]
13   val v vbc, vbe, vce             # declare variables
14   val i iec, icc, iba, ico, ir
15   val q qbc, qbe
16   electrical cp                   # local node
17   group {vbc,vbe} v               # extraction groups
18   group {iba,ico,ir} i
19   group {qbc,qbe} q
20   parameters {
21   # calculate thermal voltage and 4 functions of model param.
22                           # ... 1'st call to MAST function
23     work = bjtm_pars(model,temp)
24   }                              # end of parameters section
```


Copyright © 1985-2004 Synopsys, Inc.

```
25   values {
26      # calculate basic quantities of npn and pnp trans.
27     vbc = v(b) - v(cp)
28     vbe = v(b) - v(e)
29     vce = v(cp) - v(e)
30                         # ... 2'ond call to MAST function
31     (iec,qbc,icc,qbe) = bjtm_values(model,work,vbc,vbe)
32
33      # calculate base, collector, and resistor currents
34     iba = iec/model->br + icc/model->bf
35     ico = icc - iec - iec/model->br
36     if (model->rc ~= 0) ir = (v(c) - v(cp)) / model->rc
37     else                ir = 0
38   }                          # end values section
39   control_section {
40      # if no collector resistance, collapse nodes c and cp
41     if (model->rc == 0) collapse(c,cp)
42      # specify Newton steps
43     newton_step((vbc,vbe),nv)
44      # initial conditions and start value
45     initial_condition(vbe, ic[1])
46     initial_condition(vce, ic[2])
47     start_value (vbe, 0.6)
48      # small-signal parameters
49     device_type("bjtm", "example")
50     small_signal(ibase,current,"base current",iba)
51     small_signal(icoll,current,"collector current",ico)
52     small_signal(vbe,voltage,"base-emitter voltage", vbe)
53     small_signal(vbc,voltage,"base-collector voltage", vbc)
54     small_signal(rc,resistance,"collector resistance",\
55                  model->rc)
56   }                          # end control_section
57   equations {
58      # current at base, internal collector, and emitter
59     i(b->e)  += iba + d_by_dt(qbe)
60     i(b->cp) += d_by_dt(qbc)
61     i(cp->e) += ico
62
63      # current at collector resistor, if present
64     if (model->rc ~= 0) i(c->cp) += ir
65   }                        # end equations section
66 }                          # end template body
```

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

Copyright © 1985-2004 Synopsys, Inc.

For an overview description of the three types of function calls in this template, refer to the topic titled "Function Call Overview - bjtm MAST Template".

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.)

   ```
   3  bjtm_arg..model model = ()# use local parameters
   4                            # from "companion" template
     # ...
   8  bjtm_arg..work  work      # use local parameters
   9                            # 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 (see the topic titled "Local Parameters Function bjtm_pars") provides values to the `work` parameter as the first call from the **bjtm** template:

   ```
   22                    # 1'st call to MAST function
   23    work = bjtm_pars(model,temp)
   ```

   This shows the syntax for calling an external MAST function:

   (*variable_list*) = *name*(*argument_list*)

Copyright © 1985-2004 Synopsys, Inc.

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:

```
30                    # 2'ond call to MAST function
31  (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.

Copyright © 1985-2004 Synopsys, Inc.

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.

```
1  template  bjtm_arg = model, work
2
3    # the bjt model...
4    struc {
5      enum{_n, _p} type
6      number is=1e-16, bf=100, br=1,
7      cje=0, vje=.75, mje=.33,
8      cjc=0, vjc=.75, mjc=.33, rc = 0
9    } model
10
11 # working parameters for local bjt calculations...
12   struc {
13     number vt,
14     qbe0, qbc0, vje0, vjc0
15   } work
16
17 {   # empty template body...
18 }
```


Copyright © 1985-2004 Synopsys, Inc.

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:

```
8  bjtm_arg..work  work  # use local parameters
9                        # 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

Copyright © 1985-2004 Synopsys, Inc.

```
1  function (work)= bjtm_pars(model,temp)
2
3    bjtm_arg..work work     # output from this function
4    bjtm_arg..model model   # input to this function
5    number temp             # input to this function
6  {
7    # The following include file declares math_boltz constant
8    # (is k in bjt template), the math_charge constant
9    # (is qe in bjt template), and the math_ctok constant
10 <consts.sin
11
12   # Calculation of thermal voltage and 4 other quantities
13
14   work->vt = math_boltz * (temp + math_ctok) / math_charge;
15   work->qbe0 = model->cje * model->vje / (1.0 - model->mje);
16   work->qbc0 = model->cjc * model->vjc / (1.0 - model->mjc);
17   work->vje0 = 2.0 * model->vje / model->mje;
18   work->vjc0 = 2.0 * model->vjc / model->mjc;
19 }
```

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`).

Copyright © 1985-2004 Synopsys, Inc.

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:

```
10 <consts.sin
```


Copyright © 1985-2004 Synopsys, Inc.

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:

```
14  work->vt = math_boltz * (temp + math_ctok) / math_charge;
15  work->qbe0 = model->cje * model->vje / (1.0 - model->mje);
16  work->qbc0 = model->cjc * model->vjc / (1.0 - model->mjc);
17  work->vje0 = 2.0 * model->vje / model->mje;
18  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".

```
1   function (iec,qbc,icc,qbe) =  bjtm_values(model,work,vbc,vbe)
2
3     val i iec, icc          # output from function
4     val q qbc, qbe          # output from function
5     bjtm_arg..model model   # input to function
6     bjtm_arg..work work     # input to function
7     val v vbc, vbe          # input to function
8   {
9      # calculate basic quantities of npn and pnp trans.
10    if (model->type == _n) {
11      iec = model->is * (limexp(vbc/work->vt) - 1)
12      icc = model->is * (limexp(vbe/work->vt) - 1)
13    }
14    else {
15      iec = -model->is * (limexp(-vbc/work->vt) - 1)
16      icc = -model->is * (limexp(-vbe/work->vt) - 1)
17    }
```


Copyright © 1985-2004 Synopsys, Inc.

```
18    # calculate charges
19   if(model->type == _n) {
20    if (vbc<0) {
21     qbc = work->qbc0*(1-((1-vbc/model->vjc)**(1-model->mjc)))
22    }
23    else {
24     qbc = model->cjc*vbc*(1 + vbc/work->vjc0)
25    }
26    if (vbe<0) {
27     qbe = work->qbe0*(1-((1-vbe/model->vje)**(1-model->mje)))
28    }
29    else {
30     qbe = model->cje*vbe*(1 + vbe/work->vje0)
31    }
32   }                   # end "if type _n" condition
33   else { # if model is not of type _n
34    if(vbc > 0) {
35     qbc = -work->qbc0*(1-((1+vbc/model->vjc)**(1-model->mjc)))
36    }
37    else {
38     qbc = model->cjc*vbc*(1-vbc/work->vjc0)
39    }
40    if(vbe > 0) {
41     qbe = -work->qbe0*(1-((1+vbe/model->vje)**(1-model->mje)))
42    }
43    else {
44     qbe = model->cje*vbe*(1-vbe/work->vje0)
45    }
46   }                    # end "if not type _n" condition
47 }
```

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`):

```
1  function(iec,qbc,icc,qbe)=bjtm_values(model,work,vbc,vbe)
```


Copyright © 1985-2004 Synopsys, Inc.

Header Declaration

The following five lines from the **bjtm_values** function declare the input and output parameters appearing in the header:

```
3  val i iec, icc          # output from function
4  val q qbc, qbe          # output from function
5  bjtm_arg..model model   # input to function
6  bjtm_arg..work work     # input to function
7  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 `val`s 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:

```
8  {                  # start template body
9    # calculate basic quantities of npn and pnp trans.
10   if (model->type == _n) {
11     iec = model->is * (limexp(vbc/work->vt) - 1)
12     icc = model->is * (limexp(vbe/work->vt) - 1)
13   }
14   else {
15     iec = -model->is * (limexp(-vbc/work->vt) - 1)
16     icc = -model->is * (limexp(-vbe/work->vt) - 1)
17   }
```


Copyright © 1985-2004 Synopsys, Inc.

```
18     # calculate charges
19  if(model->type == _n) {
20   if (vbc<0) {
21    qbc = work->qbc0*(1-((1-vbc/model->vjc)**(1-model->mjc)))
22   }
23   else {
24    qbc = model->cjc*vbc*(1 + vbc/work->vjc0)
25   }
26   if (vbe<0) {
27    qbe = work->qbe0*(1-((1-vbe/model->vje)**(1-model->mje)))
28   }
29   else {
30    qbe = model->cje*vbe*(1 + vbe/work->vje0)
31   }
32  }                   # end “if type _n” condition
33  else { # if model is not of type _n
34   if(vbc > 0) {
35    qbc = -work->qbc0*(1-((1+vbc/model->vjc)**(1-model->mjc)))
36   }
37   else {
38    qbc = model->cjc*vbc*(1-vbc/work->vjc0)
39   }
40   if(vbe > 0) {
41    qbe = -work->qbe0*(1-((1+vbe/model->vje)**(1-model->mje)))
42   }
43   else {
44    qbe = model->cje*vbe*(1-vbe/work->vje0)
45   }
46  }                 # end “if not type _n” condition
47 }
```


Copyright © 1985-2004 Synopsys, Inc.

chapter 5

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.

Copyright © 1985-2004 Synopsys, Inc.

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
```


Copyright © 1985-2004 Synopsys, Inc.

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

  Refer to the *Installation and Configuration Guide*, under topic titled "Using C or FORTRAN Routines Called by Templates" for a complete and up-to-date description of how to make foreign routines 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
```


Copyright © 1985-2004 Synopsys, Inc.

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.

Copyright © 1985-2004 Synopsys, Inc.

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:

1. 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.

Copyright © 1985-2004 Synopsys, Inc.

2. 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**) that is described in the topic titled "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

Copyright © 1985-2004 Synopsys, Inc.

- 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.

Copyright © 1985-2004 Synopsys, Inc.

- 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`).

```
1  element template bjtf c b e = model, ic
2    electrical c, b, e
3    struc {                      # the transistor model
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
```


Copyright © 1985-2004 Synopsys, Inc.

```
11 {                             # begin template body
12    # declare local param., vals, and extraction groups
13   number work[5]
14   struc {number bp,inc;} \
15     nv[*] = [(0,.1),(2,0)]
16   val v vbc, vbe, vce
17   val i iec, icc, iba, ico, ir
18   val q qbc, qbe
19   electrical cp               # local node
20   group {vbc,vbe} v #...extraction groups
21   group {iba,ico, ir} i
22   group {qbc,qbe} q
23   foreign bjt                 # ... foreign routine
24   control_section {
25    # If no collector res., collapse nodes c and cp
26   if(model->rc == 0) collapse(c,cp)
27    # specification of sample points and newton steps
28   newton_step((vbc,vbe),nv)
29     # initial conditions, start values, and
30     #  small-signal parameters removed for brevity
31   }

32   parameters {
33     # Calculate thermal voltage and 4 functions of model
34     #  parameters. They are stored in a work vector
35     work = bjt(1,model,temp)             #...foreign call
36   }
37
38   values {
39     vbc = v(b,cp)
40     vbe = v(b,e)
41     vce = v(cp,e)
42      # calculate currents and charges
43     if(model->type == _n) {
44       (iec,qbc) = bjt(2,model,work,vbc)  #...foreign call
45       (icc,qbe) = bjt(3,model,work,vbe)  #...foreign call
46     }
47     else {
48       (iec,qbc) = bjt(2,model,work,-vbc) #...foreign call
49       (icc,qbe) = bjt(3,model,work,-vbe) #...foreign call
50     }
51      #calculate base, collector, and resistor currents
52     iba = iec/model->br + icc/model->bf
53     ico = icc - iec - iec/model->br
54     if (model->rc ~= 0) ir = (v(c,cp)) / model->rc
55     else                ir = 0
56   }                            # end of values section
```


Copyright © 1985-2004 Synopsys, Inc.

```
57   equations {
58      # current at base, internal collector, and emitter
59     i(b->e)  += iba + d_by_dt(qbe)
60     i(b->cp) += d_by_dt(qbc)
61     i(cp->e) += ico
62
63      # current at collector resistor, if present
64     if (model->rc ~= 0) i(c->cp) += ir
65   }                      # end of equations section
66 }                        # 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.

Copyright © 1985-2004 Synopsys, Inc.

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:

   ```
   23   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 `val`s (that is, a pair of simple numeric values) for the second and third calls.

2. The first call to the `bjt` foreign routine is:

   ```
   35   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".

Copyright © 1985-2004 Synopsys, Inc.

3. The calls to the `bjt` routine in the following lines are made to compute currents and charges:

```
43    if(model->type == _n) {
44      (iec,qbc) = bjt(2,model,work,vbc)
45      (icc,qbe) = bjt(3,model,work,vbe)
46    }
47    else {
48      (iec,qbc) = bjt(2,model,work,-vbc)
49      (icc,qbe) = bjt(3,model,work,-vbe)
50    }
```

   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 Book 2 and are therefore not discussed here.

Copyright © 1985-2004 Synopsys, Inc.

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 __stdcall 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 upper-case 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. Refer to the *Installation and Configuration Guide*, under topic titled “Using C or FORTRAN Routines Called by Templates” for more information.

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

Copyright © 1985-2004 Synopsys, Inc.

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:

```
4   #define JOB                    in[0]
5   #define PARAMETERS             1
6   #define BC_CHARACTERISTICS     2
7   #define BE_CHARACTERISTICS     3
```


Copyright © 1985-2004 Synopsys, Inc.

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:

```
 8  #define MODEL_TYPE  in[1]
 9  #define _N 1  /* npn */
10 #define _P 2  /* pnp */
11 #define MODEL_IS   in[2]
12 #define MODEL_BF   in[3]
13 #define MODEL_BR   in[4]
14 #define MODEL_CJE  in[5]
15 #define MODEL_VJE  in[6]
16 #define MODEL_MJE  in[7]
17 #define MODEL_CJC  in[8]
18 #define MODEL_VJC  in[9]
19 #define MODEL_MJC  in[10]
20 #define MODEL_RC   in[11]
```

In addition, the system temperature, which is a local parameter is defined as follows:

```
21 #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.

Copyright © 1985-2004 Synopsys, Inc.

Each array element has been given a name that indicates its use in the following lines:

```
24 #define WORK_P_SIZE  out[0]
25 #define WORK_P_VT    out[1]
26 #define WORK_P_QBE0  out[2]
27 #define WORK_P_QBC0  out[3]
28 #define WORK_P_VJE0  out[4]
29 #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:

```
51 #define K   1.381e-23  /*Boltzmann's constant */
52 #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:

```
75 if (JOB == PARAMETERS) {
76  /* Calculate thermal voltage and four other quant. */
77  WORK_P_VT = K * (TEMP + 273.15) / QE;
78  WORK_P_QBE0 = MODEL_CJE * MODEL_VJE / (1.0 - MODEL_MJE);
79  WORK_P_QBC0 = MODEL_CJC * MODEL_VJC / (1.0 - MODEL_MJC);
80  WORK_P_VJE0 = 2.0 * MODEL_VJE / MODEL_MJE;
81  WORK_P_VJC0 = 2.0 * MODEL_VJC / MODEL_MJC;
```

The second part consists of defining the size of the `work` array as follows:

```
83  /* Define the array size */
84  WORK_P_SIZE = 5;
85 }
```

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.

Copyright © 1985-2004 Synopsys, Inc.

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):

```
32 #define WORK_V_SIZE   in[12]
33 #define WORK_V_VT     in[13]
34 #define WORK_V_QBE0   in[14]
35 #define WORK_V_QBC0   in[15]
36 #define WORK_V_VJE0   in[16]
37 #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.

```
41 #define VBC          in[18]
42 #define IEC          out[0]
43 #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:

```
86 else if (JOB == BC_CHARACTERISTICS) {
87   /* Calculation of iec and qbc from vbc */
88   IEC = MODEL_IS * (exp(VBC / WORK_V_VT) - 1.0);
89   if (VBC < 0.0) {
90     QBC=WORK_V_QBC0 *
91     (1.0 - pow(1.0 - VBC/MODEL_VJC, 1.0 - MODEL_MJC));
92   }
93   else {
94     QBC = MODEL_CJC * VBC * (1.0 + VBC / WORK_V_VJC0);
95   }
96   if (MODEL_TYPE == _P) {
97     IEC = -IEC;
98     QBC = -QBC;
99   }
100 }
```


Copyright © 1985-2004 Synopsys, Inc.

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`:

```
47 #define VBE          in[18]
48 #define ICC          out[0]
49 #define QBE          out[1]

86 else if (JOB == BE_CHARACTERISTICS) {
87   /* Calculation of icc and qbe from vbc */
88   ICC = MODEL_IS * (exp(VBE / WORK_V_VT) - 1.0);
89   if (VBE < 0.0) {
90     QBE=WORK_V_QBE0 *
91     (1.0 - pow(1.0 - VBE/MODEL_VJE, 1.0 - MODEL_MJE));
92   }
93   else {
94     QBE = MODEL_CJE * VBE * (1.0 + VBE / WORK_V_VJE0);
95   }
96   if (MODEL_TYPE == _P) {
97     ICC = -ICC;
98     QBE = -QBE;
99   }
100 }
```

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*.

Copyright © 1985-2004 Synopsys, Inc.

Complete BJT C Routine

The following is a listing of the complete `bjt` routine, written in the C language.

```
1  /* Define names for the input and output of the foreign*/
2  /* routine. First the part that is common to all usage:*/
3  /* result = bjt(job, model...) */
4  #define JOB  in[0]
5  #define PARAMETERS 1
6  #define BC_CHARACTERISTICS 2
7  #define BE_CHARACTERISTICS 3
8  #define MODEL_TYPE  in[1]
9  #define _N  1                  /* npn */
10 #define _P  2                  /* pnp */
11 #define MODEL_IS    in[2]
12 #define MODEL_BF    in[3]
13 #define MODEL_BR    in[4]
14 #define MODEL_CJE   in[5]
15 #define MODEL_VJE   in[6]
16 #define MODEL_MJE   in[7]
17 #define MODEL_CJC   in[8]
18 #define MODEL_VJC   in[9]
19 #define MODEL_MJC   in[10]
20 #define MODEL_RC    in[11]
21 #define TEMP        in[12]   /* local parameter */
```


Copyright © 1985-2004 Synopsys, Inc.

```
22 /* Define names for 1st call     */
23 /* work = bjt(1, model, temp)  */
24 #define WORK_P_SIZE   out[0]
25 #define WORK_P_VT     out[1]
26 #define WORK_P_QBE0   out[2]
27 #define WORK_P_QBC0   out[3]
28 #define WORK_P_VJE0   out[4]
29 #define WORK_P_VJC0   out[5]
30 /* Define names for 2nd and 3rd calls   */
31 /* result = bjt(job, model, work,...)       */
32 #define WORK_V_SIZE    in[12]
33 #define WORK_V_VT      in[13]
34 #define WORK_V_QBE0    in[14]
35 #define WORK_V_QBC0    in[15]
36 #define WORK_V_VJE0    in[16]
37 #define WORK_V_VJC0    in[17]
38
39 /* Define names for calculation of base/collector charac. */
40 /* (iec, qbc) = bjt(2, model, work, vbc) */
41 #define VBC            in[18]
42 #define IEC            out[0]
43 #define QBC            out[1]
44
45 /* Define names for calculation of base/emitter charac. */
46 /* (icc, qbe) = bjt(3, model, work, vbe) */
47 #define VBE            in[18]
48 #define ICC            out[0]
49 #define QBE            out[1]
```


Copyright © 1985-2004 Synopsys, Inc.

```
50 /* Other defines */
51 #define K    1.381e-23   /* Boltzmann`s constant */
52 #define QE   1.602e-19   /* electron charge        */
53
54 /*---------------------------------------------------*/
55 /*The following two include statements are
56   system-provided files used to declare input/output
57   channels and the exp() and pow() mathematical functions*/
58 #include <stdio.h>
59 #include <math.h>
60 #include “saberApi.h” /* Specify the complete path here to
61                       “<install_home>/include/saberApi.h” */
62
63 #if defined(_MSC_VER)
64 #define bjt BJT
65 #endif
66
67  /*The following line works for SunOS 5.5.1 - 5.6 platforms
68   void bjt_(double*in, long*nin, long*ifl, long*nifl, */
69
70  /* The following line works for HP platforms */
71 void bjt(double*in, long*nin, long*ifl, long*nifl,
72  double*out, long*nout, long*ofl, long*nolf, double*undef,
73  long*ier)
74 {
```


Copyright © 1985-2004 Synopsys, Inc.

```
75   if (JOB == PARAMETERS) {
76    /* Calculate thermal voltage and four other quan. */
77     WORK_P_VT = K * (TEMP + 273.15) / QE;
78     WORK_P_QBE0 = MODEL_CJE * MODEL_VJE / (1.0 - MODEL_MJE);
79     WORK_P_QBC0 = MODEL_CJC * MODEL_VJC / (1.0 - MODEL_MJC);
80     WORK_P_VJE0 = 2.0 * MODEL_VJE / MODEL_MJE;
81     WORK_P_VJC0 = 2.0 * MODEL_VJC / MODEL_MJC;
82
83     /* Define the array size */
84     WORK_P_SIZE = 5;
85   }
86   else if (JOB == BC_CHARACTERISTICS) {
87     /* Calculation of iec and qbc from vbc */
88     IEC = MODEL_IS * (exp(VBC / WORK_V_VT) - 1.0);
89     if (VBC < 0.0) {
90       QBC=WORK_V_QBC0 *
91       (1.0 - pow(1.0 - VBC/MODEL_VJC, 1.0 - MODEL_MJC));
92     }
93     else {
94       QBC = MODEL_CJC * VBC * (1.0 + VBC / WORK_V_VJC0);
95     }
96     if (MODEL_TYPE == _P) {
97       IEC = -IEC;
98       QBC = -QBC;
99     }
100  }
101  else if (JOB == BE_CHARACTERISTICS) {
102    /* Calculation of icc and qbe from vbc */
103    ICC = MODEL_IS * (exp(VBE / WORK_V_VT) - 1.0);
104    if (VBE < 0.0) {
105      QBE=WORK_V_QBE0 *
106      (1.0 - pow(1.0 - VBE/MODEL_VJE, 1.0 - MODEL_MJE));
107    }
108    else {
109      QBE = MODEL_CJE * VBE * (1.0 + VBE / WORK_V_VJE0);
110    }
111    if (MODEL_TYPE == _P) {
112      ICC = -ICC;
113      QBE = -QBE;
114    }
115  }
```


Copyright © 1985-2004 Synopsys, Inc.

```
116  else { /*If the bjt routine is called with an
117           unrecognized first argument, do the following*/
118    fprintf(stderr, "Bad job: %f\n", JOB);
119  }
120}
```

ASCII text of this example is located in:
install_home`/example/MASTtemplates/structured/bjt.c`

Copyright © 1985-2004 Synopsys, Inc.

Copyright © 1985-2004 Synopsys, Inc.

chapter 6

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

Copyright © 1985-2004 Synopsys, Inc.

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*)

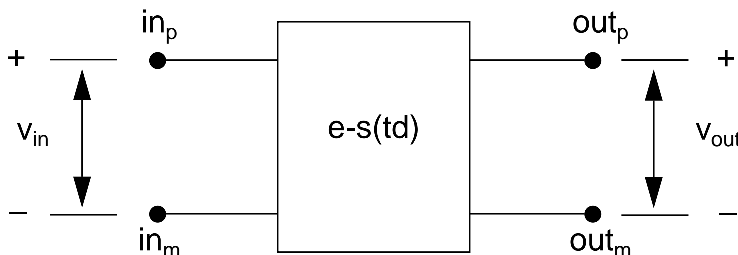
where *delayed_value* and *reference_value* are either a val variable, a branch variable, or a var variable; and *time* is a parameter that specifies the duration of the constant delay between *delayed_value* and *reference_value*. For example, suppose a template equation contains the following statement:

```
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

Copyright © 1985-2004 Synopsys, Inc.

Ideal Delay Line (dline) MAST Template

The **dline** template is shown as follows:

```
1  template dline inp inm outp outm = td, a
2    electrical inp, inm, outp, outm
3    number td = 0.0,     # ideal delay, with default
4            a = 1.0      # gain, with default
5  {
6    var i iout     # current from outp to outm
7    val v vout,    # voltage developed across outp and outm
8        vin,       # controlling voltage
9        vdl        # delayed voltage
10   values  {
11     vout = v(outp) - v(outm)
12     vin = v(inp) - v(inm)
13     vdl = vin * a
14   }
15   equations  {
16     i(outp->outm) += iout
17     iout: vout = delay(vdl, td)
18   }
19 }
```

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.

Copyright © 1985-2004 Synopsys, Inc.

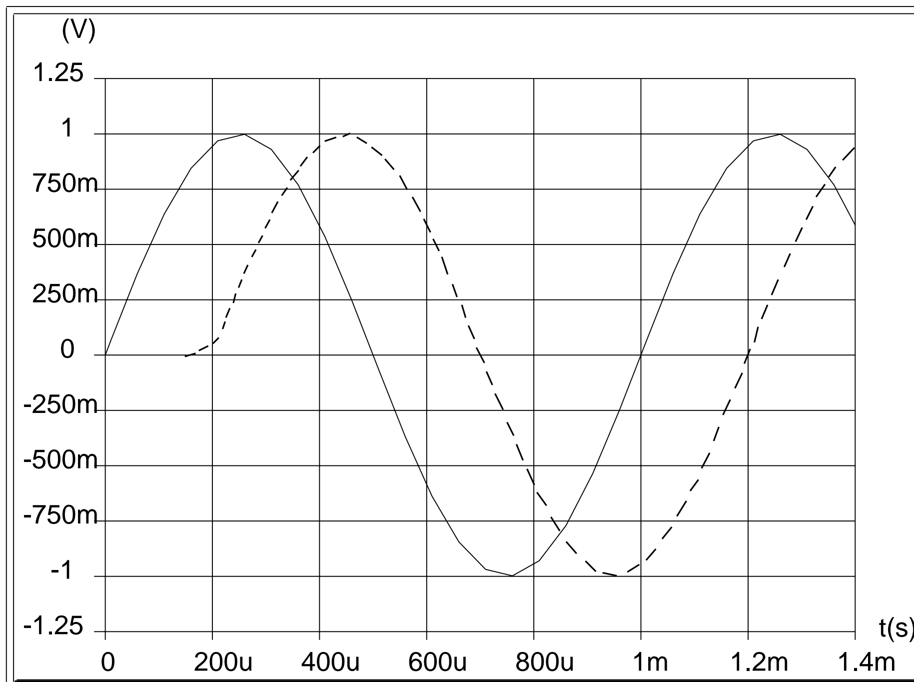
It is incorrect syntax to multiply the `delay` function by any quantity. For example, the following template equation would be incorrect:

```
# 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

Copyright © 1985-2004 Synopsys, Inc.

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 µs. 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

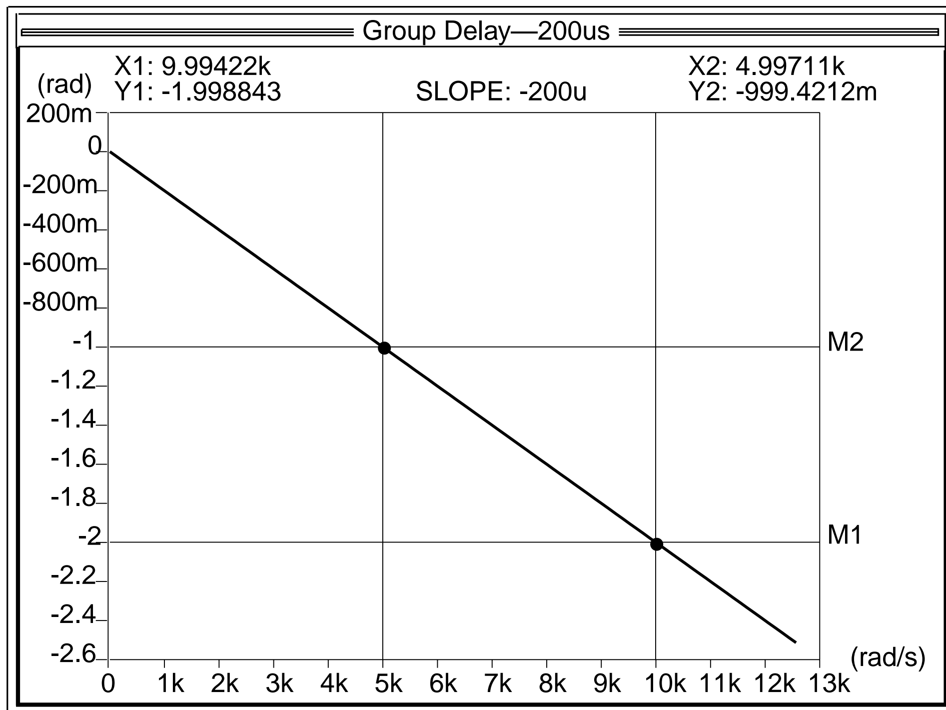
The **dline** template is also effective for a small-signal AC analysis. If the multiplier, `a`, is left at the default value of 1, **dline** has no effect on the magnitude of the input signal. However, the phase is shifted as a function of frequency.

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
```


Copyright © 1985-2004 Synopsys, Inc.

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 μs, 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

Copyright © 1985-2004 Synopsys, Inc.

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.

Copyright © 1985-2004 Synopsys, Inc.

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.

Copyright © 1985-2004 Synopsys, Inc.

The vsource_2 MAST Template

The template for this voltage source (**vsource_2**) is listed below.

```
1  element template vsource_2 p m  = supply, tran
2    electrical p, m
3    number supply = 0
4    union {
5      number                         off
6      struc {number vo, va, f, td;}  sin
7      struc {number v1,v2,tau;}      exp
8      struc {number v1,v2,tstep,tr;} step
9    } tran = (off = 1)
10 {
11   number pi = 3.14159
12   val v vn, vs
13   var i is
14   number td,vo,va,w,ss,v1,v2,tau,tstep,tr,slew
15   parameters {
16    # define intermediate values, depends on selected output
17     if (union_type (tran,sin))  {
18       td = tran->sin->td
19       vo = tran->sin->vo
20       va = tran->sin->va
21       w  = 2*pi*tran->sin->f
22       ss = 0.05/tran->sin->f
23     }
24     else if (union_type (tran,exp)) {
25       v1  = tran->exp->v1
26       v2  = tran->exp->v2
27       tau = tran->exp->tau
28     }
```


Copyright © 1985-2004 Synopsys, Inc.

```
29     else if (union_type (tran,step)) {
30       tstep = tran->step->tstep
31       v1    = tran->step->v1
32       v2    = tran->step->v2
33       tr    = tran->step->tr
34       slew  = (v2-v1)/tr
35     }
36   }                           # end parameters section
37   values {
38     vn = v(p) - v(m)
39     if (dc_domain|time_domain) {
40       if (union_type (tran,sin))  {
41         if (time <= td) {
42           vs = vo
43           next_time = td
44         }
45         else {
46           vs = vo + va*sin(w*(time-td))
47           step_size = ss
48         }
49       }                                # end tran->sin
50       else if (union_type (tran,exp)) {
51         vs = v1 + (v2-v1)*(1-exp(-(time/tau)))
52       }                                # end tran->exp
```


Copyright © 1985-2004 Synopsys, Inc.

```
53        else if (union_type (tran,step)) {
54          if (dc_domain|(time < tstep)) {
55            vs = v1
56            next_time = tstep
57          }
58          else if ((time >= tstep) & (time < tstep+tr)){
59            vs = v1 + (time-tstep)*slew
60            next_time = tstep + tr
61          }
62          else {
63            vs = v2
64          }
65        }
66        else vs = supply
67      }                       # end dc_domain|time_domain
68      else vs = 0
69    }                         # end values section
70    equations {
71      i(p) += is
72      i(m) -= is
73      is : vn = vs
74    }                         # end equations section
75  }
```

ASCII text of this example is located in:
install_home`/example/MASTtemplates/structured/vsource_2.sin`

Copyright © 1985-2004 Synopsys, Inc.

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`.

```
2   electrical p, m
3   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*]**)**]

Copyright © 1985-2004 Synopsys, Inc.

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,

```
4   union {
5     number                        off
6     struc {number vo, va, f, td;}  sin
7     struc {number v1,v2,tau;}      exp
8     struc {number v1,v2,tstep,tr;} step
9   } 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.

Copyright © 1985-2004 Synopsys, Inc.

Sine Wave Output (sin) Declaration

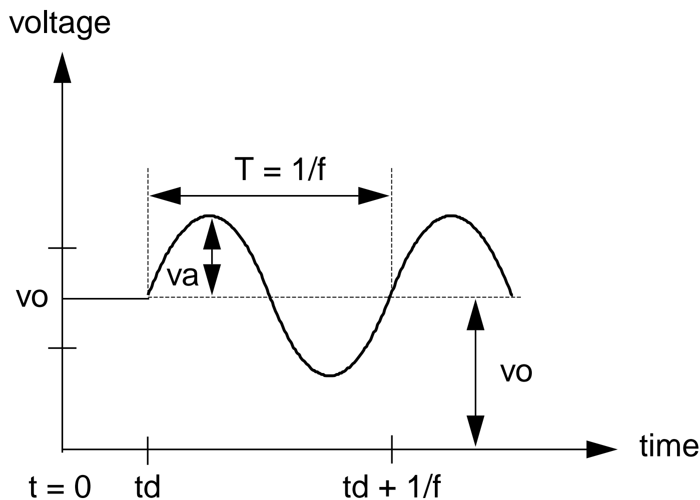
The equation for defining a sine wave, as shown in the following figure, is:

$$v = vo + va * \sin(2\pi f\,(time - td)) \qquad (1)$$

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
```


Copyright © 1985-2004 Synopsys, Inc.

Exponential Wave Output (exp) Declaration

The exponential signal, as shown in the following figure, is defined with the following equation:

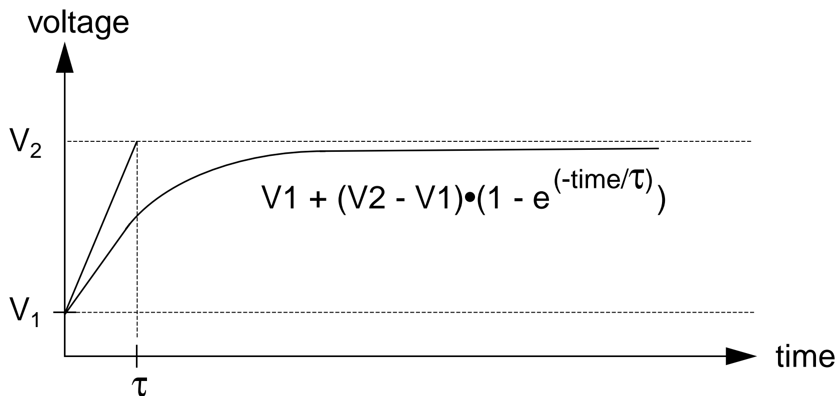
$$v = v1 + (v2 - v1) * (1 - e^{-(time/tau)}) \quad (2)$$

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
```

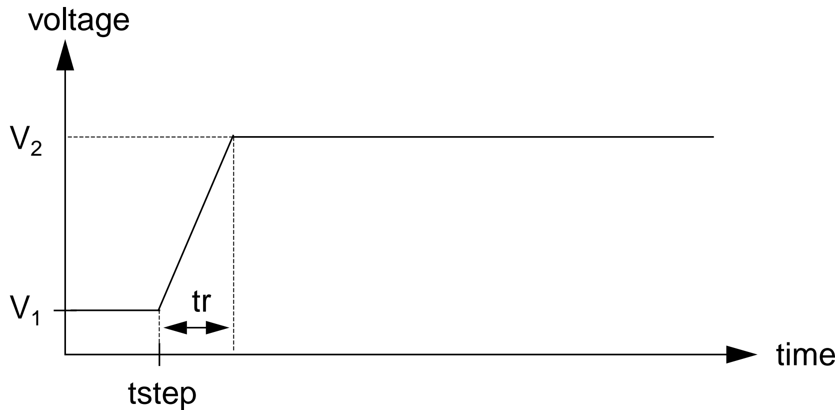


Describing an exponential waveform

Copyright © 1985-2004 Synopsys, Inc.

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.

Copyright © 1985-2004 Synopsys, Inc.

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:

```
3   number supply = 0
4   union {
5     number off
6     struc {number vo, va, f, td;}      sin
7     struc {number v1, v2, tau;}        exp
8     struc {number v1, v2, tstep, tr;}  step
9   } 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:

- 0V offset (`vo=0`)
- 4.3V amplitude (`va=4.3`)
- 1kHz frequency (`f=1k`)
- 0s 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.

Copyright © 1985-2004 Synopsys, Inc.

Local Declarations

The following declarations are required for use throughout the template:

- `pi`, the name used to represent the number π, 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:

```
11   number pi = 3.14159
12   val v vn, vs
13   var i is
14   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.

```
70   equations {           ### FROM vsource_2
71     i(p) += is
72     i(m) -= is
73     is : vn = vs
74   }                     # end equations section
```

```
37   equations {          ### FROM vsource_1
38    i(p->m) += is
39    is: v(p)-v(m)=vs # determine current contributed
40                     #  by source
41   }                 # end of equations section
```


Copyright © 1985-2004 Synopsys, Inc.

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***)**

Copyright © 1985-2004 Synopsys, Inc.

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.

```
17   if (union_type (tran,sin)) {
18     td  = tran->sin->td
19     vo  = tran->sin->vo
20     va  = tran->sin->va
21     w   = 2*pi*tran->sin->f
22     ss  = 0.05/tran->sin->f
23   }
24   else if (union_type (tran,exp)) {
25     v1  = tran->exp->v1
26     v2  = tran->exp->v2
27     tau = tran->exp->tau
28   }
29   else if (union_type (tran,step)) {
30     tstep = tran->step->tstep
31     v1    = tran->step->v1
32     v2    = tran->step->v2
33     tr    = tran->step->tr
34     slew  = (v2-v1)/tr
35   }
```

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 (`->`).

Copyright © 1985-2004 Synopsys, Inc.

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:

```
39   if (dc_domain | time_domain) {
40     if (union_type (tran, sin)) {
      # define vs as a sine wave}
50     else if (union_type (tran, exp)) {
      # define vs as an exponential wave}
53     else if (union_type (tran, step)) {
      # define vs as step function}
65   }

66   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.

Copyright © 1985-2004 Synopsys, Inc.

Sine Wave Output

The sine wave voltage is defined with respect to time as:

v = vo + va * sin(2*pi*f*(time - td)) (3)

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:

```
41   if (time <= td) {
41     vs = vo
```

The complete sine wave definition then becomes:

```
40   if (union_type (tran,sin)) {
41     if (time <= td) {
42       vs = vo
43       next_time = td
44     }
45     else {     # if (time > td)
46       vs = vo + va*sin(w*(time-td))
47       step_size = ss
48     }
49   }
```

Also, notice the use of the assignment to the `step_size` variable as follows:

```
47   step_size = ss
```


Copyright © 1985-2004 Synopsys, Inc.

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:

```
22   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:

$$v = v1 + (v2 - v1) * 1 - e^{-(time/tau)} \quad (4)$$

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:

```
50   else if (union_type (tran,exp)) {
51     vs = v1 + (v2-v1)*(1-exp(-(time/tau)))
52   }
```

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.

Copyright © 1985-2004 Synopsys, Inc.

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:

```
53   else if (union_type (tran,step)) {
54     if (dc_domain|(time < tstep)) {
55       vs = v1
56       next_time = tstep
57     }
58     else if ((time >= tstep) & (time < tstep+tr)){
59       vs = v1 + (time-tstep)*slew
60       next_time = tstep + tr
61     }
62     else {
63     vs = v2
64     }
65   }
```

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.

Copyright © 1985-2004 Synopsys, Inc.

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:

```
39   if (dc_domain|time_domain) {
40     if (union_type (tran,sin)) {
41       if (time <= td) {
42         vs = vo
43         next_time = td
44       }
45       else {     # if (time > td)
46         vs = vo + va*sin(w*(time-td))
47         step_size = ss
48       }
49     }
50     else if (union_type (tran,exp)) {
51       vs = v1 + (v2-v1)*(1-exp(-(time/tau)))
52     }
```


Copyright © 1985-2004 Synopsys, Inc.

```
53      else if (union_type (tran,step)) {
54        if (dc_domain|(time < tstep)) {
55          vs = v1
56          next_time = tstep
57        }
58        else if ((time >= tstep) & (time < tstep+tr)){
59          vs = v1 + (time-tstep)*slew
60          next_time = tstep + tr
61        }
62        else {
63          vs = v2
64        }
65      }
66      else vs = supply
67    }
68    else vs = 0
```


Copyright © 1985-2004 Synopsys, Inc.

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))
```


Copyright © 1985-2004 Synopsys, Inc.

Copyright © 1985-2004 Synopsys, Inc.

chapter 7

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).

Copyright © 1985-2004 Synopsys, Inc.

- 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).

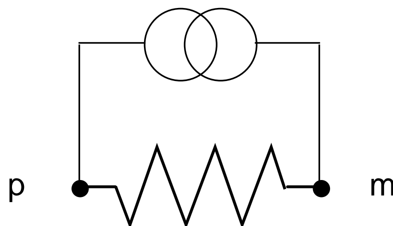
If there is more than one noise source, the control section must contain a separate variable, definition, and statement for each.

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

Copyright © 1985-2004 Synopsys, Inc.

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

Copyright © 1985-2004 Synopsys, Inc.

resistor_2 Template Topics

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 $A/\sqrt{Hz}$. 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
```


Copyright © 1985-2004 Synopsys, Inc.

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]*`)`

Copyright © 1985-2004 Synopsys, Inc.

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.

Copyright © 1985-2004 Synopsys, Inc.

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).

Copyright © 1985-2004 Synopsys, Inc.

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.

Copyright © 1985-2004 Synopsys, Inc.

chapter **8**

Statistical Modeling

Statistical modeling (also known as a Monte Carlo analysis) includes the features that are described in the following topics:

- Varying Values in a Simple Voltage Divider
- Probability Density Functions (PDFs)
- Cumulative Density Functions (CDFs)
- Correlating Distributions
- Modifying Uniform and Normal Default Distributions
- Parameterized PDF and CDF Specifications
- The random MAST Function
- Use of the statistical MAST Simvar Variable
- Worst-Case Statistical MAST Modeling

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.

Copyright © 1985-2004 Synopsys, Inc.

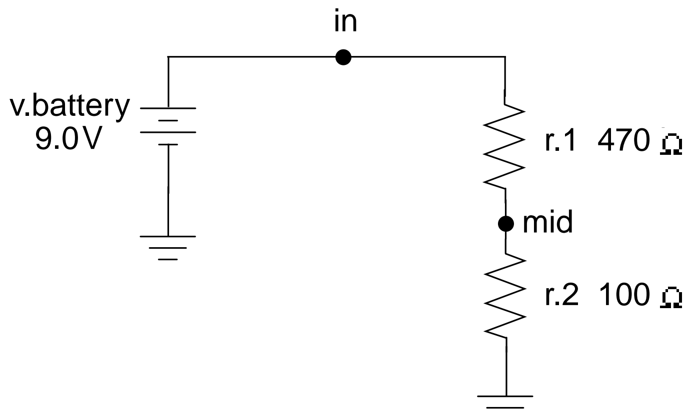
Introduction

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
```


Copyright © 1985-2004 Synopsys, Inc.

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.

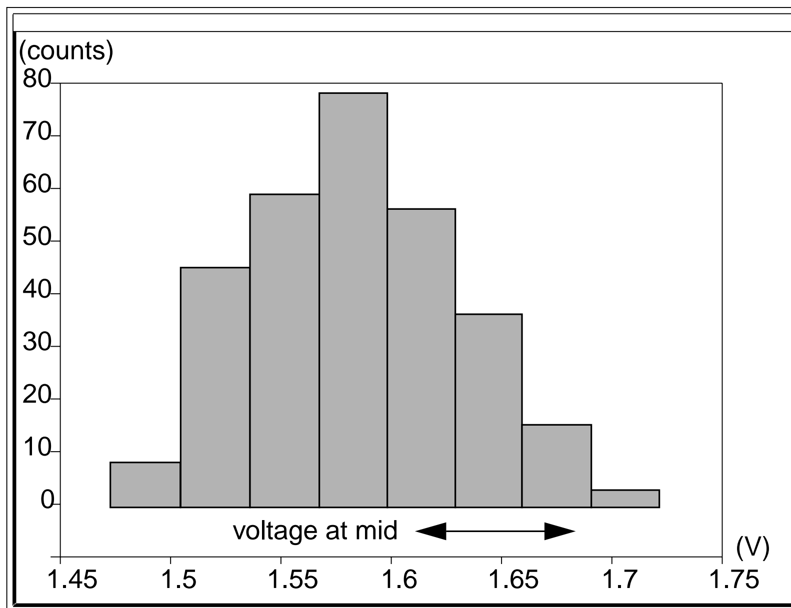
A simple DC analysis using this circuit produces the result that the voltage at `mid` is 1.5789 volts. To add the variations to this model, include the `uniform` and `normal` intrinsic distribution functions in the netlist:

```
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

Copyright © 1985-2004 Synopsys, Inc.

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)

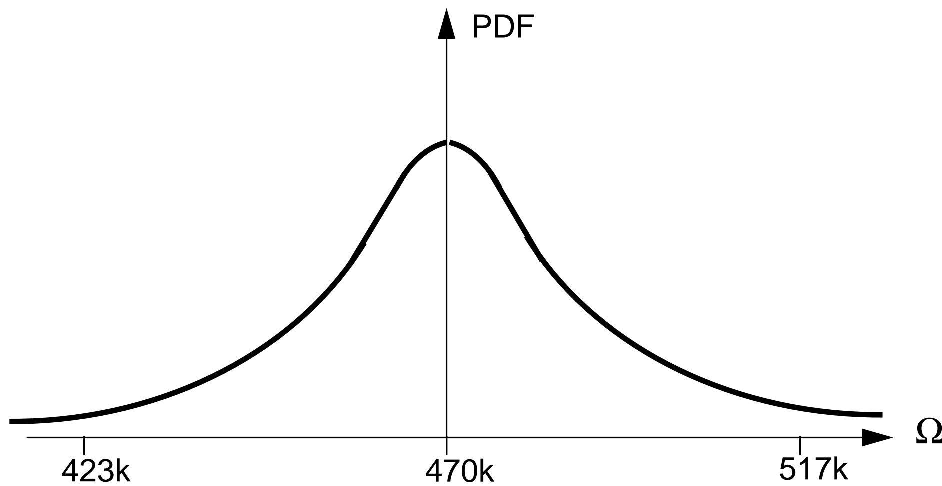
The primary tool for describing model parameter variations is the probability density function (PDF). Among the types of PDFs are the following:

- 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`

Copyright © 1985-2004 Synopsys, Inc.

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:

1. Begin with the basic, normalized, distribution, called the *prototype distribution* (MAST provides three intrinsic functions: `normal`, `uniform`, and `pwl`).

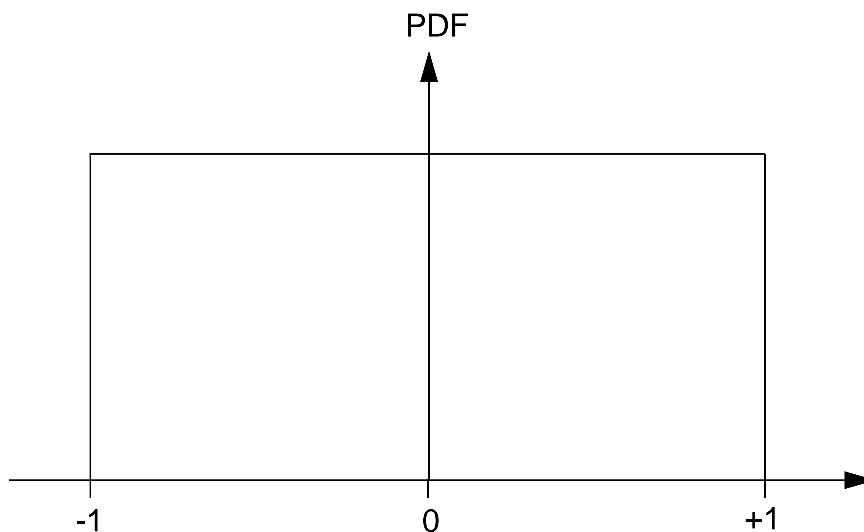
Copyright © 1985-2004 Synopsys, Inc.

2. 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

Copyright © 1985-2004 Synopsys, Inc.

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.

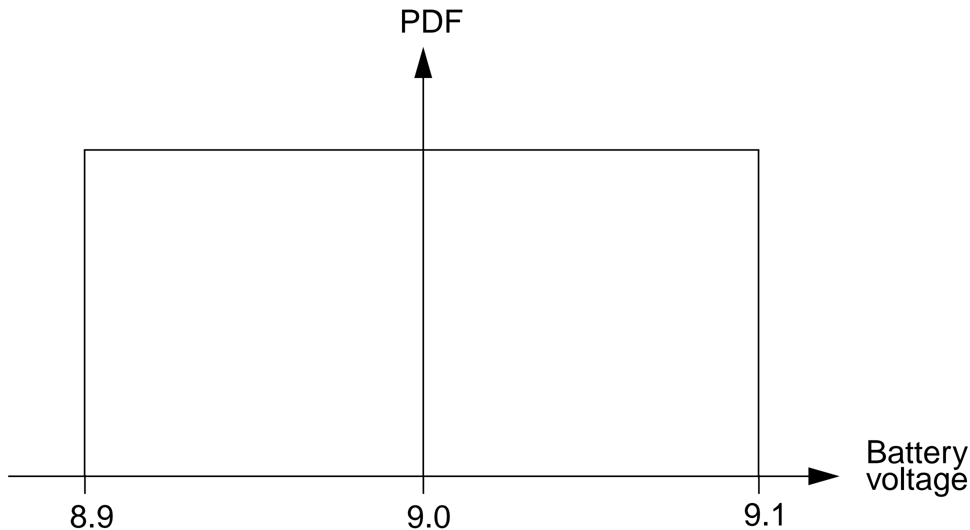
Note that these correspondences are maintained even if the prototype distribution is changed.

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)
```


Copyright © 1985-2004 Synopsys, Inc.

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".

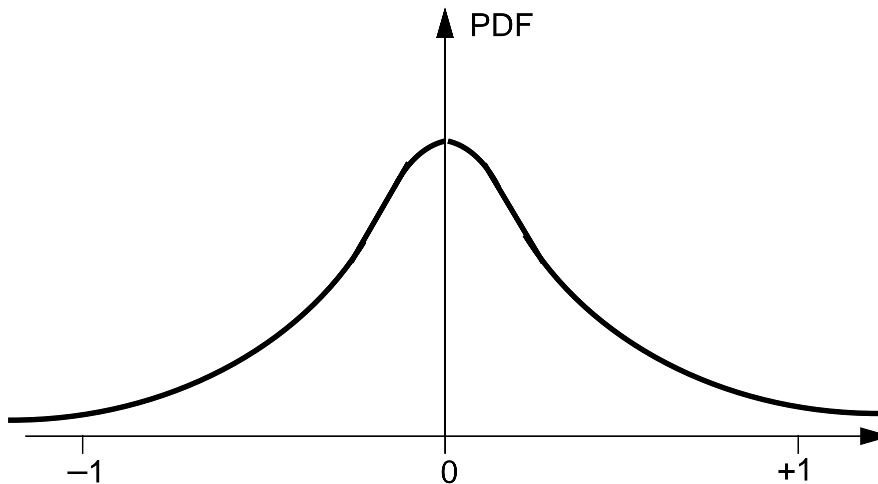
Normal Probability Density Function

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 3σ and -3σ points, respectively, where σ is the normal distribution's standard deviation.

Copyright © 1985-2004 Synopsys, Inc.

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 -3σ value (when using the default) of the distribution, where σ is the standard deviation of the normal distribution. You may change the multiplier of s from its default of -3.
upper_limit	The $+3\sigma$ value of the distribution, where σ is the standard deviation of the normal distribution. You may change the multiplier of σ 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 3σ limit.

The second form of the normal distribution shown above applies if the normal distribution is symmetrical with respect to *nominal_value*.

Copyright © 1985-2004 Synopsys, Inc.

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* – 3σ, and *nominal_value* + 3σ, where the two 3σ 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 -3σ.
- 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:

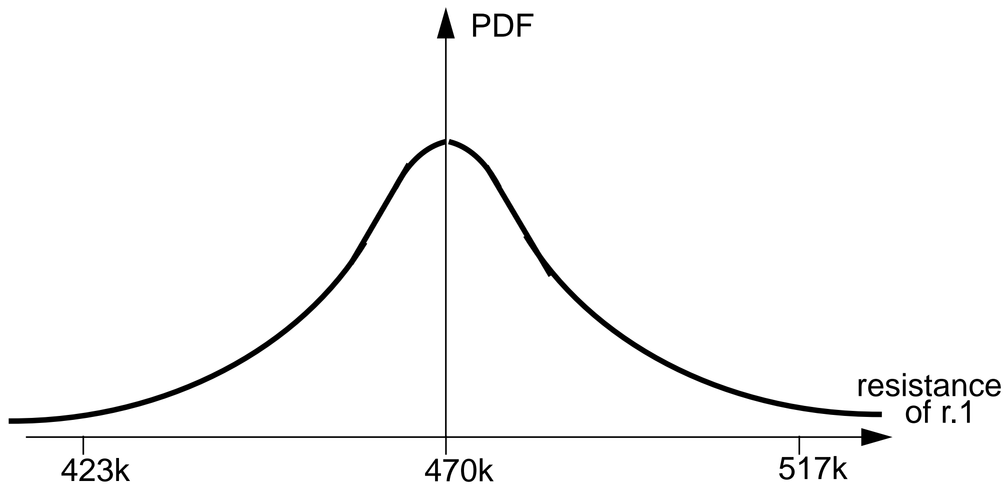
```
r.1 in mid = normal(470k, 0.1)
```

This means that the resistance values for resistor `r.1` follow the normal distribution, with a nominal value of 470 kΩ and a tolerance of 0.1. Thus, the -3σ limit is $470k\Omega - (470k\Omega \bullet 0.1) = 423k\Omega$, and the $+3\sigma$ limit is $470k\Omega + (470k\Omega \bullet 0.1) = 517k\Omega$. Because the 3σ value is $470k\Omega \bullet 0.1$ or $47k\Omega$, then the value of one standard deviation (σ) would be $47k\Omega/3$ or $15.67k\Omega$.

The figure below shows the actual value normal distribution. The tolerance specifies the symmetrical $\pm 3\sigma$ limits. Using 3σ as the tolerance means that,

Copyright © 1985-2004 Synopsys, Inc.

given a randomly selected resistor from the batch, the probability that its resistance lies inside the tolerance (i.e., between $\pm 3\sigma$) 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.

Copyright © 1985-2004 Synopsys, Inc.

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.

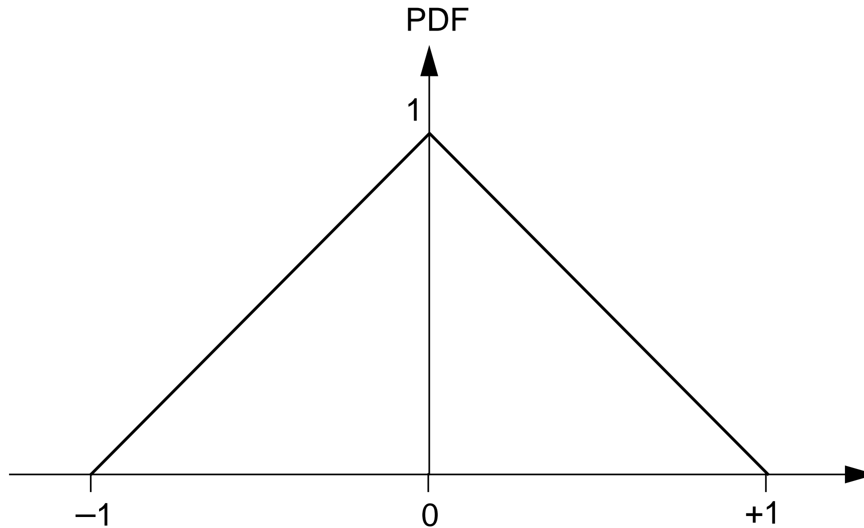
Using the Standard Template

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.

Copyright © 1985-2004 Synopsys, Inc.

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 ≥ 0.

Copyright © 1985-2004 Synopsys, Inc.

- The first x value (x_1) must be <0, and the last (x_n) must be > 0. (The simulator uses these values as truncation bounds when assigning random values to the distribution.)
- The integral of the PDF, from x_1 to x_n, 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.

Copyright © 1985-2004 Synopsys, Inc.

- 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 Statisical 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):

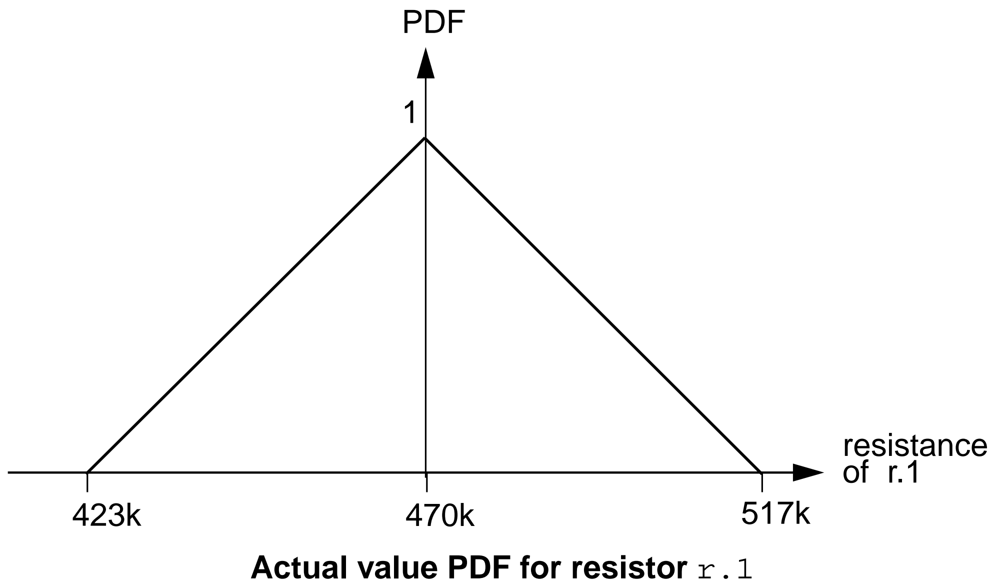
```
# creates prototype pwl PDF
standard..p_pwl ppwl1=(type=_pdf,pwl=[(-1,0),(0,1),(1,0)])

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.

Copyright © 1985-2004 Synopsys, Inc.



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`).

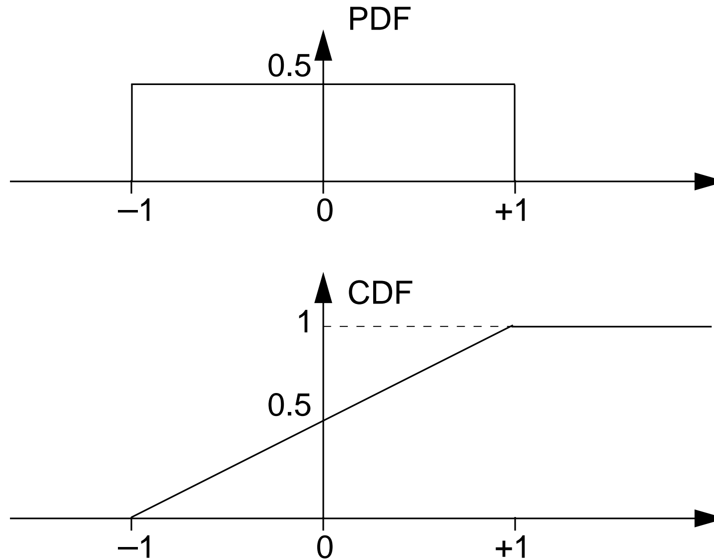
Note that in each example either the tolerance or both upper and lower limits must be `undef`. It is an error to specify *numeric* values for all three, even though all three must have values. The `pwl` intrinsic function, unlike the `uniform` and `normal` functions, cannot infer the tolerance or the 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

Copyright © 1985-2004 Synopsys, Inc.

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

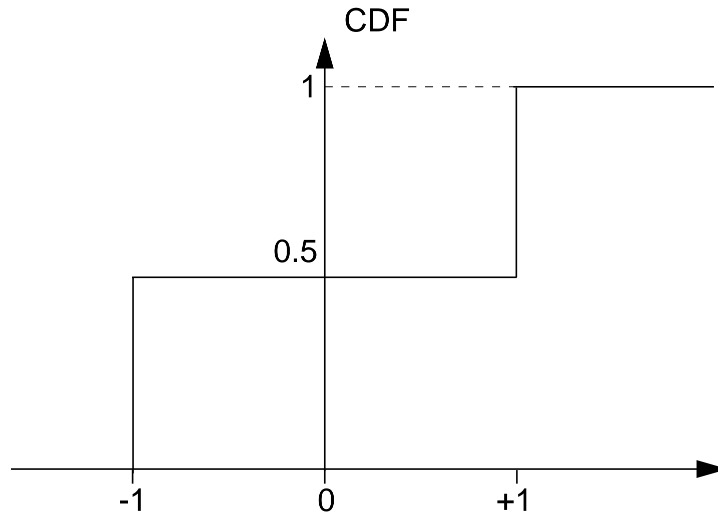
The CDF, like the PDF, is a function of x, where x ranges from −*infinity* to *infinity*. The value of a CDF at x is the integral of the PDF, evaluated between −*infinity* and x.

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.

Copyright © 1985-2004 Synopsys, Inc.

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:

1. Create a prototype CDF.
2. Map actual values to the prototype CDF.
3. 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".

Copyright © 1985-2004 Synopsys, Inc.

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`:

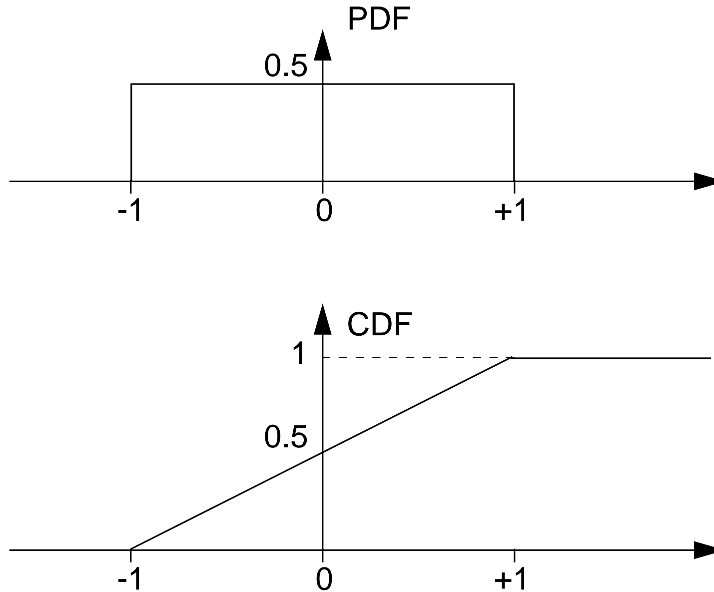
```
<distrib.sin
```

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.

Copyright © 1985-2004 Synopsys, Inc.

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 ≥ 0.

Copyright © 1985-2004 Synopsys, Inc.

- The first x value (x_1) must be < 0, and the last (x_n) 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.

Copyright © 1985-2004 Synopsys, Inc.

- 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)
```

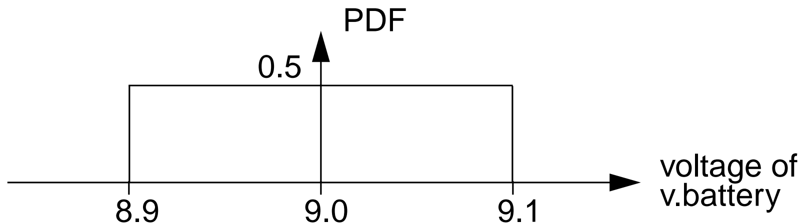
This method specifies the actual value CDF by giving the nominal value (`9`), the tolerance (`0.01111`), the lower and upper bounds (both `undef`), and the name of the prototype CDF (`cpwl1`).

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

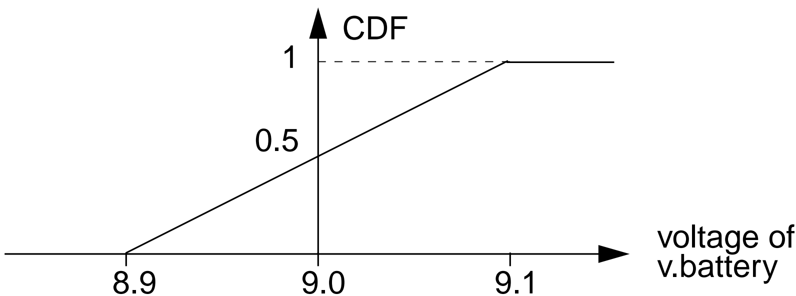
Copyright © 1985-2004 Synopsys, Inc.

`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

Copyright © 1985-2004 Synopsys, Inc.

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

Copyright © 1985-2004 Synopsys, Inc.

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 titled "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

Copyright © 1985-2004 Synopsys, Inc.

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:

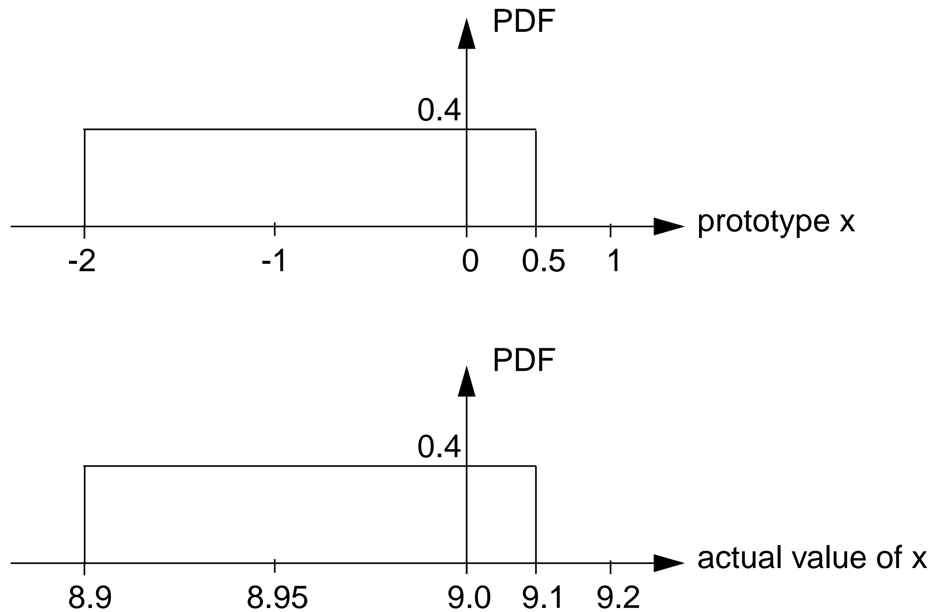
nominal_value	the value that the distribution is to have in a deterministic (non-statistical) environment.
tolerance	either `undef` or a numeric value between -1 and 1—if it is a numeric value, then the upper and lower limits of the distribution are *nominal_value* + *tolerance** *nominal_value* and *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

Copyright © 1985-2004 Synopsys, Inc.

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)
```


Copyright © 1985-2004 Synopsys, Inc.

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

Copyright © 1985-2004 Synopsys, Inc.

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:

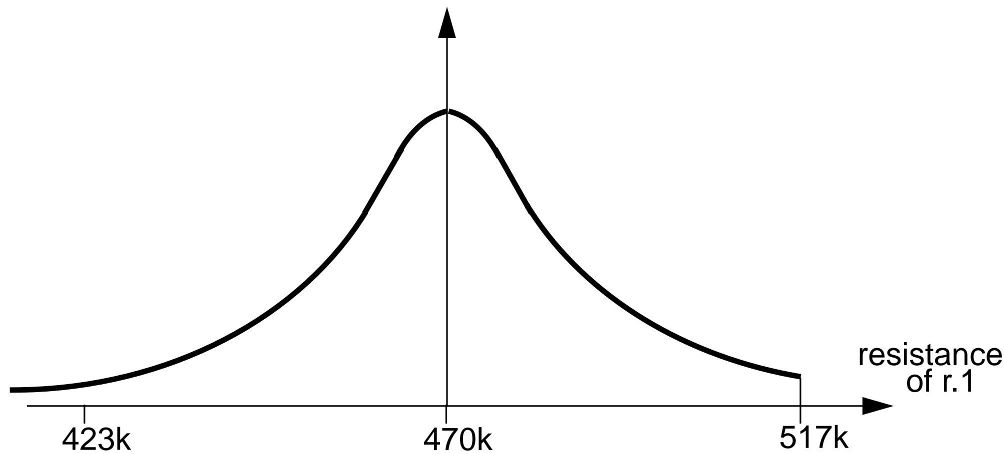
```
standard..p_normal pnorm = (std_dev=1/4, max=1)
r.1 in mid = normal(470k, 0.1, undef, undef,
                    pnorm)
```

The above example produces a distribution somewhat like that for `r.1` in the original example (shown below):

```
r.1 in mid = 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 *upper_limit* of the modified distribution was specified as `max=1`, the upper end is truncated at 517k. The following figure illustrates this modified distribution.

Copyright © 1985-2004 Synopsys, Inc.



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 (`undef`), upper limit (`undef`), 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`.

Copyright © 1985-2004 Synopsys, Inc.

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->max = 1
r.1 in mid = 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.

Copyright © 1985-2004 Synopsys, Inc.

- 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. For information about seeding, refer to the description of the `mc` command in SaberBook.

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.

Copyright © 1985-2004 Synopsys, Inc.

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
}
```


Copyright © 1985-2004 Synopsys, Inc.

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.

Copyright © 1985-2004 Synopsys, Inc.

chapter 9

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

Add the required `stress_measure` statements to the control section of the template as follows:

- ❑ Place a `stress_measure` statement for each stress measure you want to implement in the control section of your template.

Copyright © 1985-2004 Synopsys, Inc.

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.

Copyright © 1985-2004 Synopsys, Inc.

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. For additional details, see the topic titled Modeling Thermal Networks in Electrical Circuits in the SaberBook online documentation system.

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

Copyright © 1985-2004 Synopsys, Inc.

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 Book 1 of the *Guide to Writing MAST Templates; Variables and Arguments*.

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.

- ❑ 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=()
```


Copyright © 1985-2004 Synopsys, Inc.

- 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
```

- 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.
```

- 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")
```


Copyright © 1985-2004 Synopsys, Inc.

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.

- ❑ 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).
```


Copyright © 1985-2004 Synopsys, Inc.

- ❑ 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.
```

- ❑ 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`.

Copyright © 1985-2004 Synopsys, Inc.

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.

- ❑ 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.

Copyright © 1985-2004 Synopsys, Inc.

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.

❑ 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.

❑ 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.

Copyright © 1985-2004 Synopsys, Inc.

- ❑ 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.

Copyright © 1985-2004 Synopsys, Inc.

MAST Example Including Stress Statements

A complete listing of the example resistor template is shown below. Stress-related statements are shown in bold.

```
1  #**************************************************************
2  # Constant resistor (called by: r )
3  # Zero value is not allowed and will generate an error message.
4  # Geometric description is allowed.
5  #**************************************************************
6
7  #**************************************************************
8  # This template created by Analogy, Inc. for exclusive use with
9  # the Saber simulator.
10 # Copyright 1987,1988,1989,1993 Analogy, Inc.
11 # This template may not be reproduced in any way (physically or
12 # electronically) without permission from Analogy, Inc.
13 # The content of this template is subject to change without
14 # notice. Analogy does not assume liability for the use of this
15 # template or the results obtained from using it.
16 #*************************************************************
17
18 element template r  p m = rnom,tc,tnom,nons,model,l,w,
19                           ratings,rth_ja,rth_jc,
20                           rth_hs,part_type,part_class
21
22   #...declaration of connections:
23   electrical  p,m
24   process..imodel model = ()# Process model for resistor.
25
```


Copyright © 1985-2004 Synopsys, Inc.

```
26   #...declaration of arguments (tnom is in degrees celsius)
27
28   number rnom=undef,   # Nominal resistance.
29          tnom=27,      # Nominal temperature.
30          tc[2]=[0,0],  # Temperature coefficients.
31          nons=0.0,     # Resistor will be noiseless if
32                        # non-zero value.
33          l=0.0,        # Optional length of resistor.
34          w=0.0,        # Optional width of resistor.
35          rth_ja=undef, # Junction-Ambient Thermal
36                        # resistance (deg C/W)
37          rth_jc=undef, # Junction-Case Thermal
38                        # resistance (deg C/W)
39          rth_hs=undef  # Heatsink Thermal
40                        # resistance (deg C/W)
41   #...Bring in external numbers
42   external number temp, include_stress, r_tol, r_pdmax
43   external standard..pdist pdist
44
45   struc {
46     number pdmax_ja= undef,# Max. power diss. w/out heatsink
47     pdmax_jc=undef,        # Max. power diss. w/ heatsink
48     tjmax=undef,           # Max. temperature
49     tjmin=undef,           # Min. temperature
50     vmax=undef             # Max. voltage
51   } ratings=()
52
53   string part_type="resistor",# type of the device
54          part_class="generic" # class of the device
55
```


Copyright © 1985-2004 Synopsys, Inc.

```
56   export val tc tempj  # instantaneous junction temperature
57   export val p  pwrd   # instantaneous power dissipation
58   export val i  i      # instantaneous current
59
60 #++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
61   # Start the definition
62 {
63   #...Quantities useful for output:
64
65   val v v
66   val p power
67   val ni nsr
68   val tc temp_case
69   val rth rth_hs_tjmax
70
71   #...Define a group for extraction purposes
72   group {nsr}        noise
73   group {power,pwrd} pwr
74   #...Define quantities used later
75
76   number r,        # Final resistance.
77          g,        # Final Conductance.
78          nx,       # Noise flag.
79          xl,       # Effective resister length.
80          xw,       # Effective resister width.
81          dl,       # Final value of geometry reduction.
82          rth_eff, # Final value of thermal resistance.
83          pdmax,   #Final value of power dissipation rating.
84          xvmax,   # Final value of max. voltage rating.
85          xtjmax,  # Final value of max temp rating.
86          xtjmin   # Final value of min temp rating
87   number  tempj_tnom=25  # Ref_rating for temp stress.
88
89   #...Bring in mathematical constants
90 <consts.sin
```


Copyright © 1985-2004 Synopsys, Inc.

```
91 #+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
92
93   parameters {
94     #...Check input parameters.
95     if ( (rnom == undef) & ((model->rsh == 0)|(l == 0)|
96      ((w == 0)&(model->wdf == 0))) ) {
97       # Resistance is not specified.
98       saber_message("TMPL_S_ALT_SPEC",  instance(),
99         "resistance","rnom","model->rsh, l, and w")
100    }
101
102    #...Include temperature effects
103    if (rnom ~= undef) {
104      #...Resistor specification.
105      if (rnom == inf) {
106       r = inf
107      }
108      else {
109       #...Call function to apply distribution
110       #...to resistor value
111       r = distfunc(rnom,r_tol,pdist)
112       r = r*(1 + tc[1]*(temp-tnom) + tc[2]*((temp-tnom)**2))
113      }
114    }
```


Copyright © 1985-2004 Synopsys, Inc.

```
115    else {
116      #...Process specification.
117      #...Check input parameters.
118      if ((model->dl == undef)|(model->dl < 0)) {
119       dl = 0
120      }
121      else {
122       dl = model->dl
123      }
124      if ( ((w == 0)|(w == undef)) &
125       ((model->wdf == 0)|(model->wdf == undef)) ) {
126        saber_message("TMPL_S_ALT_SPEC",  instance(),
127         "resistor width","w","model->wdf")
128      }
129      #...Take into account the geometry
130      #...reduction of the length.
131      xl = l - dl
132
133      if (xl <= 0) {
134        saber_message("TMPL_S_POS",instance(),
135         "effective resistor length")
136      }
137
138      #...Take into account the geometry
139      #...reduction of the width.
140      if ((w > 0)&(w ~= undef)) {
141        xw = w - dl
142      }
143      else {
144        xw = model->wdf
145      }
```


Copyright © 1985-2004 Synopsys, Inc.

```
146      #...Calculate the resistance from
147      #...the sheet resistance.
148      if (xw > 0) {
149        #...Call function to apply distribution
150        #...to resistor value
151        r = distfunc(model->rsh*(xl/xw),r_tol,pdist)
152        r = r*(1 + tc[1]*(temp-tnom) + tc[2]*((temp-tnom)**2))
153      }
154      else {
155        saber_message("TMPL_S_POS",instance(),
156         "effective resistor width")
157      }
158    }
159
160    #...Calculate conductance and print message if r=0
161    if (r == 0) {
162      saber_message("TMPL_S_RANGE_NE_0",instance(),
163       "resistance value")
164      g = 0
165    }
166    else if (r < 0) {
167      #...negative resistance
168      saber_message("TMPL_W_GE_REL_VALUE",instance(),
169       "resistance value","zero")
170      g = 1/r
171    }
```


Copyright © 1985-2004 Synopsys, Inc.

```
172    else if (r == inf) {
173      g = 0
174    }
175    else {
176      g = 1/r
177    }
178    #...Bulletproofing on ratings
179    (rth_eff,pdmax) = thermpar(rth_ja,rth_jc,rth_hs,
180      if(ratings->pdmax_ja==undef) then r_pdmax
181       else ratings->pdmax_ja,
182      ratings->pdmax_jc)
183
184    xvmax = ratingbp(ratings->vmax,"vmax")
185
186    xtjmax = ratings->tjmax
187    xtjmin = ratings->tjmin
188    if(xtjmin ~= undef & xtjmin ~= inf) {
189      if(xtjmax ~= undef & xtjmax~=inf & xtjmax < xtjmin) {
190        saber_message("TMPL_S_REL_VALUE",instance(),"tjmax",
191          "tjmin")
192        xtjmin = undef
193      }
194    }
195    #...Determine the noise "switch" multiplier
196    if ((nons == 0) & (r > 0)) {
197      nx = 1
198    }
199    else {
200      nx = 0
201    }
202  }
```


Copyright © 1985-2004 Synopsys, Inc.

```
203#+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
204  values {
205    #...Definition of output quantities.
206    v = v(p) - v(m)
207    i = g*v
208    # If r=0, i=0. The value for i is wrong but a
209    # message has been printed out indicating that r=0.
210
211    #...Calculate noise generator
212    if (freq_domain & nx) {
213      nsr = sqrt(abs(4.0*math_boltz*(temp + math_ctok)*g))
214    }
215    else {
216      nsr = 0.0
217    }
218    #...Determine power term for extraction
219
220    power = v*i
221    pwrd = power
222    tempj = temp + pwrd*rth_eff
223
224    if(rth_jc ~= undef & rth_jc ~= inf & rth_jc > 0) {
225      if (pwrd ~= 0) {
226        rth_hs_tjmax = (xtjmax - temp)/pwrd - rth_jc
227      }
228      else rth_hs_tjmax = inf
229    temp_case = tempj - pwrd*rth_jc
230    }
231  }
```


Copyright © 1985-2004 Synopsys, Inc.

```
232#++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
233  control_section {
234    #...device type and class
235    device_type(part_type,part_class)
236
237    #...Specify noise source
238    noise_source(nsr,p,m)
239
240    #...Specify the stress measures
241    if(include_stress) {
242      stress_measure(pdmax,power,"Max Power Diss.",
243        pwrd,winmax,pdmax)
244      stress_measure(pdavg,power,"Avg Power Diss.",
245        pwrd,average,pdmax)
246      stress_measure(tjmax,temperature,"Max Temperature",
247        tempj,winmax,xtjmax, tempj_tnom)
248      stress_measure(tjavg,temperature,"Avg Temperature",
249        tempj,average,xtjmax,tempj_tnom)
250      stress_measure(tjmin,temperature,"Min Temperature",
251        tempj,min,xtjmin,tempj_tnom)
252      stress_measure(vmax,voltage,"Max Voltage",
253        abs(v),max,xvmax)
254    }
255  }
256#++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
257  equations {
258    i(p->m) += i
259  }
260}
```


Copyright © 1985-2004 Synopsys, Inc.

Copyright © 1985-2004 Synopsys, Inc.

appendix A

Un-Structured Modeling Approach - Examples

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:

```
1  element template bjtm c b e = model, ic
2    electrical c,b,e
3    bjtm_arg..model model = () # use arguments from
4                               # "companion" template
5    number ic[2]=[undef,undef]
6    external number temp
7  {                            # begin template body
8    # use local parameters from "companion" template
9    bjtm_arg..work work
10
11   struc {
12     number bp,inc;
13   } nv[*] = [(0,.1),(2,0)]
14   electrical cp                   #...local node
15   branch ibe=i(b->e),  vbe=v(b,e)
16   branch ibc=i(b->cp), vbc=v(b,cp)
17   branch ice=i(cp->e), vce=v(cp,e)
18   val i iec,icc,iba,ico           #...declare variables
19   val q qbc,qbe
20   group {vbc,vbe} v               #...extraction groups
21   group {iba,ico} i
22   group {qbc,qbe} q
23

24   control_section {
25    #...If no collector resistance, collapse nodes c and cp
26     if(model->rc == 0) collapse(c,cp)
27
28     #...specification of sample points and newton steps
29     newton_step((vbc,vbe),nv)
30   }
31
```


Copyright © 1985-2004 Synopsys, Inc.

```
32   # calculate thermal voltage and 4 funct. of model param.
33        #... 1'st call to MAST function
34   work = bjtm_pars(model,temp)
35
36   #...calculation of currents and charges
37        #...2'nd call to MAST function
38     (iec,qbc,icc,qbe) = bjtm_values(model,work,vbc,vbe)
39
40
41   #...calculate base and collector currents for extraction
42   iba = iec/model->br + icc/model->bf
43   ico = icc - iec - iec/model->br
44
45   #...calculation of branch currents
46   ibe = iec/model->br + icc/model->bf + d_by_dt(qbe)
47   ibc = d_by_dt(qbc)
48   ice = icc - iec - iec/model->br
49
50   #...current through collector resistor if present
51   if(model->rc ~= 0) i(c->cp) = v(c,cp)/model->rc
52 }
```

The functions 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

Copyright © 1985-2004 Synopsys, Inc.

Ideal Delay Line - Unstructured dline Template

The ideal delay line template (**dline**) is shown below using the delay function and the unstructured modeling approach:

```
1  template dline inp inm outp outm = td, a
2    electrical inp, inm, outp, outm
3    number td=0.0,                               # time delay
4                a=1.0                            # gain
5
6  {
7    branch iout=i(outp->outm) # output current
8    branch vin=v(inp,inm), vout= v(outp,outm) # input & output
9                                              # voltages
10   val v vdl                                    # delayed voltage
11
12   vdl = vin*a
13   vout=delay(vdl, td)
14 }
```


Copyright © 1985-2004 Synopsys, Inc.

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:

```
1  element template  vsource_2 p m = supply, tran
2    electrical p, m
3    number supply = 0
4    union {
5      number                          off
6      struc {number vo, va, f, td;}   sin
7      struc {number v1,v2,tau;}       exp
8      struc {number v1,v2,tstep,tr;}  step
9    } tran = (off=1)
10 {
11   number pi = 3.14159
12   branch is=i(p->m), vn=v(p,m)
13   val v vs
14
15   # define intermediate values depending on selected output
16   if (union_type (tran,sin)) {
17     td = tran->sin->td
18     vo = tran->sin->vo
19     va = tran->sin->va
20     w  = 2*pi*tran->sin->f
21     ss = 0.05/tran->sin->f
22   }
23   else if (union_type (tran,exp)) {
24     v1  = tran->exp->v1
25     v2  = tran->exp->v2
26     tau = tran->exp->tau
27   }
```


Copyright © 1985-2004 Synopsys, Inc.

```
28   else if (union_type (tran,step)) {
29     tstep = tran->step->tstep
30     v1    = tran->step->v1
31     v2    = tran->step->v2
32     tr    = tran->step->tr
33     slew  = (v2-v1)/tr
34   }
35   # determine vs, which is set equal to vn in temp. equ.
36
37   if (dc_domain|time_domain) {
38     if (union_type (tran,sin)) {
39       if (time <= td) {
40         vs = vo
41         next_time = td
42       }
43       else {     # if (time > td)
44         vs = vo + va*sin(w*(time-td))
45         step_size = ss
46       }
47     }                                # end tran->sin
48     else if (union_type (tran,exp)) {
49       vs = v1 + (v2-v1)*(1-exp(-(time/tau)))
50     }                                # end tran->exp
51     else if (union_type (tran,step)) {
52       if (dc_domain|(time < tstep)) {
53         vs = v1
54         next_time = tstep
55       }
56       else if ((time >= tstep) & (time < tstep+tr)){
57         vs = v1 + (time-tstep)*slew
58         next_time = tstep + tr
59       }
60       else {
61         vs = v2
62       }
63     }                               # end tran->step
64     else vs = supply
65   }                                 # end dc_domain|time_domain
66
67   else vs = 0
68
69   # template equation...find branch voltage, vn
70   vn = vs
71 }
```


Copyright © 1985-2004 Synopsys, Inc.

Copyright © 1985-2004 Synopsys, Inc.

appendix B

Making User Templates Visible for Unix and NT

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

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

Copyright © 1985-2004 Synopsys, Inc.

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

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

Copyright © 1985-2004 Synopsys, Inc.

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

SaberSketch 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. For more information on precompiled files, refer to the *Guide to Writing MAST Templates* manual, the topic titled Predefined MAST Declarations.

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.

Copyright © 1985-2004 Synopsys, Inc.

Step 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.

Copyright © 1985-2004 Synopsys, Inc.

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.

Step 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 (SaberSketch, Design Architect, Artist, or ViewDraw) for details. If you are using the SaberSketch design editor use the following instructions.

Making Symbols Available in SaberSketch

To make symbols available in SaberSketch, 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.

SaberSketch 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 SaberSketch 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.

Copyright © 1985-2004 Synopsys, Inc.

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.

Step 1. 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 the topic titled Foreign Routines in MAST in Book 1 of the *Guide to Writing MAST Templates Manual*.

Copyright © 1985-2004 Synopsys, Inc.

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.

Step 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.

Copyright © 1985-2004 Synopsys, Inc.

Step 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).

Copyright © 1985-2004 Synopsys, Inc.

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

Step 1. Compile the subroutines using the appropriate compiler.

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

Step 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).

Step 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`.

Copyright © 1985-2004 Synopsys, Inc.

Step 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.

Step 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*

Copyright © 1985-2004 Synopsys, Inc.

Making User Templates Visible for NT

This topic describes the following:

How Applications Find Files

Making Symbols Available in SaberSketch

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

	SaberSketch	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.

Copyright © 1985-2004 Synopsys, Inc.

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 SaberSketch 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.*

   SaberSketch 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.

Copyright © 1985-2004 Synopsys, Inc.

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 SaberSketch

To make symbols available in SaberSketch, 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.

SaberSketch 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 SaberSketch 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.

Copyright © 1985-2004 Synopsys, Inc.

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 SaberSketch 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.

Step 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.

Copyright © 1985-2004 Synopsys, Inc.

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

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

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

Copyright © 1985-2004 Synopsys, Inc.

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

Copyright © 1985-2004 Synopsys, Inc.

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.

Copyright © 1985-2004 Synopsys, Inc.

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:

Step 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.

Step 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.

Step 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.

Copyright © 1985-2004 Synopsys, Inc.

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

Step 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.

Step 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.

Copyright © 1985-2004 Synopsys, Inc.

Copyright © 1985-2004 Synopsys, Inc.

Index

A

Add a Way to Disable Stress (Optional) 9-8
Add a Way to Specify Device Type and Class (Optional) 9-9
Add Stress Ratings 9-4
Add stress_measure Statements to Template 9-1
Add Thermal Resistances (Optional) 9-6
Adding Noise to a Resistor MAST Template 7-2
Adding Noise to a Voltage Source MAST Template 7-6
Adding Noise to the MAST diode Template 7-7
Assigning Internal Values 6-19

B

Basic Model Equations 2-19
Basic Versus Advanced Modeling iii
bjtm_arg Declaration Template 4-9

C

Calculated Values Function bjtm_values 4-13
Calling the Foreign Routines 5-11
Characteristic Equation 2-4
Characteristic Equations 1-5
Comparison and Summary of Linearization Techniques 3-6
Complete BJT C Routine 5-19
Control Section 2-32, 3-9, 7-5
Control Section—Newton Steps 1-9
Conventions v
Correlating Distributions 8-23
Cumulative Density Functions (CDFs) 8-16

D

Declaring and Calling the Routine From a Template 5-5
Default Sample Points 3-16
Defining Template Arguments 5-14
Delayed Sine Wave AC Analysis 6-5
Delayed Sine Wave Transient Analysis 6-4
Density of Sample Points 3-16
Determine if Specified Variables are Accessible 9-4
Determining Union Elements 6-19
device_type - MAST Small Signal Parameter Statement 2-9

E

Ebers-Moll MAST Model for the Bipolar Transistor 2-13
Equation and Values Sections 1-18
Equations Section 2-36, 3-8, 6-18
Expanding the Multiple-Output Voltage Source 6-7
Expression for Noise 7-5

F

First Call—Setting Up Return Parameters 5-15
Five Fields
- small_signal Statement 2-10
Four Fields
- small_signal Statement 2-10
Function Call Overview - bjtm MAST Template 4-7

Copyright © 1985-2004 Synopsys, Inc.

G

General Approach 3-12
General Foreign Function Call Syntax 5-10
Guidelines for Splitting a MAST Template into Separate Functions 4-2

H

Header and Header Declarations 1-5
Header Declarations 1-14, 2-4, 2-21, 6-12, 7-4
How the Applications Find Files B-1

I

Ideal Delay Line - Unstructured dline Template A-3
Ideal Delay Line (dline) MAST Template 6-3
Implementing a MAST Foreign Routine in C 5-13
Initial Conditions 2-8
Intermediate Calculations 2-29
Intrinsic Piecewise Linear Cumulative Density Function 8-18
Intrinsic Probability Density Functions 8-5
Introduction 5-2, 7-1, 8-2

J

Junction Capacitance 2-28

L

Local Declarations 6-18, 7-4
Local Parameters 2-24
Local Parameters Function bjtm_pars 4-10

M

Making Symbols Available in SaberSketch B-13
Making User Templates Visible for NT B-11
Making User Templates visible for Unix B-1
MAST dline Template Summary 6-6
MAST Example Including Stress Statements 9-11
Modeling a Simple Voltage Limiter with MAST 1-3
Modeling a Voltage Divider with MAST 1-11
Modeling a Voltage Squarer - MAST vsqr Template 3-7
Modeling an Ideal Diode with MAST 2-2
Modeling Nonlinear Devices with MAST 2-1
Modeling Temperature 2-5
Modeling the Bipolar Transistor Using Foreign Routines 5-6
Modeling the Bipolar Transistor Using MAST Functions 4-2
Modifying a Normal Prototype Distribution 8-28
Modifying a Uniform Prototype Distribution 8-25
Modifying the BJT Template to Use a Foreign Routine 5-8
Modifying Uniform and Normal Default Distributions 8-24
Multiple-Output Voltage Source - Unstructured vsource_2 Template A-4

N

Netlist Examples 6-27
Newton Step Parameters 1-16
Newton Steps 2-5
Nonlinear Elements 1-2
Normal Probability Density Function 8-8

Copyright © 1985-2004 Synopsys, Inc.

O

Overview 4-1, 6-8

P

Parameterized PDF and CDF Specifications 8-31
Parameters Section - MAST vdiv Template 1-15
Performing Calculations (Defining Signals) 6-21
Piecewise linear approximation (Method 2) 3-4
Piecewise Linear Evaluation (Method 3) 3-4
Piecewise Linear Probability Density Function 8-11
Preface i
Preparing to Write the MAST bjt Template 2-20
Probability Density Functions (PDFs) 8-4
Purpose of Newton Steps 1-8

R

Revision History v

S

Schematic Entry Versus Netlist iv
Second and Third Calls—Performing Calculations 5-17
Simulation Linearization Techniques 3-2
Simulation Techniques for Evaluating Nonlinearities 3-2
small_signal - MAST Small Signal Parameter Statement 2-10
Small-Signal Parameters 2-8
Specific Approach (voltage squarer) 3-13
Specifying Sample Points 3-14
Splitting Functionality Between a MAST Template and a Foreign Function 5-7
ss_partial - MAST Small Signal Parameter Statement 2-11
Starting Value 2-8

T

Taking the Slope (Method 1) 3-3
Template Equation 2-7
Template Header 3-8
The bjtm Template 4-5
The bjtm Template Architecture Using MAST Functions 4-3
The random MAST Function 8-32
The vsource_2 MAST Template 6-9
Thermal Voltage 2-28

U

Understanding Sample Points 3-10
Uniform Probability Density Function 8-6
Union Type Parameters 6-12
Use of the statistical MAST Simvar Variable 8-33
Using a FORTRAN Function in a MAST Template 5-2
Using a MAST Function Instead of a Foreign Routine 4-1
Using C or FORTRAN Routines Called by Templates B-6, B-15
Using Custom Models From Your Capture Tool B-5
Using MAST Functions - Unstructured bjtm Template A-1
Using Templates Written in MAST B-3, B-14
Using the MAST delay Function in an Ideal Delay Line 6-2
Using the Standard Template 8-12

V

Values and Equations Sections 1-5
Values Section 3-8
Varying Values in a Simple Voltage Divider 8-2

Copyright © 1985-2004 Synopsys, Inc.

W

What This Manual is About ii
What You Need to Know to Use This Manual i
Worst-Case Statistical MAST Modeling 8-34
Writing the FORTRAN Routine 5-3

Copyright © 1985-2004 Synopsys, Inc.

BOOKSHELF

OVERVIEW W-2004.12

Saber® Quick Start

Saber® User Guide

Saber® Examples User Guide

DESIGN - GENERAL W-2004.12

Saber® Sketch User Guide

Saber® Parts Gallery Reference Manual

Saber® Property Editor Reference Manual

DESIGN - SPECIFIC W-2004.12

Saber® Harness Quick Start

Saber® Harness User Guide

Saber® Sketch iQBus User Guide

SIMULATION W-2004.12

Saber® Simulator Command Reference Manual

Saber® Simulator Guide Reference Manual

Saber®HDL Command Reference Manual

Saber® Netlist Options Reference Manual

Saber® Simulator Real Time (RT) Interface User Guide

ANALYSIS W-2004.12

CosmosScope™ Reference Manual

CosmosScope™ MATLAB® Interface User Guide

Saber® Simulator Testify Quick Start

Saber® Simulator Testify User Guide

POINT TOOLS W-2004.12

CosmosScope™ Calculator Reference Manual

Saber® and CosmosScope™ Command Line Tool User Guide

Saber® and CosmosScope™ Draw Tool User Guide

Saber® and CosmosScope™ Macro Recorder User Guide

Saber® and CosmosScope™ Report Tool Reference Manual

Saber® and CosmosScope™ StateAMS Reference Manual

Saber® Design Browser Tool Reference Manual

Saber® Model Architect Tool User Guide

MODELING AND MODEL LIBRARIES W-2004.12

Saber® Library and Model User Guide

Saber® Managing Symbols and Models User Guide

Saber® MAST Language Reference Manual

Saber® MAST Language, Book 1, User Guide

Saber® MAST Language, Book 2, User Guide

AIM W-2004.12

Saber® and CosmosScope™ AIM User Guide

Saber® AIM Reference Manual

INTEGRATION WITH THIRD-PARTY PRODUCTS W-2004.12

Saber® Frameway for Cadence Design Framework II User Guide

Saber® Frameway for Mentor Graphics ePD User Guide

Saber® Frameway for Mentor Graphics Falcon Framework User Guide

Saber® Frameway Integrations Quick Start

Saber® Simulator Co-Simulation With ModelSim Quick Start

Saber® Simulator Co-Simulation With ModelSim User Guide

Saber® Simulator Co-Simulation With Verilog User Guide

Saber® Simulink Co-simulation Interface User Guide