Spectre® Circuit Simulator User Guide

Product Version 5.0
July 2002

© 2000 Cadence Design Systems, Inc. All rights reserved.
Printed in the United States of America.

Cadence Design Systems, Inc., 555 River Oaks Parkway, San Jose, CA 95134, USA

Trademarks: Trademarks and service marks of Cadence Design Systems, Inc. (Cadence) contained in this document are attributed to Cadence with the appropriate symbol. For queries regarding Cadence's trademarks, contact the corporate legal department at the address shown above or call 1-800-862-4522.

All other trademarks are the property of their respective holders.

Restricted Print Permission: This publication is protected by copyright and any unauthorized use of this publication may violate copyright, trademark, and other laws. Except as specified in this permission statement, this publication may not be copied, reproduced, modified, published, uploaded, posted, transmitted, or distributed in any way, without prior written permission from Cadence. This statement grants you permission to print one (1) hard copy of this publication subject to the following conditions:

1. The publication may be used solely for personal, informational, and noncommercial purposes;
2. The publication may not be modified in any way;
3. Any copy of the publication or portion thereof must include all original copyright, trademark, and other proprietary notices and this permission statement; and
4. Cadence reserves the right to revoke this authorization at any time, and any such use shall be discontinued immediately upon written notice from Cadence.

Disclaimer: Information in this publication is subject to change without notice and does not represent a commitment on the part of Cadence. The information contained herein is the proprietary and confidential information of Cadence or its licensors, and is supplied subject to, and may be used only by Cadence's customer in accordance with, a written agreement between Cadence and its customer. Except as may be explicitly set forth in such agreement, Cadence does not make, and expressly disclaims, any representations or warranties as to the completeness, accuracy or usefulness of the information contained in this document. Cadence does not warrant that use of such information will not infringe any third party rights, nor does Cadence assume any liability for damages or costs of any kind that may result from use of such information.

Restricted Rights: Use, duplication, or disclosure by the Government is subject to restrictions as set forth in FAR52.227-14 and DFAR252.227-7013 et seq. or its successor.
cadence
# Contents

1. Introducing the Spectre Circuit Simulator
   - Improvements over SPICE
     - Improved Capacity
     - Improved Accuracy
     - Improved Speed
     - Improved Reliability
     - Improved Models
     - Spectre Usability Features and Customer Service
   - Analog HDLs
   - RF Capabilities
   - Mixed-Signal Simulation
   - Environments

2. Getting Started with Spectre
   - Using the Example and Displaying Results
   - Sample Schematic
   - Sample Netlist
     - Elements of a Spectre Netlist
   - Instructions for a Spectre Simulation Run
     - Following Simulation Progress
     - Screen Printout
   - Viewing Your Output
   - Starting AWD
5

Parameter Specification and Modeling Features

Instance (Component or Analysis) Parameters

Types of Parameter Values

Parameter Dimension

Parameter Ranges

Help on Parameters

Scaling Numerical Literals

Parameters Statement

Circuit and Subcircuit Parameters

Parameter Declaration

Parameter Inheritance

Parameter Namespace

Parameter Referencing
Multiple Analyses ............................................................... 134
Multiple Analyses in a Subcircuit ........................................ 136
  Example ........................................................................ 137
DC Analysis ...................................................................... 138
AC Analysis ...................................................................... 139
Transient Analysis ............................................................. 140
  Trading Off Speed and Accuracy with a Single Parameter Setting 140
  Controlling the Accuracy ................................................ 142
  Setting the Integration Method ........................................ 143
  Improving Transient Analysis Convergence ....................... 144
  Controlling the Amount of Output Data ........................... 144
Other Analyses (sens, fourier, dcmatch, and stb) .................. 147
  Sensitivity Analysis ...................................................... 147
  Fourier Analysis .......................................................... 150
  DC Match Analysis ....................................................... 151
  Stability Analysis ........................................................ 154
Advanced Analyses (sweep and montecarlo) ......................... 156
  Sweep Analysis .......................................................... 156
  Monte Carlo Analysis ................................................... 161
Special Analysis (Hot-Electron Degradation) ....................... 175
  Hot-Electron Degradation Analysis ................................ 176
  Output Options for Hot-Electron Degradation Analysis ...... 177
  Example of Hot-Electron Degradation ............................. 178

7
Control Statements ............................................................. 183
The alter and altergroup Statements .................................... 183
  Changing Parameter Values for Components .................... 184
  Changing Parameter Values for Models ............................ 184
  Further Examples of Changing Component Parameter Values 185
  Changing Parameter Values for Circuits ........................... 185
The check Statement .......................................................... 186
The ic and nodeset Statements ............................................ 187
  Setting Initial Conditions for All Transient Analyses ........ 187
  Supplying Solution Estimates to Increase Speed ............... 189
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Specifying State Information for Individual Analyses</td>
<td>189</td>
</tr>
<tr>
<td>The info Statement</td>
<td>192</td>
</tr>
<tr>
<td>Specifying the Parameters You Want to Save</td>
<td>193</td>
</tr>
<tr>
<td>Specifying the Output Destination</td>
<td>194</td>
</tr>
<tr>
<td>Examples of the info Statement</td>
<td>194</td>
</tr>
<tr>
<td>Printing the Node Capacitance Table</td>
<td>194</td>
</tr>
<tr>
<td>The options Statement</td>
<td>198</td>
</tr>
<tr>
<td>options Statement Form at</td>
<td>199</td>
</tr>
<tr>
<td>options Statement Example</td>
<td>199</td>
</tr>
<tr>
<td>Setting Tolerances</td>
<td>199</td>
</tr>
<tr>
<td>Additional options Statement Settings You Might Need to Adjust</td>
<td>200</td>
</tr>
<tr>
<td>The paramset Statement</td>
<td>200</td>
</tr>
<tr>
<td>The save Statement</td>
<td>201</td>
</tr>
<tr>
<td>Saving Signals for Individual Nodes and Components</td>
<td>201</td>
</tr>
<tr>
<td>Saving Groups of Signals</td>
<td>207</td>
</tr>
<tr>
<td>The set Statement</td>
<td>210</td>
</tr>
<tr>
<td>The shell Statement</td>
<td>211</td>
</tr>
<tr>
<td>The statistics Statement</td>
<td>211</td>
</tr>
</tbody>
</table>

8 Specifying Output Options

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Signals as Output</td>
<td>212</td>
</tr>
<tr>
<td>Saving Signals for Individual Nodes and Components</td>
<td>213</td>
</tr>
<tr>
<td>Saving Main Circuit Signals</td>
<td>213</td>
</tr>
<tr>
<td>Saving Subcircuit Signals</td>
<td>215</td>
</tr>
<tr>
<td>Examples of the save Statement</td>
<td>215</td>
</tr>
<tr>
<td>Saving Individual Currents with Current Probes</td>
<td>216</td>
</tr>
<tr>
<td>Saving Power</td>
<td>217</td>
</tr>
<tr>
<td>Saving Groups of Signals</td>
<td>219</td>
</tr>
<tr>
<td>Formatting the save and nestlvl Parameters</td>
<td>219</td>
</tr>
<tr>
<td>The save Parameter Options</td>
<td>219</td>
</tr>
<tr>
<td>Saving Subcircuit Signals</td>
<td>220</td>
</tr>
<tr>
<td>Saving Groups of Currents</td>
<td>220</td>
</tr>
<tr>
<td>Saving All AHDL Variables</td>
<td>222</td>
</tr>
<tr>
<td>Listing Parameter Values as Output</td>
<td>222</td>
</tr>
</tbody>
</table>
Specifying the Parameters You Want to Save ........................................... 223
Specifying the Output Destination ............................................................ 223
Examples of the info Statement ................................................................. 224
Prepapring Output for Viewing ................................................................. 224
Output Formats Supported by the Spectre Simulator .................................... 224
Defining Output File Formats .................................................................... 225
Accessing Output Files .............................................................................. 225
How the Spectre Simulator Creates Names for Output Directories and Files .... 226
Filenames for SPICE Input Files ................................................................. 228
Specifying Your Own Names for Directories ............................................. 228

9
Running a Simulation .................................................................................. 229
Starting Simulations ................................................................................. 229
Specifying Simulation Options ................................................................... 229
Determining Whether a Simulation Was Successful ...................................... 230
Checking Simulation Status ....................................................................... 230
Interrupting a Simulation ......................................................................... 231
Recovering from Transient Analysis Terminations ..................................... 231
Creating Recovery Files from the Command Line ...................................... 232
Setting Recovery File Specifications for a Single Analysis .......................... 233
Restarting a Transient Analysis .................................................................. 233
Controlling Command Line Defaults .......................................................... 233
Examining the Spectre Simulator Defaults ................................................ 234
Setting Your Own Defaults ........................................................................ 234
References for Additional Information about Specific Defaults ................. 235
Overriding Defaults ................................................................................... 235

10
Time-Saving Techniques .......................................................................... 236
Specifying Efficient Starting Points ........................................................... 236
Reducing the Number of Simulation Runs .................................................. 236
Adjusting Speed and Accuracy ................................................................... 237
Saving Time by Starting Analyses from Previous Solutions ....................... 237
Saving Time by Specifying State Information ............................................ 237
### Setting Initial Conditions for All Transient Analyses

- Setting Initial Conditions for All Transient Analyses
- Supplying Solution Estimates to Increase Speed
- Specifying State Information for Individual Analyses
- Saving Time by Modifying Parameters during a Simulation
- Changing Circuit or Component Parameter Values
- Modifying Initial Settings of the State of the Simulator
- Saving Time by Selecting a Continuation Method

### Managing Files

- About Spectre Filename Specification
- Creating Filenames That Help You Manage Data
- Creating Filenames by Modifying Input Filenames
- Description of Spectre Predefined Percent Codes
- Customizing Percent Codes
- Creating Filenames from Parts of Input Filenames

### Identifying Problems and Troubleshooting

- Error Conditions
  - Invalid Parameter Values That Terminate the Program
  - Singular Matrices
  - Internal Error Messages
  - Time Is Not Strictly Increasing
- Spectre Warning Messages
  - P-N Junction Warning Messages
  - Parameter Is Unusually Large or Small
  - gmin Is Large Enough to Noticeably Affect the DC Solution
  - Minimum Timestep Used
  - Syntax Errors
  - Topology Messages
  - Model Parameter Values Clamped
  - Invalid Parameter Warnings
  - Redefine Primitives Messages
  - Initial Condition Messages
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Output Messages</td>
<td>262</td>
</tr>
<tr>
<td>Customizing Error and Warning Messages</td>
<td>262</td>
</tr>
<tr>
<td>Selecting Limits for Parameter Value Warning Messages</td>
<td>263</td>
</tr>
<tr>
<td>Selecting Limits for Operating Region Warnings</td>
<td>270</td>
</tr>
<tr>
<td>Range Checking on Subcircuit Parameters</td>
<td>271</td>
</tr>
<tr>
<td>Formatting the paramtest Component</td>
<td>271</td>
</tr>
<tr>
<td>Controlling Program-Generated Messages</td>
<td>273</td>
</tr>
<tr>
<td>Specifying Log File Options</td>
<td>273</td>
</tr>
<tr>
<td>Correcting Convergence Problems</td>
<td>274</td>
</tr>
<tr>
<td>Correcting DC Convergence Problems</td>
<td>274</td>
</tr>
<tr>
<td>Correcting Transient Analysis Convergence Problems</td>
<td>277</td>
</tr>
<tr>
<td>Correcting Accuracy Problems</td>
<td>277</td>
</tr>
<tr>
<td>Suggestions for Improving DC Analysis Accuracy</td>
<td>277</td>
</tr>
<tr>
<td>Suggestions for Improving Transient Analysis Accuracy</td>
<td>278</td>
</tr>
</tbody>
</table>

A

Example Circuits                                                        | 279  |
| Notes on the BSIM3v3 Model                                              | 280  |
| Spectre Syntax                                                         | 280  |
| SPICE BSIM 3v3 Model                                                    | 280  |
| Spectre BSIM 3v3 Model                                                  | 281  |
| Ring Oscillator Spectre Deck for Inverter Ring with No Fanouts         | 281  |
| Ring Oscillator Spectre Deck for Two-Input NAND Ring with No Fanouts   | 283  |
| Ring Oscillator Spectre Deck for Three-Input NAND Ring with No Fanouts | 284  |
| Ring Oscillator Spectre Deck for Two-Input NOR Ring with No Fanouts    | 286  |
| Ring Oscillator Spectre Deck for Three-Input NOR Ring with No Fanouts  | 287  |
| Opamp Circuit (opamp.cir)                                               | 289  |
| Opamp Circuit 2 (opamp1.cir)                                           | 289  |
| Original Open-Loop Opamp (openloop.sp)                                 | 289  |
| Modified Open-Loop Opamp (openloop1.sp)                                | 290  |
| Example Model Directory (q35d4h5.modsp)                                 | 290  |
Preface

This manual assumes that you are familiar with the development, design, and simulation of integrated circuits and that you have some familiarity with SPICE simulation. It contains information about the Spectre® circuit simulator.

Spectre is an advanced circuit simulator that simulates analog and digital circuits at the differential equation level. The simulator uses improved algorithms that offer increased simulation speed and greatly improved convergence characteristics over SPICE. Besides the basic capabilities, the Spectre circuit simulator provides significant additional capabilities over SPICE. SpectreHDL (Spectre High-Level Description Language) and Verilog®-A use functional description text files (modules) to model the behavior of electrical circuits and other systems. SpectreRF adds several new analyses that support the efficient calculation of the operating point, transfer function, noise, and distortion of common RF and communication circuits, such as mixers, oscillators, sample holds, and switched-capacitor filters.

This Preface discusses the following topics:

- Related Documents on page 13
- Typographic and Syntax Conventions on page 14
- References on page 15

Related Documents

The following can give you more information about the Spectre circuit simulator and related products:

- The Spectre circuit simulator is often run within the analog circuit design environment, under the Cadence design framework II. To see how the Spectre circuit simulator is run under the analog circuit design environment, read the Cadence Analog Design Environment User Guide.

- To learn more about specific parameters of components and analyses, consult the Spectre online help (spectre -h) or the Spectre Circuit Simulator Reference manual.

- To learn more about the equations used in the Spectre circuit simulator, consult the Spectre Circuit Simulator Device Model Equations manual.
The Spectre circuit simulator also includes a waveform display tool, Analog Waveform Display (AWD), to use to display simulation results. For more information about AWD, see the Analog Waveform User Guide.

For more information about using the Spectre circuit simulator with SpectreHDL, see the SpectreHDL Reference manual.

For more information about using the Spectre circuit simulator with Verilog-A, see the Verilog-A Language Reference manual.

If you want to see how SpectreRF is run under the analog circuit design environment, read SpectreRF Help.

For more information about RF theory, see SpectreRF Theory.

For more information about how you work with the design framework II interface, see Cadence Design Framework II Help.

For more information about specific applications of Spectre analyses, see The Designer’s Guide to SPICE & Spectre1.

Typographic and Syntax Conventions

This list describes the syntax conventions used for the Spectre circuit simulator.

`literal` Nonitalic words indicate keywords that you must enter literally. These keywords represent command (function, routine) or option names, filenames and paths, and any other sort of type-in commands.

`argument` Words in italics indicate user-defined arguments for which you must substitute a name or a value. (The characters before the underscore (_) in the word indicate the data types that this argument can take. Names are case sensitive.

`|` Vertical bars (OR-bars) separate possible choices for a single argument. They take precedence over any other character.

`[ ]` Brackets denote optional arguments. When used with OR-bars, they enclose a list of choices. You can choose one argument from the list.

Braces are used with OR-bars and enclose a list of choices. You must choose one argument from the list.

Three dots (...) indicate that you can repeat the previous argument. If you use them with brackets, you can specify zero or more arguments. If they are used without brackets, you must specify at least one argument, but you can specify more.

Important

The language requires many characters not included in the preceding list. You must enter required characters exactly as shown.

References

Text within brackets ([ ]) is a reference. See Appendix A. “References” of the Spectre Circuit Simulator Reference manual for more detailed information.
Introducing the Spectre Circuit Simulator

This chapter discusses the following:

- **Improvements over SPICE** on page 16
- **Analog HDLs** on page 20
- **RF Capabilities** on page 21
- **Mixed-Signal Simulation** on page 23
- **Environments** on page 23

The Spectre® circuit simulator is a modern circuit simulator that uses direct methods to simulate analog and digital circuits at the differential equation level. The basic capabilities of the Spectre circuit simulator are similar in function and application to SPICE, but the Spectre circuit simulator is not descended from SPICE. The Spectre and SPICE simulators use the same basic algorithms—such as implicit integration methods, Newton-Raphson, and direct matrix solution—but every algorithm is newly implemented. Spectre algorithms, the best currently available, give you an improved simulator that is faster, more accurate, more reliable, and more flexible than previous SPICE-like simulators.

**Improvements over SPICE**

The Spectre circuit simulator has many improvements over SPICE.

**Improved Capacity**

The Spectre circuit simulator can simulate larger circuits than other simulators because its convergence algorithms are effective with large circuits, because it is fast, and because it is frugal with memory and uses dynamic memory allocation. For large circuits, the Spectre circuit simulator typically uses less than half as much memory as SPICE.
Improved Accuracy

Improved component models and core simulator algorithms make the Spectre circuit simulator more accurate than other simulators. These features improve Spectre accuracy:

- **Advanced metal oxide semiconductor (MOS) and bipolar models**
  - The Spectre BSIM3v3 is a physics-based metal-oxide semiconductor field effect transistor (MOSFET) model for simulating analog circuits.
  - The Spectre models include the MOS0 model, which is even simpler and faster than MOS1 for simulating noncritical MOS transistors in logic circuits and behavioral models, MOS 9, EKV, BTA-HVMOS, BTA-SOI, VBIC95, TOM2, HBT, and many more.

- **Charge-conserving models**
  The capacitance-based nonlinear MOS capacitor models used in many SPICE derivatives can create or destroy small amounts of charge on every time step. The Spectre circuit simulator avoids this problem because all Spectre models are charge-conserving.

- **Improved Fourier analyzer**
  The Spectre circuit simulator includes a two-channel Fourier analyzer that is similar in application to the SPICE .FOURIER statement but is more accurate. The Spectre simulator’s Fourier analyzer has greater resolution for measuring small distortion products on a large sinusoidal signal. Resolution is normally greater than 120 dB. Furthermore, the Spectre simulator’s Fourier analyzer is not subject to aliasing, a common error in Fourier analysis. As a result, the Spectre simulator can accurately compute the Fourier coefficients of highly discontinuous waveforms.

- **Better control of numerical error**
  Many algorithms in the Spectre circuit simulator are superior to their SPICE counterparts in avoiding known sources of numerical error. The Spectre circuit simulator improves the control of local truncation error in the transient analysis by controlling error in the voltage rather than the charge.

  In addition, the Spectre circuit simulator directly checks Kirchhoff’s Current Law (also known as Kirchhoff’s Flow Law) at each time step, improves the charge-conservation accuracy of the Spectre circuit simulator, and eliminates the possibility of false convergence.

- **Superior time-step control algorithm**
Spectre Circuit Simulator User Guide
Introducing the Spectre Circuit Simulator

The Spectre circuit simulator provides an adaptive time-step control algorithm that reliably follows rapid changes in the solution waveforms. It does so without limiting assumptions about the type of circuit or the magnitude of the signals.

- More accurate simulation techniques

Techniques that reduce reliability or accuracy, such as device bypass, simplified models, or relaxation methods, are not used in the Spectre circuit simulator.

- User control of accuracy tolerances

For some simulations, you might want to sacrifice some degree of accuracy to improve the simulation speed. For other simulations, you might accept a slower simulation to achieve greater accuracy. With the Spectre circuit simulator, you can make such adjustments easily by setting a single parameter.

Improved Speed

The Spectre circuit simulator is designed to improve simulation speed. The Spectre circuit simulator improves speed by increasing the efficiency of the simulator rather than by sacrificing accuracy.

- Faster simulation of small circuits

The average Spectre simulation time for small circuits is typically two to three times faster than SPICE. The Spectre circuit simulator can be over 10 times faster than SPICE when SPICE is hampered by discontinuity in the models or problems in the code. Occasionally, the Spectre circuit simulator is slower when it finds ringing or oscillation that goes unnoticed by SPICE. This can be improved by setting the macromodels option to yes.

- Faster simulation for large circuits

The Spectre circuit simulator is generally two to five times faster than SPICE with large circuits because it has fewer convergence difficulties and because it rapidly factors and solves large sparse matrices.

Improved Reliability

The Spectre circuit simulator offers you the following improvements in reliability:

- Improved convergence

Spectre proprietary algorithms ensure convergence of the Newton-Raphson algorithm in the DC analysis. The Spectre circuit simulator virtually eliminates the convergence problems that earlier simulators had with transient simulation.
Helpful error and warning messages

The Spectre circuit simulator detects and notifies you of many conditions that are likely to be errors. For example, the Spectre circuit simulator warns of models used in forbidden operating regions, of incorrectly wired circuits, and of erroneous component parameter values. By identifying such common errors, the Spectre circuit simulator saves you the time required to find these errors with other simulators.

The Spectre circuit simulator lets you define soft parameter limits and sends you warnings if parameters exceed these limits.

Thorough testing

Automated tests, which include over 1,000 test circuits, are constantly run on all hardware platforms to ensure that the Spectre circuit simulator is consistently reliable and accurate.

Benchmark suite

There is an independent collection of SPICE netlists that are difficult to simulate. You can obtain these circuits from the Microelectronics Center of North Carolina (MCNC) if you have File Transfer Protocol (FTP) access on the Internet. You can also get information about the performance of several simulators with these circuits.

The Spectre circuit simulator has successfully simulated all of these circuits. Sometimes the netlists required minor syntax corrections, such as inserting balancing parentheses, but circuits were never altered, and options were never changed to affect convergence.

Improved Models

The Spectre circuit simulator has MOSFET Level 0–3, BSIM1, BSIM2, BSIM3, BSIM3v3, EKV, MOS9, JFET, TOM2, GaAs MESFET, BJT, VBIC, HBT, diode, and many other models. It also includes the temperature effects, noise, and MOSFET intrinsic capacitance models.

The Spectre Compiled Model Interface (CMI) option lets you integrate new devices into the Spectre simulator using a very powerful, efficient, and flexible C language interface. This CMI option, the same one used by Spectre developers, lets you install proprietary models.

Spectre Usability Features and Customer Service

The following features and services help you use the Spectre circuit simulator easily and efficiently:

You can use Spectre soft limits to catch errors created by typing mistakes.
Spectre Circuit Simulator User Guide
Introducing the Spectre Circuit Simulator

- Spectre diagnosis mode, available as an options statement parameter, gives you information to help diagnose convergence problems.

- You can run the Spectre circuit simulator standalone or run it under the Cadence™ analog design environment. To see how the Spectre circuit simulator is run under the analog circuit design environment, read the *Cadence Analog Design Environment User Guide*. You can also run the Spectre circuit simulator in the Composer-to-Spectre direct simulation environment. The environment provides a graphical user interface for running the simulation.

- The Spectre circuit simulator gives you an online help system. With this system, you can find information about any parameter associated with any Spectre component or analysis. You can also find articles on other topics that are important to using the Spectre circuit simulator effectively.

- The Spectre circuit simulator also includes a waveform display tool, Analog Waveform Display (AWD), to use to display simulation results. For more information about AWD, see the *Analog Waveform User Guide*.

- If you experience a stubborn convergence or accuracy problem, you can send the circuit to Customer Support to get help with the simulation. For current phone numbers and e-mail addresses, see the following web site: [http://sourcelink.cadence.com/supportcontacts.html](http://sourcelink.cadence.com/supportcontacts.html)

Analog HDLs

The Spectre circuit simulator works with two analog high-level description languages (AHDLs): SpectreHDL and Verilog®-A. These languages are part of the Spectre Verilog-A option. SpectreHDL is proprietary to Cadence and is provided for backward compatibility. The Verilog-A language is an open standard, which was based upon SpectreHDL. The Verilog-A language is preferred because it is upward compatible with Verilog-AMS, a powerful and industry-standard mixed-signal language.

Both languages use functional description text files (modules) to model the behavior of electrical circuits and other systems. Each programming language allows you to create your own models by simply writing down the equations. The AHDL lets you describe models in a simple and natural manner. This is a higher level modeling language than previous modeling languages, and you can use it without being concerned about the complexities of the simulator or the simulator algorithms. In addition, you can combine AHDL components with Spectre built-in primitives.

Both languages let designers of analog systems and integrated circuits create and use modules that encapsulate high-level behavioral descriptions of systems and components. The behavior of each module is described mathematically in terms of its terminals and
external parameters applied to the module. Designers can use these behavioral descriptions in many disciplines (electrical, mechanical, optical, and so on).

Both languages borrow many constructs from Verilog and the C programming language. These features are combined with a minimum number of special constructs for behavioral simulation. These high-level constructs make it easier for designers to use a high-level description language for the first time.

**RF Capabilities**

SpectreRF adds several new analyses that support the efficient calculation of the operating point, transfer function, noise, and distortion of common analog and RF communication circuits, such as mixers, oscillators, sample and holds, and switched-capacitor filters.

SpectreRF adds four types of analyses to the Spectre simulator. The first is periodic steady-state (PSS) analysis, a large-signal analysis that directly computes the periodic steady-state response of a circuit. With PSS, simulation times are independent of the time constants of the circuit, so PSS can quickly compute the steady-state response of circuits with long time constants, such as high-Q filters and oscillators.

You can also embed a PSS analysis in a sweep loop (referred to as an SPSS analysis in the Cadence analog design environment), which allows you to easily determine harmonic levels as a function of input level or frequency, making it easy to measure compression points, intercept points, and voltage-controlled oscillator (VCO) linearity.

The second new type of analysis is the periodic small-signal analysis. After completing a PSS analysis, SpectreRF can predict the small-signal transfer functions and noise of frequency translation circuits, such as mixers or periodically driven circuits such as oscillators or switched-capacitor or switched-current filters. The periodic small-signal analyses—periodic AC (PAC) analysis, periodic transfer function (PXF) analysis, and periodic noise (Pnoise) analysis—are similar to Spectre’s AC, XF, and Noise analyses, but the traditional small-signal analyses are limited to circuits with DC operating points. The periodic small-signal analyses can be applied to circuits with periodic operating points, such as the following:

- Mixers
- VCOs
- Switched-current filters
- Phase/frequency detectors
- Frequency multipliers
- Chopper-stabilized amplifiers
The third SpectreRF addition to Spectre functionality is periodic distortion (PDISTO) analysis. PDISTO analysis directly computes the steady-state response of a circuit driven with a large periodic signal, such as an LO (local oscillation) or a clock, and one or more tones with moderate level. With PDISTO, you can model periodic distortion and include harmonic effects. PDISTO computes both a large signal, the periodic steady-state response of the circuit, and also the distortion effects of a specified number of moderate signals, including the distortion effects of the number of harmonics that you choose. This is a common scenario when trying to predict the intermodulation distortion of a mixer, amplifier, or a narrow-band filter. In this analysis, the tones can be large enough to create significant distortion, but not so large as to cause the circuit to switch or clip. The frequencies of the tones need not be periodically related to each other or to the large signal LO or clock. Thus, you can make the tone frequencies very close to each other without penalty, which allows efficient computation of intermodulation distortion of even very narrow band circuits.

The fourth analysis that SpectreRF adds to the Spectre circuit simulator is the envelope-following analysis. This analysis computes the envelope response of a circuit. The simulator automatically determines the clock period by looking through all the sources with the specified name. Envelope-following analysis is most efficient for circuits where the modulation bandwidth is orders of magnitude lower than the clock frequency. This is typically the case, for example, in circuits where the clock is the only fast varying signal and other input signals have a spectrum whose frequency range is orders of magnitude lower than the clock frequency. For another example, the down conversion of two closely placed frequencies can also generate a slow-varying modulation envelope. The analysis generates two types of output files, a voltage versus time (td) file, and an amplitude/phase versus time (fd) file for each specified harmonic of the clock fundamental.

In summary, with periodic small-signal analyses, you apply a small signal at a frequency that might not be harmonically related to the periodic response of the undriven system, the clock. This small signal is assumed to be small enough so that the circuit is unaffected by its presence.

With PDISTO, you can apply one or two additional signals at frequencies not harmonically related to the large signal, and these signals can be large enough to drive the circuit to behave non-linearly.
For complex nonlinear circuits, hand calculation of noise or transfer function is virtually impossible. Without SpectreRF, these circuits must be breadboarded to determine their performances. SpectreRF eliminates unnecessary breadboarding, saving time.

**Mixed-Signal Simulation**

You can use the Spectre circuit simulator coupled with the Verilog®-XL simulator in the Cadence analog design environment to simulate mixed analog and digital circuits efficiently. This mixed-signal simulation solution can easily handle complex designs with tens of thousands of transistors and tens of thousands of gates. The digital Verilog data can come from the digital designer as either an RTL block or gates out of synthesis.

**Environments**

The Spectre circuit simulator is fully integrated into the Cadence® design framework II for the Cadence analog design environment and also into the Cadence analog workbench design system. You can also use the Spectre circuit simulator by itself with several different output format options.

Assura interactive verification, Dracula® distributed multi-CPU option, and Assura hierarchical physical verification produce a netlist that can be read into the Spectre circuit simulator. However, only interactive verification when used with the Cadence analog design environment automatically attaches the stimulus file. All other situations require a stimulus file as well as device models.
Getting Started with Spectre

This chapter discusses the following topics:

- Using the Example and Displaying Results on page 24
- Sample Schematic on page 25
- Sample Netlist on page 26
- Instructions for a Spectre Simulation Run on page 30
- Viewing Your Output on page 32

Using the Example and Displaying Results

In this chapter, you examine a schematic and its Spectre® circuit simulator netlist to get an overview of Spectre syntax. You also follow a sample circuit simulation. The best way to use this chapter depends on your past experience with simulators.

Carefully examine the schematic (“Sample Schematic” on page 25) and netlist (“Sample Netlist” on page 26) and compare Spectre netlist syntax with that of SPICE-like simulators you have used. If you have prepared netlists for SPICE-like simulators before, you can skim “Elements of a Spectre Netlist” on page 27. With this method, you can learn a fair amount about the Spectre simulator in a short time.

Approach this chapter as an overview. You will probably have unanswered questions about some topics when you finish the chapter. Each topic is covered in greater depth in subsequent chapters. Do not worry about learning all the details now.

To give you a complete overview of a Spectre simulation, the example in this chapter includes the display of simulation results with Analog Waveform Display (AWD), a waveform display tool that is included with the Spectre simulator. If you use another display tool, the procedures you follow to display results are different. This user guide does not teach you how to display waveforms with different tools. If you need more information about how to display Spectre results, consult the documentation for your display tool.
The example used in this chapter is a small circuit, an oscillator; you run a transient analysis on the oscillator and then view the results. The following sections contain the schematic and netlist for the oscillator. If you have used SPICE-like simulators before, looking at the schematic and netlist can help you compare Spectre syntax with those of other simulators. If you are new to simulation, looking at the schematic and netlist can prepare you to understand the later chapters of this book.

You can also get more information about command options, components, analyses, controls, and other selected topics by using the `spectre -h` command to access the Spectre online help.

**Sample Schematic**

A schematic is a drawing of an electronic circuit, showing the components graphically and how they are connected together. The following schematic has several annotations:

- **Names of components**
  
  Each component is labeled with the name that appears in the instance statement for that component. The names for components are in italics (for example, *Q2*).

- **Names of nodes**
  
  Each node in the circuit is labeled with its unique name or number. This name can be either a name you create or a number. Names of nodes are in boldface type (for example, *b1*). Ground is node 0.

- **Sample instance statements**
  
  The schematic is annotated with instance statements for some of the components. Arrows connect the components in the schematic with their corresponding instance statements.
Sample Netlist

A netlist is an ASCII file that lists the components in a circuit, the nodes that the components are connected to, and parameter values. You create the netlist in a text editor such as vi or

**Bold Type** = Names of nodes. All connections to ground have the same node name.

**Italic Type** = Names of components (also appear in the instance statement for each component).

= Link between components and instance statements.
emacs or from one of the environments that support the Spectre simulator. The Spectre simulator uses a netlist to simulate a circuit.

// BJT ECP Oscillator
// simulator lang=spectre

Iee (e 0) isource dc=1mA

Q1 (cc b1 e) npn
Q2 (out b2 e) npn

L1 (cc out) inductor l=1uH

C1 (cc out) capacitor c=1pf
C2 (out b1) capacitor c=272.7pF
C3 (b1 0) capacitor c=3nF
C4 (b2 0) capacitor c=3nF

R1 (b1 0) resistor r=10k
R2 (b2 0) resistor r=10k

ic bc=1

model npn bjt type=npn bf=80 rb=100 vaf=50 \ cjs=2pf tf=0.3ns tr=6ns cje=3pf cjc=2pf

OscResp tran stop=80us maxstep=10ns

Elements of a Spectre Netlist

This section briefly explains the components, models, analyses, and control statements in a Spectre netlist. All topics discussed here (such as model statements or the simulator lang command) are presented in greater depth in later chapters. If you want more complete reference information about a topic, consult these discussions.

Title Line

The first line is taken to be the title. It is used verbatim when labeling output. Any statement you place in the first line is ignored as a comment. For more information about comment lines, see "Basic Syntax Rules" on page 58.
Simulation Language

The second line of the sample netlist indicates that the netlist is in the Spectre Netlist Language, instead of SPICE. For more information about the simulator lang command, see Chapter 3, “SPICE Compatibility.”

Instance Statements

The next section in the sample netlist consists of instance statements. To specify a single component in a Spectre netlist, you place all the necessary information for the component in a netlist statement. Netlist statements that specify single components are called instance statements. (The instance statement also has other uses that are described in Chapter 4, “Spectre Netlists.”)

To specify single components within a circuit, you must provide the following information:

- A unique component name for the component
- The names of nodes to which the component is connected
- The master name of the component (identifies the type of component)
- The parameter values associated with the component

A typical Spectre instance statement looks like this:

\[ \text{Component name} \rightarrow \text{R16 (4 0) resistor r=100} \rightarrow \text{Parameter value} \]

\[ \text{Node names} \rightarrow \text{Master name} \]

**Note:** You can use balanced parentheses to distinguish the various parts of the instance statement, although they are optional:

- R1 (1 2) resistor r=1
- Q1 (c b e s) npn area=10
- Gm (1 2) (3 4) vccs gm=.01
- R7 (x y) rmod (r=1k w=2u)

Component Names

Unlike SPICE, the first character of the component name has no special meaning. You can use any character to start the component name. For example:
Load (out o) resistor r=50
Balun (in o pout nout) transformer

**Note:** You can find the exact format for any component in the parameter listings for that component in the Spectre online help.

**Master Names**

The type of a component depends on the name of the master, not on the first letter of the component name (as in SPICE); this feature gives you more flexibility in naming components. The master can be a built-in primitive, a model, a subcircuit, or an AHDL component.

**Parameter Values**

Real numbers can be specified using scientific notation or common engineering scale factors. For example, you can specify a 1 pF capacitor value either as `c=1pf` or `c=1e-12`. Depending on whether you are using the Spectre Netlist Language or SPICE, you might need to use different scale factors for parameter values. Only ANSI standard scale factors are used in Spectre netlists. For more information about scale factors, see “Instance Statements” on page 61.

**Control Statements**

The next section of the sample netlist contains a control statement, which sets initial conditions.

**Model Statements**

Some components allow you to specify parameters common to many instances using the `model` statement. The only parameters you need to specify in the instance statement are those that are generally unique for a given instance of a component.

You need to provide the following for a `model` statement:

- The keyword `model` at the beginning of the statement
- A unique name for the model (reference by master names in instance statements)
- The master name of the model (identifies the type of model)
- The parameter values associated with the model

The following example is a `model` statement for a bjt. The model name is `npn`, and the component type name is `bjt`. The backslash (\) tells you that the statement continues on the
next line. The backslash must be the last character in the line because it escapes the carriage return.

```plaintext
model npn bjt type=npn bf=80 rb=100 vaf=50 \ cjs=2pf tf=0.3ns tr=6ns cje=3pf cjc=2pf
```

When you create an instance statement that refers to a `model` statement for its parameter values, you must specify the model name as the master name. For example, an instance statement that receives its parameter values from the previous `model` statement might look like this:

```plaintext
Q1  (vcc b1 e vcc) npn
```

Check documentation for components to determine which parameters are expected to be provided on the instance statement and which are expected on the model statement.

### Analysis Statements

The last section of the sample netlist has the analysis statement. An analysis statement has the same syntax as an instance statement, except that the analysis type name replaces the master name. To specify an analysis, you must include the following information in a netlist statement:

- A unique name for the analysis statement
- Possibly a set of node names
- The name of the type of analysis you want
- Any additional parameter values associated with the analysis

To find the analysis type name and the parameters you can specify for an analysis, consult the parameter listing for that analysis in the Spectre online help (`spectre -h`).

The following analysis statement specifies a transient analysis. The analysis name is `stepResponse`, and the analysis type name is `tran`.

```plaintext
stepResponse tran stop=100ns
```

### Instructions for a Spectre Simulation Run

When you complete a netlist, you can run the simulation with the `spectre` command.

➤ To run a simulation for the sample circuit, type the following at the command line:

```plaintext
spectre osc.cir
```

**Note:** `osc.cir` is the file that contains the netlist.
Following Simulation Progress

As the simulation runs, the Spectre simulator sends messages to your screen that show the progress of the simulation and provide statistical information. In the simulation of osc.cir, the Spectre simulator prints some warnings and notifications. The Spectre simulator tells you about conditions that might reduce simulation accuracy. When you see a Spectre warning or notification, you must decide whether the information is significant for your particular simulation.

Screen Printout

The printout for the osc.cir simulation looks like this:

spectre (ver. 4.4.3.52_339 -- 02 Jul 1998).


Circuit inventory:

nodes 5
equations 9
bjt 2
capacitor 4
inductor 1
isource 1
resistor 2
vsource 1

***************************************************
Transient Analysis `OscResp': time = (0 s -> 80 us)
***************************************************

.....9.....8......7......6......5......4......3......2.....1......0
Number of accepted tran steps = 11090.
Initial condition solution time = 10 ms.
Intrinsic tran analysis time = 15.53 s.
Total time required for tran analysis `OscResp' was 15.58 s.

Aggregate audit (9:24:38 AM, Mon Jul 6, 1998):
Time used: CPU = 17.7 s, elapsed = 23 s, util. = 77%.
Virtual memory used = 1.39 Mbytes.
spectre completes with 0 errors, 0 warnings, and 0 notices.
Viewing Your Output

The waveform display tool for this simulation example is AWD, a display tool you receive when you purchase the Spectre simulator. In this section you will learn the following:

- How to start AWD
- How to display waveforms with the Results Browser
- How to extract zero-crossing points of a waveform
- How to select new colors for waveforms
- How to access data from multiple directories

Starting AWD

There are several ways to start AWD.

Using the -dataDir Command Line Option

To start AWD for the simulation example, type the following at the command line (from the directory where the Spectre simulator was run):

```
awd -dataDir osc.raw
```

Using Just the awd Command

1. To start AWD for the simulation example, type the following at the command line (from the directory where the Spectre simulator was run):

```
awd osc.raw
```

2. In the Browse Project Hierarchy form, click OK.

For this command to work, you need to specify which simulator you are using. The default is Spectre. To specify the simulator, create a .cdsinit file and put the following line in it:

```
envSetVal( "asimenv.startup" "simulator" 'string "spectre" )
```

Save the .cdsinit file in the directory from which you start AWD or in your home directory. AWD looks for the .cdsinit file first in the directory from which AWD was started; if the file is not there, AWD looks for it in your home directory.

Another option is to place the following entry in a .cdsenv file:

```
asimenv.startup simulator string "spectre"
```
Save the `.cdsenv` file in your home directory. If AWD does not find the `.cdsenv` file in your home directory, AWD looks for the system-defined default settings.
AWD and Other Windows

Once AWD is running, four windows appear on your screen. They are shown in the following figures.

Command Interpreter Window (CIW)

Calculator Window
Results Browser

100m
Apache/
MM/
Navajo/
adm/
barbaral/
log/
lost+found/
nsmail/
opt/
osc%
osc.cir
osc,ram/
psf@
stuff/
swap2
tmp/
var/
Displaying a Waveform

1. In the Results Browser, scroll, if necessary, until `osc.raw` is visible and then click on `osc.raw`. 
The display in the Results Browser now looks like this:

![Results Browser diagram]

The lowest level of the hierarchy in the Results Browser now contains the names of the waveforms you can display.

2. To display a waveform, click on its name in the Results Browser with the right mouse button.
For example, if you click on `out`, the following display appears in the Waveform Window:
If you zoom in on the right side of the waveform (Zoom – Zoom In) and adjust the X axis (Axes – X Axis), you see the following:

You can add a more meaningful title with the Annotation – Title command.

**Extracting Zero-Crossing Points**

To extract zero-crossing points of a waveform using the calculator:

1. Make certain that the Evaluate Buffer button in the calculator is disabled.
2. Select the waveform in the Results Browser by clicking the left mouse button over the name of the waveform.

3. Choose *cross* from the Special Functions menu in the calculator

   The Threshold Crossing form appears.

4. Type 0 in the *Threshold Value* field, type 0 in the *Edge Number* field, and set *Edge Type* to either.

5. Click *OK*.

6. Enable the *Evaluate Buffer* button in the calculator.

This returns the list of zero-crossing points (time co-ordinates) of the waveform in the CIW.

Selecting New Colors for Waveforms

To select a new color for a waveform:

1. In the Waveform Window, choose *Curves – Edit*.

   The Curves form appears.

2. In the Curves form, select the curve name you want and the pen color.

3. Click *OK*.

Accessing Data from Multiple Directories

There are two ways to access data from multiple directories:

- If you are accessing data from multiple directories and each unique data directory path ends with a directory called psf, use the *awd* command by itself and click *OK* in the Browse Project Hierarchy form. In the Results Browser, expand the hierarchy down to a data directory, click the middle mouse button over the waveform name, and select the *Create ROF* command from the middle mouse button menu. Then expand this data tree to access the data.

  To access a different data directory, back up in the hierarchy in the Results Browser and then expand down to some other data directory. Again, select the *Create ROF* command for that particular psf directory and expand down to the data. Data can be overlaid on top of data from a previous data directory.

- If you are accessing data from multiple directories and the data directory path names do not end with psf as the last directory, you can still use the *awd* command by itself.
When you expand down to a data directory and select the *Create ROF* command, the system creates a soft link from that data directory to `psf`. You can now expand down to the data. If you then move to a different data directory in the Results Browser, you have to again select the *Create ROF* command, even if a run object file already exists. This removes the old link (if the two data directories exist in the same level of hierarchy) and reestablishes the `psf` link to the new data directory. Now you can expand down to data. If you want to plot from the calculator a net you had previously plotted already (for example, overlay the same net from two different runs), you need to first flush out the cache memory.

### Learning More about AWD

The AWD display tool gives you a number of additional options for displaying your data. To learn more about using AWD with the Spectre simulator, see the Cadence application note, “Using `awd` with Standalone Spectre,” available from Cadence Customer Support, and the *Analog Waveform User Guide* about displaying results for the Cadence analog design environment.
SPICE Compatibility

While SPICE is an industry-standard language, there are many variations of SPICE syntax on the market today. Most of the language is common across the major commercial simulators, but each vendor has extended or modified it with different capabilities and/or slightly different syntax. For the convenience of SPICE users, the Spectre® circuit simulator provides a SPICE Reader as an extension to its native language that accepts most variations of SPICE input. The SPICE Reader provides support for SPICE2, SPICE3, and common extensions found in other simulators such as PSPICE and HSPICE. This chapter focuses on how to use the SPICE Reader with Spectre. This chapter is not intended to describe the syntax variations of SPICE, as they are well documented in other places, nor an exact listing of syntax supported by the SPICE Reader.

- Reading SPICE Netlists on page 42
- Language Differences on page 48
- Scope of the Compatibility on page 50

Reading SPICE Netlists

Spectre provides the SPICE Reader to let you simulate existing SPICE netlists with Spectre. While it is possible to gain access to many of Spectre’s features with a SPICE netlist, the Spectre syntax is a much more powerful language designed to give the Spectre user features beyond those provided by SPICE. Whenever possible, you are encouraged to take advantage of Spectre’s native language. When Spectre syntax is not an acceptable solution, the SPICE Reader can be used to ensure access to as much of Spectre’s capabilities as possible.

Running the SPICE Reader

In order for Spectre to properly process a SPICE netlist, the SPICE Reader must be enabled. By default the SPICE Reader is disabled when you invoke Spectre. Use the command spectrespp rather than the spectre command to enable the SPICE Reader. For example, the following command invokes Spectre with the SPICE Reader for the input netlist amp.sp:

```
spectrespp amp.sp
```
The Spice Reader runs on the top-level netlist, as well as on any included files that contain SPICE syntax. The SPICE Reader converts the netlist to Spectre native syntax before it sends the design to the Spectre netlist reader. From the 5.0 release, any `.measure` statements found in the Spice input file are converted to MDL (Measurement Description Language) before being passed on to Spectre. The conversion is done via the creation of an MDL control file, which is written to the current directory. This control file takes on the name of the netlist, with a suffix of `.mdl`. The simulation that immediately follows the conversion uses the MDL option to Spectre with this control file to evaluate the `.measure` expressions. Results are written to a file with the same name as the control file (netlist) with a `.measure` suffix. For example, the `amp.sp` SPICE netlist above creates a control file named `amp.mdl`, and a measurement output file named `amp.measure`. You simply enter the `spectrespp` command with the netlist, and the rest of the steps are completed for you automatically. The MDL control file is left behind for future use.

**Note:** In previous releases, the command line option `+spp` was used to invoke the SPICE Reader. While this method is still supported, it is advised that you use the `spectrespp` command if `.measure` statements are included in the input file. For more information on MDL, see the `SpectreMDL User Guide and Reference`.

For backward compatibility, a limited SPICE2 netlist reader is embedded into the Spectre netlist reader. This reader supports much of SPICE2G6 syntax and has as its primary use the processing of SPICE-style model cards. If the SPICE Reader is not used, this limited SPICE netlist reader is used. This is only for backwards compatibility and might be removed in a future release, to be replaced by the SPICE Reader. This limited SPICE netlist reader is no longer documented as of the 4.4.3 release. In the rest of this chapter, “SPICE Reader” refers to the full SPP SPICE Reader, and not this limited version.

### Running the SPICE Reader from Analog Artist

When using Spectre within Analog Artist, the need to read SPICE netlists is usually limited to model files and information from extraction tools that comes into the netlist by included files. For all other design data, Analog Artist produces netlists in the Spectre native syntax. You can enable the use of the SPICE Reader by choosing the Setup → Environment form and toggling the Use SPICE Netlist Reader (spp) button to Y. Be sure that you do not have any relative paths specified for included files, such as model files, and that the Include Path field in the Setup → Simulation Files field is blank. The SPICE Reader’s inability to handle relative paths is documented in the `Spectre Known Problems and Solutions`.

### Running the SPICE Reader from the Command Line

When you run Spectre from the command line (outside of the Artist/Composer environment), the SPICE Reader is disabled by default. As described in “Running the SPICE Reader” on
page 42, you must use the `spectrespp` command for the SPICE Reader to process the top-level netlist:

```
spectrespp opamp.sp
```

This command calls Spectre with the `spp` and `mdl` command line options. There are several command line options that can be used in conjunction with `spectrespp`:

### Specifying the design file

The command line option `-design` is used to identify the input file. In the absence of any command line options, the single argument after the `spectrespp` command is assumed to be the input netlist. Therefore the following commands are identical:

```
spectrespp opamp.sp
spectrespp -design opamp.sp
```

### Specifying the spp binary to be used

By default, the `spp` binary that is located in the Cadence hierarchy, `<CDS_INSTALL>/tools/dfII/bin/spp`, is used by the `spectrespp` command. You can specify the use of other `spp` executables with the `-sppbin` command line option. In the following example, `spp` from the current directory is used:

```
spectrespp -sppbin ./spp opamp.sp
```

### Specifying the location of the measurement output

Also by default, the output of the `.measure` statements is written to a file with the base name of the netlist and a suffix of `.measure`. In the example of `opamp.sp`, a measurement file named `opamp.measure` would be created. This can be overridden with the `-measure` command line option:

```
spectrespp -measure design.measure -design opamp.sp
```

Notice that when specifying the `-measure` option, you must also specify the `-design` option to identify the netlist.

All other command line options are passed to Spectre. Please type `spectrespp -usage` for the online help.
Using the SPICE Reader to Convert Netlists

In many cases, the SPICE data to be used with Spectre is a library of models or reusable blocks (that is, subcircuits). Rather than using the SPICE Reader to interpret these files each time they are used, you can convert the netlist to Spectre's native syntax. This is done by running the SPICE Reader alone (spp) and using the -convert option:

```
spp -convert <netlist.spice> netlist.spectre
```

An example is given below.

```
spp -convert < amp.sp > amp.scs
```

converts the SPICE netlist `amp.sp` to an equivalent Spectre netlist with the name `amp.scs`.

The -convert option removes special embedded codes for tracking the line numbers of the original netlist, returning a cleanly converted netlist. This option does not descend into included files to convert them but only converts the include statement. For this reason, included files must be converted independently. In this case, the SPICE Reader cannot always resolve conflicts and ambiguities, and the resulting netlist files must be verified. Great care must be taken when converting hierarchically arranged netlists. Some of the possible issues include the following:

- **Namespace conflicts**

  Spectre has a single namespace for instance names, nodes, master/model names, and parameters. When running on an entire netlist, the SPICE Reader can map these name conflicts. In the convert mode, where only sections of the netlist can be considered, `spp` might not resolve all conflicts.

- **Distinguishing instance models from instance parameters**

  For some devices, such as resistors, SPICE instance statements can use an order-dependent mechanism to define the values of parameters. An example could be

  ```
  r1 1 0 myres
  ```

  The field `myres` could be either a model name or a parameter representing the resistance value in SPICE. The SPICE Reader maps this to a parameter if a `.param` statement is encountered with this assignment or to a model name if a model card with the name `myres` is found. If neither is present, the field is assumed to be a model name. In the case of the -convert mode, where these assignments could be part of included files, this assumption could be incorrect, and the instance statement would need to be fixed.

- **Global nodes**

  In Spectre, global nodes must be at the very beginning of the netlist, while they can be present anywhere in a SPICE netlist. If a converted file contains global nodes, these
must be moved to the top level of the netlist before the file can be included in a higher level netlist. Multiple global statements cause fatal errors in Spectre.

**Location of measurement data**

The SPICE Reader converts Hspice style `.measure` statements to Measurement Description Language (MDL) statements. The SPICE Reader does this by creating an MDL control file, which can then be used in the simulation. In normal operation (using the `sptrespp <netlist>` command), the control file that is created takes on the name of the netlist, with the suffix `.mdl`. When a netlist is converted with the `-convert` option, the MDL control file that is created is always named `Netlist.mdl`. If this control file is used in a subsequent simulation, the results output is written to a file named `Netlist.measure`. For example, say that you have an SPICE netlist named `amp.sp`. In order to convert this netlist without executing the simulation, use `spp` with the `-convert` option:

```
spp -convert < amp.sp > amp.scs
```

Upon completion of this command, the Spectre equivalent for `amp.sp` is in the `amp.scs` file. In addition, a file called `Netlist.mdl` is created. This additional file contains the MDL expressions that correspond to any `.measure` statements in the original netlist. In order to run the Spectre simulation using the new netlist and control file, use the following command:

```
spectremdl -batch Netlist.mdl -design amp.scs
```

Upon completion of the simulation, you’ll find a file named `Netlist.measure`, which contains the values calculated for each of the MDL expressions.

For more information on the expression evaluation capabilities in Spectre, see the *SpectreMDL User Guide and Reference*.

**Netlist parameters**

In Spectre, parameters are order dependent; that is, they must be defined before being used. When the SPICE Reader is run on the entire design, parameters are moved to the top of the netlist, but, in convert mode, they are moved to the top of each file. If a parameter is accessed before inclusion, it will fail until it is fixed.

**Order-dependent statements**

Some statements in SPICE are mapped into multiple statements in Spectre. Things like temp sweeps, analysis sweeps, and alter sections all get converted into multiline statements. If temp statements, analysis statements, or alter statements are not in the same file, they might not get properly unfolded.

Here is an example of a Mos characterization deck written in SPICE:

```plaintext
M1 1 2 0 0 NCH
VDR 1 0 2
```
VG 2 0 1
.OP
.MODEL NCH NMOS LEVEL=49
+ TOX=1e-9 VTH0=0.5 VBOX=5
.TEMP 25 75 100
.INCLUDE "dcAnalysis.inc"
.END

where dcAnalysis.inc is
.DC VDR 0 5 0.1 VG 0 2 0.1

The netlist produced with spp -convert does not include the sweep statement within the temperature sweep:

// Mos characterization deck
simulator lang=spectre insensitive=yes
m1 ( 1 2 0 0 ) nch
vdr ( 1 0 ) vsourcB dc=2
vg ( 2 0 ) vsource dc=1
model nch bsim3v3 type=n tox=1e-9 vth0=0.5 vbox=5
include "dcAnalysis.inc"
swp_tmp1 sweep param=temp values=[25 75 100] {
  sppSaveOptions options save=allpub
  analysisOP1 dc oppoint=logfile
}
// end

You need to edit the netlist to insert the included analysis into the temperature sweep:

// Mos characterization deck
simulator lang=spectre insensitive=yes
m1 ( 1 2 0 0 ) nch
vdr ( 1 0 ) vsourcB dc=2
vg ( 2 0 ) vsource dc=1
model nch bsim3v3 type=n tox=1e-9 vth0=0.5 vbox=5
swp_tmp1 sweep param=temp values=[25 75 100] {
  sppSaveOptions options save=allpub
  analysisOP1 dc oppoint=logfile
  analysisDCswp1 sweep param=dc dev=vg start=0 stop=2 step=0.1  {
    analysisDCswp2 dc param=dc dev=vdr start=0 stop=5 step=0.1
  }
}
// end

• Mixed language

Spectre allows the use of both SPICE and Spectre syntax in the same netlist. The SPICE Reader does not read the Spectre statements; thus, statements in the SPICE sections that are dependent on Spectre statements might get improperly mapped. Except for model and subcircuit names, do not have any dependencies between these two language sections.
Language Differences

Comparing the SPICE and Spectre Languages

When you run the SPICE Reader, all SPICE statements are mapped directly to Spectre input. There are several important differences between these two languages that are addressed by the SPICE Reader.

To get a better idea at the differences between the two languages, an example netlist is shown below. First is the SPICE version of the netlist for an oscillator (from “Sample Netlist” on page 26):

```
BJT ECP Oscillator
Vcc 1 0 5
Iee 2 0 1MA
Q1 1 3 2 NPN
Q2 4 5 2 NPN
L1 4 5 1UH
C1 1 5 1PF
C2 5 3 272.7PF
C3 3 0 3NF
C4 5 6 3NF
R1 3 0 10K
R2 5 6 10K
.ic V(3)=1
.model NPN NPN BF=80 RB=100
  + VAF=50 CJS=2PF TF=0.3NS TR=6NS CJE=3PF CJC=2PF
.option method=trap
.tran 1us 80us 10ns
```

Here is the Spectre version of the same netlist:

```
// BJT ECP Oscillator
simulator lang=spectre
Vcc (vcc 0) vsource dc=5
Iee (e 0) isource dc=1mA
Q1 (vcc b1 e) npn
Q2 (out b2 e) npn
L1 (vcc out) inductor l=1uH
C1 (vcc out) capacitor c=1pf
C2 (out b1 ) capacitor c=272.7pf
C3 (b1 0) capacitor c=3nf
C4 (b2 in ) capacitor c=3nf
R1 (b1 0) resistor r=10k
R2 (b2 in ) resistor r=10k
ic b1=1
.model npn bjt type=npn bf=80 rb=100 vaf=50 \ cjs=2pf tf=0.3ns tr=6ns cje=3pf cjc=2pf
OscResp tran stop=80us maxstep=10ns method=trap
```
This example shows several of the basic differences between SPICE and Spectre netlists:

- Spectre netlists define the language context at the beginning of the netlist with the `simulator lang=spectre` statement. By default, netlists are assumed to be in SPICE unless the `simulator lang=spectre` statement is encountered or unless the netlist file ends in the extension `.scs`. The SPICE Reader will insert this line at the beginning of each SPICE-to-Spectre converted netlist.

- In the SPICE language, the component type is determined by the first letter of the component name, which limits the component types to 26. The Spectre Netlist Language has unlimited component types since the component type is determined by the master name instead of the first letter of the component name. An example of using the master name is the `resistor` field in the instance statements for R1 and R2.

- In many versions of SPICE, the component name, node numbers, and other fields are limited to a maximum of 8 characters. The Spectre netlister allows each of these fields to be greater than 200 characters.

- In the SPICE language, instance statements often have their parameter values listed in a particular order. In the Spectre language, you can specify parameter values in any order using `parameter name =`. This allows for a more readable netlist.

- Spectre netlists are case sensitive and use lower case for all keywords. SPICE netlists are case insensitive. This means that you can name one model `myModel` and another `mymodel`, and Spectre treats them as two distinct models. In SPICE, they would be considered to be the same model.

  Spectre always tries to match the case of a model/subcircuit’s name when there is a call to a model/subcircuit from an instance line, an `alter` statement, or other part of the netlist. During the netlist conversion process, the SPICE Reader converts included models/subcircuits to lower case, which can prevent Spectre from identifying the right model/subcircuit for a given device. For this reason, the SPICE Reader will insert the flag `insensitive=yes` at the beginning of each converted netlist on the simulator command statement (that is, `simulator lang=spectre insensitive=yes`). The `insensitive=yes` language directive marks the models/subcircuits that were converted so that Spectre can perform a lower case compare.

- Because the SPICE language is case insensitive, the scale factors are different from the Spectre scale factors (see “Scaling Numerical Literals” on page 81 for more information). The Spectre Netlist Language uses the ANSI standard (SI) scale factors.

- Options that are analysis specific are found on the global `.options` card in SPICE. In Spectre, options pertaining to an analysis are listed on the analysis card.

- Spectre supports the idea of named analyses, which allows you to include multiple analyses of the same type in a single netlist. The SPICE Reader takes advantage of Spectre’s ability to have more than one analysis of each type, mapping each analysis
statement to the equivalent Spectre analysis and providing each a unique name. If multiple analysis of the same type are listed within the SPICE deck, each will be simulated instead of the “last one wins” mode of SPICE

■ In both Spectre and Spice, the first line is always the title card and ignored by the simulator.

■ A card (other than the title or a comment) can be continued onto the next card by putting the continuation character (+) in the first column of the next card. Spectre also supports continuations at the end of the line in its native mode, with the backslash character (\).

■ In many versions of Spice, there are punctuation characters (such as parentheses and commas) that are treated as white space. Spectre will treat all characters as relevant input. The SPICE Reader will treat all of these characters as literals.

■ The Spectre simulator does not require an .end statement; it assumes the end of the input file is the end of the input statements. The .end statement, if included in a SPICE netlist, is treated like the end of file; and the SPICE Reader does not read beyond the .end.

Reading SPICE and Spectre Files

When the Spectre simulator opens a netlist file, the simulator determines whether the file is a SPICE or Spectre netlist and interprets it accordingly. This is true regardless of whether the file is the main netlist or an included netlist. The Spectre simulator recognizes Spectre files because they start with the line simulator lang=spectre or the filename ends in the Spectre circuit simulator extension of .scs. Files that do not start with this line or end in the .scs extension are taken to be SPICE files. Thus, SPICE files do not need to be modified to work with the Spectre simulator as the SPICE Reader will automatically map them correctly to Spectre input. The .scs extension allows users to write netlists, included files, and library files using only the Spectre Netlist Language without having to switch from SPICE mode.

To learn more about controlling the input language, see “General Input Compatibility" on page 51.

Scope of the Compatibility

Many current variants of SPICE have their own alterations to the SPICE input language. The SPICE Reader in Spectre does support many of these variations using the following sources and guidelines.
Sources

- The *SPICE Version 2G User’s Guide* [vladimirescu81] is the reference for the SPICE2 input language. The SPICE Reader accepts input formats from this document for all SPICE capabilities the Spectre simulator supports.

  The SPICE Reader might not accept variations from SPICE 2G6 that are not in the user guide.

- *The SPICE3 Version 3e User’s Manual* [B.JohnsonXX] is the reference for SPICE3 batch input language. The SPICE Reader accepts input formats (batch mode only) from this document for all SPICE capabilities the Spectre simulator supports.

- *The SPICE BOOK* [vladimirescu81] is a general reference for SPICE2/3-based syntax’s with common extensions.

- Publicly available netlists provide references for many of the extensions to SPICE.

Compatibility Guidelines

- Mapping of SPICE must not adversely affect simulation results.

- Where simulators follow different syntax rules, support for each is attempted. For example, the TC property is an instance property in SPICE2 and model parameters (TC1,TC2) in SPICE3—both are supported.

- When a conflict exists between simulator syntax rules, one method is selected and documented.

General Input Compatibility

By default, the SPICE Reader processes its input in the following manner:

- Accepts the SPICE instance (A-Z) and control (dot) cards as well as the Spectre native input language.

- Reads in the first line as the title card.

- Generally ignores the case of alphabetic characters in the input (The program internally converts them to lowercase). Quoted strings are exceptions to this rule.

- Interprets scale factors following numerical values according to SPICE conventions.

- Accepts the SPICE comment and continuation conventions.
Spectre Circuit Simulator User Guide
SPICE Compatibility

- Accepts the /, +, -, and * characters as valid parts of a node or an instance or parameter name. You can also use the Escape backslash convention to include otherwise illegal characters in the names.

- Spectre statements are not allowed in the SPICE mode, and SPICE statements are not allowed in the Spectre mode.

- In simulator lang=spectre mode, you can refer to model names and subcircuit names defined with the simulator lang=spice mode .model and .subckt statements, even though they are in another portion of the netlist.

Spectre always tries to match the case of a model/subcircuit’s name when there is a call to a model/subcircuit from an instance line, an alter statement, or other part of the netlist. During the netlist conversion process, the SPICE Reader converts included models/subcircuits to lower case, which can prevent Spectre from identifying the right model/subcircuit for a given device. For this reason, the SPICE Reader will insert the flag insensitive=yes at the beginning of each converted netlist on the simulator command statement (that is, simulator lang=spectre insensitive=yes). The insensitive=yes language directive marks the models/subcircuits that were converted so that Spectre can perform a lower case compare.

- The SPICE Reader treats the * character as a math function if found inside of an expression and as a comment if found outside of an expression.

- Lines beginning with #<CPP keywords> are not treated as comments by the SPICE Reader but will result in an syntax error unless you run CPP on the netlist.

- You can force the SPICE Reader to treat all bjt devices as having four terminals with the -q4term command line option to spp. By default, the fourth field after the device name of an instance statement is treated as a terminal, unless one of the following occurs:
  - There is a model name within the netlist that is named the same as the fourth field.
  - It is the last item on the instance line.

For example:

```
q1 1 2 3 4 mybjt // The 4 is treated as a device terminal
q1 1 2 3 nch mybjt
```

The nch is treated as a device terminal unless

- mybjt is defined as a parameter in the netlist
- nch is defined as a model in the netlist

In the second case, the -q4term option can be used to override the default behavior of the converter.
You can gain full access to Spectre and its features with a SPICE netlist by controlling the input language. Spectre provides three ways to gain access to its language in a SPICE netlist:

1. Change the input language to Spectre using the `simulator lang` command
2. Append Spectre parameters to a SPICE statement
3. Hide Spectre text from SPICE

Each of these methods is discussed in the following sections.

**Spectre’s simulator lang Command**

You can use the Spectre Netlist Language or an extended form of the SPICE2/3 input language for your simulation. You specify which language with a `simulator lang` command.

You can change the input-processing mode with the `simulator lang` command. A `simulator lang=spice` command causes subsequent input to be processed in the SPICE extension mode; a `simulator lang=spectre` command causes the input processing to conform to the Spectre Netlist Language. (Remember that you can give the `simulator lang` command anywhere that you can enter an instance statement, and it affects all subsequent input regardless of whether the command is part of an `if` statement.) Use of Spectre syntax in the SPICE mode or SPICE syntax in the Spectre mode results in an error.

The `simulator lang` command has the following format:

```
simulator lang=spice
simulator lang=spectre
```

Although the `simulator lang` command is expected to be the first line in the netlist, it does not have to be. The commands take effect immediately and remain in effect until the end of input or until the next `simulator lang` command. For example, you can include Spectre lines in the middle of a SPICE netlist by surrounding the Spectre lines with the `simulator lang` command:

```
r1 1 0 1k
v1 1 0 1
simulator lang=spectre
r2 1 0 resistor r=1k
simulator lang=spice
.op
```

If you do not specify an input language, the Spectre simulator uses SPICE mode by default. You do not need to place `simulator lang` commands around `include` statements, but you must specify `simulator lang=spectre` in include files written in the Spectre Netlist Language.
Spectre Circuit Simulator User Guide
SPICE Compatibility

Spectre allows you to specify the netlist input format to be Spectre without the simulator lang command by naming the file with the Spectre circuit simulator extension of .scs. Files ending in this extension are treated as Spectre syntax.

Appending Spectre Parameters

While you can gain full access to Spectre features by switching to the Spectre syntax, you can also gain access to specific parameters on the SPICE input statements. The SPICE Reader maps each SPICE statement to the Spectre equivalent statement. Mapping of parameters is done first for order-dependent parameters and then for name=value based parameters. The SPICE reader tries to map parameters in the name=value form if they are known; otherwise, they are passed directly to Spectre as is. This allows you to add Spectre parameters to the end of any SPICE statement. For example,

* Additional Spectre parameter to resistor
  Vin input 0 dc=1 AC 1 0
  X1 mainnetlist in out
  Rload out 0 1k isnoisy=no //isnoisy is spectre parameter
  .include "./main.ckt"
  .ac dec 10 1k 100k
  .noise v(out)
  .end

If the netlist is to be run by a SPICE simulator at a later time, these Spectre parameters must be hidden or removed. See the next section for information on how to hide Spectre text.

Hiding Spectre Text from SPICE

You can create a SPICE-mode input file that you can use with both a SPICE-based simulator and the Spectre simulator. When you run the Spectre simulator, you can use Spectre features unavailable in SPICE. For example, if you want to specify a Spectre parameter that SPICE does not support, you write the Spectre statements so they do not cause syntax errors when you run the SPICE-based simulator. To create such an input file, follow these steps:

1. Insert the special prefix *spectre: at the beginning of a line in your SPICE netlist.

   The line should start with an asterisk, optionally followed by blanks or tabs, followed by the string spectre: .

   * spectre: (Spectre text)

   Be sure to leave a space after the colon.

2. Place Spectre-specific input language statements after the : Note special prefix.

   The Spectre simulator recognizes the Spectre statements after the prefix as input, but the SPICE-based simulator ignores the whole line as a comment.

   *spectre: Rp 15 0 resistor r=50m
SPICE Compatibility

**Note:** These statements must be in the Spectre context of a netlist.

3. If you need to place the Spectre input in continuation lines, begin each line of a continuation with a (+).

   Vbias vbb 0 1.2
   *spectre: + tc=0.001
   .options reltol=0.001
   *spectre: + diagnose=yes

**Compatibility Limitations**

**Conflicts between Spectre and Spice**

- The Spectre simulator does not support all SPICE capabilities (for example, the `.disto` and `.PZ` analyses).

- Slight modifications are often necessary to make input acceptable to both languages. Superficial punctuation differences, such as ignoring extra parentheses or equal signs (=), fit this category. For example, the statement `.temp = 125`, which is legal in SPICE, cannot be mapped by the SPICE Reader. Removing the extraneous punctuation solves the problem.

- Compatibility is limited in the area of unsupported primitive devices in Spectre. You are encouraged to use the extensive capabilities of Verilog-A to model such devices.

- Parameters that are to be passed into a subcircuit from an instance statement must be specified on the SPICE subcircuit line. Allowing the instance to define a subcircuit parameter is not allowed in Spectre.

- Incompatibilities remain for some rarely used SPICE2 features because the authors have concentrated their efforts on creating compatibility with commonly used features.

**Conflicts between SPICE variations**

- Node 00 is treated as a number (that is, zero) in SPICE2 and a string (that is, 00) in SPICE3 and thus are interpreted differently. The SPICE Reader treats these like SPICE3.

- Leading spaces are ignored in SPICE2 and treated as a comment in SPICE3. The SPICE Reader treats these as SPICE2 does.

- `.lib` in PSPICE provides a file to search for each instantiated model; in HSPICE, it acts as a section-specific include statement. The SPICE Reader treats these as HSPICE does.
.alter in SPICE2 allows multiple lines to be relisted with deltas following this statement. In SPICE3, the .alter is an interactive command and only allows one parameter to be changed per alter statement. The SPICE Reader treats these as SPICE2 does.

In some versions of SPICE, global parameters values override values specified on instances of subcircuits. The SPICE Reader treats all parameter values specified on any instance including subcircuits as the final value instead of allowing a global to override it.

For more information on specific SPICE Reader limitations, please see the SPP Known Problems and Solutions on SourceLink.
Spectre Netlists

This chapter discusses the following topics:

- Netlist Statements on page 57
- Instance Statements on page 61
- Analysis Statements on page 64
- Control Statements on page 66
- Model Statements on page 68
- Input Data from Multiple Files on page 70
- Multidisciplinary Modeling on page 73
- Inherited Connections on page 75

Netlist Statements

A Spectre® circuit simulator netlist describes the structure of circuits and subcircuits by listing components, the nodes that the components are connected to, the parameter values that are used to customize the components, and the analyses that you want to run on the circuit. You can use SpectreHDL or Verilog®-A to describe the behavior of new components that you can use in a netlist like built-in components. You can also define new components as a collection of existing components by using subcircuits.

A netlist consists of four types of statements:

- Instance Statements on page 61
- Analysis Statements on page 64
- Control Statements on page 66
- Model Statements on page 68
Before you can create statements for a Spectre netlist, you must learn some basic syntax rules of the Spectre Netlist Language.

**Netlist Conventions**

These are the netlist conventions followed by the Spectre simulator.

**Associated Reference Direction**

The reference direction for the voltage is positive when the voltage of the + terminal is higher than the voltage of the – terminal. A positive current arrives through the + terminal and leaves through the – terminal.

![Current](image)

**Ground**

Ground is a common reference point in an electrical circuit. The value of ground is zero.

**Node**

A node is an infinitesimal connection point. The voltage everywhere on a node is the same. According to Kirchhoff’s Current Law (also known as Kirchhoff’s Flow Law), the algebraic sum of all the flows out of a node at any instant is zero.

**Basic Syntax Rules**

The following syntax rules apply to all statements in the Spectre Netlist Language:
Field separators are blanks, tabs, punctuation characters, or continuation characters.

Punctuation characters such as equal signs (=), parentheses (()), and colons (::) are significant to the Spectre simulator.

You can extend a statement onto the next line by ending the current line with a backslash (\). You can use a plus sign (+) to indicate line continuation at the beginning of the next line. The preferred style is the \ at the end of the line.

You can indicate a comment line by placing a double slash (//) at the beginning of the comment. The comment ends at the end of that line. You can also use an asterisk (*) to indicate a comment line. The preferred style is using //.

You can indicate a comment for the remainder of a line by placing a space and double slash (//) anywhere in the line.

**Spectre Language Modes**

The Spectre netlist supports two language modes:

- Spectre mode (mostly discussed here)
- SPICE mode

You can specify the language mode for subsequent statements by specifying `simulator lang=mode`, where `mode` is `spectre` or `spice`.

Spectre mode input is fully case sensitive. In contrast, except for letters in quoted strings, SPICE mode converts input characters to lowercase.

**Creating Component and Node Names**

When you create node and component names in the Spectre simulator, follow these rules.

- Use unique names for components and nodes. The Spectre Netlist Language uses the same syntax for the names for nodes, device instances, models, nets, subcircuit parameters, enumerated named values for parameters, and netlist parameters that are expressions, so there is no way for the Spectre simulator to distinguish between duplicate names.
- Do not use the name of an analysis (such as `tran`) or built-in primitive (such as `resistor`).
- Except for node names, names must begin with a letter or underscore. Node names can also be integers.
■ If the node name is an integer, leading zeros are significant in the Spectre language mode.

■ Names can contain any number of letters, digits, or underscores.

■ Names cannot contain nonalphanumeric characters, blanks, tabs, punctuation characters, or continuation characters (\, +, -, //, *) unless they are escaped with a backslash (\). The exception is bang (!), for example, vdd!.

■ The following reserved words (case specified) should not be used as node names, net names, instance names, model names, or parameter names.

```
M_1_PI  M_2_PI  M_2_SQRTPi  M_DEGPERRAD
M_EM_LN10M_LN2M_LOG10E
M_LOG2EM_PI  M_PI_2M_PI_4
M_SQRT1_2  M_SQRT2M_TWO_PI  P_C
P_CELSIUS0  P_EPS0P_HP_K
P_Q  P_U0abs  acos
acosh  altergroup  asin  asinh
atanatan2  atanh  ceil
correlate  cos  cosh  else
end  endsexp  export
floor  for  function  global
hypot  ic  if  inline
int  library  local  log
log10  march  max  min
modelnodeset  parameters  paramset
plot  pow  print  pwr
real  returnsave  sens
sinsinh  sqrt  statistics
subckt  tan  tanh  to
truncate  vary
```
Escaping Special Characters in Names

If you have old netlists that contain names that do not follow Spectre syntax rules, you might still be able to run these netlists with the Spectre simulator. The Spectre Netlist Language permits the following exceptions to its normal syntax rules to accommodate old netlists. Use these features only when necessary.

If you place a backslash (\) before any printable ASCII character, including spaces and tabs, you can include the character in a name.

You can create a name from the following elements in the order given:

A string of digits, followed by
- Letters, underscores, or backslash-escaped characters, followed by
- A digit, followed by
- Underscores, digits, or backslash-escaped characters

This accommodates model or subcircuit libraries that use names like \2N2222.

Instance Statements

In this section, you will learn to place individual components into your netlist and to assign parameter values for them.

Formatting the Instance Statement

To specify components, you use the instance statement. You format the instance statement as follows:

```
name [(node1 ... nodeN[])] master [[param1=value1]
...[paramN=valueN]]
```

When you specify components with the instance statement, the fields have the following values:

- `name`: The unique name you give to the statement. (Unlike SPICE instance names, the first character in Spectre instance names is not significant.)

- `[(node1 ... nodeN[])]:` The names you give to the nodes that connect to the component.
You have the option of putting the node names in parentheses to improve the clarity of the netlist. (See the examples later in this chapter.)

master

This is the name of one of the following:
A built-in primitive (such as resistor)
A model
A subcircuit
An AHDL module (SpectreHDL or Verilog-A language)

Note: The instance statement is used to call subcircuits and refer to AHDL modules as well as to specify individual components. For more information about subcircuit calls, see “Subcircuits” on page 90. For more information about the SpectreHDL product, see the SpectreHDL Reference manual. For more information about Verilog-A, see the Verilog-A Language Reference manual.

parameter1=value1...parameterN=valueN

This is an optional field you can repeat any number of times in an instance statement. You use it to specify parameter values for a component. Each parameter specification is an instance parameter, followed by an equal sign, followed by your value for the parameter. You can find a list of the available instance parameters for each component in the Spectre online help (spectre -h). As with node names, you can place optional parentheses around parameter specifications to improve the clarity of the netlist. For example, (c=10E-12).

For subcircuits and ahdl modules, the available instance parameters are defined in the definition of the subcircuit or AHDL module. In addition, all subcircuits have an implicit instance parameter \( m \) for defining multiplicity. For more details about \( m \), see “Identical Components or Subcircuits in Parallel” on page 63.

Examples of Instance Statements

In this example, a capacitor named \( c1 \) connects to nodes 2 and 3 in the netlist. Its capacitance is 10E-12 Farads.

\[ c1 \ (2 \ 3) \ \text{capacitor} \ c=10E-12F \]

The following example specifies a component whose parameters are defined in a model statement. In this statement, \( npn \) is the name of the model defined in a model statement that
contains the model parameter definitions; \( Q1 \) is the name of the component; and \( o1, i1, \) and \( b2 \) are the connecting nodes of the component.

\[ Q1 \ (o1 \ i1 \ b2) \ npn \]

**Note:** The **model** statement is described in “Model Statements” on page 68. You can specify additional parameters for an individual component in an instance statement that refers to a **model** statement. You can find a list of available instance parameters and a list of available model parameters for a component in the Spectre online help for that component (spectre-h).

### Basic Instance Statement Rules

When you prepare netlists for the Spectre simulator, remember these basic rules:

- You must give each instance statement a unique name.
- If the *master* is a model, you need to specify the model.

### Identical Components or Subcircuits in Parallel

If your circuit contains identical devices (or subcircuits) in parallel, you can specify this condition easily with the multiplication factor (\( m \)).

### Specifying Identical Components in Parallel

If you specify an \( m \) value in an instance statement, it is as if \( m \) identical components are in parallel. For example, capacitances are multiplied by \( m \), and resistances are divided by \( m \). Remember the following rules when you use the multiplication factor:

- You can use \( m \) only as an instance parameter (not as a model parameter).
- The \( m \) value need not be an integer. The \( m \) value can be any positive real number.
- The multiplication factor does not affect short-channel or narrow-gate effects in MOSFETs.
- If you use the \( m \) factor with components that naturally compute branch currents, such as voltage sources and current probes, the computed current is divided by \( m \). Terminal currents are unaffected.
- You can set the built-in \( m \) factor property on a subcircuit to a parameter and then alter it.
Example of Using m to Specify Parallel Components

In the following example, a single instance statement specifies four 4000-Ohm resistors in parallel.

\[ R_0 \ (d \ c) \ \text{resistor} \ r=4k \ m=4 \]

The preceding statement is equivalent to

\[ R_0 \ (d \ c) \ \text{resistor} \ r=1k \]

Specifying Subcircuits in Parallel

If you place a multiplication factor parameter in a subcircuit call, you model \( m \) copies of the subcircuit in parallel. For example, suppose you define the following subcircuit:

```
subckt LoadOutput a b
   r1 (a b) resistor r=50k
   c1 (a b) capacitor c=2pF
ends LoadOutput
```

If you place the following subcircuit call in your netlist, the Spectre simulator models five \( \text{LoadOutput} \) cells in parallel:

\[ x1 \ (\text{out} \ 0) \ \text{LoadOutput} \ m=5 \]

Analysis Statements

In this section, you will learn to place analyses into your netlist and to assign parameter values for them. For more information on analyses, see Chapter 6, “Analyses,” and the Spectre online help (spectre -h).

Basic Formatting of Analysis Statements

You format analysis statements in the same way you format component instance statements except that you usually do not put a list of nodes in analysis statements. You specify most analysis statements as follows:

\[ \text{Name} \ [(\text{node}_1 \ldots \text{node}_N[])] \ \text{Analysis Type parameter=value} \]

\text{Name} is a unique name you give to the analysis.

\[(\text{node}_1\ldots\text{node}_N[])\] are the names you give to the nodes that connect to the analysis. You have the option of putting the node names in parentheses to improve the clarity of the netlist. (See the examples later in this chapter.) For most analyses, you do not need to specify any nodes.
**Analysis Type** is the Spectre name of the type of analysis you want, such as ac, tran, or xf. You can find this name by referring to the topics list in the Spectre online help (spectre -h).

`parameter=value` is the list of parameter values you specify for the analysis. You can specify values for any number of parameters. You can find parameter listings for an analysis by referring to the Spectre online help (spectre -h).

**Note:** The noise, xf, pnoise, and pxf analyses let you specify nodes, p and n, which identify the output of the circuit. When you use this option, you should use the full analysis syntax as follows:

```
Name> [p n] Analysis Type parameter=value
```

If you do not specify the p and n terminals, you must specify the output with a probe component.

**Examples of Analysis Statements**

The following examples illustrate analysis statement syntax.

```
XferVsTemp xf start=0 stop=50 step=1 probe=Rload \ 
   param=temp freq=1kHz
```

This statement specifies a transfer function analysis (xf) with the user-supplied name `XferVsTemp`. With all transfer functions computed to a probe component named `Rload`, it sweeps temperature from 0 to 50 degrees in 1-degree steps at frequency 1 kHz. (For long statements, you must place a backslash (\) at the end of the first line to let the statement continue on the second line.)

```
Sparams sp stop=0.3MHz lin=100 ports=[Pin Pout]
```

This statement requests an S-parameter analysis (sp) with the user-supplied name `Sparams`. A linear sweep starts at zero (the default) and continues to .3 MHz in 100 linear steps. The `ports` parameter defines the ports of the circuit; ports are numbered in the order given.

The following example statement demonstrates the proper format to specify optional output nodes (`p n`):

```
FindNoise (out gnd) noise start=1 stop=1MHz
```

**Basic Analysis Rules**

When you prepare netlists for the Spectre simulator, remember these basic analysis rules:
The Spectre simulator has no default analysis. If you do not put any analysis statements into a netlist, the Spectre simulator issues a warning and exits.

For most analyses, if you specify an analysis that has a prerequisite analysis, the Spectre simulator performs the prerequisite analysis automatically. For example, if you specify an AC analysis, the Spectre simulator automatically performs the prerequisite DC analysis. However, if you want to run a pac, pxf, or pnoise analysis, you must specify the prerequisite pss analysis.

You specify analyses in the order you want the Spectre simulator to perform them.

You can perform more than one of the same type of analysis in a single Spectre run. Consequently, you can perform several analyses of the same type and vary parameter values with each analysis.

You must give each analysis or control statement a unique name. The Spectre simulator requires these unique names to identify output data and error messages.

Control Statements

The Spectre simulator lets you place a sequence of control statements in the netlist. You can use the same control statement more than once. Spectre control statements are discussed throughout this manual. The following are control statements:

- alter
- altergroup
- check
- ic
- info
- nodeset
- options
- paramset
- save
- set
- shell
- statistics
Formatting the Control Statement

Control statements often have the same format as analysis statements. Like analysis statements, many control statements must have unique names. These unique names let the Spectre simulator identify the control statement if there are error messages or other output associated with the control statement. You specify most control statements as follows:

\(<\text{Name}>\ <\text{Control Statement Type}>\ <\text{parameter=value…}>\)

\(<\text{Name}>\) is a unique name you give to the control statement.

\(<\text{Control Statement Type}>\) is the Spectre name of the type of control statement you want, such as \text{alter}. You can find this name by referring to the topics list in the Spectre online help (\text{spectre-h}).

\(<\text{parameter=value…}>\) is the list of parameter values you specify for the control statement. You can specify values for any number of parameters. You can find parameter listings for a control statement by referring to the Spectre online help (\text{spectre-h}).

Examples of Control Statements

\text{SetTemp alter param=temp value=27}

The preceding example of an \text{alter} statement sets the temperature for the simulation to 27˚C. The name for the \text{alter} statement is \text{SetTemp}, and the name of the control statement type is \text{alter}.

You cannot alter a device from one primitive type to another. For example,

\text{inst1 (1 2) capacitor c=1pF}
\text{alterfail altergroup{}
  \text{inst1 (1 2) resistor r=1k}
\text{}}

is illegal.

Another example of a control statement is the \text{altergroup} statement, which allows you to change the values of any modifiable device or netlist parameters for any analyses that follow. Within an alter group, you can specify parameter, instance, or model statements; the corresponding netlist parameters, instances, and models are updated when the \text{altergroup} statement is executed. These statements must be bound within braces. The opening brace is required at the end of the line defining the alter group. Alter groups cannot be nested or be instantiated inside subcircuits. Also, no topology changes are allowed to be specified in an alter group.

The following is the syntax of the \text{altergroup} statement:
Name altergroup ... {
    <netlist parameter statements> ...
}
and/or
    <device instance statements> ...
and/or
    <model statements> ...
and/or
    <parameter statements> ...
}

The following is an example of the altergroup statement:

v1 1 0 vsource dc=1
R1 1 0 Resistor R=1k
dc1 dc // this analysis uses a 1k resistance value
a1 altergroup {
    R1 1 0 Resistor R=5k
}
dc2 dc // this analysis uses a 5k value

Model Statements

Model statements are designed to allow certain parameters, which are expected to be shared over many instances, to be given once. However, for any given component, it is predetermined which parameters can be given on model statements for that component.

This section gives a brief overview of the Spectre model statement. For a more detailed discussion on modeling issues (including parameterized models, expressions, subcircuits, and model binning), see Chapter 5, “Parameter Specification and Modeling Features.”

Formatting the model Statement

You format the model statement as follows:

model <name> <master> [[<param1>=<value1>] ... [[<param2>=<value2>]]

The fields have the following values:

model The keyword model (.model is used for SPICE mode).

<name> The unique name you give to the model.
The master name of the component, such as resistor, bjt, or tline. This field can also contain the name of an AHDL module. (For more information about using the Spectre simulator with SpectreHDL, see the SpectreHDL Reference manual.)

This is an optional field you can repeat any number of times in a model statement. Each parameter specification is a model parameter, followed by an equal sign, followed by the value of the parameter. You can find a list of the available model parameters for each component in the parameter listings of Spectre online help (spectre -h).

Examples of model Statements

The following examples give parameters for a tline model named tuner and a bjt model named NPNbjt.

```
model tuner tline f=1MHz alphac=9.102m dcr=105m
model NPNbjt bjt type=npn bf=100 js=0.1fA
```

**Note:** The backslash (\) is used as a continuation character in this lengthy model statement.

```
model NPNbjt2 bjt \
  type=npn is=3.38e-17 bf=205 nf=0.978 vaf=22 \ 
  ikf=2.05e-2 ise=0 ne=1.5 br=62 nr=1 var=2.2 isc=0 \ 
  nc=1.5 rb=115 re=1 rc=30.5 cje=1.08e-13 vje=0.995 \ 
  mje=0.46 tf=1e-11 xtf=1 itf=1.5e-2 cjc=2.2e-13 \ 
  vjc=0.42 mjc=0.22 xcjc=0.1 tr=4e-10 cjs=1.29e-13 \ 
  vjs=0.65 mjs=0.31 xtb=1.5 eg=1.232 xti=2.148 fc=0.875
```

The following example creates two instances of a bjt transistor model:

```
a1 (C B1 E S) NPNbjt
a2 (C B2 E S) NPNbjt2
```

**Basic model Statement Rules**

When you use the model statement,

- You can have several model statements for a particular component type, but each instance statement can refer to only one model statement.

- Occasionally, a component allows a parameter to be specified as either an instance parameter or as a model parameter. If you specify it as a model parameter, it acts as a default for instances. If you specify it as an instance parameter, it overrides the model parameter.
Values for model parameters can be expressions of netlist parameters.

Input Data from Multiple Files

If you want to use data from multiple files, you use the `include` statement to insert new files. When the Spectre simulator reads the `include` statement in the netlist, it finds the new file, reads it, and then resumes reading the netlist file.

If you have older netlist files you want to incorporate into your new, larger netlist, the `include` statement is particularly helpful. Instead of creating a completely new netlist, you can use the `include` statement to insert your old files into the netlist at the location you want.

**Note:** The Spectre simulator always assumes that the file being included is in SPICE language mode unless the extension of the filename is `.scs`.

Formatting the include Statement

You can use any of two formatting options for the `include` statement. When you want to use C preprocessor (CPP) macro-processing capabilities within your inserted file, use the second format (`#include`). These are the two format options:

```plaintext
include "filename"
#include "filename"
```

The first option (`include`) is performed by the Spectre simulator itself. The second option (`#include`) is performed by the CPP. You must use the `#include` option when you have macro substitution in the inserted file.

**Note:** CPP is not supported in Spectre Direct.

Rules for Using the include Statement

Remember the following rules and guidelines when using the `include` statement:

- You must use the `#include` format if you want the CPP to process the inserted file. Also, you must specify that the CPP be run using the `-E` command line option when you start the Spectre simulator.

- Regardless of which include format you use, you can use the `-I` command line option, followed by a path, to have the Spectre simulator look for the inserted files in a specified directory, in addition to the current directory, just as you would for the CPP `#include`. 
If \textit{filename} is not an absolute path specification, it is considered relative to the directory of the including file that the Spectre simulator is reading, not from the directory in which the Spectre simulator was called.

You must surround the \textit{filename} in quotation marks.

You can place a space and then a backslash-escaped newline (\) between include and \textit{filename} for line continuation.

You can place other include statements in the inserted file (recursive inclusion).

With any of the include formats, you can set the language mode for the inserted file by placing a \texttt{simulator lang} command at the beginning of the file. For more information about the \texttt{simulator lang} command, see Chapter 3, “SPICE Compatibility.” The Spectre simulator assumes that the file to be included is in the SPICE language unless one of the following conditions occurs:

- The file to be included has a \texttt{simulator lang=spectre} line at the beginning of the file. (The first line is not a comment title line, even in SPICE mode.)
- The file to be included has a .scs file extension.

With the include format, if you change the language mode in an inserted file, the language mode returns to that of the original file at the end of the inserted file.

You cannot start a statement in an original file and end it in an inserted file or vice versa.

You can use include "~/\textit{filename}". and the Spectre simulator looks for \textit{filename} in your home directory. This does not work for \texttt{#include}.

You can use environment variables in your include statements. For example,

\begin{verbatim}
include "$MYMODELS/filename"
\end{verbatim}

The Spectre simulator looks for \textit{filename} in the directory specified by \texttt{$MYMODELS}. This works for include, but not for \texttt{#include}.

There are two major differences between using \texttt{#include} and include:

- You can specify \texttt{#include} to run CPP and use macros and \#-defined constants.
- \texttt{#include} does not expand special characters or environment variables in the filename.

\textbf{Example of include Statement Use}

In the following include statement example, the Spectre simulator reads initial program options and then inserts two files, \texttt{cmos.mod} and \texttt{opamp.ckt}. After reading these files, it returns to the original file and reads further data about power supplies.
// example of using include statement
global gnd vdd vss
simulator lang=spectre
parameters VDD=5
include "cmos.mod"
include "opamp.ckt"
// power supplies
Vdd   vdd   gnd   vsource dc=VDD
Vss   vss   gnd   vsource dc=-VDD

Reading Piecewise Linear (PWL) Vector Values from a File

You could type the following component description into a netlist:

v4 in 0  vsrce type=pwl wave=[0 0 1n 0 2n 5 10n 5 11n 0 12n 0]

You could also enter the vector values from a file, in which case the component description might look like this:

v4 in 0  vsrce type=pwl file="test.in"

You can use the -I command line option, followed by a path, to have the Spectre simulator look for the inserted files in a specified directory if they cannot be found in the current directory.

If you place PWL vector values in an input file that is read by the component, do not specify scale factors in your parameter values.

If you use an input file, the values in the file must look like this—without scale factors:

0  0
1e-9  0
2e-9  5
10e-9  5
11e-9  0
12e-9  0

Using Library Statements

Another way to insert new files is to use the library statements. There are two statements: one to refer to a library and one that defines the library. A library is a way to group statements into multiple sections and selectively include them in the netlist by using the name of the section.

As for the include statement, the default language of the library file is SPICE unless the extension of the file is .scs; then the default language is the Spectre Netlist Language.
Library Reference

This statement refers to a library section. This statement can be nested. To see more information on including files, see `spectre -h include`. The name of the section has to match the name of the section defined in the library. The following is the syntax for library reference:

```
include "file" section=Name
```

where `file` is the name of the library file to be included. The library reference statement looks like an include statement, except for the specification of the library section. When the file is being inserted, only the named section is actually included.

Library Definition

The library definition has to be in a separate file. The library has to have a name. Each section in the library has to be named because this name is used by the library reference statement to identify the section to include. The `statements` within each section can be any valid statement. This is important to remember when using libraries in conjunction with alter groups because the `altergroup` statement is restrictive in what can be specified.

The optional names are allowed at the end of the section and library. These names must match the names of the section or library.

The following is the syntax for library definition:

```
library libraryName
    section sectionName
        <statements>
        endsection [sectionName]
    section anotherName
        <statements>
        endsection [anotherName]
library [libraryName]
```

One common use of library references is within `altergroup` statements. For example:

```
al altergroup {
    //change models to "FAST" process corner
    include "MOSLIB" section=FAST
}
```

Multidisciplinary Modeling

Multidisciplinary modeling involves setting tolerances and using predefined quantities.
Setting Tolerances with the quantity Statement

Quantities are used to hold convergence-related information about particular types of signals, such as their units, absolute tolerances, and maximum allowed change per Newton iteration. With the `quantity` statement, you can create quantities and change the values of their parameters. You set these tolerances with the `abstol` and `maxdelta` parameters, respectively. You can set the `huge` parameter, which is an estimate of the probable maximum value of any signal for that quantity. You can also set the `blowup` parameter to define an upward bound for signals of that quantity. If a signal exceeds the `blowup` parameter value, the analysis stops with an error message.

Generally, a reasonable value for the absolute tolerance of a quantity is $10^6$ times smaller than its greatest anticipated value. A reasonable definition for the `huge` value of a quantity is $10$ to $10^3$ times its greatest expected value. A reasonable definition of the `blowup` value for a quantity is $10^6$ to $10^9$ times its greatest expected value.

Predefined Quantities

The Spectre Netlist Language has seven predefined quantities that are relevant for circuit simulation, and you can set tolerance values for any of them. These seven predefined quantities are

- Electrical current in Amperes (named `I`)
  (Default absolute tolerance = 1 pA)
- Magnetomotive force in Amperes (named `MMF`)
  (Default absolute tolerance = 1 pA-turn)
- Electrical potential in Volts (named `V`)
  (Default absolute tolerance = 1 \( \mu \)V; Default maximum allowable change per iteration = 300mV)
- Magnetic flux in Webers (named `Wb`)
  (Default absolute tolerance = 1 nWb)
- Temperature in Celsius (named `Temp`)
  (Default absolute tolerance = 100 \( \mu \)C)
- Power in Watts (named `Pwr`)
  (Default absolute tolerance = 1 nW)
Unitless (named u)

(Default absolute tolerance = 1 x 10^{-6})

For more information, see spectre -h quantity.

**Quantity Statement Example**

The electrical potential quantity has a normal default setting of 1 µV for absolute tolerance (abstol) and 300 mV for maximum change per Newton iteration (maxdelta). You can change abstol to 5 µV, reset maxdelta to 600 mV, define the estimate of the maximum voltage to be 1000 V, and set the maximum permitted voltage to be 10^9 with the following statement:

```plaintext
VoltQuant quantity name="V" abstol=5uV maxdelta=600mV huge=1000V blowup=1e9
```

*VoltQuant is a unique name you give to the quantity statement.*

The keyword *quantity* is the primitive name for the statement.

The name parameter identifies the quantity you are changing. (V is the name for electrical potential.)

abstol, maxdelta, huge, and blowup are the parameters you are resetting.

**Note:** The *quantity* statement has other uses besides setting tolerances. You can use the *quantity* statement to create new quantities or to redefine properties of an existing quantity, and you can use the *node* statement to set the quantities for a particular node. For more information about the *quantity* statement, see the Spectre online help (spectre -h quantity) and the *SpectreHDL Reference* manual. For more information on the *node* statement, see spectre -h node. The following is an example of a *node* statement:

```plaintext
setToMagnetic t1 t2 node value="Wb" flow="MMF" strength=insist
```

**Inherited Connections**

Inherited connections is an extension to the connectivity model that allows you to create global signals and override their names for selected branches of the design hierarchy. The flexibility of inherited connections allows you to use

- Multiple power supplies in a design
- Overridable substrate connections
- Parameterized power and ground symbols
You can use an inherited connection so that you can override the default connection made by a signal or terminal. This method can save you valuable time. You do not have to re-create an entire subbranch of your design just to change one global signal.

For more detailed information on how to use inherited connections and net expressions with various Cadence® tools in the design flow, see the *Inherited Connections Flow Guide*. 
Parameter Specification and Modeling Features

You can use the Spectre® circuit simulator models and AHDL modules in Spectre netlists. This chapter describes the powerful modeling capabilities of Spectre, including:

- Instance (Component or Analysis) Parameters on page 77
- Parameters Statement on page 82
- Expressions on page 84
- Subcircuits on page 90
- Inline Subcircuits on page 96
- Binning on page 104
- Scaling Physical Dimensions of Components on page 115
- N-Ports on page 117

Instance (Component or Analysis) Parameters

In this section, you will learn about the types of component or analysis parameter values the Spectre circuit simulator accepts and how to specify them.

Types of Parameter Values

Spectre component or analysis parameters can take the following types of values:

- Real or integer expression, consisting of
  - Literals
  - Arithmetic or Boolean operators
Parameter Specification and Modeling Features

- Predefined circuit or subcircuit parameters
- Built-in constants (fixed values) or mathematical functions (software routines that calculate equations)
- Real or integer constants
- The name of a component instance or model
- The name of a component parameter
- A character string (must be surrounded by quotation marks)
- A name from a predefined set of names available to specify the parameter value (enumerated types)

Parameter Dimension

Component or analysis parameters can be either scalar or vector.

If a component or analysis parameter value is a group of numbers or names, you specify the group as a vector of values by enclosing the list of items in square brackets ([ ])—for example, `coeffs=[0 0.1 0.2 0.5]` to specify the parameter values 0, 0.1, 0.2, and 0.5. You can specify a group of number pairs (such as time-voltage pairs to specify a waveform) as a vector of the individual numbers.

Remember these guidelines when you specify vectors of value:

- You can mix numbers and netlist or subcircuit parameter names in the same vector (`coeff=[0 coeff1 coeff2 0.5]`).
- You cannot leave a list of items empty.
- You can use expressions (such as formulas) to specify numbers within a vector. When you use a vector of expressions, each expression must be surrounded by parentheses (`coeff=[0 (A*B) C 0.5]`).
- You can use subcircuit parameters within vectors.

Parameter Ranges

Parameter ranges have hard limits and soft limits. Hard limits are enforced by the Spectre simulator; if you violate them, the Spectre simulator issues an error and terminates. You specify soft limits; if you violate them, the Spectre simulator issues a warning and continues. Soft limits are used to define reasonable ranges for parameter values and can help find “unreasonable” values that are likely errors. You can change soft limits, which are defined in
one or more files. Use the +param command line option to use the suggested parameter range limits.

You can specify limits for any scalar parameter that takes either a real number, an integer, or an enumeration. To specify the limits of a parameter that takes enumerations, use the indices or index values associated with the enumerations. For example, consider the region parameter of the bjt. There are four possible regions (see spectre -h bjt):

- off
- fwd
- rev
- sat

Each enumeration is assigned a number starting at 0 and counting up. Thus,

- off=0
- fwd=1
- rev=2
- sat=3

The specification bjt 3 <= region <= 1 indicates that a warning is printed if region=rev because the conditions 3 <= region and region <= 1 exclude only region=2 and region 2 is rev.

For more information on parameter range checking, see “Checking for Invalid Parameter Values” on page 95.

**Help on Parameters**

There are four main ways to get online help about Spectre component or analysis parameters.

**spectre -help**

When you type spectre -help name, where name is the name of a component or analysis, you get the following information:

- Parameter names
  
  Related parameters are grouped together.
Parameter defaults

Units

Parameter description

For analyses and controls, parameters are listed in the “Parameters” section. At the end of the longer parameter listings is the parameter index. This index lists the parameters alphabetically and gives the number that corresponds to where the parameter is in the numbered list.

For components, parameters are divided into up to four sections: “Instance Parameters,” “Model Parameters,” “Output Parameters,” and “Operating Point Parameters.” At the end of longer parameter listings is the parameter index. This index indicates where to find a parameter’s description with a letter and a number. The letter refers to the section (for example, I refers to the instance parameters section, M refers to the model parameters section, O refers to the output parameters section, and OP refers to the operating-point parameters section), and the number refers to where the parameter is in the numbered list.

spectre -helpsort

When you type `spectre -helpsort name`, where `name` is the name of a component or analysis, you get the same information as you do with `spectre -h name`, but the parameters are sorted alphabetically instead of divided into related groups.

spectre -helpfull

When you type `spectre -helpfull name`, where `name` is the name of a component or analysis, you get related parameters grouped as you do with `spectre -h name`, and you get the following additional information:

- Parameter type
- Parameter dimension—scalar or vector
- Parameter range

spectre -helpsortfull

When you type `spectre -helpsortfull name`, where `name` is the name of a component or analysis, you get the same information as you do with `spectre -helpsort name`, but the parameters are sorted alphabetically instead of divided into related groups.
Scaling Numerical Literals

If a parameter value is an integer or a floating-point number, you can scale it in the following ways:

- Follow the number with an `e` or an `E` and an integer exponent (for example, `2.65e3`, `5.32e-4`, `1E-14`, `3.04E6`)
- Use scale factors (for example `5u`, `3.26k`, `4.2m`)

You cannot use both scale factors and exponents in the same parameter value. For example, the Spectre simulator ignores the `p` in a value such as `1.234E-3p`.

**Caution**

*The Spectre simulator also accepts additional data files, such as the waveform and noise files accepted by the independent sources or the S-parameter file accepted by the N-port. Generally, these files do not accept numbers with scale factors.*

The Spectre mode (`simulator lang=spectre`) accepts only the following ANSI standard (SI) scale factors:

\[
\begin{align*}
T &= 10^{12} \\
G &= 10^{9} \\
M &= 10^{6} \\
K &= 10^{3} \\
k &= 10^{3} \\
\_ &= 1 \\
\% &= 10^{-2} \\
c &= 10^{-2} \\
m &= 10^{-3} \\
u &= 10^{-6} \\
n &= 10^{-9} \\
p &= 10^{-12} \\
f &= 10^{-15} \\
a &= 10^{-18}
\end{align*}
\]

**Note:** SI scale factors are case sensitive.

The Spectre simulator allows you to specify units, but only if you specify a scale factor. If specified, units are ignored. Thus,

\[
c=1\text{pf} \quad // \text{units} = "f" \\
l=1\text{uH} \quad // \text{units} = "H"
\]

are accepted, but

\[
r=50\text{Ohms}
\]

is rejected because units are provided without a scale factor. For the last example, use

\[
r=50\_\text{Ohms}
\]
SPICE mode \( \text{simulator lang=spice} \) accepts only the following SPICE scale factors:

\[
\begin{align*}
& t=10^{12} \quad g=10^9 \quad \text{meg}=10^6 \quad \text{k}=10^3 \quad p=10^{-12} \\
& m=10^{-3} \quad \text{mil}=25.4 \times 10^6 \quad u=10^6 \quad n=10^{-9} \quad f=10^{-15}
\end{align*}
\]

Note: SPICE scale factors are not case sensitive. Any other scale factor is ignored (treated as 1.0).

Caution

If you are not clear about the scaling rules for each simulation mode, you can cause errors in your simulation. For example, \( 1.0 \text{M} \) is interpreted as \( 10^{-3} \) in the SPICE mode but as \( 10^6 \) in the Spectre mode.

Parameters Statement

In this section, you will learn about the circuit and subcircuit parameters (collectively known as netlist parameters) as defined by the \texttt{parameters} statement.

Circuit and Subcircuit Parameters

The Spectre Netlist Language allows real-valued parameters to be defined and referenced in the netlist, both at the top-level scope and within subcircuit declarations (run \texttt{spectre -h subckt} for more details on parameters within subcircuits).

The format for defining parameters is as follows:

\[
\text{parameters } \texttt{param=value } <\texttt{param=value}> \ldots
\]

Once defined, you can use parameters freely in expressions. The following are examples:

```
simulator lang=spectre
parameters p1=1 p2=2 // declare some parameters
r1 1 0 resistor r=p1   // use a parameter, value=1
r2 1 0 resistor r=p1+p2 // use parameters in an expression, value=3
x1 s1 p4=8    // subckt "s1" is defined below, pass in value 8 for "p4"

subckt s1
  parameters p1=4 p3=5 p4=6 // note: no "p2" here, p1 "redefined"
  r1 1 0 resistor r=p1     // local definition used: value=4
  r2 1 0 resistor r=p2     // inherit from parent(top-level) value=2
  r3 1 0 resistor r=p3     // use local definition, value=5
  r4 1 0 resistor r=p4     // use passed-in value, value=8
  r5 1 0 resistor r=p1+p2/p3 //use local+inherited/local=(4+2/5)=4.4
ends
```
Parameter Declaration

Parameters can be declared anywhere in the top-level circuit description or on the first line of a subcircuit definition. Parameters must be declared before they are used (referenced). Multiple parameters can be declared on a single line. When parameters are declared in the top-level, their values are also specified. When parameters are declared within subcircuits, their default values are specified. The value or default value for a parameter can be a constant, expression, a reference to a previously defined parameter, or any combination of these.

Parameter Inheritance

Subcircuit definitions inherit parameters from their parent (enclosing subcircuit definition, or top-level definition). This inheritance continues across all levels of nesting of subcircuit definitions; that is, if a subcircuit $s1$ is defined, which itself contains a nested subcircuit definition $s2$, then any parameters accessible within the scope of $s1$ are also accessible from within $s2$. Also, any parameters declared within the top-level circuit description are also accessible within both $s1$ and $s2$. However, any subcircuit definition can redefine a parameter that it inherited. In this case, if no value is specified for the redefined parameter when the subcircuit is instantiated, then the redefined parameter uses the locally defined default value, rather than inheriting the actual parameter value from the parent. See how the $r2$ resistor is used in the examples in “Circuit and Subcircuit Parameters” on page 82.

Parameter Namespace

Parameter names must not conflict with node, component, or analysis names. That is, it is not possible to reference a parameter called $r1$ if there is an instance of a resistor (or other device or analysis) called $r1$. Parameter names must also not be used where a node name is expected.

Parameter Referencing

Spectre netlist parameters can be referenced anywhere that a numeric value is normally specified on the right-hand side of an $=$ sign or within a vector, where the vector itself is on the right-hand side of an $=$ sign. This includes referencing of parameters in expressions (run `spectre -h expressions` for more details on netlist expression handling), as indicated in the preceding examples. You can use expressions containing parameter references when specifying component or analysis parameter values (for example specifying the resistance of...
a resistor or the stop time of a transient analysis, as outlined in the preceding example), when specifying model parameter values in model statements (for example specifying $bf=p1*0.8$ for a bipolar model parameter, $bf$), or when specifying initial conditions and nodesets for individual circuit nodes.

### Altering/Sweeping Parameters

Just as certain Spectre analyses (such as `sweep`, `alter`, `ac`, `dc`, `noise`, `sp`, and `xf`) can sweep component instance or model parameters, they can also sweep netlist parameters. Run `spectre -h analysis` to see the particular details for any of these analyses, where `analysis` is the analysis of interest.

### Expressions

An expression is a construct that combines operands with operators to produce a result that is a function of the values of the operands and the semantic meaning of the operators. Any legal operand is also an expression in itself. Legal operands include numeric constants and references to top-level netlist parameters or subcircuit parameters. Calls to algebraic and trigonometric functions are also supported. The complete lists of operators, algebraic, and trigonometric functions are given after some examples.

The following are examples:

```plaintext
simulator lang=spectre
parameters p1=1 p2=2           // declare some top-level parameters
r1 (1 0) resistor r=p1        // the simplest type of expression
r2 (1 0) resistor r=p1+p2     // a binary (+) expression
r3 (1 0) resistor r=5+6/2    // expression of constants, = 8
x1 s1 p4=8 // instantiate a subcircuit, defined in the following lines
subckt s1
parameters p1=4 p3=5 p4=p1+p3       // subcircuit parameters
    r1 (1 0) resistor r=p1     // another simple expression
    r2 (1 0) resistor r=p2*p2   // a binary multiply expression
    r3 (1 0) resistor r=(p1+p2)/p3      // a more complex expression
    r4 (1 0) resistor r=sqrt(p1+p2)    // an algebraic function call
    r5 (1 0) resistor r=3+atan(p1/p2)  // a trigonometric function call
    r6 (1 0) RESMOD r=(p1 ? p4+1 : p3)  // the ternary operator

// a model statement, containing expressions
model RESMOD resistor tc1=p1+p2 tc2=sqrt(p1*p2)
// some expressions used with analysis parameters
time_sweep tran start=0 stop=(p1+p2)*50e-6 // use 5*50e-6 = 150 us
// a vector of expressions (see notes on vectors below)
dc_sweep dc param=p1 values=[0.5 1 +p2 (sqrt(p2*p2)) ]    // sweep p1
```
Where Expressions Can Be Used

The Spectre Netlist Language allows expressions to be used where numeric values are expected on the right-hand side of an = sign or within a vector, where the vector itself is on the right-hand side of an = sign. Expressions can be used when specifying component or analysis instance parameter values (for example, specifying the resistance of a resistor or the stop time of a transient analysis, as outlined in the preceding example), when specifying model parameter values in model statements (for example, specifying $bf=p1*0.8$ for a bipolar model parameter, $bf$), or when specifying initial conditions and nodesets for individual circuit nodes.

Operators

The operators in the following table are supported, listed in order of decreasing precedence. Parentheses can be used to change the order of evaluation. For a binary expression like $a+b$, $a$ is the first operand and $b$ is the second operand. All operators are left associative, with the exceptions of the “to the power of” operator (**) and the ternary operator (? : ), which are right associative. For logical operands, any nonzero value is considered true. The relational and equality operators return a value of 1 to indicate true or 0 to indicate false. There is no short-circuiting of logical expressions involving && and ||.

<table>
<thead>
<tr>
<th>Operator</th>
<th>Symbol(s)</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unary +, Unary −</td>
<td>+, −</td>
<td>Value of the operand, negative of the operand.</td>
</tr>
<tr>
<td>To the power of</td>
<td>**</td>
<td>First operand to be raised to the power of the second operand.</td>
</tr>
<tr>
<td>Multiply, Divide</td>
<td>*, /</td>
<td>Product, quotient of the operands.</td>
</tr>
<tr>
<td>Binary Plus/Minus</td>
<td>+, −</td>
<td>Sum, difference of the operands.</td>
</tr>
<tr>
<td>Shift</td>
<td>&lt;&lt;, &gt;&gt;</td>
<td>First operand shifted left by the number of bits specified by the second operand; first operand shifted right by the number of bits specified by the second operand.</td>
</tr>
<tr>
<td>Relational</td>
<td>&lt;, &lt;=, &gt;, &gt;=</td>
<td>Less than, less than or equal, greater than, greater than or equal, respectively.</td>
</tr>
<tr>
<td>Equality</td>
<td>==, !=</td>
<td>True if the operands are equal; true if the operands are not equal.</td>
</tr>
<tr>
<td>Bitwise AND</td>
<td>&amp;</td>
<td>Bitwise AND (of integer operands).</td>
</tr>
</tbody>
</table>
Algebraic and Trigonometric Functions

The trigonometric and hyperbolic functions expect their operands to be specified in radians. The `atan2()` and `hypot()` functions are useful for converting from Cartesian to polar form.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
<th>Domain</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>log(x)</code></td>
<td>Natural logarithm</td>
<td>$x &gt; 0$</td>
</tr>
<tr>
<td><code>log10(x)</code></td>
<td>Decimal logarithm</td>
<td>$x &gt; 0$</td>
</tr>
<tr>
<td><code>exp(x)</code></td>
<td>Exponential</td>
<td>$x &lt; 80$</td>
</tr>
<tr>
<td><code>sqrt(x)</code></td>
<td>Square root</td>
<td>$x &gt; 0$</td>
</tr>
<tr>
<td><code>min(x,y)</code></td>
<td>Minimum value</td>
<td>All $x$, all $y$</td>
</tr>
<tr>
<td><code>max(x,y)</code></td>
<td>Maximum value</td>
<td>All $x$, all $y$</td>
</tr>
<tr>
<td><code>abs(x)</code></td>
<td>Absolute value</td>
<td>All $x$</td>
</tr>
<tr>
<td><code>pow(x,y)</code></td>
<td>$x$ to the power of $y$</td>
<td>All $x$, all $y$</td>
</tr>
<tr>
<td><code>sin(x)</code></td>
<td>Sine</td>
<td>All $x$</td>
</tr>
<tr>
<td><code>cos(x)</code></td>
<td>Cosine</td>
<td>All $x$</td>
</tr>
<tr>
<td><code>tan(x)</code></td>
<td>Tangent</td>
<td>All $x$, except $x = n \times (\pi/2)$, where $n$ is odd</td>
</tr>
<tr>
<td><code>asin(x)</code></td>
<td>Arc-sine</td>
<td>$-1 \leq x \leq 1$</td>
</tr>
<tr>
<td><code>acos(x)</code></td>
<td>Arc-cosine</td>
<td>$-1 \leq x \leq 1$</td>
</tr>
<tr>
<td><code>atan(x)</code></td>
<td>Arc-tangent</td>
<td>All $x$</td>
</tr>
<tr>
<td><code>atan2(x,y)</code></td>
<td>Arc-tangent of $x/y$</td>
<td>All $x$, all $y$</td>
</tr>
</tbody>
</table>
Using Expressions in Vectors

Expressions can be used as vector elements, as in the following example:

dc_sweep dc param=p1 values=[0.5 1 +p2 (sqrt(p2*p2)) ]  // sweep p1

**Note:** When expressions are used within vectors, anything other than constants, parameters, or unary expressions (unary +, unary -) must be surrounded by parentheses. Vector elements should be space separated. The preceding `dc_sweep` example shows a vector of four elements: 0.5, 1, +p2, and `sqrt(p2*p2)`. Note that the square root expression is surrounded by parentheses.

Built-in Constants

You can use built-in constants to specify parameter values.
The Spectre Netlist Language contains the built-in mathematical and physical constants listed in the following table. Mathematical constants start with \texttt{M_}, and physical constants start with \texttt{P_}.

<table>
<thead>
<tr>
<th>Constant Name</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{M_E}</td>
<td>2.7182818284590452354</td>
<td>(e) or (\text{escp}(1))</td>
</tr>
<tr>
<td>\texttt{M_LOG2E}</td>
<td>1.442695040889634074</td>
<td>(\log_2(e))</td>
</tr>
<tr>
<td>\texttt{M_LOG10E}</td>
<td>0.43429448190325182765</td>
<td>(\log_{10}(e))</td>
</tr>
<tr>
<td>\texttt{M_LN2}</td>
<td>0.69314718055994530942</td>
<td>(\ln(2))</td>
</tr>
<tr>
<td>\texttt{M_LN10}</td>
<td>2.30258509299404568402</td>
<td>(\ln(10))</td>
</tr>
<tr>
<td>\texttt{M_PI}</td>
<td>3.14159265358979323846</td>
<td>(\pi)</td>
</tr>
<tr>
<td>\texttt{M_TWO_PI}</td>
<td>6.28318530717958647652</td>
<td>(2\pi)</td>
</tr>
<tr>
<td>\texttt{M_PI_2}</td>
<td>1.57079632679489661923</td>
<td>(\pi/2)</td>
</tr>
<tr>
<td>\texttt{M_PI_4}</td>
<td>0.78539816339744830962</td>
<td>(\pi/4)</td>
</tr>
<tr>
<td>\texttt{M_1_PI}</td>
<td>0.31830988618379067154</td>
<td>(1/\pi)</td>
</tr>
<tr>
<td>\texttt{M_2_PI}</td>
<td>0.63661977236758134308</td>
<td>(2/\pi)</td>
</tr>
<tr>
<td>\texttt{M_2_SQRTP}\texttt{I}</td>
<td>1.12837916709551257390</td>
<td>(2/\sqrt{\pi})</td>
</tr>
<tr>
<td>\texttt{M_SQR2}\texttt{2}</td>
<td>1.41421356237309504880</td>
<td>(\sqrt{2})</td>
</tr>
<tr>
<td>\texttt{M_SQR1_2}</td>
<td>0.70710678118654752440</td>
<td>(\sqrt{1/2})</td>
</tr>
<tr>
<td>\texttt{M_DEGPERR}\texttt{RAD}</td>
<td>57.2957795130823208772</td>
<td>Number of degrees per radian (equal to (180/\pi))</td>
</tr>
<tr>
<td>\texttt{P_Q}</td>
<td>1.6021918\times10^{-19}</td>
<td>Charge of electron in coulombs</td>
</tr>
<tr>
<td>\texttt{P_C}</td>
<td>2.997924562\times10^{8}</td>
<td>Speed of light in vacuum in meters/second</td>
</tr>
<tr>
<td>\texttt{P_K}</td>
<td>1.3806226\times10^{-23}</td>
<td>Boltzmann's constant in joules/Kelvin</td>
</tr>
<tr>
<td>\texttt{P_H}</td>
<td>6.6260755\times10^{-34}</td>
<td>Planck's constant in joules times seconds</td>
</tr>
<tr>
<td>\texttt{P_EPS0}</td>
<td>8.85418792394420013968\times10^{-12}</td>
<td>Permittivity of vacuum in farads/meter</td>
</tr>
</tbody>
</table>
User-Defined Functions

The user-defined function capability allows you to build upon the provided set of built-in mathematical and trigonometric functions. You can write your own functions and call these functions from within any expression. The Spectre function syntax resembles that of C or the SpectreHDL language.

Defining the Function

The following is a simple example of defining a function with arguments of type real and results of type real:

```c
real myfunc( real a, real b ) {
    return a+b*2+sqrt(a*sin(b));
}
```

When you define a function, follow these rules:

- Functions can be declared only at the top level and cannot be declared within subcircuits.
- Arguments to user-defined functions can only be real values, and the functions can only return real values. You must use the keyword real for data typing.
- The Spectre function syntax does not allow references to netlist parameters within the body of the function, unless the netlist parameter is passed in as a function argument.
- The function must contain a single return statement.

Note: If you create a user-defined function with the same name as a built-in function, the Spectre simulator issues a warning and runs the user-defined function.

Calling the Function

Functions can be called from anywhere that an expression can currently be used in the Spectre simulator. Functions can call other functions; however, the function must be declared before it can be called. The following example defines the function myfunc and then calls it:
real myfunc( real a, real b ) {
    return a+b*2+sqrt(a*sin(b));
}

real yourfunc( real a, real b ) {
    return a+b*myfunc(a,b);// call "myfunc"
}

The next example shows how a user-defined function can be called from an expression in the Spectre netlist:
r1 (1 0) resistor r=myfunc(2.0, 4.5)

Predefined Netlist Parameters

There are two predefined netlist parameters:

- temp is the circuit temperature (in degrees Celsius) and can be used in expressions.
- tnom is the measurement (nominal) temperature (in degrees Celsius) and can be used in expressions.

For example:

r1 1 0 res r=(temp-tnom)*15+10k
ol options TEMP=55

Note: If you change temp or tnom using a set statement, alter statement, or simulator options card, all expressions with temp or tnom are reevaluated. Hence, you can use the temp parameter for temperature-dependent modeling (this does not include self-heating, however).

Subcircuits

The Spectre simulator helps you simplify netlists by letting you define subcircuits that you can place any number of times in the circuit. You can nest subcircuits, and a subcircuit definition can contain both instances and definitions of other subcircuits. The main applications of subcircuits are to describe the circuit hierarchy and to perform parameterized modeling. In this section, you learn to define subcircuits and to call them into the main circuit.

Formatting Subcircuit Definitions

You format subcircuit definitions as follows:

subckt SubcircuitName [() node1 ... nodeN []]
[parameters name1=value1 ... [nameN=valueN]]

instance, model, ic, or nodeset statements—or further subcircuit definitions

ends [SubcircuitName]

subckt

The keyword subckt (.subckt is used in SPICE mode).

SubcircuitName

The unique name you give to the subcircuit.

(node1...nodeN)

The external or connecting nodes of the subcircuit to the main circuit.

parameters name1=value1...nameN=

This is an optional parameter specification field. You can specify default values for subcircuit calls that refer to this subcircuit. The field contains the keyword parameters followed by the names and values of the parameters you want to specify.

component instance statement

The instance statements of your subcircuit, other subcircuit definitions, component statements, analysis statements, or model statements.

ends SubcircuitName

The keyword ends (or .ends in SPICE mode), optionally followed by the subcircuit name.

A Subcircuit Definition Example

The following subcircuit named twisted models a twisted pair. It has four terminals—p1, n1, p2, and n2. The parameter specification field for the subcircuit sets subcircuit call default values for parameters zodd, zeven, veven, vodd, and len. Remember that the specified parameters are defaults for subcircuit calls, not for the instance statements in the subcircuit. For example, if the subcircuit call leaves the zodd parameter unspecified, the value of zodd in odd is 50. If, however, the subcircuit call sets zodd to 100, the value of zodd in odd is 100.
subckt twisted (p1 n1 p2 n2)
  parameters zodd=50 zeven=50 vodd=1 len=0
  odd (p1 n1 p2 n2) tline z0=zodd vel=vodd len=len
  tf1a (p1 0 e1 c1) transformer t1=2 t2=1
  tf1b (n1 0 c1 0) transformer t1=2 t2=1
  even (e1 0 e2 0) tline z0=zeven vel=veven len=len
  tf2a (p2 0 e2 c2) transformer t1=2 t2=1
  tf2b (n2 0 c2 0) transformer t1=2 t2=1
ends twisted

Indenting the contents of a subcircuit definition is not required. However, it is recommended to make the subcircuit definition more easily identifiable.

Subcircuit Example

GaAs Traveling Wave Amplifier
// GaAs Traveling-wave distributed amplifier (2-26.5GHz)
// Designed by Jerry Orr, MWTD Hewlett-Packard Co.
// 1986 MTT symposium; unpublished material.
global gnd vdd
simulator lang=spectre

// Models
model nGaAs gaas type=n vto=-2 beta=0.012 cgs=.148p cgd=.016p fc=0.5
subckt cell (o g1 g2)
  TL (o gnd d gnd) tline len=355u vel=0.36
  Gt (d g2 s) nGaAs
  Ctgd (d s) capacitor c=0.033p
  Cgg (g2 gnd) capacitor c=3p
  Gb (s g1 gnd) nGaAs
  Cbgd (s gnd) capacitor c=0.033p
  Ro (d c) resistor r=4k
  Co (c gnd) capacitor c=0.165p
ends cell
subckt stage (i0 o8)
  // Devices
  Q1 (o1 i1 b2) cell
  Q2 (o2 i2 b2) cell
  Q3 (o3 i3 b2) cell
  Q4 (o4 i4 b2) cell
  Q5 (o5 i5 b2) cell
  Q6 (o6 i6 b2) cell
  Q7 (o7 i7 b2) cell

  // Transmission lines
  TLI1 (i0 gnd i1 gnd) tline len=185u z0=96 vel=0.36
  TLI2 (i1 gnd i2 gnd) tline len=675u z0=96 vel=0.36
  TLI3 (i2 gnd i3 gnd) tline len=675u z0=96 vel=0.36
  TLI4 (i3 gnd i4 gnd) tline len=675u z0=96 vel=0.36
  TLI5 (i4 gnd i5 gnd) tline len=675u z0=96 vel=0.36
  TLI6 (i5 gnd i6 gnd) tline len=675u z0=96 vel=0.36
  TLI7 (i6 gnd i7 gnd) tline len=675u z0=96 vel=0.36
  TLI8 (i7 gnd i8 gnd) tline len=340u z0=96 vel=0.36
  TL01 (o0 gnd o1 gnd) tline len=360u z0=96 vel=0.36
TLo2  (o1  gnd  o2  gnd)  tline  len=750u  z0=96  vel=0.36
TLo3  (o2  gnd  o3  gnd)  tline  len=750u  z0=96  vel=0.36
TLo4  (o3  gnd  o4  gnd)  tline  len=750u  z0=96  vel=0.36
TLo5  (o4  gnd  o5  gnd)  tline  len=750u  z0=96  vel=0.36
TLo6  (o5  gnd  o6  gnd)  tline  len=750u  z0=96  vel=0.36
TLo7  (o6  gnd  o7  gnd)  tline  len=750u  z0=96  vel=0.36
TLo8  (o7  gnd  o8  gnd)  tline  len=220u  z0=96  vel=0.36

// Bias network
// drain bias
Ldd  (vdd  o0)  inductor  l=1u
R1  (o0  b1)  resistor  r=50
C1  (b1  gnd)  capacitor  c=9p

// gate 2 bias
R2  (b1  b2)  resistor  r=775
R3  (b2  gnd)  resistor  r=465
C2  (b2  gnd)  capacitor  c=21p

// gate 1 bias
R4  (i8  b3)  resistor  r=50
R5  (b3  gnd)  resistor  r=500
C3  (b3  gnd)  capacitor  c=12p

ends stage

// Two stage amplifier
P1  (in  gnd)  port  r=50  num=1  mag=0
Cin  (in  in1)  capacitor  c=1n
X1  (in1  out1)  stage
Cmid  (out1  in2)  capacitor  c=1n
X2  (in2  out2)  stage
Cout  (out2  out)  capacitor  c=1n
P2  (out  gn)  port  r=50  num=2

// Power Supply
Vpos vdd gnd vsource dc=5

// Analyses
OpPoint dc
Sparams sp start=100M stop=100G dec=100

Rules to Remember

When you use subcircuits,

- You must place the same number of nodes in the same order in subcircuit definitions and their respective subcircuit calls.

- Models and subcircuits defined within a subcircuit definition are accessible only from within that subcircuit. You cannot use the model names in a subcircuit definition in statements from outside the subcircuit. You can, however, use both the model name and the subcircuit definitions in new subcircuits within the original subcircuit. Local models or subcircuits hide nodes or subcircuits with the same names defined outside the subcircuit.

- When you use model statements within subcircuit definitions, where model parameters are expressions of subcircuit parameters definitions, a new model is created for every
instance of the subcircuit. These different models are “expanded models,” which are
derived from the original model statement. Each of the new models has a unique name,
and component instances created from the original model statement are instances of
the new model created for the subcircuit. The full name of each new model is the
flattened name of the subcircuit, followed by a dot (.), followed by the name of the model
as given in the model statement. If you request the output of model data, you can see
these expanded models in the output.

- Subcircuit parameter names are local only. You cannot access the value of a subcircuit
  parameter outside of the scope of the subcircuit in which it was declared.
- Parameter names must be lowercase if you want to instantiate components from SPICE
  mode.

**Calling Subcircuits**

To call a subcircuit, place an instance statement in your netlist. The nodes for this instance
statement are the connections to the subcircuit, and the master field in the subcircuit call
contains the name of the subcircuit. You can enter parameters in a subcircuit call to override
the parameters in the subcircuit definition for that subcircuit call.

The following example shows a subcircuit call and its corresponding subcircuit definition.
**cell** is the name of the subcircuit being called; **Q1** is the unique name of the subcircuit call;
and **o1, i1, and b2** are the connecting nodes to the subcircuit from the subcircuit call. When
you call the subcircuit, the Spectre simulator substitutes these connecting node names in the
subcircuit call for the connecting nodes in the subcircuit definition: **o1** is substituted for **o**,
**i1** is substituted for **g1**, and **b2** is substituted for **g2**. The length of transmission line **TL** is
changed to 500 µm for this subcircuit call from its 355 µm default value in the subcircuit
definition.

```
parameters length=355um
  TL o gnd d gnd  tline len=length vel=0.36
  Gt d g2 s     ngAs
  Ctg d s g2     capacitor c=0.033p
  Cgg g2 gnd     capacitor c=3p
  Gb s g1 gnd    ngAs
  Cbgd s gnd     capacitor c=0.033p
  Ro d c         resistor r=10k
  Co c gnd       capacitor c=0.165p
ends cell
```
Modifying Subcircuit Parameter Values

The following example is a passive Bessel three-pole bandpass filter with default parameter values for bandwidth, termination resistance, and center frequency. The \( \text{bw} \), \( r_0 \), and \( f_c \) parameters are given default values in the subcircuit, but you can change these values when you call the subcircuit. Changing the value of \( \text{bw} \), \( r_0 \), and \( f_c \) in a subcircuit call changes the values of many parameters in instance statements that refer to these three parameters.

```
/*
/* define passive 3-pole bandpass filter
subckt filter (n1 n2)
  parameters bw=1 r0=1 fc=1
  C1 (n1 0) capacitor c=0.3374 / (6.2832 * bw * r0)
  L1 (n1 0) inductor l=(r0 * bw) / (0.3374 * 6.2832 * fc * fc)
  C2 (n1 n12) capacitor c=bw / (0.9705 * 6.2832 * fc * fc * r0)
  L2 (n12 n2) inductor l=(r0 * 0.9705) / (6.2832 * bw)
  C3 (n2 0) capacitor c=2.2034 / (6.2832 * bw * r0)
  L3 (n2 0) inductor l=(r0 * bw)/(2.2034 * 6.2832 * fc * fc)
ends filter

// instantiate 50 Ohm filter with 10.4MHz center frequency and 1MHz bandwidth
F1 (in out) filter bw=1MHz fc=10.4MHz r0=50

// instantiate 1 Ohm filter with 10Hz center frequency and 1Hz bandwidth
F2 (n1 n2) filter fc=10
```

You can use such parameterized subcircuits when the Spectre simulator is reading either Spectre or SPICE syntax. Unlike subcircuit calls in SPICE, the names of Spectre subcircuit calls do not have to start with an \( \text{x} \). This is useful if you want to replace individual components in an existing netlist with subcircuits for more detailed modeling.

Checking for Invalid Parameter Values

When you define subcircuits such as the one in “Modifying Subcircuit Parameter Values” on page 95, you might want to put error checking on the subcircuit parameter values. Such error checking prevents you from entering invalid parameter values for a given subcircuit call.

The Spectre \text{paramtest} component lets you test parameter values and generate necessary error, warning, and informational messages. For example, in the example of the bandpass filter in “Modifying Subcircuit Parameter Values” on page 95, the center frequency needs to be greater than half the bandwidth.

Here is a version of the previous three-pole filter that issues an error message if the center frequency is less than or equal to half the bandwidth:
* define passive 3-pole bandpass filter
subckt filter (n1 n2)
  parameters bw=1 r0=1 fc=1
  checkFreqs paramtest errorif=((bw/2-fc)>=0) \ 
    message="center frequency must be greater than half the bandwidth"
  C1 n1 0   capacitor c=0.3374 / (6.2832 * bw * r0)
  L1 n1 0   inductor l=(r0 * bw) / (0.3374 * 6.2832 * fc * fc)
  C2 n1 n12 capacitor c=bw / (0.9705 * 6.2832 * fc * fc * r0)
  L2 n12 n2 inductor l=(r0 * 0.9705) / (6.2832 * bw)
  C3 n2 0   capacitor c=2.2034 / (6.2832 * bw * r0)
  L3 n2 0   inductor l=(r0 * bw) / (2.2034 * 6.2832 * fc * fc)
ends filter

The `paramtest` component `checkFreqs` has no terminals and no effect on the simulation results. It monitors its parameters and issues a message if a given condition is satisfied by evaluating to a nonzero number. In this case, `errorif` specifies that the Spectre simulator issues an error message and stops the simulation. If you specify `printif`, the Spectre simulator prints an informational message and continues the simulation. If you specify `warnif`, the Spectre simulator prints a warning and continues.

For more information about specific parameters available with the `paramtest` component, see the parameter listings in the Spectre online help (`spectre -h`).

### Inline Subcircuits

An inline subcircuit is a special case where one of the instantiated devices or models within the subcircuit does not get its full hierarchical name but inherits the subcircuit call name. The inline subcircuit is called in the same manner as a regular subcircuit. You format inline subcircuit definitions as follows:

```
inline subckt <SubcircuitName> [[<node1> ... <nodeN>]]
```

Depending on the use model, the body of the inline subcircuit typically contains one of the following:

- Multiple device instances, one of which is the `inline` component
- Multiple device instances (one of which is the `inline` component) and one or more parameterized models
- A single `inline` device instance and a parameterized model to which the device instance refers
- Only a single parameterized model
The inline component is denoted by giving it the same name as the inline subcircuit itself. When the subcircuit is flattened, as shown in the following section, the inline component does not acquire a hierarchical name such as X1.M1 but rather acquires the name of the subcircuit call itself, X1. Any noninline components in the subcircuit acquire the regular hierarchical name, just as if the concept of inline subcircuits never existed.

Typically, a modeling engineer writes inline subcircuit definitions for a circuit design engineer to use.

**Modeling Parasitics**

You can model parasitics by adding parasitics components to a base component using inline subcircuits. The body of the inline subcircuit contains one inline component, the base component, and several regular components, which are taken to represent parasitics.

The following example of an inline subcircuit contains a MOSFET instance and two parasitic capacitances:

```plaintext
inline subckt s1 (a b) // "s1" is name of subcircuit
parameters l=1u w=2u
s1 (a b 0 0) mos_mod l=l w=w // "s1" is "inline" component
cap1 (a 0) capacitor c=1n
cap2 (b 0) capacitor c=1n
ends s1
```

The following circuit creates a simple MOS device instance M1 and calls the inline subcircuit s1 twice (M2 and M3):

```plaintext
M1 (2 1 0 0) mos_mod
M2(5 6) s1 l=6u w=7u
M3(6 7) s1
```

This circuit flattens to the following equivalent circuit:

```plaintext
M1 (2 1 0 0) mos_mod
M2 (5 6 0 0) mos_mod l=6u w=7u // the "inline" component
M2.cap1 (5 0) capacitor c=1n // inherits call name
M2.cap2 (6 0) capacitor c=1n // a regular hierarchical name
M3 (6 7 0 0) mos_mod l=1u w=2u // the "inline" component
M3.cap1 (6 0) capacitor c=1n // inherits call name
M3.cap2 (7 0) capacitor c=1n
```

The final flattened names of each of the three MOSFET instances are M1, M2 and M3. (If s1 was a regular subcircuit, the final flattened names would be M1, M2.s1, and M3.s1.) However, the parasitic capacitors have full hierarchical names.

You can create an instance of the inline subcircuit cell in the same way as creating an instance of the specially tagged inline device. You can use save statements to probe this instance in the same way as a regular device, without having to
Realize that the instance is actually embedded in a subcircuit

Know that there are possible additional parasitic devices present

Figure out the hierarchical name of the device of interest

A modeling engineer can create several of these inline subcircuits and place them in a library for the design engineer to use. The library then includes a symbol cell view for each inline subcircuit. The design engineer then places a symbol cell view on a design, which behaves just as if a primitive were being used. The design engineer can then probe the device for terminal currents and operating-point information.

Probing the Device

The Spectre simulator allows the following list of items to be saved or probed for primitive devices, including devices modeled as the inline components of inline subcircuits:

- All terminal currents
  ```
  save m1:currents
  ```

- Specific (index) terminal current
  ```
  save m1:1 //#1=drain
  ```

- Specific (named) terminal current
  ```
  save m1:s //"s"=source
  ```

- Save all operating-point information
  ```
  save m1:oppoint
  ```

- Save specific operating-point information
  ```
  save m1:vbe
  ```

- Save all currents and operating-point information
  ```
  save m1
  ```

**Note:** If the device is embedded in a regular subcircuit, you have to know that the device is a subcircuit and find out the appropriate hierarchical name of the device in order to save or probe the device. However, with inline components, you can use the subcircuit call name, just as if the device were not in a subcircuit.

Operating-point information for the inline component is reported with respect to the terminals of the inline component itself and not with respect to the enclosing subcircuit terminals. This results in the following cautions.
Caution: Parasitic Elements in Series with Device Terminals

If the parasitic elements are in series with the device terminals, the reported operating-point currents are correct, but reported operating-point voltages might be incorrect. For example, consider the case of an inline MOSFET device with parasitic source and drain resistances:

```
inline subckt mos_r (d g s b)
    parameters p1=1u p2=2u
    mos_r (dp g sp b) mos_mod l=p1 w=p2               // "inline" component
    rd (d dp) resistor r=10                          // series drain resistance
    rs (s sp) resistor r=10                          // series source resistance
ends mos_r
```

If an instance M1 is created of this mos_r inline subcircuit and the operating point of M1 is probed, the drain-to-source current ids is reported correctly. However, the reported vds is not the same as V(d) – V(s), the two wires that connect the subcircuit drain and source terminals. Instead, vds is V(dp) – V(sp), which are nodes internal to the inline subcircuit.

Caution: Parasitic Elements in Parallel with Device Terminals

If the parasitic elements are in parallel with the device terminals, the reported voltages are correct, but the reported currents might be incorrect. For example, consider the following case of a MOSFET with source-to-bulk and drain-to-bulk diodes:

```
inline subckt mos_d (d g s b)
    parameters p1=1u p2=2u
    mos_d (d g s b) mos_mod l=p1 w=p2               // "inline" component
    d1(d b) diodel r=10                             // drain-bulk diode
    d2(s b) diodel r=10                             // source-bulk diode
ends mos_d
```

Here, the operating-point vds for the inline component is reported correctly because there are no extra nodes introduced by the inline subcircuit model. However, the reported ids for the inline device is not the same as the current flowing into terminal d because some of the current flows into the transistor and some through diode d1.

Parameterized Models

Inline subcircuits can be used in the same way as regular subcircuits to implement parameterized models. When an inline subcircuit contains both a parameterized model and an inline device referencing that model, you can create instances of the device, and each instance automatically gets an appropriately scaled model assigned to it.

For example, the instance parameters of an inline subcircuit can represent emitter width and length of a BJT device. Within that subcircuit, a model statement can be created that is parameterized for emitter width and length and scales accordingly. When you instantiate the subcircuit, you supply the values for the emitter width and length, and the device is
instantiated with an appropriate geometrically scaled model. Again, the inline device does not get a hierarchical name and can be probed in the same manner as if it were a simple device and not actually embedded in a subcircuit.

In the following example, a parameterized model is declared within an inline subcircuit for a bipolar transistor. The model parameters are the emitter width, emitter length, emitter area, and the temperature delta (\(\text{trise}\)) of the device above nominal. Ninety-nine instances of a 4x4 transistor are then placed, and one instance of a transistor with \(\text{area}=50\) is placed. Each transistor gets an appropriately scaled model.

* declare a subcircuit, which instantiates a transistor with
* a parameterized model. The parameters are emitter width
* and length.

```plaintext
inline subckt bjtmod (c b e s)
  parameters le=1u we=2u area=le*we trise=0
  model mod1 bjt type=npn bf=100+(le+we)/2*(area/1e-12)
    is=1e-12*(le/we)*(area/1e-12)
  bjtmod (c b e s) mod1 trise=trise //"inline" component
ends bjtmod
```

* some instances of this subck
```
q1 (2 3 1 0) bjtmod le=4u we=4u // trise defaults to zero
q2 (2 3 2 0) bjtmod le=4u we=4u trise=2
q3 (2 3 3 0) bjtmod le=4u we=4u
.
q99 (2 3 99 0) bjtmod le=4u we=4u
q100 (2 3 100 0) bjtmod le=1u area=50e-12
```

Since each device instance now gets its own unique model, this approach lends itself to statistical modeling of on-chip mismatch distributions, in which each device is taken to be slightly different than all the others on the same chip. The value of \(\text{bf}\) is the same for the first 99 transistors. For more details, see “Monte Carlo Analysis” on page 161.

### Inline Subcircuits Containing Only Inline model Statements

You can create an inline subcircuit that contains only a single model statement (and nothing else), where the model name is identical to the subcircuit name. This syntax is used to generate a parameterized model statement for a given set of parameters, where the model is then exported and can be referenced from one level outside the subcircuit. Instantiating the inline subcircuit in this case merely creates a model statement with the same name as the subcircuit call. The Spectre simulator knows that because the subcircuit definition contains only a model statement and no other instances, the definition is to be used to generate named models that can be accessed one level outside of the subcircuit. Regular device instances one level outside of the subcircuit can then refer to the generated model.

Because these are now instances of a model rather than of a subcircuit, you can specify device instance parameters with enumerated types such as \(\text{region=fwd}\).
This technique is shown in the following example:

* declare an inline subcircuit that "exports" a parameterized model
inline subckt bjtmod
   parameters le=1u we=2u area=le*we
   model bjtmod bjt type=npn bf=100+(le+we)/2*(area/1e-12) \n       is=1e-12*(le/we)*(area/1e-12)
ends bjtmod

* now create two "instances" of the inline subcircuit, that is,
* create two actual models, called mod1, mod2

mod1 bjtmod le=4u we=4u
mod2 bjtmod le=1u area=50e-12

* 99 instances of mod1 (all share mod1)
* and 1 instance of mod2.
q1 (2 3 1 0) mod1 region=fwd
q2 (2 3 2 0) mod1 trise=2
.
q99 (2 3 99 0) mod1
q100 (2 3 100 0) mod2

Because the syntax of creating an instance of a model is the same as the syntax of creating an instance of a subcircuit in the Spectre netlist, you can easily replace model instances with more detailed subcircuit instances. To do this, replace the model statement itself with a subcircuit definition of the same name.

When you do this in the Spectre Netlist Language, you do not have to change the instance statements. In SPICE, inline subcircuits start with x, so you need to rename all instance statements to start with x if you replaced a model with a subcircuit.

Process Modeling Using Inline Subcircuits

Another modeling technique is to specify geometrical parameters, such as widths and lengths, to a device such as a bipolar transistor and then to create a parameterized model based on that geometry. In addition, the geometry can be modified according to certain process equations, allowing you to model nonideal etching effects, for example.

You use inline subcircuits and some include files for process and geometric modeling, as in the following example files. The ProcessSimple.h file defines the process parameters and the bipolar and resistor devices:

```c
// File: ProcessSimple.h
simulator lang=spectre

// define process parameters, including mismatch effects
parameters RSHSP=200 RSHPI=5k // sheet resistance, pinched sheet res
   +  SPDW=0 SNDW=0 // etching variation from ideal
   +  XISN=1 XBFN=1 XRSP=1 // device "mismatch" (mm)
```
parameters
+  XISNafac=100m XISNbfac=1m // IS scaling factors for mm eqns
+  XBFNafac=100m XBFNbfac=1m // BF " " " " " "
+  XRSPafac=100m XRSPbfac=1m // RS " " " " " "
+  RSHPnom=200 RSHPINom=5k // sheet resistance nom. values
+  FRSHPI=RSHPI/RSHPINom // ratio of PI sheet res to nom

// define "simple" bipolar and resistor devices

// a "base" TNSA subckt, that is, a simple "TNSA" bipolar transistor
// subcircuit, with model statement
inline subckt TNSA_B (C B E S)
  parameters MULT=1 IS=1e-15 BF=100
  model modX bjt type=npn is=IS bf=BF // a model statement
  TNSA_B (C B E S) modX m=MULT // "inline" device instance
ends TNSA_B

// a "base" resistor
// a simple "RPLR" resistor subcircuit
inline subckt RPLR_B (A B)
  parameters R MULT=1
  RPLR_B (A B) resistor r=R m=MULT // "inline" device
ends RPLR_B

// define process/geometry dependent bipolar and resistor devices

// a "geometrical/process" TNSA subcircuit
// a BJT subcircuit, with process and geometry effects modeled
// bipolar model parameters IS and BF are functions of effective
// emitter area/perimeter taking process factors (for example,
// nonideal etching) into account
inline subckt TNSA_PR (C B E S)
  parameters WE LE MULT=1 dIS=0 dBF=0
  +  WEA=WE+SNDW // effective or "Actual" emitter width
  +  LEA=LE+SNDW // effective or "Actual" emitter length
  +  AE=WEA*LEA // effective emitter area
  +  IS=1e-18*FRSHPI*AE*(1+(XISNafac/sqrt(AE)+XISNbfac)
  +  *(dIS/2+XISN-1)/sqrt(MULT))
  +  BF=100*FRSHPI*(1+(XBFNafac/sqrt(AE)+XBFNbfac)
  +  *(dBF/2+XBFN-1)/sqrt(MULT))
  TNSA_PR (C B E S) TNSA_B IS=IS BF=BF MULT=MULT // "inline"
ends TNSA_PR

// a "geometrical/process" RPLR resistor subcircuit
// resistance is function of effective device geometry, taking
// process factors (for example, nonideal etching) into account
inline subckt RPLR_PR (A B)
  parameters Rnom WB MULT=1 dR=0
  +  LB=Rnom*WB/RSHSPnom
  +  AB=LB*(WB+SPDW)
  RPLR_PR (A B) RPLR_B R=RSHSP*LB/(WB+SPDW)*
  +  (1+(XRSPafac/sqrt(AB)+XRSPbfac)*(dR/2+XRSP-1)/sqrt(MULT))
ends RPLR_PR

The following file, Plain.h, provides the designer with a plain device interface without
geometrical or process modeling:
The following file, Process.h, provides the designer with the geometrical device interface:

```plaintext
// File: Process.h
simulator lang=spectre
inline subckt TNSA (C B E S)
    parameters WE=1u LE=1u MULT=1 dIS=0 dBF=0
    TNSA (C B E S) TNSA_PR WE=WE LE=LE \ 
        MULT=MULT dIS=dIS dBF=dBF       // call TNSA_PR
ends TNSA

inline subckt RPLR (A B)
    parameters Rnom=1 WB=10u MULT=1 dR=0
    RPLR (A B) RPLR_PR Rnom=Rnom WB=WB \ 
        MULT=MULT dR=dR                   // call RPLR_PR
ends RPLR
```

The following example is a differential amplifier netlist showing how a design can combine process modeling with process and geometry effects:

```plaintext
// a differential amplifier, biased with a 1mA current source
simulator lang=spectre
include "ProcessSimple.h"
include "Process.h"

E1 (1 0) vsource dc=12

// pullup resistors, 4k ohms nominal
R1 (1 2) RPLR Rnom=4k WB=5 // 5 units wide, model will calc length
R2 (1 3) RPLR Rnom=4k WB=10 // 10 units wide, model will calc length

// the input pair
TNSA1 (2 4 5 0) TNSA WE=10 LE=10
TNSA2 (3 4 5 0) TNSA WE=10 LE=10

// no differential input voltage, both inputs tied to same source
E4 (4 0) vsource dc=5

// current source biasing
J5 (5 0) isource dc=1m
```
Binning

Binning is the process of partitioning a device with different sizes into different models. Before BSIM 3v3, it was very difficult to fit all the devices with a single model statement over very wide ranges of device sizes. To improve fitting accuracy, you might characterize devices into several models with each model valid only for a limited range of device sizes.

For example, suppose you have a device with a length (L) from 0.25 µm to 100 µm and a width (W) from 0.25 µm to 100 µm. You might want to divide your device as follows (not drawn to scale).

In this example, nine models are used. These devices are divided into bins. For devices whose length lies between 1 µm and 10 µm and width lies between 10 µm and 100 µm, Mod6 is used.

The process of generating these models is called binning. The binning process is usually identical for all simulators because the equations for binning are always the same:

\[ P = P_0 + P_l/\text{Leff} + P_w/\text{Weff} + P_p/(\text{Leff} \times \text{Weff}) \]
There are two ways to do binning with the Spectre simulator:

- **Auto model selection**
  
  For more information on auto model selection, see the following section.

- **Conditional instances**
  
  For more information on using conditional instances, see “Conditional Instances” on page 106. For more information on using inline subcircuits for model selection, see “Scaling Physical Dimensions of Components” on page 115.

### Auto Model Selection

Automatic model selection is a simulator feature that automatically assigns the correct models to devices based on their device sizes without using conditional instantiation.

Binning is usually used together with automatic model selection. Model selection is now automatic for MOSFETs, BSIM1, BSIM2, and BSIM3. Help on any one of these devices (for example, `spectre -h mos1`) gives you more details.

For the auto model selector program to find a specific model, the models to be searched need to be grouped together within braces. Such a group is called a model group. An opening brace is required at the end of the line defining each model group. Every model in the group is given a name followed by a colon and the list of parameters. Also, you need to specify the device length and width using the four geometric parameters `lmax`, `lmin`, `wmax`, and `wmin`. The selection criteria to choose a model is as follows:

```
lmin <= inst_length < lmax and wmin <= inst_width < wmin
```

For example:

```plaintext
model ModelName ModelType {
  1:  lmin=2 lmax=4 wmin=1 wmax=2 vto=0.8
  2:  lmin=1 lmax=2 wmin=2 wmax=4 vto=0.7
  3:  lmin=2 lmax=4 wmin=4 wmax=6 vto=0.6
}
```

Then for a given instance

```
M1 1 2 3 4 ModelName  w=3  l=1.5
```

the program searches all the models in the model group with the name `ModelName` and then picks the first model whose geometric range satisfies the selection criteria. In the preceding example, the auto model selector program chooses `ModelName.2`. 
Conditional Instances

You can specify different conditions that determine which components the Spectre simulator instantiates for a given simulation. The determining conditions are computed from the values of parameters. You specify these conditions with the structural if statement. This statement lets you put if-else statements in the netlist.

You can also use conditional instantiation with inline subcircuits. For more information on using inline subcircuits, see “Scaling Physical Dimensions of Components” on page 115.

Formatting the if Statement

You format the structural if statement as follows:

\[
\text{if } <\text{condition}> <\text{statement1}> [\text{else } <\text{statement2}>]
\]

Where:
- `<statement1>` and `<statement2>` fields contain one or more instance statements or if statements. The else part of the statement is optional.
- `<condition>` fields are Boolean-valued expressions where any nonzero value is taken as “true.”

An if Statement Example

The following example illustrates the use of the if statement. There are additional if statements in the `<statement1>` and `<statement2>` fields.

\[
\text{if (rseries == 0) }
   \begin{array}{l}
   \text{c1 (a b) capacitor c=c}
   \text{if (gparallel != 0) gp1 a b resistor r=1/gparallel}
   \end{array}
\text{else }
   \begin{array}{l}
   \text{r2 (a x) resistor r=rseries}
   \text{c2 (x b) capacitor c=c}
   \text{if (gparallel != 0) gp2 x b resistor r=1/gparallel}
   \end{array}
\]

In this example, the Spectre simulator puts different instance statements into the simulation depending on the values of two parameters, rseries and gparallel.

- If both rseries and gparallel are zero, the Spectre simulator includes the instance statement for capacitor c1. If rseries is zero and gparallel is nonzero, the Spectre simulator includes the instance statements for capacitor c1 and resistor gp1.
- If rseries is nonzero and gparallel is zero, the Spectre simulator includes the instance statements for resistor r2 and capacitor c2.
If neither \texttt{rseries} nor \texttt{gparallel} is zero, the Spectre simulator includes the instance statements for resistor \texttt{r2}, capacitor \texttt{c2}, and resistor \texttt{gp2}.

**Rules to Remember**

When you use the \texttt{if} statement,

- If the \texttt{<statement1>} or \texttt{<statement2>} fields contain multiple statements, place these fields within braces (\{\}).
- End the \texttt{<statement1>} and \texttt{<statement2>} fields with newlines.
- Use a continuation character if you want to place a newline between the \texttt{if} and \texttt{condition} statements.
- When the \texttt{<statement1>} or \texttt{<statement2>} field is a single instance or \texttt{if} statement, an \texttt{else} statement is associated with the closest previous \texttt{if} statement.

**Binning by Conditional Instantiation**

You can use conditional instantiation to select an appropriate model based on certain ranges of specified parameters (model binning). This technique lets you decide and implement which parameters to bin on and is valid for any device that supports a model.

The Spectre Netlist Language has conditional instantiation. For example:

\begin{verbatim}
subckt s1 (d g s b) 
  parameters l=1u w=1u
  if (l < 0.5u) {
    m1 (d g s b) shortmod l=l w=w // short-channel model
  } else {
    m2 (d g s b) longmod l=l w=w // long-channel model
  }
  model shortmod vto=0.6 gamma=2 ..etc
  model longmod vto=0.8 gamma=66 ..etc
ends s1
\end{verbatim}

Based on the value of parameter \texttt{l}, one of the following is chosen:

- A short-channel model
- A long-channel model

Previously, the transistor instances had to have unique names (such as \texttt{m1} and \texttt{m2} in the preceding example), even though only one of them could actually be chosen. Now, you can use the same name for both instances, provided certain conditions are met. The following example shows a powerful modeling approach that combines model binning (based on area) with inline subcircuits for a bipolar device:
// general purpose binning and inline models

simulator lang=spectre
parameters VDD=5

vcc (2 0) vsource dc=VDD
vin (1 0) vsource dc=VDD

q1 (1 2 0 0) npn_mod area=350e-12 // gets 20x20 scaled model
q2 (1 3 0 0) npn_mod area=25e-12 // gets 10x10 scaled model
q3 (1 3 0 0) npn_mod area=1000e-12 // gets default model

inline subckt npn_mod (c b e s) //generalized binning, based on area
parameters area=5e-12
if ( area < 100e-12 ) {
    npn_mod (c b e s) npn10x10 // 10u * 10u, inline device
} else if ( area < 400e-12 ) {
    npn_mod (c b e s) npn20x20 // 20u * 20u, inline device
} else {
    npn_mod (c b e s) npn_default // 5u * 5u, inline device
}
model npn_default bjt is=3.2e-16 va=59.8
model npn10x10 bjt is=3.5e-16 va=61.5
model npn20x20 bjt is=3.77e-16 va=60.5
ends npn_mod

The transistors end up having the name that they were called with (q1, q2, and q3), but each
has the correct model chosen for its respective area. Model binning can be now achieved
based on any parameter and for any device, such as resistor or bjt.

A Warning about Conditional Instantiation

If you run a sweep analysis on the previous circuit, sweeping the parameter area for device
instance q1 from 350 x 10^{-12} to 50 x 10^{-12}, the Spectre simulator issues an error and stops
once the value of area exceeds 100 x 10^{-12}. This is because the condition in if (area <
100e-12) changes from being true to being false, and this can result in a topology change
that is not supported. In general, this error is produced for any analysis that changes the value
of the conditional expression.

Rules for General-Purpose Model Binning

The following set of rules exists for general-purpose model binning as outlined in the
preceding example. Within subcircuit npn_mod, the inline device npn_mod is referenced
three times, each with a different model. Allowing multiple “instances” or “references” to the
same-named device is possible only under the following strict topological conditions:

- The reference to the same-named device is possible only in a structural if statement
  that has both an if part and an else part.
Both the if part and the else part must be either a simple one-statement block or another structural if statement to which these same rules apply.

Both the if part and the else part must evaluate to a single device instance, whose instance name, terminal list, and type of primitive are identical.

Multiple references to the same-named device are only possible if there can only ever be one single instance of this device after all expressions have been evaluated, and each instance must be connected to the same nodes and represent the same device.

Examples of Conditional Instances

The following two examples show how to use conditional instances.

Fully Differential CMOS Operational Amplifier

This netlist describes and analyzes a CMOS operational amplifier and demonstrates several sophisticated uses of Spectre features. The example includes top-level netlist parameters, model library/section statements, multiple analyses, and configuring a test circuit with the alter statement.

The first file in the example is the main file for the circuit. This file contains the test circuit and the analyses needed to measure the important characteristics of the amplifier.

The main file is followed by files that describe the amplifier and the various models that can be selected. These additional files are placed in the netlist with include statements.

Voltage-controlled voltage sources transform single-ended signals to differential- and common-mode signals and vice versa. This approach does not generate or dissipate any power. The power dissipation reported by the Spectre simulator is that of the differential amplifier.

```plaintext
// Fully Differential Operational Amplifier Test Circuit
#define PROCESS_CORNER TYPICAL
simulator lang=spectre
global gnd vdd vss
parameters VDD=5.0_V GAIN=0.5
include "cmos.mod" section=typical
include "opamp.ckt"
// power supplies
Vdd (vdd gnd) vsource dc=VDD
Vss (vss gnd) vsource dc=-VDD
// compute differential input
Vcm (cmin gnd) vsource type=dc dc=0 val0=0 val1=2 width=1u delay=10ns
```
Spectre Circuit Simulator User Guide
Parameter Specification and Modeling Features

Vdif (in gnd) vsource type=dc dc=0 val0=0 val1=2 width=1u \
    delay=10ns
Rdif (in gnd) resistor
Eicmp (pin cmin in gnd) vcvs gain=GAIN
Eicmn (nin cmin in gnd) vcvs gain=-GAIN
// feedback amplifier
A1 (pout nout pvg nvg) opamp
Cf1 (pout t1) capacitor c=8p
Cf2 (nout t2) capacitor c=8p
Vt1 (t1 nvg) vsource mag=0
Vt2 (t2 pvg) vsource mag=0
C11 (pout gnd) capacitor c=8p
C12 (out gnd) capacitor c=8p
C11 (pin pvg) capacitor c=2p
C12 (nin nvg) capacitor c=2p
// compute differential output
Edif (out gnd pout nout) vcvs gain=1
Rodif (out gnd) resistor
Ecmp (cmout mid pout gnd) vcvs gain=GAIN
Ecmn (mid gnd nout gnd) vcvs gain=GAIN
Rocm (cmout gnd) resistor
// Perform measurements
//
spectre options save=lvlpub nestlvl=1
printParams info what=output where=logfile
// operating point
opPoint dc readns="%C:r.dc" write="%C:r.dc"
printOpPoint info what=oppoint where=logfile
// differential-mode characteristics
// closed-loop gain, Av = Vdif:p
// power supply rejection ratio, Vdd PSR = Vdd:p, Vss PSR = Vss:p
dmXferFunctions xf start=1k stop=1G dec=10 probe=Rodif
dmNoise noise start=1k stop=1G dec=10 \ 
    oprobe=Edif oportv=1 iprobe=Vdif iportv=1
// step response
dmEnablePulse alter dev=Vdif param=type value=pulse annotate=no
dmStepResponse tran stop=2us errpreset=conservative
dmDisablePulse alter dev=Vdif param=type value=dc annotate=no
// loop gain, Tv = -t1/nvg
// open-loop gain, av = out/(pvg-nvg)
dmEnableTest1 alter dev=Vt1 param=mag value=1 annotate=no
dmEnableTest2 alter dev=Vt2 param=mag value=-1 annotate=no
dmLoopGain ac start=1k stop=1G dec=10
// common-mode characteristics
// closed-loop gain, Av = Vcm:p
// power supply rejection ratio, Vdd PSR = Vdd:p, Vss PSR =Vss:p
cmXferFunctions xf start=1k stop=1G dec=10 probe=Rocm
cmNoise noise start=1k stop=1G dec=10 \ 
    oprobe=Rocm oportv=1 iprobe=Vcm iportv=1
// step response
cmEnablePulse alter dev=Vcm param=type value=pulse annotate=no
cmStepResponse tran stop=2us errpreset=conservative
cmDisablePulse alter dev=Vcm param=type value=dc annotate=no
// loop gain, Tv = -t1/nvg
// open-loop gain, av = 2*cmout/(pvg+nvg)
cmEnableTest1 alter dev=Vt1 param=mag value=1 annotate=no
cmEnableTest2 alter dev=Vt2 param=mag value=1 annotate=no
cmLoopGain ac start=1k stop=1G dec=10

July 2002 110 Product Version 5.0
The following file, opamp.ckt, contains the differential amplifier.

```
// opamp.ckt: Fully Differential CMOS Operational Amplifier
//
// This circuit requires the use of the cmos process models
// simulator lang=spectre
// Folded-Cascode Operational Amplifier
subckt opamp (pout nout pin nin)
  // input differential pair
  M1 (4 pin 1 1) nmos w=402.4u l=7.6u
  M2 (5 nin 1 1) nmos w=402.4u l=7.6u
  M18 (1 12 vss vss) nmos w=242.4u l=7.6u
  // upper half of folded cascode
  M3 (4 11 vdd vdd) pmos w=402.4u l=7.6u
  M4 (5 11 vdd vdd) pmos w=402.4u l=7.6u
  M5 (nout 16 4 vdd) pmos w=122.4u l=7.6u
  M6 (pout 16 5 vdd) pmos w=122.4u l=7.6u
  // lower half of folded cascode
  M7 (nout 13 8 vss) nmos w=72.4u l=7.6u
  M8 (pout 13 9 vss) nmos w=72.4u l=7.6u
  M9 (8 12 vss vss) nmos w=122.4u l=7.6u
  M10 (9 12 vss vss) nmos w=122.4u l=7.6u
  // common-mode feedback amplifier
  M11 (10 nout vss vss) nmos w=12.4u l=62.6u
  M12 (10 pout vss vss) nmos w=12.4u l=62.6u
  M13a (11 gnd vss vss) nmos w=12.4u l=62.6u
  M13b (11 gnd vss vss) nmos w=12.4u l=62.6u
  M14 (10 10 vdd vdd) pmos w=52.4u l=7.6u
  M15 (11 10 vdd vdd) pmos w=52.4u l=7.6u
  Cc1 (nout 11) capacitor c=1p
  Cc2 (pout 11) capacitor c=1p
  // bias network
  M16 (12 12 vss vss) nmos w=22.4u l=7.6u
  M17 (21 12 vss vss) nmos w=22.4u l=7.6u
  M19 (13 13 14 vss) nmos w=22.4u l=11.6u
  M20 (13 21 16 vdd) pmos w=52.4u l=7.6u
  M21 (21 16 17 vdd) pmos w=26.4u l=11.6u
  M22 (16 16 17 vdd) pmos w=26.4u l=11.6u
  D1 (15 vss) dnp area=400n
  D2 (14 15) dnp area=400n
  D3 (18 17) dnp area=400n
  D4 (vdd 18) dnp area=400n
  Ib (gnd 12) isource dc=10u
ends opamp
```

The following file, cmos.mod, contains the models and uses the library and section statements to bin models into various process “corners.” See “Process File” on page 112 for a more sophisticated example, which also includes automatic model selection.

```
// cmos.mod: Spectre MOSFET model parameters --- CMOS process
//
// Empirical parameters, best, typical, and worst cases.
//
// simulator lang=spectre
library cmos_mod
```
Process File

This example of automatic model selection with the conditional if statement is more sophisticated than the previous example (in "Fully Differential CMOS Operational Amplifier" on page 109). With this example, you ask for an nmos transistor, and the Spectre simulator automatically selects the appropriate model according to the process corner, the circuit temperature, and the device width and length. The selection is done by the parameterized inline subcircuit and the conditional if statement.

The paramtest statements create warnings if the model selection parameters are out of range. Models are assigned to instances depending on the initial values of their parameters when the circuit is input. Note that the instances are not reassigned to new models if the selection parameters later change. However, the models are updated if the circuit temperature is changed. Notice that even though nmos is defined as a subcircuit, it is called as if it is a MOSFET primitive.
The MOSFET model parameters are deleted to keep this example to a reasonable length, and ordinarily both the N- and P-channel models are in the same file. If you added the model parameters and a `pmos` model, this file could replace `cmos.mod` in the previous example.

```plaintext
// N-CHANNEL MOS --- 1u NMOS PROCESS
//
// The following group of models represent N-channel MOSFETs over
// the following ranges:
//  1u <= l <= oo
//  1u <= w <= oo
//  0 <= T <= 55
// process corners = {FAST, TYPICAL, SLOW}
// Warnings are issued if these limits are violated.
//
// This model takes 4 parameters, l, w, ls, and ld. The defaults for
// each of these parameters is 1um.
//
// +-----------+
// |        |           |        |---
//     +--------+           +--------+  ^
//     |        |           |        | |
//     |<- ls ->|<--- l --->|<- ld ->|  w
//     |        |           |        ||
//     +--------+           +--------+  v
//              +-----------+
//       Source      Gate       Drain
//
// simulator lang=spectre
//
// Complain if global constants are out-of-range.
//
// TooCold paramtest warnif=(temp < 0) \ 
// message="The nmos model is not accurate below 0 C."
//
// TooHot paramtest warnif=(temp > 55) \ 
// message="The nmos model is not accurate above 55 C."
//
// Define inline subcircuit that implements automatic model selection
// this uses user-supplied parameters l and w which represent device
// geometry, and the built-in parameter temp, to perform model binning
// based on both geometry and temperature.
//
// inline subckt nmos (d g s b)
// parameters l=1um w=1um ls=1um ld=1um
//
// Complain if subcircuit parameters are out-of-range.
//
// TooShort paramtest warnif=(l < 1um) \ 
// message="Channel length for nmos must be greater than 1u."
//
// TooThin paramtest warnif=(w < 1um) \ 
// message="Channel width for nmos must be greater than 1u."
//
// TooNarrow paramtest warnif=(ls < 1um) warnif=(ld < 1um) \ 
// message="Stripe width for nmos must be greater than 1u."
//
// Model selection
```
include "models.scs"  // include all model definitions
include "select_models.scs " section=typical // include code to auto
// choose models
ends nmos
//
// Set the temperature
//
nmosSetTempTo27C alter param=temp value=27

The following file, models.scs, contains the models, which depend on circuit temperature (temp). Each model parameter can take on one of two values, depending on the value of parameter temp.

// // Fast models.
//
model fast_1x1 mos3 type=n vto=(temp >= 27_C) ? 0.8 : 0.77  // ...etc
model fast_1x3 mos3 type=n vto=(temp >= 27_C) ? 0.9 : 0.78  // ...etc
model fast_3x1 mos3 type=n vto=(temp >= 27_C) ? 0.95 : 0.79  // ...etc
model fast_3x3 mos3 type=n vto=(temp >= 27_C) ? 0.98 : 0.81  // ...etc
//
// Typical models.
//
model typ_1x1 mos3 type=n vto=(temp >= 27_C) ? 0.90 : 0.75  // ...etc
model typ_1x3 mos3 type=n vto=(temp >= 27_C) ? 0.91 : 0.73  // ...etc
model typ_3x1 mos3 type=n vto=(temp >= 27_C) ? 0.97 : 0.77  // ...etc
model typ_3x3 mos3 type=n vto=(temp >= 27_C) ? 0.99 : 0.84  // ...etc
//
// Slow models.
//
model slow_1x1 mos3 type=n vto=(temp >= 27_C) ? 0.92 : 0.76  // ...etc
model slow_1x3 mos3 type=n vto=(temp >= 27_C) ? 0.93 : 0.74  // ...etc
model slow_3x1 mos3 type=n vto=(temp >= 27_C) ? 0.98 : 0.78  // ...etc
model slow_3x3 mos3 type=n vto=(temp >= 27_C) ? 0.98 : 0.89  // ...etc

The following file, select_models.scs, uses the structural if statement to select models based on parameters l and w. Note that for MOS devices, this could also be achieved by using auto model selection (see spectre -h mos3), but this example illustrates model binning and model selection based on the structural if statement, which can be used to bin based on any combination of parameters, (not necessarily predefined device geometry models) and for any device type (that is, not limited to MOS devices only).

library select_models
section fast // select models for fast device, based on subckt
//parameters l,w
if (l <= 3um) {
  if (w <= 3um) {
    nmos (d g s b) fast_1x1 l=l w=w ls=ls ld=ld
  } else {
    nmos (d g s b) fast_1x3 l=l w=w ls=ls ld=ld
  }
} else {
  if (w <= 3um) {
    nmos (d g s b) fast_3x1 l=l w=w ls=ls ld=ld
  } else {
Scaling Physical Dimensions of Components

Selected components allow their physical dimensions to be scaled with global scale factors. When you want to scale the physical dimensions of these instances and models, you use the scale and scalem parameters in the options statement. Use scale for instances and scalem for models. The default value for both scale and scalem is 1.0. (For more information about the options statement, see “options Statement Form at” on page 199 and the documentation for the options statement in Spectre online help spectre -h options.)

You can scale the following devices with scale and scalem:

- Capacitors—length (l) or width (w)
- Diodes—length (l) or width (w)
Resistors—length \( l \) or width \( w \)

Physical resistors \( \text{phy\_res} \)—length \( l \) or width \( w \)

MOSFET—length or width

Finding Default Measurement Units

The effects of \text{scale} and \text{scalem} depend on the default measurement units of the components you scale. To find the default measurement units for a component parameter, look up that parameter in the parameter listings of your Spectre online help (\text{spectre -h}). The default for measurement units is in parentheses. For example, the \( (m) \) in this entry from the \text{mos8} parameter descriptions in the Spectre online help (\text{spectre -h}) tells you that the default measurement unit for channel width is meters.

\[ w \text{ (m)} \quad \text{Channel width} \]

Effects of \text{scale} and \text{scalem} with Different Default Units

\text{scale} and \text{scalem} affect only parameters whose default measurement units are in meters. For example, \text{vmax} is affected because its units are \text{m/sec}, but \text{ucrit} is not affected because its units are \text{V/cm}. Similarly, \text{nsub} is not affected because its units are \text{1/cm\textsuperscript{3}}. You can check the effects of \text{scale} and \text{scalem} by adding an info statement with what=output and look for the effective length \( (\text{leff}) \) or width \( (\text{weff}) \) of a device. For more information about the info statement, see “Specifying the Parameters You Want to Save” on page 193.

The following table shows you the effects of \text{scale} and \text{scalem} with different units:

<table>
<thead>
<tr>
<th>\text{scale or scalem}</th>
<th>Units</th>
<th>Scaling action</th>
</tr>
</thead>
<tbody>
<tr>
<td>\text{scale}</td>
<td>\text{meters (m)}</td>
<td>Multiplied by \text{scale}</td>
</tr>
<tr>
<td>\text{scalem}</td>
<td>\text{meters (m)}</td>
<td>Multiplied by \text{scalem}</td>
</tr>
<tr>
<td>\text{scale}</td>
<td>\text{meters\textsuperscript{n} (m\textsuperscript{n})}</td>
<td>Multiplied by \text{scale\textsuperscript{n}}</td>
</tr>
<tr>
<td>\text{scalem}</td>
<td>\text{meters\textsuperscript{n} (m\textsuperscript{n})}</td>
<td>Multiplied by \text{scalem\textsuperscript{n}}</td>
</tr>
<tr>
<td>\text{scale}</td>
<td>\text{1/ meters (1/m)}</td>
<td>Divided by \text{scale}</td>
</tr>
<tr>
<td>\text{scalem}</td>
<td>\text{1/ meters (1/m)}</td>
<td>Divided by \text{scalem}</td>
</tr>
</tbody>
</table>
**String Parameters**

You can use quoted character strings as parameter values. String values you might use as Spectre parameter values include filenames and labels. When you use a string as a parameter, enclose the string in quotation marks. In the following example, the value for the parameter named `file` is the character string `Spara.data`.

```plaintext
model sp_data nport file="Spara.data"
```

**N-Ports**

When you model N-ports, you must first create a file listing the S-parameters. Then you complete the modeling by using this file as input for an `nport` statement. You can create the S-parameter file listing in two different ways.

- You can run an `sp` analysis and create a listing of S-parameter estimates automatically.
- If you already know the S-parameter values, you can create an S-parameter listing manually with a text editor such as `vi`.

The S-parameter data file describes the characteristics of a linear N-port over a list of frequencies. The format of the data file used by the Spectre simulator is flexible and self documenting. The Spectre simulator native format describes N-ports with an arbitrary number of ports, specifies the reference resistance of each port, mentions the frequency with no hidden scale factors, and allows the S-parameters to be given in several formats.

**N-Port Example**

This example demonstrates the use of the `nport` statement.

```plaintext
// Two port test circuit
global gnd
simulator lang=spectre
// Models
```
model sp_data nport file="Spara.data"

// Components

Port1  (i1  gnd)  port  num=1
TL1    (i1  gnd)  o1    gnd)  tline  z0=25  f=1M
Port2  (o1  gnd)  port  num=2
Port3  (i2  gnd)  port  r=50  num=3
X1     (o2  gnd)  o2    gnd)  sp_data
Port4  (o2  gnd)  port  r=50  num=4

// Analyses
Op_Point dc
Sparams sp stop=0.3MHz lin=100 port=Port1 port=Port3

Creating an S-Parameter File Automatically

To create an S-parameter file automatically, run an sp analysis that sweeps frequency and set the output file parameter to file="filename". The parameter filename is the name you select for the S-parameter file. For more information about specifying an sp analysis, see Chapter 6, "Analyses," and the parameter listings for the sp analysis in the Spectre online help (spectre -h sp).

Creating an S-Parameter File Manually

To create an S-parameter file manually in a text editor, observe the following guidelines and rules:

- The Spectre simulator accepts the following formats for S-parameters: real-imag, mag-deg, mag-rad, db-deg, and db-rad. The formats do not have to be the same for each S-parameter. For clarity, use a comma to separate the two parts of an S-parameter.
- Begin each file with a semicolon. Semicolons indicate comment lines.
- Use spaces, commas, and newlines as delimiters.
- You can enter S-parameters in any order.
- You can specify any number of frequency points. The frequency points do not have to be equally spaced, but the frequency index must be in ascending or descending order.
- You must place the frequency specification before the S-parameters, and you must separate the frequency specification from the S-parameters with a colon.
There is no limit to the number of ports. If either port number is greater than nine, place a colon between the two port numbers when you specify the S-parameter format (S13:15).

**Note:** The s-parameters can be given in any order and in any supported format, but the order and format used must be consistent throughout the file and must match the order and format specified in the format line.

### Example of an S-Parameter File

The following is a correctly formatted S-parameter file:

```plaintext
; S-parameter data file nport.data.
; Generated by spectre from circuit file `gendata1' during analysis swp.
; 4:42:38 PM, Tue Jul 28, 1998
reference resistance
port2=50       ; is port p2
port1=50       ; is port p1

format freq: s22(real,imag) s12(real,imag)
           s21(real,imag) s11(real,imag)

0.000000000e+00: 0, 0 1, 0 2.22045e-16, 0
1.000000000e+05: 0.665775, -1.89151e-05 0.334225, 0.334224, -0.00010347
2.000000000e+05: 0.665775, -3.78302e-05 0.334225, 0.334224, -0.00020694
3.000000000e+05: 0.665775, -5.67453e-05 0.334224, 0.334224, -0.00031041
4.000000000e+05: 0.665775, -7.56604e-05 0.334224, 0.334224, -0.00041388
5.000000000e+05: 0.665775, -9.45756e-05 0.334224, 0.334224, -0.00051735
6.000000000e+05: 0.665775, -1.13439e-04 0.334224, 0.334224, -0.00062082
7.000000000e+05: 0.665775, -1.32406e-04 0.334224, 0.334224, -0.00072429
8.000000000e+05: 0.665775, -1.51324e-04 0.334224, 0.334224, -0.000827759
9.000000000e+05: 0.665775, -1.70237e-04 0.334223, 0.334223, -0.000931229
1.000000000e+06: 0.665776, -1.89152e-05 0.334223, 0.334223, -0.0010347
```

For polar formats, the format is the same, except that the appropriate polar format indication replaces the (real, imag) indication of the previous example.

Reference resistance
port2=50
port1=50

Format freq:

s11(mag,rad)
s21(mag,rad)
s12(mag,rad)
s22(mag,rad)

<table>
<thead>
<tr>
<th>freq</th>
<th>s11</th>
<th>s21</th>
<th>s12</th>
<th>s22</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.0</td>
<td>1,0</td>
<td>0,0</td>
<td>0,0</td>
<td>1,0</td>
</tr>
</tbody>
</table>

Reading the S-Parameter File

After you create the S-parameter file, you must place instructions in the netlist for the Spectre simulator to read it. You give these instructions with an nport model statement. The following example shows how to enter an S-parameter file into a netlist. This model statement reads S-parameters from the file Spara.data.

model Sdata nport file="Spara.data"

If the S-parameter file is not in the same directory as the Spectre simulator, you can use a path to the S-parameter file as a value for the nport statement file parameter, or you can specify a search path using the -I command line argument.

S-Parameter File Format Translator

The S-parameter data file format translator (sptr) is a separate program from the Spectre simulator. It translates files from LIBRA, MHARM, HPMNS, or Linear Neutral Model (LNM) format to Spectre format or from Spectre format to LIBRA, MHARM, HPMNS, or LNM format. The input or output can be files that you specify or standard input or output. For standard input or output, the input must be LIBRA format, and the output must be Spectre format. The translator supports the S-parameter file format translation with any number of ports.

Command Arguments

The following is a synopsis of the command line and arguments used to run the translator.
sptr [file format] [freq. scale] [input file] [output file]

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>[file format]</td>
<td>-sim hpmns: Input or output is in HPMNS S-parameter file format.</td>
</tr>
<tr>
<td></td>
<td>-sim libra: Input or output is in LIBRA S-parameter file format.</td>
</tr>
<tr>
<td></td>
<td>-sim mharm: Input or output is in MHARM S-parameter file format.</td>
</tr>
<tr>
<td></td>
<td>-sim lnm: Input or output is in LNM S-parameter file format.</td>
</tr>
<tr>
<td></td>
<td>The file format default is Spectre. If the option is not specified,</td>
</tr>
<tr>
<td></td>
<td>the input file must be in LIBRA or Spectre format. The translator</td>
</tr>
<tr>
<td></td>
<td>always automatically checks the input file format to determine the</td>
</tr>
<tr>
<td></td>
<td>output file format. For example, if you specify the option as -sim</td>
</tr>
<tr>
<td></td>
<td>hpmns and the input file is found to be in Spectre or HPMNS</td>
</tr>
<tr>
<td></td>
<td>format, the translator generates an output file for HPMNS or</td>
</tr>
<tr>
<td></td>
<td>Spectre format. If the input file format does not match the option,</td>
</tr>
<tr>
<td></td>
<td>the translator reports an error message and exits.</td>
</tr>
<tr>
<td>[freq. Scale]</td>
<td>-f FreqScale: If the frequency scale is not explicitly given, it is</td>
</tr>
<tr>
<td></td>
<td>taken to be FreqScale. Thus, TrueFreq = GivenFreq*FreqScale. The default is</td>
</tr>
<tr>
<td></td>
<td>1. This option is ignored if the scale is specified on the first frequency</td>
</tr>
<tr>
<td></td>
<td>or the line beginning with #.</td>
</tr>
<tr>
<td>[input file]</td>
<td>This is the input filename. If both filenames are not given or if the</td>
</tr>
<tr>
<td></td>
<td>input file is specified as -, the input is taken from the standard input.</td>
</tr>
<tr>
<td>[output file]</td>
<td>This is the output filename. If it is not given or if it is specified as</td>
</tr>
<tr>
<td></td>
<td>-, the output is delivered to the standard output.</td>
</tr>
</tbody>
</table>

**Input File Syntax**

This section describes the input file syntax for the five formats.

**LIBRA Format**

An input file for LIBRA format must have a header and a data section. The header consists of all comments. An exclamation mark begins a comment, and the first line of the header must be a comment line with the keyword `Libra`. A line beginning with `#` can include information
on frequency scale factor, impedance data, complex number format definition, and so on. The following is an input file example with five frequency data points and three ports:

```
! Libra (TM) Ver. 3.500.103.3 Cfg. (800 14999 5 0 8613 0 1 2F3FCF9DE)
! Wed May 12 10:54:31 1998 ./makes3p.ckt
# GHZ  S  MA  R  50.0000
! SCATTERING PARAMETERS :
1.00000 0.99602 -7.18399 0.06256 84.5886 0.06256 84.5886
0.06256 84.5886 0.99801 -3.59201 0.00196 172.762
0.06256 84.5886 0.00196 172.762 0.99801 -3.59201
2.00000 0.98439 -14.2738 0.12351 79.2690 0.12351 79.2690
0.12351 79.2690 0.99220 -7.13736 0.00775 165.618
0.12351 79.2690 0.00775 165.618 0.99220 -7.13736
3.00000 0.96600 -21.1845 0.18146 74.1238 0.18146 74.1238
0.18146 74.1238 0.98299 -10.5954 0.01704 158.654
0.18146 74.1238 0.01704 158.654 0.98299 -10.5954
4.00000 0.94210 -27.8472 0.23535 69.2195 0.23535 69.2195
0.23535 69.2195 0.97101 -13.9360 0.02938 151.939
0.23535 69.2195 0.02938 151.939 0.97101 -13.9360
5.00000 0.91416 -34.2126 0.28449 64.6032 0.28449 64.6032
0.28449 64.6032 0.95696 -17.1412 0.04424 145.522
0.28449 64.6032 0.04424 145.522 0.95696 -17.1412
```

While reading the file, the translator ignores all comment lines beginning with '!'. If a scale factor is presented on the line beginning with '#', the translator ignores the -f option. If the scale factor is missing on the line, the translator assumes the scale factor to be HZ unless you use the -f option. The scale factor can be HZ, KHZ, MHZ, GHZ, or THZ in uppercase or lowercase letters. If an impedance value is given on this line, it needs to follow the key character R and a space. The complex number format needs to be specified on the line beginning with '#' and can be RI, MA, or DB in uppercase letters:

RI The complex number is expressed in the form of real and imaginary.

MA The complex number is expressed in the form of magnitude and angle.

DB The complex number is expressed in the form of db (magnitude) and angle.

The data section needs to only include frequency and S-parameter data. The data entries at each frequency point are required to comply with the following formats.

**Single port:**
```
freq. S11(mag) S11(deg)
```

**Two ports:**
```
freq. S11(mag) S11(deg) S21(mag) S21(deg) S12(mag) S12(deg) S22(mag) S22(deg)
```

**Three ports:**
```
freq. S11(mag) S11(deg) S21(mag) S21(deg) S12(mag) S12(deg) S22(mag) S22(deg)
```
and so on, where the complex number format is specified as MA on the # line. The frequency entry must be first and can include a scale factor. The data entries need to be separated with at least a space. The data entries at the first frequency point specify the number of ports.

MHARM Format

The following is an example MHARM file with three frequency data points and three ports.

```
1.0GHz  0.99602  -7.18399  0.06256  84.5886  0.06256  84.5886
  0.06256  84.5886  0.99801  -3.59201  0.00196  172.762
  0.06256  84.5886  0.00196  172.762  0.99801  -3.59201

2.0GHz  0.98439  -14.2738  0.12351  79.2690  0.12351  79.2690
  0.12351  79.2690  0.00775  165.618
  0.12351  79.2690  0.00775  165.618  0.99220  -7.13736

An input file for MHARM format has no header. The first line must be the data line. The first data entry at each frequency must be the frequency value, which can include a frequency scale factor. If the frequency scale factor is not given as shown previously, it can be given by using the command option \(-f\). In general, the data entry format is required to be the same as described for the LIBRA format.

HPMNS Format

An input file for HPMNS format is expected to have a header preceding its data section. The following is an HPMNS S-parameter file with two ports and two frequency points.

```
File Format: MDS
Revision:  2.01
Title: sp.net
Date: Oct  6 17:15:53 1998
Plotname: SP scatter[1] <sp.net>.  freq=(2 GHz->18 GHz)
Flags: complex
No. Sweep Variables: 0
Sweep Variables: 
No. Variables: 10
Variables:  0 freq  frequency type=real indep=yes mixop=neg
  1  S[1,1]  s-param type=complex indep=no
  2  S[1,2]  s-param type=complex indep=no
  3  S[2,1]  s-param type=complex indep=no
  4  S[2,2]  s-param type=complex indep=no
When reading a file for HPMNS format, the translator ignores anything before the line beginning with Variables. Specify the S-parameter variable in the variable definition section as described in the previous example, which is between the Variables line and Values line. Data entries must comply with the formats specified. Use a comma to separate the real and imaginary numbers. The index number is the first number of each frequency entry line, starting from 0 and increasing monotonically. The translator reads the frequency value from the second entry on each index line; the frequency value must be given explicitly, without a hidden scale factor. The translator reads the S-parameter data from the first line to the $K_2 + 1$ line after each index line for a k-port file. The impedance values of the ports pick up the data from the $K_2 + 2$ line to the $K_2 + K + 2$ line and are read only at the first frequency point. Each port impedance can be treated only as frequency independent. The translator ignores the data between the $K_2 + K + 2$ line and the next index line. The data file must mark the end of the file with #.

**LNM Format**

The following is an example of an LNM file. This file has two sets of S-parameter data, which correspond to two width values.

```
! MS bend model as a function of frequency, strip width (W) and dielectric constnat (ER) !
! 10GHz <= FREQ <= 30GHz
! 0.6mm <= W <= 0.8mm
! 2 <= ER <= 7
```
VAR MODELFMT = MAFET NEUTRAL MODEL FORMAT
VAR NPORTS = 2 ; 2-ports
VAR MODELTYP = S ; S-parameters
VAR FORMAT = RI ; S data in real-imaginary format
VAR R = 50;
VAR PARAMETER = W UNIT:MM INTERPOLATION:YES
VAR PARAMETER = ER INTERPOLATION:YES
VAR PARAMETER = FREQ UNIT:GHZ INTERPOLATION:YES
!
VAR W = 0.6
VAR ER = 2
BEGIN ACDATA
% FREQ reS11 imS11 reS21 imS21 reS12 imS12 reS22 imS22
10.0000 -0.05 -0.24 0.96 -0.12 0.96 -0.12 -0.01 -0.24
15.0000 -0.10 -0.33 0.92 -0.18 0.92 -0.18 -0.03 -0.34
20.0000 -0.16 -0.40 0.87 -0.24 0.87 -0.24 -0.07 -0.42
25.0000 -0.23 -0.44 0.81 -0.30 0.81 -0.30 -0.11 -0.49
30.0000 -0.29 -0.47 0.75 -0.35 0.75 -0.35 -0.18 -0.53
END
!
VAR W = 0.8
VAR ER = 2
BEGIN ACDATA
% FREQ reS11 imS11 reS21 imS21 reS12 imS12 reS22 imS22
10.0000 -0.05 -0.22 0.96 -0.17 0.96 -0.17 -0.03 -0.23
15.0000 -0.10 -0.31 0.91 -0.26 0.91 -0.26 -0.08 -0.31
20.0000 -0.16 -0.37 0.85 -0.35 0.85 -0.35 -0.14 -0.37
25.0000 -0.21 -0.41 0.78 -0.43 0.78 -0.43 -0.23 -0.40
30.0000 -0.27 -0.43 0.69 -0.51 0.69 -0.51 -0.34 -0.38
END

A line beginning with ! is treated as a comment and ignored. The translator reads lines beginning with VAR, which is uppercase, to pick up the number of ports given by parameter NPORT, impedance value given by R, frequency scale factor given by FREQ UNIT, and complex number format given by FORMAT. The possible values of and treatments for the frequency scale factor and the complex number format comply with what was described for LIBRA format. The translator treats a line beginning with % as the S-parameter format definition. Data is expected after the % line. Currently, the translator supports only a single set of data. Anything after the first END line is ignored.

Spectre Format

A Spectre S-parameter file for three ports looks as follows:

; S-parameter data file 'port2.data'.
; Generated by spectre from circuit file 'gendata' during analysis swp.
; 12:13:06 PM, Fri May 8, 1998
reference resistance
    port3=137 ; is port p3
    port2=137 ; is port p2
    port1=137 ; is port p1
format freq:  s33(real,imag) s23(real,imag)
              s13(real,imag) s32(real,imag)
              s22(real,imag) s12(real,imag)
              s31(real,imag) s21(real,imag)
An S-parameter file for Spectre format must have a header. The header must have a comment beginning with a semicolon as the first line, must define the reference resistance of ports and the S-parameter formats, and can include any number of comment and blank lines. When reading the file, the translator ignores all the lines beginning with semicolons, spaces, commas, and newlines in the header. The translator reads the numbers immediately after = on the lines after the reference resistance line as impedance data of the ports. The format section is treated as the format definition of S-parameter data entries. You can enter the S-parameters in any order, but the frequency must be first and is separated from the S-parameters with a colon. Each S-parameter can be expressed as (real, imag), (mag, deg), (mag, rad), (db, deg), or (db, rad). You can use commas to separate the two parts of an S-parameter. Any number of frequency points can be presented. They do not need to be equally spaced, but the frequency index must be monotonic, and the frequency data must be given explicitly, with no hidden scale factors. There is no limit to the number of ports. S-parameters use the syntax S13:15 when either port number is greater than 9.

### Output File Syntax

Output file syntax is described in this section. Some notes are given as follows:

- **[time]** Shows the time when the output file is generated.
- **[file]** Shows the input file name.
- **[format]** Shows the complex number format definition.
- **[resistance]** Shows the input impedance value of ports or 50 if no input impedance is given.
LIBRA Format

A LIBRA output file has the following form:

```
! Libra (TM) Ver. 3.500.103.3
! [time]
! Translated file: [file]
# S [format] R [resistance]
! SCATTERING PARAMETERS :
```

[Data section] shows the frequency and S-parameter data in the format described in “LIBRA Format” on page 121.

MHARM Format

The output MHARM file complies with the format described in “MHARM Format” on page 123.

HPMNS Format

The general format for an HPMNS output file is as follows:

File Format: Spectre format converted to MDS-like format.
Some editing required to make it true MDS format.
Title: [file]
Date: [time]
No. Variables: [number of variables]
Variables: 0  freq  frequency  type=real  indep=yes
  1  S[1,1]  s-param type=complex  indep=no
  2  S[1,2]  s-param type=complex  indep=no1
    .
    k^2  S[k,K]  s-param type=complex  indep=no
    k^2+1  PORTZ[1]  port-impedance type=complex  indep=n0
    .
    K^2+K  PORT[K]  port-impedance type=complex  indep=n0
Values:
[Data section]
```

k is the number of ports. [number of variables] shows the total number of frequency, S-parameters, and ports. [Data section] shows S-parameter data in the format specified in the Variables section of the header.

LNM Format

An output file for LNM format looks like the following:

```
! LNM format
VAR MODELFMT  = MAFET NEUTRAL MODEL FORMAT
VAR NPORTS    = [number]
```
VAR MODELTYPE = S
VAR FORMAT = [format]
VAR R = [resistance]
BEGIN ACDATA
% [S-parameter format]
[Data Section]

[number] shows the number of ports. [S-parameter format] specifies the S-parameter data format that must be followed. The S-parameter data for a LNM output file is in the LIBRA format.

Spectre Format

An output file for Spectre format looks like the following:

; S-parameter data translated from file 'file'.
; [time]
reference resistance
  port1=[resistance]
  .
  portk=[resistance]
format freq: S11(mag,deg) S21(mag,deg)
  .
  $kk(mag,deg)
[Data section]

Comment lines can appear before the [time] line. These comment lines are notes on complex number formats in the input file. [Data section] enters S-parameter data in the format specified in the header.

Error Handling

The translator provides the following error-handling mechanisms.

- When the incorrect filename has been given, the translator exits with an error message such as sptr: Unable to open input file 'filename'. No such file or directory.

- When the input file format does not match the -sim option, the translator exits and reports an error message such as Input format does not match -sim option.

- When reading a file that is inconsistent with the standard format (for example, upon scanning a line in the data section that does not match the specified format), the translator exits with an error message such as sptr: error in file at line XXX.

Generally, the translator ignores blank lines in data sections when reading a file.
Analyses

This chapter discusses the following topics:

- **Types of Analyses** on page 129
- **Analysis Parameters** on page 133
- **Probes in Analyses** on page 133
- **Multiple Analyses** on page 134
- **Multiple Analyses in a Subcircuit** on page 136
- **DC Analysis** on page 138
- **AC Analysis** on page 139
- **Transient Analysis** on page 140
- **Other Analyses** (sens, fourier, dcmatch, and stb) on page 147
- **Advanced Analyses** (sweep and montecarlo) on page 156
- **Special Analysis** (Hot-Electron Degradation) on page 175

### Types of Analyses

This section gives a brief description of the Spectre® circuit simulator analyses you can specify. Spectre analyses frequently let you sweep parameters, estimate or specify the DC solution, specify options that promote convergence, and select annotation options. You can specify sequences of analyses, any number in any order. For a more detailed description of each analysis and its parameters, consult the Spectre online help (spectre -h).

- **DC analysis** (dc)—Finds the DC operating point or DC transfer curves of the circuit.
- **AC/small signal analyses**
  - AC analysis (ac)—Linearizes the circuit about the DC operating point and computes the steady-state response of the circuit to a given small-signal sinusoidal stimulus.
This analysis is useful for obtaining small-signal transfer functions. For more information about the AC analysis, see Chapter 4, “Analysis Statements,” of the Spectre Circuit Simulator Reference.

- **Noise analysis (noise)**—Linearizes the circuit about the DC operating point and computes the total-noise spectral density at the output. The output can be either a voltage or a current. If you specify an input probe, the Spectre simulator computes the transfer function and the equivalent input-referred noise for a noise-free network. For more information about the noise analysis, see Chapter 4, “Analysis Statements,” of the Spectre Circuit Simulator Reference.

- **Transfer function analysis (xf)**—Linearizes the circuit about the DC operating point and performs a small-signal analysis. It calculates the transfer function from every source in the circuit to a specified output. The output can be either a voltage or a current. For more information about the transfer function analysis, see Chapter 4, “Analysis Statements,” of the Spectre Circuit Simulator Reference.

- **S-parameter analysis (sp)**—Linearizes the circuit about the DC operating point and computes S-parameters of the circuit taken as an N-port. You define the ports of the circuit with port statements. You must place at least one port statement in the circuit. The Spectre simulator turns on each port sequentially and performs a linear small-signal analysis. The Spectre simulator converts the response of the circuit at each port into S-parameters. For more information about the S-parameter analysis, see Chapter 4, “Analysis Statements,” of the Spectre Circuit Simulator Reference.

### Transient analyses

- **Transient analysis (tran)**—Computes the transient response of the circuit over a specified time interval. You can specify initial conditions for this analysis. If you do not specify initial conditions, the analysis starts from the DC steady-state solution. You can influence the speed of the simulation by setting parameters that control accuracy requirements and the number of data points saved. For more information about the transient analysis, see “Transient Analysis” on page 140,

- **Time-domain reflectometer analysis (tdr)**—Linearizes the circuit about the DC operating point and computes the reflection and transmission coefficients versus time. This is the time-domain equivalent of the S-parameter analysis. For more information about time-domain reflectometer analysis, see Chapter 4, “Analysis Statements,” of the Spectre Circuit Simulator Reference.

### RF analyses

- **Envelope-Following Analysis (envlp)**—Computes the envelope response of a circuit. The simulator automatically determines the clock period by looking through all the sources with the specified name. Envelope-following analysis is most efficient
for circuits where the modulation bandwidth is orders of magnitude lower than the clock frequency. This is typically the case, for example, in circuits where the clock is the only fast varying signal and other input signals have a spectrum whose frequency range is orders of magnitude lower than the clock frequency. For another example, the down conversion of two closely placed frequencies can also generate a slow-varying modulation envelope. The analysis generates two types of output files, a voltage versus time (td) file, and an amplitude/phase versus time (fd) file for each of specified harmonic of the clock fundamental. For more information about the envelope-following analysis, see SpectreRF Help.

- **Periodic AC analysis** (pac)—After the periodic steady-state analysis computes a periodic solution, this analysis computes the response of a circuit in periodic steady-state to a small sinusoidal stimulus. The frequency of the sinusoidal stimulus is not constrained by the period of the large-periodic solution. The periodic AC analysis lets you sweep the frequency over a range of values. This analysis is similar to the AC analysis, except that it linearizes the circuit about a periodically varying operating point and accurately models frequency conversion effects. For more information about the periodic AC analysis, see SpectreRF Help.

- **Periodic distortion analysis** (pdisto)—Models periodic distortion and includes harmonic effects. pdisto computes both a large signal, the periodic steady-state response of the circuit, and also the distortion effects of a specified number of moderate signals, including the distortion effects of the number of harmonics that you choose. For more information about the periodic distortion analysis, see SpectreRF Help.

- **Periodic noise analysis** (pnoise)—After the periodic steady-state analysis computes a periodic solution, this analysis linearizes the circuit about the periodic steady-state and performs a small-signal analysis. The small-signal analysis computes the total-noise spectral density at the output. If you specify an input probe, the transfer function and the input noise for an equivalent noise-free network are computed. pnoise is like noise except it models frequency conversion effects and the effect of time-varying bias on noise sources. For more information about the periodic noise analysis, see SpectreRF Help.

- **Periodic steady-state analysis** (pss)—Computes the periodic steady-state response of a circuit at either a given fundamental frequency or the corresponding steady-state analysis period. The pss analysis also determines the circuit's periodic operating point. This operating point is used for periodic time-varying small-signal analyses, such as pac, pxf, and pnoise. For more information about the periodic steady-state analysis, see SpectreRF Help.

- **Periodic transfer function analysis** (pxf)—After the periodic steady-state analysis computes a periodic solution, this analysis linearizes the circuit about the periodic steady-state and performs a small-signal analysis. The small-signal analysis
computes the transfer function from every independent source in the circuit to a designated output. The variable of interest at the output can be voltage or current. The analysis is similar to the transfer function analysis, except that it linearizes the circuit about a periodically varying operating point and accurately models frequency conversion effects. For more information about the periodic transfer function analysis, see SpectreRF Help.

Other analyses

- Sensitivity analysis (sens)—Determines the sensitivity of output variables to input design parameters. The results are expressed as a ratio of the change in an output analysis variable to the change in an input design parameter. The output for the sens command is sent to the rawfile or to an ASCII file. For more information about sensitivity analysis, see “Sensitivity Analysis” on page 147.

- Fourier analysis (fourier)—Measures the Fourier coefficients of two different signals at a specified fundamental frequency without loading the circuit. The algorithm used is based on the Fourier integral rather than the discrete Fourier transform and therefore is not subject to aliasing. Even on broad-band signals, it computes a small number of Fourier coefficients accurately and efficiently. Therefore, this Fourier analysis is suitable on clocked sinusoids generated by sigma-delta converters, pulse-width modulators, digital-to-analog converters, sample-and-holds, and switched-capacitor filters as well as on the traditional low-distortion sinusoids produced by amplifiers or filters. For more information about the Fourier analysis, see Chapter 4, “Analysis Statements,” of the Spectre Circuit Simulator Reference.

- DC Match Analysis (dcmatch)—Computes the statistical deviation in the DC operating point of the circuit caused by device mismatch. For more information about the dcmatch analysis, see DC Match Analysis on page 151

- Stability Analysis (stb)—Linearizes the circuit about the DC operating point and computes loop gain, gain margin, and phase margin for a specific feedback loop or an active device. The stability of the circuit can be determined from the loop gain waveform. The probe parameter must be specified to perform stability analysis. For more information about the stability analysis, see Chapter 4, “Analysis Statements,” of the Spectre Circuit Simulator Reference.

Advanced analyses

- Sweep analysis (sweep)—Sweeps a parameter executing a list of analyses (or multiple analyses) for each value of the parameter. The sweeps can be linear or logarithmic. Swept parameters return to their original values after the analysis. Sweep statements can be nested. For more information about the sweep analysis, see “Sweep Analysis” on page 156.
Monte Carlo analysis (montecarlo)—Varies netlist parameters according to specified distributions and correlations, runs nested child analyses, and extracts specified circuit-performance measurements. You can apply both process and device-to-device mismatch variations and tag device instances as correlated or “matched pairs.” Use the Cadence analog circuit design environment Calculator expressions to measure the circuit performance. You can use the analog design environment graphics tools to plot scalar performance data, such as slew rates and bandwidths, as a histogram or scattergram. You can also display waveform data as cloud (family) plots. For more information about the Monte Carlo analysis, see “Monte Carlo Analysis” on page 161. For more information about using the Monte Carlo analysis with the analog design environment, see the Advanced Analysis Tools User Guide.

Hot-electron degradation analysis—Lets you control the age of the circuit when simulating hot-electron degradation. For more information about the hot-electron degradation analysis, see “Special Analysis (Hot-Electron Degradation)” on page 175.

Analysis Parameters

You specify parameter values for analysis and control statements just as you specify those for component and model statements, but many analysis parameters have no assigned default values. You must assign values to these parameters if you want to use them. To assign values to these parameters, simply follow the parameter keyword with an equal sign (=) and your selected value. For example, to set the points per decade (dec) value to 10, you enter dec=10.

Note: Some parameters require text strings, usually filenames, as values. You must enclose these text strings in quotation marks to use them as parameter values.

When analysis parameters do have default values, these values are given in the parameter listings for that analysis in the Spectre online help (spectre -h).

A listing like the following tells you that the default value for parameter lin is 50 steps:

```
lin=50
```

Emission coefficient parameters

Probes in Analyses

Some Spectre analyses require that you set probes. Remember the following guidelines when you set probes:

You can name any component instance as a probe.
If the probe component measures a branch current, you can use it as either a current
probe or a voltage probe. Component instances that do not calculate branch currents
can be used only as voltage probes.

If the probe component has more than two terminals, you specify which pair of terminals
to use as the probe by specifying a port of the probe. In the following instance statement,
port 1 is nodes 5 and 8, and port 2 is nodes 2 and 4.

```
bjt1 5 8 2 4 bmod1
```

If the probe component measures more than one branch current, you specify which
branch current to use as the probe by specifying a port. In the following instance
statement, current port 1 is the branch from node 4 to node 5, and port 2 is the branch
from node 8 to node 9.

```
tline2 4 5 8 9 tline
```

Every component has a default probe type and port number.

In the following example, the netlist contains a resistor named Rcom and a voltage source
named Vcm. These components are used as probes for the
noise analysis statement. The parameters oprobe and iprobe specify the probe
components, and the parameters oportv and iportv specify the port numbers.

```
cmNoise noise start=1k stop=1G dec=10 oprobe=Rcom \
oportv=1 iprobe=Vcm iportv=1
```

**Multiple Analyses**

This netlist demonstrates the Spectre simulator’s ability to run many analyses in the order you
prefer. In this example, the Spectre simulator completely characterizes an operational
amplifier in one run. In analysis OpPoint, the program computes the DC solution and saves
it to a state file whose name is derived from the name of the netlist file. On subsequent runs,
the Spectre simulator reads the state information contained in this state file and speeds
analysis by using this state information as an initial estimate of the solution.

Analysis Drift computes DC solutions as a function of temperature. The Spectre simulator
computes the solution at the initial temperature and saves this solution to a state file to use
as an estimate in the next analysis and in subsequent simulations.

Analysis XferVsTemp computes the small-signal characteristics of the amplifier versus
temperature. Analysis XferVsTemp starts up quickly because it begins with the initial
temperature of the DC solution that was placed in a state file by the previous analysis.

Analysis OpenLoop computes the loop gain of an amplifier in closed-loop configuration.
Analysis OpenLoop starts quickly because it begins with the initial temperature of the DC
solution that was placed in a state file during analysis OpPoint.
Analysis XferVsFreq computes several small-signal quantities of interest such as closed-loop gain, the rejection ratio of the positive and negative power supply, and output resistance. The analysis again starts quickly because the operating point remains from the previous analysis.

Analysis StepResponse computes the step response that permits the measurement of the slew-rate and settling times. The alter statement please4 then changes the input stimulus from a pulse to a sine wave. Finally, the Spectre simulator computes the response to a sine wave in order to calculate distortion.

// ua741 operational amplifier
global gnd vcc vee
simulator lang=spectre
Spectre options audit=detailed limit=delta maxdeltav=0.3 \
  save=lvlpub nestlvl=1
// ua741 operational amplifier
model NPNdiode diode is=.1f imax=5m
model NPNbjt bjt type=npn bf=80 vaf=50 imax=5m \
  cje=3p cjc=2p cjs=2p tf=.3n tr=6n rb=100
model PNPbjt bjt type=pnp bf=10 vaf=50 imax=5m \
  cje=6p cjc=4p tf=1n tr=20n rb=20
subckt ua741 (pIn nIn out)
  // Transistors
  Q1 1 pIn 3 vee NPNbjt
  Q2 1 nIn 2 vee NPNbjt
  Q3 5 16 3 vcc PNPbjt
  Q4 4 16 2 vcc PNPbjt
  Q5 5 8 7 vee NPNbjt
  Q6 4 8 6 vee NPNbjt
  Q7 vcc 5 8 vee NPNbjt
  Q9 16 1 vcc vcc PNPbjt
  Q14 vcc 13 15 vee NPNbjt
  Q16 vcc 4 9 vee NPNbjt
  Q17 11 9 10 vee NPNbjt
  Q18 13 12 17 vee NPNbjt
  Q20 vee 17 14 vcc PNPbjt
  Q23 vee 11 17 vcc PNPbjt
  // Diodes
  Q8 vcc 1 NPNdiode
  Q19 13 12 NPNdiode
  // Resistors
  R1 7 vee resistor r=1k
  R2 6 vee resistor r=1k
  R3 8 vee resistor r=50k
  R4 9 vee resistor r=50k
  R5 10 vee resistor r=100
  R6 12 17 resistor r=40k
  R8 15 out resistor r=27
  R9 14 out resistor r=22
  // Capacitors
  C1 4 11 capacitor c=30p
  // Current Sources
  I1 16 vee isource dc=19u
  I2 vcc 11 isource dc=550u
Spectre Circuit Simulator User Guide
Analyses

```
I3  vcc 13 isource dc=180u
ends ua741
// Sources
Vpos  vcc gnd vsource dc=15
Vneg  vee gnd vsource dc=-15
Vin   pin gnd vsource type=pulse dc=0  
      val0=0 val1=10 width=100u period=200u rise=2u  
      fall=2u tdi=0 tau1=20u tau2=100u  
      freq=10k ampl=10 delay=5u  
      file="sine10" scale=10.0 stretch=200.0e-6
Vfb   nin out vsource
// Op Amps
OA1   pin nin out ua741
// Resistors
Rload out gnd resistor r=10k
// Analyses
// DC operating point
please1 alter param=temp value=25 annotate=no
OpPoint dc print=yes readns="%C:r.dc25"
write="%C:r.dc25"
// Temperature Dependence
Drift dc start=0 stop=50.0 step=1 param=temp  
readns="%C:r.dc0" write="%C:r.dc0"
XferVsTemp xf start=0 stop=50 step=1 probe=Rload  
param=temp freq=1kHz readns="%C:r.dc0"
// Gain
please2 alter dev=Vfb param=mag value=1 annotate=no
OpenLoop ac start=1 stop=10M dec=10 readns="%C:r.dc25"
please3 alter dev=Vfb param=mag value=0 annotate=no
// XF
XferVsFreq xf start=1_Hz stop=10M dec=10 probe=Rload
// Transient
StepResponse tran stop=250u
please4 alter dev=Vin param=type value=sine
SineResponse tran stop=150u
```

Multiple Analyses in a Subcircuit

You might want to run complex sets of analyses many times during a simulation. To simplify this process, you can group the set of analyses into a subcircuit. Because subcircuit definitions can contain analyses and control statements, you can put the analyses inside a single subcircuit and perform the multiple analyses with one call to the subcircuit. The Spectre simulator performs the analyses in the order you specify them in the subcircuit definition. Generally, you do not mix components and analyses in the same subcircuit definition. For more information about formats for subcircuit definitions and subcircuit calls, see Chapter 6, “Analyses.”
Example

The following example illustrates how to create and call subcircuits that contain analyses.

Creating Analysis Subcircuits

```
subckt sweepVcc()
  parameters start=0 stop=10 Ib=0 omega=1G steps=100
  setIbb alter dev=Ibb param=dc value=Ib
  SwpVccDC dc start=start stop=stop dev=Vcc lin=steps/2SwpVccAC ac dev=Vcc
  start=start stop=stop lin=steps \freq=omega/6.283185
ends sweepVcc
```

This example defines a subcircuit called `sweepVcc` that contains the following:

- A list of parameters with default values for `start`, `stop`, `Ib`, `omega`, and `steps`

  This list of defaults is optional.

  These defaults are for the subcircuit call. For example, if you call `sweepVcc` and do not specify values for the `start` and `stop` parameters in the subcircuit call, the sweeps for analyses `SwpVccDC` and `SwpVccAC` start at 0 and end at 10, the values specified as defaults. If, however, you specify `start=1` and `stop=5` as parameter values in the subcircuit call, the `start` and `stop` parameters in `SwpVccDC` and `SwpVccAC` take the values 1 and 5, respectively.

- A control statement, `setIbb`, which alters the `dc` parameter of the component named `Ibb` to the numerical value of `Ib`

- Two analysis statements, `SwpVccDC` and `SwpVccAC`, which run a DC analysis followed by an AC analysis

Calling Analysis Subcircuits

Each subcircuit call for `sweepVcc` in the netlist causes all the analyses in the `sweepVcc` to be performed. Each of the following statements is a subcircuit call to subcircuit `sweepVcc`:

- `Ibb1uA sweepVcc stop=2 Ib=1u`
- `Ibb3uA sweepVcc stop=2 Ib=3u`
- `Ibb10uA sweepVcc stop=2 Ib=10u`
- `Ibb30uA sweepVcc stop=2 Ib=30u`
- `Ibb100uA sweepVcc stop=2 Ib=100u`

Note the following important syntax features:

- Each subcircuit call has a unique name.
- Each subcircuit call overrides the default values for the `stop` and `Ib` parameters.
The start, steps, and omega parameters are not defined in the subcircuit calls. They take the default values assigned in the subcircuit.

DC Analysis

The DC analysis finds the DC operating point or DC transfer curves of the circuit. To generate transfer curves, specify a parameter and a sweep range. The swept parameter can be circuit temperature, a device instance parameter, a device model parameter, a netlist parameter, or a subcircuit parameter for a particular subcircuit instance. You can sweep the circuit temperature by giving the parameter name as `param=temp` with no `dev`, `mod`, or `sub` parameter. You can sweep a top-level netlist parameter by giving the parameter name with no `dev`, `mod`, or `sub` parameter. You can sweep a subcircuit parameter for a particular subcircuit instance by specifying the subcircuit instance name with the `sub` parameter and the subcircuit parameter name with the `param` parameter. After the analysis has completed, the modified parameter returns to its original value.

The syntax is as follows:

```
Name dc parameter=value ...
```

You can specify sweep limits by giving the end points or by providing the center value and the span of the sweep. Steps can be linear or logarithmic, and you can specify the number of steps or the size of each step. You can give a step size parameter (`step, lin, log, dec`) and determine whether the sweep is linear or logarithmic. If you do not give a step size parameter, the sweep is linear when the ratio of stop to start values is less than 10, and logarithmic when this ratio is 10 or greater. If you specify the `oppoint` parameter, Spectre computes and outputs the linearized model for each nonlinear component.

Nodesets help find the DC or initial transient solution. You can supply them in the circuit description file with nodeset statements, or in a separate file using the `readns` parameter. When nodesets are given, Spectre computes an initial guess of the solution by performing a DC analysis while forcing the specified values onto nodes by using a voltage source in series with a resistor whose resistance is `rforce`. Spectre then removes these voltage sources and resistors and computes the true solution from this initial guess.

Nodesets have two important uses. First, if a circuit has two or more solutions, nodesets can bias the simulator towards computing the desired one. Second, they are a convergence aid. By estimating the solution of the largest possible number of nodes, you might be able to eliminate a convergence problem or dramatically speed convergence.

When you simulate the same circuit many times, we suggest that you use both the `write` and `readns` parameters and give the same filename to both parameters. The DC analysis then converges quickly even if the circuit has changed somewhat since the last simulation, and the nodeset file is automatically updated.
You may specify values to force for the DC analysis by setting the parameter force. The values used to force signals are specified by using the force file, the ic statement, or the ic parameter on the capacitors and inductors. The force parameter controls the interaction of various methods of setting the force values. The effects of individual settings are

force=none: Any initial condition specifiers are ignored.

force=node: The ic statements are used, and the ic parameter on the capacitors and inductors are ignored.

force=dev: The ic parameters on the capacitors and inductors are used, and the ic statements are ignored.

force=all: Both the ic statements and the ic parameters are used, with the ic parameters overriding the ic statements.

If you specify a force file with the readforce parameter, force values read from the file are used, and any ic statements are ignored.

Once you specify the force conditions, the Spectre simulator computes the DC analysis with the specified nodes forced to the given value by using a voltage source in series with a resistor whose resistance is rforce (see options).

AC Analysis

The AC analysis linearizes the circuit about the DC operating point and computes the response to all specified small sinusoidal stimulus. For more information on specifying small sinusoidal stimulus, see Chapter 4, Component Statements Part 2, in the Spectre Simulator Reference.

The Spectre simulator can perform the analysis while sweeping a parameter. The parameter can be frequency, temperature, component instance parameter, component model parameter, or netlist parameter. If changing a parameter affects the DC operating point, the operating point is recomputed on each step. You can sweep the circuit temperature by giving the parameter name as temp with no dev or mod parameter. You can sweep a netlist parameter by giving the parameter name with no dev or mod parameter. After the analysis has completed, the modified parameter returns to its original value.

The syntax is as follows:

Name ac parameter=value ...

You can specify sweep limits by giving the end points or by providing the center value and the span of the sweep. Steps can be linear or logarithmic, and you can specify the number of
steps or the size of each step. You can give a step size parameter (step, lin, log, dec) to determine whether the sweep is linear or logarithmic. If you do not give a step size parameter, the sweep is linear when the ratio of stop to start values is less than 10, and logarithmic when this ratio is 10 or greater. All frequencies are in Hertz.

The small-signal analysis begins by linearizing the circuit about an operating point. By default, this analysis computes the operating point if it is not known or recomputes it if any significant component or circuit parameter has changed. However, if a previous analysis computed an operating point, you can set prevoppoint=yes to avoid recomputing it. For example, if you use this option when the previous analysis was a transient analysis, the operating point is the state of the circuit on the final time point.

**Transient Analysis**

You can adjust transient analysis parameters in several useful ways to meet the needs of your simulation. Setting different Spectre parameters that control the error tolerances, the integration method, and the amount of data saved let you choose between maximum speed and greatest accuracy in a simulation. You normally use reltol or errpreset to control accuracy, but you can use additional methods in special situations to improve performance.

This section also tells you about parameters you can set that improve transient analysis convergence.

**Trading Off Speed and Accuracy with a Single Parameter Setting**

You can set transient analysis speed and accuracy parameters individually, or you can set a group of parameters that control transient analysis accuracy with the errpreset parameter. You can set errpreset to three different values:

- liberal
- moderate
- conservative

At liberal, the simulation is fast but less accurate. The liberal setting is suitable for digital circuits or analog circuits that have only short time constants. At moderate, the default setting, simulation accuracy approximates a SPICE2 style simulator. At conservative, the simulation is the most accurate but also slowest. The conservative setting is appropriate for sensitive analog circuits. If you still require more accuracy than that provided by conservative, tighten error tolerance by setting reltol to a smaller value.
Description of errpreset Parameter Settings

The effect of errpreset on other parameters is shown in the following table. In this table, $T = stop - start$.

<table>
<thead>
<tr>
<th>errpreset</th>
<th>reltol</th>
<th>relref</th>
<th>method</th>
<th>maxstep</th>
<th>Iteratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>liberal</td>
<td>$\times 10.0$</td>
<td>allglobal</td>
<td>gear2</td>
<td>$\leq T/10$</td>
<td>3.5</td>
</tr>
<tr>
<td>moderate</td>
<td>$\times 1.0$</td>
<td>sigglobal</td>
<td>traponly</td>
<td>$\leq T/50$</td>
<td>3.5</td>
</tr>
<tr>
<td>conservative</td>
<td>$\times 0.1$</td>
<td>alllocal</td>
<td>gear2only</td>
<td>$\leq T/100$</td>
<td>10.0</td>
</tr>
</tbody>
</table>

The description in the previous table has the following exceptions:

- The errpreset parameter sets the value of reltol as described in the table, except that the value of reltol cannot be larger than 0.01.
- Except for reltol and maxstep, errpreset does not change the value of any parameters you have specified.
- With maxstep, the specified value in the preceding table is an upper bound. You can always specify a smaller maxstep value.

If you need to check the errpreset settings for a simulation, you can find these values in the log file.

Uses for errpreset

You adjust errpreset to the speed and accuracy requirements of a particular simulation. For example, you might set errpreset to liberal for your first simulation to see if the circuit works. After debugging the circuit, you might switch to moderate to get more accurate results. If the application requires high accuracy, or if you want to verify that the moderate solution is reasonable, you set errpreset to conservative.

You might also have different errpreset settings for different types of circuits. For logic gate circuits, the liberal setting is probably sufficient. A moderate setting might be better for analog circuits. Circuits that are sensitive to errors or circuits that require exceptional accuracy might require a conservative setting.
Controlling the Accuracy

You can control the accuracy of the solution to the discretized equation by setting the \( \text{reltol} \) and \( \text{abstol} \) (where \( x \) is the access quantity, such as \( v \) or \( i \)) parameters in an \textit{options} or \textit{set} statement. These parameters determine how well the circuit conserves charge and how accurately the Spectre simulator computes circuit dynamics and steady-state or equilibrium points.

You can set the integration error or the errors in the computation of the circuit dynamics (such as time constants), relative to \( \text{reltol} \) and \( \text{abstol} \) by setting the \textit{iteratio} parameter.

\[
\text{Tolerance}_{\text{NR}} = \text{abstol} + \text{reltol} \times \text{Ref} \\
\text{Tolerance}_{\text{LTE}} = \text{Tolerance}_{\text{NR}} \times \text{iteratio}
\]

The \text{Ref} value is determined by your setting for \text{relref}, the relative error parameter, as explained in “Adjusting Relative Error Parameters” on page 142.

In the previous equations, \( \text{Tolerance}_{\text{NR}} \) is a convergence criterion that bounds the amount by which Kirchhoff’s Current Law is not satisfied as well as the allowable difference in computed values in the last two Newton-Raphson (NR) iterations of the simulation. \( \text{Tolerance}_{\text{LTE}} \) is the allowable difference at any time step between the computed solution and a predicted solution derived from a polynomial extrapolation of the solutions from the previous few time steps. If this difference is greater than \( \text{Tolerance}_{\text{LTE}} \), the Spectre simulator shortens the time step until the difference is acceptable.

From the previous equations, you can see that tightening \( \text{reltol} \) to create more strict convergence criteria also diminishes the allowable local truncation error (\( \text{Tolerance}_{\text{LTE}} \)). You might not want the truncation error tolerance tightened because this adjustment can increase simulation time. You can prevent the decrease in the time step by increasing the \textit{iteratio} parameter to compensate for the tightening of \( \text{reltol} \).

Adjusting Relative Error Parameters

You determine the treatment of the relative error with the \text{relref} parameter. The \text{relref} parameter determines which values the Spectre simulator uses to compute whether the relative error tolerance (\( \text{reltol} \)) requirements are satisfied.

You can set \text{relref} to the following options:

- The \text{relref=pointlocal} setting compares the relative errors in quantities at each node relative to the current value of that node.
- The \text{relref=alllocal} setting compares the relative errors in quantities at each node to the largest value of that node for all past time points.
The `releff=sigglobal` or `releff=allglobal` settings compare relative errors in each of the circuit signals to the maximum of all the signals in the circuit.

In addition, the `releff=allglobal` setting compares equation residues (the amount by which Kirchhoff’s Current Law [also known as Kirchhoff’s Flow Law] is not satisfied) for each node to the maximum current floating onto the node at any time in that node’s past history.

### Setting the Integration Method

The `method` parameter specifies the integration method. You can set the `method` parameter to adjust the speed and accuracy of the simulation. The Spectre simulator uses three different integration methods: the backward-Euler method, the trapezoidal rule, and the second-order Gear method. The `method` parameter has six possible settings that permit different combinations of these three methods to be used. The following table shows the possible settings and what integration methods are allowed with each:

<table>
<thead>
<tr>
<th>Backward-Euler</th>
<th>Trapezoidal Rule</th>
<th>Second-Order Gear</th>
</tr>
</thead>
<tbody>
<tr>
<td>euler</td>
<td>☒</td>
<td></td>
</tr>
<tr>
<td>traponly</td>
<td></td>
<td>☒</td>
</tr>
<tr>
<td>trap</td>
<td>☒</td>
<td>☒</td>
</tr>
<tr>
<td>gear2only</td>
<td></td>
<td>☒</td>
</tr>
<tr>
<td>gear2</td>
<td>☒</td>
<td>☒</td>
</tr>
<tr>
<td>trapgear2</td>
<td>☒</td>
<td>☒</td>
</tr>
</tbody>
</table>

The trapezoidal rule is usually the best setting if you want high accuracy. This method can exhibit point-to-point ringing, but you can control this by tightening the error tolerances. The trapezoidal method is usually not a good choice to run with loose error tolerances because it is sensitive to errors from previous time steps. If you need to use very loose tolerances to get a quick answer, it is better to use second-order Gear.

While second-order Gear is more accurate than backward-Euler, both methods can overestimate a system’s stability. This effect is less with second-order Gear. You can also reduce this effect if you request high accuracy.
Artificial numerical damping can reduce accuracy when you simulate low-loss (high-Q) resonators such as oscillators and filters. Second-order Gear shows this damping, and backward-Euler exhibits heavy damping.

**Improving Transient Analysis Convergence**

If the circuit you simulate can have infinitely fast transitions (for example, a circuit that contains nodes with no capacitance), the Spectre simulator might have convergence problems. To avoid these problems, set $c_{\text{min}}$, which is the minimum capacitance to ground at each node, to a physically reasonable nonzero value.

You also might want to adjust the time-step parameters, $\text{step}$ and $\text{maxstep}$. $\text{step}$ is a suggested time step you can enter. Its default value is $0.001 \times (\text{stop} - \text{start})$. $\text{maxstep}$ is the largest time step permitted.

**Controlling the Amount of Output Data**

The Spectre simulator normally saves all computed data in the transient analysis. Sometimes you might not need this much data, and you might want to save only selected results. At other times, you might need to decrease the time interval between data points to get a more precise measurement of the activity of the circuit. You can control the number of output data points the Spectre simulator saves for the transient analysis in these ways:

- **With strobing**, which lets you select the time interval between the data points the Spectre simulator saves. The simulator forces a time step on each point it saves, so the data is computed, not interpolated.
- **With skipping time points**, which lets you select how many data points the Spectre simulator saves.
- **With data compression**, which eliminates repetitive recording of signal values that are unchanged.
- **With the $\text{outputstart}$ parameter**, which lets you specify the time when the Spectre simulator starts saving data points.
- **With the $\text{infotime}$ parameter**, which lets you specify specific times to save operating-point data instead of for all time points.
Telling the Spectre Simulator to Change the Time Interval between Data Points (Strobing)

Strobing changes the interval between data points. You use strobing to eliminate some unwanted high-frequency signal from the output, just as a strobe light appears to freeze rapidly rotating machinery. With strobing, you can demodulate AM signals or hide the effect of the clock in clocked waveforms. You can also dramatically improve the accuracy of external Fast Fourier Transform (FFT) routines. To perform strobing, you set the following parameters in the transient analysis:

- The strobeperiod parameter, which sets the time interval between the data points that the Spectre simulator saves
- The skipstart parameter (optional), which tells the Spectre simulator when to start strobing
  
  This parameter is also used to skip data points.
- The skipstop parameter (optional), which tells the Spectre simulator when to stop strobing
  
  This parameter is also used to skip data points.
- The strobedelay parameter (optional), which lets you set a delay between the skipstart time and the first strobe point

Telling the Spectre Simulator How Many Data Points to Save

By telling the Spectre simulator to save only every $Nth$ data point, you can reduce the size of the results database generated by the Spectre simulator. You tell the Spectre simulator to save every $Nth$ data point with the following parameters:

- With the skipcount parameter, which you set to $N$ to make the Spectre simulator save every $Nth$ data point
- The skipstart parameter, which tells the Spectre simulator when to start skipping parameters
  
  This parameter is also used in strobing.
- The skipstop parameter, which tells the Spectre simulator when to stop skipping parameters
  
  This parameter is also used in strobing.
Examples of Strobing and Skipping

In the following example, the Spectre simulator starts skipping data points at Time=10 seconds and continues to skip points until Time=35 seconds. During this 25-second period, the Spectre simulator saves only every third data point.

ExSkipSt tran skipstart=10 skipstop=35 skipcount=3

In this example, the Spectre simulator starts strobing at Time=5 seconds and continues until Time=20 seconds. During this 15-second period, the Spectre simulator saves data points every .10 seconds.

ExStrobe tran skipstart=5 skipstop=20 strobeperiod=.10

This example is identical to the previous one except that it sets a delay of 2 seconds between the skipstart time and the first strobe point.

ExStrobe2 tran skipstart=5 skipstop=20 \
   strobeperiod=.10 strobedelay=2

Data Compression

Some circuits, such as mixed analog and digital designs and circuits with switching power supplies, have substantial amounts of signal latency. If unchanged signal values for these circuits are repetitively written for each time point to a transient analysis output file, this output file can become very large. You can reduce the size of output files for such transient analyses with the Spectre simulator’s data compression feature. With data compression, the Spectre simulator writes output data for a signal only when the value of that signal changes.

Using data compression is not always appropriate. The Spectre simulator writes fewer signal values when you turn on data compression, but it must write more data for every signal value it records. For circuits with small amounts of signal latency, data compression might actually increase the size of the output file.

You turn on data compression by adding the parameter compression=yes to the transient analysis command line in a Spectre netlist.

DoTran_z12 tran start=0 stop=.003 step=0.00015 \
   maxstep=6e-06 compression=yes

You cannot apply data compression to operating-point parameters, including terminal currents that are calculated internally rather than with current probes. If you want data compression for terminal currents, you must specify that these currents be calculated with current probes. You can specify that all currents be calculated with current probes by placing useprobes=yes in an options statement.
Important

Adding probes to circuits that are sensitive to numerical noise might affect the solution. In such cases, an accurate solution might be obtained by reducing reltol.

Telling Spectre to Save Operating-Point Data at Specific Times

In addition to saving operating-point data in a dc analysis, Spectre allows this data to be saved during a transient analysis. This data is saved as a waveform and can be plotted along with node voltages and other output data. See Chapter 8, “Specifying Output Options,” to learn how to select and save this data. Often, all that is needed is the operating data at specific times, which can be achieved by linking an info analysis with the tran analysis.

To control the amount of data produced for operating-point parameters, use the following two transient analysis parameters to specify at which time points you would like to save operating point output for all devices:

\[
\text{infotimes}=\langle\text{vector of numbers}\rangle \\
\text{infoname}=\langle\text{analysis name}\rangle
\]

where analysis name points to an info analysis.

For example:

```plaintext
mytran tran stop=30n infotimes=[10n 25n] infoname=opinfo
opinfo info what=oppoint where=rawfile
```

The opinfo statement is called at 10 and 25 nanoseconds. The data for all devices is reported at these specified time points.

Other Analyses (sens, fourier, dcmatch, and stb)

There are four analyses in this category: sens, fourier, dcmatch, and stb.

Sensitivity Analysis

You can supplement the analysis information you automatically receive with the AC and DC analyses by placing sens statements in the netlist. Output for the sens command is sent to the rawfile or to an ASCII file that you can specify with the +sensdata <filename> option of the spectre command.
If you set senstype=partial, Spectre calculates partial sensitivity. This determines the sensitivity of output variables to input design parameters. Results are expressed as a ratio of the change in an output analysis variable to the change in an input design parameter. For example, if $V$ is the output value, and $P$ is the input design parameter, sensitivity is computed as follows:

$$S_{V,P} = \frac{dV}{dP}$$

If you set senstype=normalized, Spectre calculates normalized sensitivity, which removes the dependence of results on the magnitude of the output variable and input design parameters. Normalized sensitivity is computed as follows:

$$S_{V,P} = \frac{P dV}{V dP}$$

$$S_{V,P} = P \frac{dV}{dP} \text{ if } V=0$$

$$S_{V,P} = \frac{1}{V} \frac{dV}{dP} \text{ if } P=0$$

When both $P$ and $V$ are zero, partial sensitivity is used.

### Formatting the sens Command

You format the sens command as follows

```plaintext
sens [ ( output_variables_list ) ] [ to 
( design_parameters_list ) ] [ for ( analyses_list ) ]
```

where

- `output_variables_list = ovar_1 ovar_2 ...`
- `design_parameters_list = dpar_1 dpar_2 ...`
- `analyses_list = ana_1 ana_2 ...`

The `ovar_i` are the output variables whose sensitivities are calculated. These are normally node names, `deviceInstance:parameter` or `modelName:parameter` specifications. Examples are `5`, `n1`, and `Qout:betadc`.

The `dpar_j` are the design parameters to which the output variables are sensitive. You can specify them in a format similar to `ovar_i`. However, they must be input parameters that you can specify (for example, `R1:r`). If you do not specify a `to` clause, sensitivities of output variables are calculated with respect to all available instance and model parameters.
Note: The method for specifying design parameters and output variables is described in more detail in the documentation for the save statement in Chapter 8, “Specifying Output Options.”

The following table shows you the types of design and output parameters that are normally used for both AC and DC analyses:

<table>
<thead>
<tr>
<th></th>
<th>AC Analysis</th>
<th>DC Analysis</th>
</tr>
</thead>
<tbody>
<tr>
<td>Design parameters</td>
<td>Instance parameters</td>
<td>Instance parameters</td>
</tr>
<tr>
<td></td>
<td>Model parameters</td>
<td>Model parameters</td>
</tr>
<tr>
<td>Output parameters</td>
<td>Node voltages</td>
<td>Node voltages</td>
</tr>
<tr>
<td></td>
<td>Branch currents</td>
<td>Branch currents</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Instance operating-point parameters</td>
</tr>
</tbody>
</table>

You can also specify device instances or models as design parameters without further specifying parameters, but this approach might result in a number of error messages. The Spectre simulator attempts sensitivity analysis for every device parameter and sends an error message for each parameter that cannot be varied. The Spectre simulator does, however, perform the requested sensitivity analysis for appropriate parameters.

The \( \text{analyse}_k \) are the analyses for which sensitivities are calculated. These can be analysis instance names (for example, \text{opBegin} and \text{ac2}) or analysis type names (for example, \text{DC} and \text{AC}).

Examples of the sens Command

The following examples illustrate sens command format:

sens (q1:betadc 2 Out) to (vcc:dc nbjt1:rb) for (analDC)

This command computes DC sensitivities of the betadc operating-point parameter of transistor q1 and of nodes 2 and Out to the dc voltage level of voltage source vcc and to the model parameter rb of nbjt1. The values are computed for DC analysis analDC. The results are stored in the files analDC.vcc:dc and analDC.nbjt1:rb.

sens (1 n2 7) to (q1:area nbjt1:rb) for (analAC)

This command computes AC sensitivities of nodes 1, n2, and 7 to the area parameter of transistor q1 and to the model parameter rb of nbjt1. The values are computed for each frequency of the AC analysis analAC. The results are stored in the files analAC.q1:area and analAC.nbjt1:rb.
This command computes DC sensitivities of the branch current `vbb:p`, the operating-point parameter `gm` of transistor `q1`, the internal collector voltage `q1:int_c`, and the node 7 voltage to the instance parameter `q1:area` and the model parameter `nbjt:1:rb`. The values are computed for analysis `analDC1`.

This command computes the AC sensitivities of nodes 1, n2, and 7 to all available device and model parameters.

**Note:** This can result in a lot of information.

### Fourier Analysis

The ratiometric Fourier analyzer measures the Fourier coefficients of two different signals at a specified fundamental frequency without loading the circuit. The algorithm used is based on the Fourier integral rather than the discrete Fourier transform and therefore is not subject to aliasing. Even on broad-band signals, it computes a small number of Fourier coefficients accurately and efficiently. Therefore, this Fourier analyzer is suitable on clocked sinusoids generated by sigma-delta converters, pulse-width modulators, digital-to-analog converters, sample-and-holds, and switched-capacitor filters as well as on the traditional low-distortion sinusoids produced by amplifiers or filters.

The analyzer is active only during a transient analysis. For each signal, the analyzer prints the magnitude and phase of the harmonics along with the total harmonic distortion at the end of the transient analysis. The total harmonic distortion is found by summing the power in all of the computed harmonics except DC and the fundamental. Consequently, the distortion is not accurate if you request an insufficient number of harmonics. The Fourier analyzer also prints the ratio of the spectrum of the first signal to the fundamental of the second, so you can use the analyzer to compute large signal gains and immittances directly.

If you are concerned about accuracy, perform an additional Fourier transform on a pure sinusoid generated by an independent source. Because both transforms use the same time points, the relative errors measured with the known pure sinusoid are representative of the errors in the other transforms. In practice, this second Fourier transform is performed on the reference signal. To increase the accuracy of the Fourier transform, use the `points` parameter to increase the number of points. Tightening `reltol` and setting `errpreset=conservative` are two other measures to consider.

The accuracy of the magnitude and phase for each harmonic is independent of the number of harmonics computed. Thus, increasing the number of harmonics (while keeping `points` constant) does not change the magnitude and phase of the low order harmonics, but it does
improve the accuracy of the total harmonic distortion computation. However, if you do not specify points, you can increase accuracy by requesting more harmonics, which creates more points.

The large number of points required for accurate results is not a result of aliasing. Many points are needed because a quadratic polynomial interpolates the waveform between the time points. If you use too few time points, the polynomials deviate slightly from the true waveform between time points and all of the computed Fourier coefficients are slightly in error. The algorithm that computes the Fourier integral does accept unevenly spaced time points, but because it uses quadratic interpolation, it is usually more accurate using time steps that are small and nearly evenly spaced.

This device is not supported within altergroup.

**Instance Definition**

Name [p] [n] [pr] [nr] ModelName parameter=value ...
Name [p] [n] [pr] [nr] fourier parameter=value ...

The signal between terminals p and n is the test or numerator signal. The signal between terminals pr and nr is the reference or denominator signal. Fourier analysis is performed on terminal currents by specifying the term or refterm parameters. If both term and p or n are specified, then the terminal current becomes the numerator and the node voltages become the denominator. By mixing voltages and currents, it is possible to compute large signal immittances.

**Model Definition**

model modelName fourier parameter=value ...

**DC Match Analysis**

This analysis computes the 3-sigma variance of dc voltages or currents at the specified outputs due to device mismatches. You need to specify the mismatch parameters in the model cards for each device that contributes to the deviation. The analysis uses the device mismatch models to evaluate an equivalent mismatch current source that is placed in parallel to the device. These current sources have zero mean and some variance. The simulation results are displayed in descending order.

For MOSFET devices (bsim3v3), the analysis displays threshold voltage mismatch, current factor mismatch, gate voltage mismatch, and drain current mismatch. For bipolar devices (vbic), it displays base-emitter junction voltage mismatch. For resistors (limited to two terminal resistance), it displays resistor mismatches.
Spectre Circuit Simulator User Guide  
Analyses

The analysis replaces multiple simulation runs and tedious calculations in the estimation of mismatch performance. It automatically identifies the set of critical matched components during circuit design. For example, when there are matched differential pairs in the circuit, the contribution of two matched transistors will be equal in magnitude and opposite in sign.

Model Definition

name [pnode nnode] dcmatch parameter=value ...

Examples of the dcmatch command

The following example investigates the 3-sigma dc variation at the output of the current flowing through the device vd, which is a voltage source in the circuit netlist. dcmml is the name of the analysis, dcmmatch is a keyword indicating the dc mismatch analysis, and the parameter settings oprobe=vd and porti=1 specify that the output current is measured at the first port of vd. Device mismatch contributions less than 1e-3% of the maximum contribution of all mismatch devices to the output are not reported, as specified by the parameter mth. The mismatch (that is the equivalent mismatch current sources in parallel to all the devices that use model n1) is modeled by the model parameters mvtwl, mvtwl2, mvt0, mbewl, and mbe0.

dcmml dcmmatch mth=1e-3 oprobe=vd porti=1
model n1 bsim3v3 type=n ...
+ mvtwl=6.15e-9 mvtwl2=2.5e-12 mvt0=0.0 mbewl=16.5e-9 mbe0=0.0

The output of the analysis is displayed on the screen/logfile:

DC Device Matching Analysis ‘mismatch1’ at vd
Local Variation = 3-sigma random device variation
sigmaOut sigmaVth sigmaBeta sigmaVg sigmaIds
-13.8 uA 2.21 mV 357 m% 2.26 mV 1.71 % mp6
-6.99 uA 1.63 mV 269 m% 1.68 mV 1.08 % m01
-2.71 uA 1.11 mV 187 m% 1.16 mV 648 m% m02
-999 nA 769 uV 131 m% 807 uV 428 m% m04x4_4
-999 nA 769 uV 131 m% 807 uV 428 m% m04x4_3
-999 nA 769 uV 131 m% 807 uV 428 m% m04x4_2
-999 nA 769 uV 131 m% 807 uV 428 m% m04x4_1
-999 nA 769 uV 131 m% 807 uV 428 m% m04
-718 nA 1.09 mV 185 m% 1.15 mV 599 m% m40x04
-520 nA 1.55 mV 263 m% 1.63 mV 835 m% m20x04
-378 nA 2.21 mV 376 m% 2.34 mV 1.16 % m10x04
-363 nA 539 uV 92 m% 567 uV 293 m% m08
-131 nA 379 uV 64.9 m% 400 uV 203 m% m16
-46.7 nA  267 uV  45.9 m%  283 uV  142 m%  m32
vd = -3.477 mA +/- 15.91 µA (3-sigma variation)

This says that the 3-sigma variation at i(vd) due to the mismatch models is -3.477 mA ±15.91 µA. The -3.477 mA is the dc operating value of i(vdd), and 15.91 µA is the 3-sigma variation due to the device mismatches. The device mp6 contributes the most to the output variation at -13.8 µA followed by m01 which contributes -6.99 µA. The equivalent 3-sigma Vth variation of mp6 is 2.21 mV. The relative 3-sigma beta (current factor) variation of mp6 is 0.357%. The equivalent 3-sigma gate voltage variation is 2.26 mV. The relative 3-sigma Ids variation of mp6 is 1.71%.

The output can also be written in psf, and you can view the table using Analog Design Environment.

The following statement investigates the 3-sigma dc variation on output v(n1,n2). The result of the analysis is printed in a psf file and the cpu statistics of the analysis are generated.

dcmm2 n1 n2 dcmatch mth=1e-3 where=rawfile

In the following example, the output is the voltage drop across the 1st port of r3.

dcmm3 dcmatch mth=1e-3 oprobe=r3 portv=1

For the following statement, the output of the analysis is printed to a file circuitName.info.what.

dcmm4 n3 0 dcmatch mth=1e-3 where=file file="%C:r.info.what"

You can use sweep parameters on the dcmatch analysis to perform sweeps of temperature, parameters, model/instance/subcircuit parameters etc.

In the following example, the device parameter \( w \) of the device x1.mp2 is swept from 15µm to 20µm at each increment of 1µm.

dcmm6 n3 0 dcmatch mth=0.01 dev=x1.mp2 param=w
+ start=15e-6 stop=20e-6 step=1e-6 where=rawfile

In the following example, a set of analyses is performed on output v(n3,0) by sweeping the device parameter \( w \) of the device mp6 from 80µm to 90µm at each increment of 2µm.

sweep1 sweep dev=mp6 param=w start=80e-6 stop=90e-6 step=2e-6 {
  dcmm5 n3 0 dcmatch mth=1e-3 where=rawfile
}

In the following example, temperature is swept from 25°C to 100°C at increment of 25°C.

dcmm7 n3 0 dcmatch mth=0.01 param=temp
+ start=25 stop=100 step=25
For more information on the dcmatch parameters, see the *Spectre Circuit Simulator Reference*.

If you run the dcmatch analysis in Analog Design Environment, you can access the output through the Results menu and create a table of mismatch contributors.

### Stability Analysis

The loop-based and device-based algorithms are available in the Spectre circuit simulator for small-signal stability analysis. Both are based on the calculation of Bode’s return ratio. The analysis output are loop gain waveform, gain margin, and phase margin.

#### Model Definition

```
name stb parameter=value ...
```

#### Examples of the `stb` command

```
stbloop stb start=1.0 stop=1e12 dec=10 probe=Iprobe
stbdev stb start=1.0 stop=1e12 dec=10 probe=mos1
```

The analysis parameters are similar to the small-signal ac analysis except for the `probe` parameter, which must be specified to perform stability analysis. When the `probe` parameter points to a current probe or voltage source instance, the loop based algorithm will be invoked; when it points to a supported active device instance, the device based algorithm will be invoked.

The gain margin and phase margin are automatically determined from the loop gain waveform by detecting zero-crossing in the gain plot and phase plot. If margins cannot be determined for a particular stability analysis, a log file displays the corresponding reason.

#### Loop-Based Algorithm

The loop-based algorithm is based on considering the feedback loop as a lumped model with normal and reverse loop transmission. It calculates the true loop gain, which consists of normal loop gain and reverse loop gain. Stability analysis approaches for low-frequency applications assume that signal flows unilaterally through the feedback loop, and they use the normal loop gain to assess the stability of the design. However, the true loop gain provides more accurate stability information for applications involving significant reverse transmission.

You can place a probe component (current probe or zero-DC-valued voltage source) on the feedback loop to identify the loop of interest. The probe component does not change any of the circuit characteristics, and there is no special requirement on the polarity configuration of the probe component.
The loop-based algorithm provides accurate stability information for single-loop circuits and multi-loop circuits in which a probe component can be placed on a critical wire to break all loops. For a multi-loop circuit in which such a wire may not be available, the loop based algorithm can be performed only on individual feedback loops to ensure they are stable. Although the stability of all feedback loops is only a necessary condition for the whole circuit to be stable, the multi-loop circuit tends to be stable if all individual loops are associated with reasonable stability margins.

**Device Based Algorithm**

The device-based algorithm produces accurate stability information for circuits in which a critical active device can be identified such that nulling the dominant gain source of this device renders the whole network to be passive. Examples are multistage amplifier, single-transistor circuit, and S-parameter characterized microwave component.

This algorithm is often applied to assess the stability of circuit design in which local feedback loops cannot be neglected; the loop-based algorithm cannot be performed for these applications because the local feedback loops are inside the devices and are not accessible from the schematic level or netlist level to insert the probe component.

The supported active device and its dominant gain source are summarized in the table below.

<table>
<thead>
<tr>
<th>Component</th>
<th>Dominant Controlled Source</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>b3soipd</td>
<td>gm</td>
<td>Common-source transconductance</td>
</tr>
<tr>
<td>bjt</td>
<td>gm</td>
<td>Common-emitter transconductance</td>
</tr>
<tr>
<td>bsim1,2,3,3v3</td>
<td>gm</td>
<td>Common-source transconductance</td>
</tr>
<tr>
<td>btasoi</td>
<td>gm</td>
<td>Common-source transconductance</td>
</tr>
<tr>
<td>cccs</td>
<td>gain</td>
<td>Current gain</td>
</tr>
<tr>
<td>ccvs</td>
<td>rm</td>
<td>Transresistance</td>
</tr>
<tr>
<td>ekv</td>
<td>gm</td>
<td>Common-source transconductance</td>
</tr>
<tr>
<td>gaas</td>
<td>gm</td>
<td>Common-source transconductance</td>
</tr>
<tr>
<td>hbt</td>
<td>dice_dvbe</td>
<td>Intrinsic dlce/dVbe</td>
</tr>
<tr>
<td>hvmos</td>
<td>gm</td>
<td>Common-source transconductance</td>
</tr>
<tr>
<td>jfet</td>
<td>gm</td>
<td>Common-source transconductance</td>
</tr>
<tr>
<td>mos0,1,2,3</td>
<td>gm</td>
<td>Common-source transconductance</td>
</tr>
<tr>
<td>tom2</td>
<td>gm</td>
<td>Common-source transconductance</td>
</tr>
</tbody>
</table>
In general, the stability information produced by the device-based algorithm can be used to assess the stability of that particular device. Most often, a feedback network consists of a global feedback loop and numerous nested local loops around individual transistors. The loop-based algorithm can determine the stability of the whole network as long as all nested loops are stable, while the device-based algorithm can be used to ensure all local loops are stable.

For more information on the stability analysis parameters, see the Spectre Circuit Simulator Reference.

Advanced Analyses (sweep and montecarlo)

There are two advanced analyses: sweep and montecarlo.

Sweep Analysis

The sweep analysis sweeps a parameter processing a list of analyses (or multiple analyses) for each value of the parameter.

The sweeps can be linear or logarithmic. Swept parameters return to their original values after the analysis. However, certain other analyses also allow you to sweep a parameter while performing that analysis. For more details, check spectre -h for each of the following analyses. The following table shows you which parameters you can sweep with different analyses.

<table>
<thead>
<tr>
<th>Component</th>
<th>Dominant Controlled Source</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>vbic</td>
<td>dic_dvbe</td>
<td>Intrinsic dlc/dVbe</td>
</tr>
<tr>
<td>vccs</td>
<td>gm</td>
<td>Transconductance</td>
</tr>
<tr>
<td>vcvs</td>
<td>gain</td>
<td>Voltage gain</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Component</th>
<th>Dominant Controlled Source</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>vbic</td>
<td>dic_dvbe</td>
<td>Intrinsic dlc/dVbe</td>
</tr>
<tr>
<td>vccs</td>
<td>gm</td>
<td>Transconductance</td>
</tr>
<tr>
<td>vcvs</td>
<td>gain</td>
<td>Voltage gain</td>
</tr>
</tbody>
</table>

In general, the stability information produced by the device-based algorithm can be used to assess the stability of that particular device. Most often, a feedback network consists of a global feedback loop and numerous nested local loops around individual transistors. The loop-based algorithm can determine the stability of the whole network as long as all nested loops are stable, while the device-based algorithm can be used to ensure all local loops are stable.

For more information on the stability analysis parameters, see the Spectre Circuit Simulator Reference.
**Spectre Circuit Simulator User Guide**

**Analyses**

<table>
<thead>
<tr>
<th>Analysis</th>
<th>Time</th>
<th>TEMP</th>
<th>FREQ</th>
<th>A component instance parameter</th>
<th>A component model parameter</th>
<th>A netlist parameter</th>
</tr>
</thead>
<tbody>
<tr>
<td>Noise analysis (noise)</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>S-parameter analysis (sp)</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>Transfer function analysis (xf)</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>Transient analysis (tran)</td>
<td>●</td>
<td></td>
<td></td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>Time-domain reflectometer analysis (tdr)</td>
<td>●</td>
<td></td>
<td></td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>Periodic steady state analysis (pss)</td>
<td>●</td>
<td></td>
<td></td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>Periodic AC analysis (pac)</td>
<td></td>
<td>●</td>
<td></td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>Periodic transfer function analysis (pxf)</td>
<td></td>
<td>●</td>
<td></td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>Periodic noise analysis (pnoise)</td>
<td></td>
<td></td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>Envelope-following analysis (envlp)</td>
<td></td>
<td></td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>Sweep analysis (sweep)</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>DC Match Analysis (dcmatch)</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>Stability Analysis (stb)</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
</tbody>
</table>

**Note:** To generate transfer curves with the DC analysis, specify a parameter and a sweep range. If you specify the *oppoint* parameter for a DC analysis, the Spectre simulator computes the linearized model for each nonlinear component. If you specify both a DC sweep
and an operating point, the operating point information is generated for the last point in the sweep.

**Setting Up Parameter Sweeps**

To specify a parameter sweep, you must identify the component or circuit parameter you want to sweep and the sweep limits in an analysis statement. A parameter you sweep can be circuit temperature, a device instance parameter, a device model parameter, a netlist parameter, or a subcircuit parameter for a particular subcircuit instance.

Within the *sweep* analysis only, you specify child analyses statements. These statements must be bound with braces. The opening brace is required at the end of the line defining the sweep.

**Specifying the Parameter You Want to Sweep**

You specify the components and parameters you want to sweep with the following parameters:

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>dev</td>
<td>The name of an instance whose parameter value you want to sweep</td>
</tr>
<tr>
<td>sub</td>
<td>The name of the subcircuit instance whose parameter value you want to sweep</td>
</tr>
<tr>
<td>mod</td>
<td>The name of a model whose parameter value you want to sweep</td>
</tr>
<tr>
<td>param</td>
<td>The name of the component parameter you want to sweep</td>
</tr>
<tr>
<td>freq</td>
<td>For analyses that normally sweep frequency (small-signal analyses such as <em>ac</em>), if you sweep some parameter other than frequency, you must still specify a fixed frequency value for that analysis using the <em>freq</em> parameter</td>
</tr>
<tr>
<td>paramset</td>
<td>For the <em>sweep</em> analysis only; allows sweeping of multiple parameters defined by the <em>paramset</em> statement</td>
</tr>
</tbody>
</table>

For all analyses that support sweeping, to sweep the circuit temperature, use `param=temp` with no `dev`, `mod`, or `sub` parameter. You can sweep a top-level netlist parameter by giving the parameter name with no `dev`, `mod`, or `sub` parameter. You can sweep a subcircuit parameter for a particular subcircuit instance by specifying the subcircuit instance name with the `sub` parameter and the subcircuit parameter name with the `param` parameter. You can do the same thing for a particular device instance by using `dev` for the device instance name or for a particular model by using `mod` for the device model name.
Note: If frequency is a sweep option for an analysis, the Spectre simulator sweeps frequency if you leave dev, mod, and param unspecified. That is, frequency is the default swept parameter for that analysis.

Specifying Parameter Sets You Want to Sweep

For the sweep analysis only, the paramset statement allows you to specify a list of parameters and their values. This can be referred by a sweep analysis to sweep the set of parameters over the values specified. For each iteration of the sweep, the netlist parameters are set to the values specified by a row. The values have to be numbers, and the parameters’ names have to be defined in the input file (netlist) before they are used. The paramset statement is allowed only in the top level of the input file.

The following is the syntax for the paramset statement:

```
Name paramset {
    <list of netlist parameters>
    <list of values foreach netlist parameter>
    ...}
```

Here is an example of the paramset statement:

```
parameters p1=1 p2=2 p3=3
data paramset {
    p1 p2 p3
    5 5 5
    4 3 2
}
```

Combining the paramset statement with the sweep analysis allows you to sweep multiple parameters simultaneously, for example, power supply voltage and temperature.

Setting Sweep Limits

For all analyses that support sweeping, you specify the sweep limits with the parameters in the following table:

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>start</td>
<td>Start of sweep value (Default=0)</td>
<td>start and stop are used together to specify sweep limits.</td>
</tr>
<tr>
<td>stop</td>
<td>End of sweep value</td>
<td></td>
</tr>
<tr>
<td>center</td>
<td>Center value of sweep</td>
<td>center and span are used together to specify sweep limits.</td>
</tr>
<tr>
<td>span</td>
<td>Span of sweep (Default=0)</td>
<td></td>
</tr>
</tbody>
</table>
If you do not specify the step size, the sweep is linear when the ratio of the stop to the start values is less than 10 and logarithmic when this ratio is 10 or greater. If you specify sweep limits and a values array, the points for both are merged and sorted.

### Examples of Parameter Sweep Requests

This `<sweep>` statement uses braces to bound the child analyses statements.

```plaintext
swp sweep param=temp values=[-50 0 50 100 125] {
  oppoint dc oppoint=logfile
}
```

This statement specifies a linear sweep of frequencies from 0 to 0.3 MHz with 100 steps.

```plaintext
Sparams sp stop=0.3MHz lin=100
```

The previous statement could be written like this and achieve the same result.

```plaintext
Sparams sp center=0.15MHz span=0.3MHz lin=100
```

This statement specifies a logarithmic sweep of frequencies from 1 kHz through 1 GHz with 10 steps per decade.

```plaintext
cmLoopGain ac start=1k stop=1G dec=10
```

This statement is identical to the previous one except that the number of steps is set to 55.

```plaintext
cmLoopGain ac start=1k stop=1G log=55
```

This statement specifies a linear sweep of temperatures from 0 to 50 degrees in 1-degree steps. The frequency for the analysis is 1 kHz.

```plaintext
XferVsTemp xf start=0 stop=50 step=1 probe=Rload \ 
  param=temp freq=1kHz
```
This statement uses a vector to specify sweep values for device Vcc. The values specified for the sweep are 0, 2, 6, 7, 8 and 10.

SwpVccDC dc dev=Vcc values=[0 2 6 7 8 10]

Monte Carlo Analysis

The montecarlo analysis is a swept analysis with associated child analyses similar to the sweep analysis (see spectre -h sweep). The Monte Carlo analysis refers to “statistics blocks” where statistical distributions and correlations of netlist parameters are specified. (Detailed information on statistics blocks is given in “Specifying Parameter Distributions Using Statistics Blocks” on page 170.) For each iteration of the Monte Carlo analysis, new pseudorandom values are generated for the specified netlist parameters (according to their specified distributions) and the list of child analyses are then executed.

The Cadence design environment Monte Carlo option allows for scalar measurements to be linked with the Monte Carlo analysis. Calculator expressions are specified that can be used to measure circuit output or performance values (such as the slew rate of an operational amplifier). During a Monte Carlo analysis, these measurement statement results vary as the netlist parameters vary for each Monte Carlo iteration and are stored in a scalar data file for postprocessing. By varying netlist parameters and evaluating these measurement statements, the Monte Carlo analysis becomes a tool that allows you to examine and predict circuit performance variations that affect yield.

The statistics blocks allow you to specify batch-to-batch (process) and per-instance (mismatch) variations for netlist parameters. These statistically varying netlist parameters can be referenced by models or instances in the main netlist and can represent IC manufacturing process variation or component variations for board-level designs. The following description gives a simplified example of the Monte Carlo analysis flow:

perform nominal run if requested
if any errors in nominal run then stop

for each Monte Carlo iteration {
    if process variations specified then
        apply “process” variation to parameters
    if mismatch variations specified then
        for each subcircuit instance {
            apply “mismatch” variation to parameters
        }
    for each child analysis {
        run child analysis
        evaluate any export statements and
        store results in a scalar data file
    }
}
The following is the syntax for the Monte Carlo analysis:

```plaintext
Name montecarlo <parameter=value> ... 
   <analysis statements> ...
   <export statements> ... 
}
```

The Monte Carlo analysis

- Refers to the statistics block(s) for how and which netlist parameters to vary
- Generates statistical variation (random numbers according to the specified distributions)
- Runs the specified child analyses (similar to the Spectre nested sweep analysis), where the child analyses are either
  - Multiple data-producing child analyses (such as DC or AC analyses)
  - A single sweep child analysis, which itself has child analyses
- Calculates the export quantities

  Each Monte Carlo run processes export statements that implicitly refer to the result of the child analyses. These statements calculate scalar circuit output values for performance characteristics, such as slew rate.

- Organizes the export data appropriately

  Scalar data, such as bandwidth or slew rate, is calculated from an export statement and saved to an ASCII file, which can be used later for plotting a histogram or scattergram.

- After the Monte Carlo analysis is complete, all parameters are returned to their original values.
Monte Carlo Analysis Parameters

You use the following parameters for your Monte Carlo analysis.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>numruns</td>
<td>This is the number of Monte Carlo runs to perform (not including the nominal run). The default is 100 runs. The Spectre simulator performs a loop, running the specified child analyses <code>numruns</code> times (or <code>numruns+1</code> times if the <code>donominal</code> parameter is set to <code>yes</code>) and evaluating any export statements <code>numruns</code> times.</td>
</tr>
<tr>
<td>seed</td>
<td>This is the optional starting seed for the random number generator. By always specifying the same seed, you can reproduce a previous experiment. If you do not specify a seed, then each time that you run the analysis, you get different results; that is, a different stream of pseudorandom numbers is generated. If you do not specify a seed, the Spectre simulator uses the Spectre process id (PID) as a seed.</td>
</tr>
</tbody>
</table>
### Parameter Description

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>scalarfile</strong></td>
<td>This parameter specifies the name of an ASCII file where scalar data calculated by the <code>export</code> statements (the results of export expressions that resolve to scalar values) is saved. For each iteration of each Monte Carlo child analysis, Spectre writes a line to this ASCII file with one scalar expression per column (for example, slew rate or bandwidth). The default name for this file is <code>name.mcdata</code>, where <code>name</code> is the name of the Monte Carlo analysis instance. The file is created in the <code>psf</code> directory by default unless you specify a path in the filename. This file contains only the matrix of numeric values. The analog design environment Monte Carlo tool can read this ASCII file and plot the data in histograms and scattergrams. When you use the analog design environment Monte Carlo tool to generate the Spectre netlist file, Spectre merges the values of the statistically varying process parameters into this file containing the scalar data. The analog design environment statistical plotting tool can later read the data and create scatter plots of the statistically varying process parameters against each other or against the results of the export expressions. You can then see correlations between process parameter variations and circuit performance variations. This data merging takes place whenever the scalarfile and processscalarfile are written in the same directory.</td>
</tr>
<tr>
<td><strong>paramfile</strong></td>
<td>This file contains the titles, sweep variable values, and the full expression for each of the columns in <code>scalarfile</code>. The default name for this file is <code>name.mcparam</code>, where <code>name</code> is the name of the Monte Carlo analysis instance. The file is created in the <code>psf</code> directory by default unless you specify some path information in the filename.</td>
</tr>
<tr>
<td><strong>saveprocessparams</strong></td>
<td>This specifies (yes or no) whether to save scalar data for statistically varying process parameters that are subject to process variation.</td>
</tr>
</tbody>
</table>
processscalarfile

If `saveprocessparams` is set to `yes`, the process (batch-to-batch) values of all statistically varying parameters are saved to this scalar data file. You can use `saveprocessvec` to filter out a subset of parameters (in which case the Spectre simulator saves only the parameters specified in `saveprocessvec` to `processscalarfile`.)

`processscalarfile` is equivalent to `scalarfile`, except that the data in `scalarfile` contains the values of the scalar export statements, whereas the data in `processscalarfile` contains the corresponding process parameter values. The default name for this file is `instname.process.mcdata`, where `instname` is the name of the Monte Carlo analysis instance. This file is created in the `psf` directory by default unless you specify some path information in the filename. You can load `processscalarfile` and `processparamfile` into the analog design statistical postprocessing environment to plot/verify the process parameter distributions. If you later merge `processparamfile` with the data in `scalarfile`, you can then plot export scalar values against the corresponding process parameters by loading this merged file into the analog design statistical postprocessing environment.

processparamfile

This specifies the output file where process parameter scalar data labels are saved. This file contains the titles and sweep variable values for each of the columns in `processscalarfile`. These titles are the names of the process parameters. `processparamfile` is equivalent to `paramfile`, except that `paramfile` contains the name of the export expressions, whereas `processparamfile` contains the names of the process parameters. The default name for this file is `instname.process.mcpam`, where `instname` is the name of the Monte Carlo analysis instance. This file is created in the `psf` directory by default unless you specify some path information in the filename.
<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>saveprocessvec</td>
<td>This specifies an array of statistically varying process parameters (which are subject to process variation) to save as scalar data in processscalarfile. For example, saveprocessvec=[RSHSP TOX]. This acts as a filter so that you do not save all process parameters to the file. If you do not want to filter the list of process parameters, do not specify this parameter.</td>
</tr>
<tr>
<td>firstrun</td>
<td>This specifies the run that Monte Carlo begins with. The default is 1. If firstrun is specified as a number ( n ) (where ( n ) is greater than one), the previous ( n-1 ) iterations are skipped. The Monte Carlo analysis behaves as if the first ( n-1 ) iterations were run without performing the child analyses for those iterations. The subsequent stream of random numbers generated for the remaining iterations is the same as if the first ( n-1 ) iterations were run. By specifying the first iteration number, you can reproduce a particular run or sequence of runs from a previous experiment (for example, to examine an earlier case in more detail).</td>
</tr>
<tr>
<td>variations</td>
<td>This parameter specifies the type of statistical variation to apply:</td>
</tr>
<tr>
<td></td>
<td>• process—batch-to-batch variations only</td>
</tr>
<tr>
<td></td>
<td>• mismatch—per-instance variations only</td>
</tr>
<tr>
<td></td>
<td>• all—both process and mismatch variations applied</td>
</tr>
<tr>
<td></td>
<td>The default is process. This assumes that you have specified the appropriate statistical distributions in the statistics block. You cannot request that mismatch variations be applied unless you have specified mismatch statistics in the statistics block; mismatch variations work only on components inside of subcircuits. You cannot request that process variations be applied unless you have specified process statistics in the statistics block. More details on statistics blocks are given in “Specifying Parameter Distributions Using Statistics Blocks” on page 170.</td>
</tr>
</tbody>
</table>

**Note:** To reproduce a run or sequence of runs, you need to specify the same value for the seed parameter.
### donominal

This flag determines whether a nominal run of all child analyses is performed (yes or no) before starting the Monte Carlo runs. The default is yes.

If the flag is set to yes and the Spectre simulator cannot calculate an output expression during the nominal run (for example, convergence problems or incorrect export expressions), the Spectre simulator issues an error message and immediately exits the Monte Carlo analysis.

If donominal is set to no, the Spectre simulator runs the Monte Carlo iterations without performing a nominal analysis. If there are convergence problems or the output expressions cannot be successfully calculated for the first or any iteration, the Spectre simulator issues a warning and continues with the next iteration of the Monte Carlo loop.

### appendsd

This specifies whether to append scalar data to the existing scalarfile (yes) or to overwrite the existing file (no). The default is no. This flag applies to both scalarfile and processscalarfile.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>donominal</td>
<td>This flag determines whether a nominal run of all child analyses is performed (yes or no) before starting the Monte Carlo runs. The default is yes. If the flag is set to yes and the Spectre simulator cannot calculate an output expression during the nominal run (for example, convergence problems or incorrect export expressions), the Spectre simulator issues an error message and immediately exits the Monte Carlo analysis. If donominal is set to no, the Spectre simulator runs the Monte Carlo iterations without performing a nominal analysis. If there are convergence problems or the output expressions cannot be successfully calculated for the first or any iteration, the Spectre simulator issues a warning and continues with the next iteration of the Monte Carlo loop.</td>
</tr>
<tr>
<td>appendsd</td>
<td>This specifies whether to append scalar data to the existing scalarfile (yes) or to overwrite the existing file (no). The default is no. This flag applies to both scalarfile and processscalarfile.</td>
</tr>
</tbody>
</table>
Specifying the First Iteration Number

The advantages of using the `firstrun` parameter to specify the first iteration number are as follows:

- You can reproduce a particular run from a previous experiment when you know the starting seed and run number but not the corresponding seed.

- If you are a standalone Spectre user, you can run a Monte Carlo analysis of 100 runs, analyze the results, decide they are acceptable, and then decide to do a second analysis.

### Parameter Description

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>savefamilyplots</code></td>
<td>This flag specifies whether to save the multiple waveform data in parameter storage format (PSF) for family plotting. The default is <code>no</code>.</td>
</tr>
<tr>
<td></td>
<td>If the flag is set to <code>yes</code>, a separate PSF file is saved for each analysis in each Monte Carlo iteration, in addition to the export scalar results that are saved to the ASCII scalar data file at the end of each iteration. Saving the PSF files between runs allows you to:</td>
</tr>
<tr>
<td></td>
<td>- Cloud-plot overlaid waveforms in the analog design environment</td>
</tr>
<tr>
<td></td>
<td>- Define and evaluate new calculator measurements after the simulation has been run using the analog design environment calculator and/or the Analog Waveform Display tool</td>
</tr>
<tr>
<td></td>
<td><strong>Note:</strong> This feature can result in many very large data files. You might want to monitor the disk space available after each iteration or perhaps after just the nominal simulation and extrapolate to see if sufficient disk space is available.</td>
</tr>
<tr>
<td></td>
<td>If <code>savefamilyplots</code> is set to <code>no</code>, PSF files are overwritten by each Monte Carlo iteration.</td>
</tr>
<tr>
<td><code>annotate</code></td>
<td>This specifies the degree of annotation, such as percentage done. Possible values are <code>no</code>, <code>title</code>, <code>sweep</code>, or <code>status</code>, similar to the Spectre simulator’s nested sweep analysis. Use the maximum value of <code>status</code> to print a summary of which runs did not converge or had problems evaluating <code>export</code> statements and so on. The default is <code>sweep</code>.</td>
</tr>
<tr>
<td><code>title</code></td>
<td>This specifies the analysis title.</td>
</tr>
</tbody>
</table>
of 100 runs to give a total of 200 runs. By specifying the \texttt{firstrun=101} for the second analysis, the Spectre simulator retains the data for the first 100 runs and runs only the second 100 runs. This gives the same results and random sequence as if you ran just a single Monte Carlo analysis of 200 runs.

\section*{Sample Monte Carlo Analyses}

For a Monte Carlo analysis, the Spectre simulator performs a nominal run first, if requested, calculating the specified outputs. If there is any error in the nominal run or in evaluating the \texttt{export} statements after the nominal run, the Monte Carlo analysis stops.

If the nominal run is successful, then, depending on how the \texttt{variations} parameter is set, the Spectre simulator applies process variations to the specified parameters and mismatch variations (if specified) to those parameters for each subcircuit instance. If the \texttt{export} statements are specified, the corresponding performance measurements are saved as a new file or appended to an existing file.

The following Monte Carlo analysis statement specifies (using the default) that a nominal analysis is performed first. The \texttt{sweep} analysis (and all child analyses) are performed, and \texttt{export} statements are evaluated. If the nominal analysis fails, the Spectre simulator gives an error message and will not perform the Monte Carlo analysis. If the nominal analysis succeeds, the Spectre simulator immediately starts the Monte Carlo analysis. The \texttt{variations} parameter specifies that only process variations (\texttt{variations=process}) are applied; this is useful for looking at absolute performance spreads. There is a single child sweep analysis (\texttt{sw1}) so that for each Monte Carlo run, the Spectre simulator sweeps the temperature, performs the dc and transient analyses, and calculates the slew rate. The output of the slew rate calculation is saved in the scalar data file.

\texttt{mc1 montecarlo variations=process seed=1234 numruns=200 \{}\texttt{\begin{array}{l}
\texttt{sw1 sweep param=temp values=[-50 27 100] \{} \\
\texttt{dcop1 dc} \quad \text// a "child" analysis \\
\texttt{tran1 tran start=0 stop=1u} \quad \text// another "child" analysis \\
\texttt{// export calculations are sent to the scalardata file} \\
\texttt{export slewrate=oceanEval("slewRate(v("vout"),10n,t,30n,t,10,90 )"})
\end{array}\}\}

The following Monte Carlo analysis statement applies only mismatch variations, which are useful for detecting spreads in differential circuit applications. It does not perform a nominal run.

\textbf{Note:} No temperature sweep is performed.

\texttt{mc2 montecarlo donominal=no variations=mismatch seed=1234 numruns=200 \{}\texttt{\begin{array}{l}
\texttt{dcop2 dc} \\
\texttt{tran2 tran start=0 stop=1u} \\
\texttt{export slewrate=oceanEval("slewRate(v("vout"),10n,t,30n,t,10,90 )"})
\end{array}\}\}
The following Monte Carlo analysis statement applies both process and mismatch variations:

```
mc3 montecarlo saveprocessparams=yes variations=all numruns=200 {
    dcp3 dc
    tran3 tran start=0 stop=1u
    export slewrate=oceanEval("slewRate(v("vout"),10n,t,30n,t,10,90 )"
}
```

**Specifying Parameter Distributions Using Statistics Blocks**

The statistics blocks are used to specify the input statistical variations for a Monte Carlo analysis. A statistics block can contain one or more process blocks (which represent batch-to-batch type variations) and/or one or more mismatch blocks (which represent on-chip or device mismatch variations), in which the distributions for parameters are specified. Statistics blocks can also contain one or more correlation statements to specify the correlations between specified process parameters and/or to specify correlated device instances (such as matched pairs). Statistics blocks can also contain a `truncate` statement that can be used for generating truncated distributions.

The statistics block contains the distributions for parameters:

- Distributions specified in the process block are sampled once per Monte Carlo run, are applied at global scope, and are used typically to represent batch-to-batch (process) variations.
- Distributions specified in the mismatch block are applied on a per-subcircuit instance basis, are sampled once per subcircuit instance, and are used typically to represent device-to-device (on chip) mismatch for devices on the same chip.

When the same parameter is subject to both process and mismatch variations, the sampled process value becomes the mean for the mismatch random number generator for that particular parameter.

**Note:** Statistics blocks can be specified using combinations of the Spectre keywords `statistics`, `process`, `mismatch`, `vary`, `truncate`, and `correlate`. Braces `{}` are used to delimit blocks.

The following example shows some sample statistics blocks, which are discussed after the example along with syntax requirements.

```
// define some netlist parameters to represent process parameters
// such as sheet resistance and mismatch factors
parameters rshsp=200 rshpi=5k rshpi_std=0.4k xisn=1 xisp=1 xxx=20000 uuu=200
// define statistical variations, to be used
// with a MonteCarlo analysis.
statistics {
    process { // process: generate random number once per MC run
        vary rshsp dist=gauss std=12 percent=yes
        vary rshpi dist=gauss std=rshpi_std // rshpi_std is a parameter
    }
}```
vary xxx dist=lnorm std=12
vary uuu dist=unif N=10 percent=yes
...
}
mismatch { // mismatch: generate a random number per instance
    vary rshsp dist=gauss std=2
    vary xisn dist=gauss std=0.5
    vary xisp dist=gauss std=0.5
}
// some process parameters are correlated
correlate param=[rshsp rshpi] cc=0.6
// specify a global distribution truncation factor
truncate tr=6.0  // +/- 6 sigma
}
// a separate statistics block to specify correlated (i.e. matched)
// components
// where m1 and m2 are subckt instances.
statistics {
    correlate dev=[m1 m2] param=[xisn xisp] cc=0.8
}

**Note:** You can specify the same parameter (for example, rshsp) for both process and mismatch variations.

In the process block, the process parameter rshsp is varied with a Gaussian distribution, where the standard deviation is 12 percent of the nominal value (percent=yes). When percent is set to yes, the value for the standard deviation (std) is a percentage of the nominal value. When percent is set to no, the specified standard deviation is an absolute number. This means that parameter rshsp should be varied with a normal distribution, where the standard deviation is 12 percent of the nominal value of rshsp. The nominal or mean value for such a distribution is the current value of the parameter just before the Monte Carlo analysis starts. If the nominal value of the parameter rshsp was 200, the preceding example specifies a process distribution for this parameter with a Gaussian distribution with a mean value of 200 and a standard deviation of 24 (12 percent of 200). The parameter rshpi (sheet resistance) varies about its nominal value with a standard deviation of 0.4 K-ohms/square.

In the mismatch block, the parameter rshsp is then subject to further statistical variation on a per-subcircuit instance basis for on-chip variation. Here, it varies a little for each subcircuit instance, this time with a standard deviation of 2. For the first Monte Carlo run, if there are multiple instances of a subcircuit that references parameter rshsp, then (assuming variations=all) it might get a process random value of 210, and then the different instances might get random values of 209.4, 211.2, 210.6, and so on. The parameter xisn also varies on a per-instance basis, with a standard deviation of 0.5. In addition, the parameters rshsp and rshpi are correlated with a correlation coefficient (cc) of 0.6.

**Specifying Distributions**

Parameter variations are specified using the following syntax:
vary PAR_NAME dist=type {std=<value> | N=<value>} {percent=yes|no}

Three types of parameter distributions are available: Gaussian, log normal, and uniform, corresponding to the type keywords gauss, lnorm, and unif, respectively. For both the gauss and the lnorm distributions, you specify a standard deviation using the std keyword.

The following distributions (and associated parameters) are supported:

- **Gaussian**
  This distribution is specified using dist=gauss. For the Gaussian distribution, the mean value is taken as the current value of the parameter being varied, giving a distribution denoted by Normal(mean,std). Using the example in “Specifying Parameter Distributions Using Statistics Blocks,” parameter rshpi is varied with a distribution of Normal(5k,0.4k). The nominal value for the Gaussian distribution is the value of the parameter before the Monte Carlo analysis is run. The standard deviation can be specified using the std parameter. If you do not specify the percent parameter, the standard deviation you specify is taken as an absolute value. If you specify percent=yes, the standard deviation is calculated from the value of the std parameter multiplied by the nominal value and divided by 100; that is, the value of the std parameter specifies the standard deviation as that percentage of the nominal value.

- **Log normal**
  This distribution is specified using dist=lnorm. The log normal distribution is denoted by

  \[
  \log(x) = \text{Normal}(\log(\text{mean}), \text{std})
  \]

  where \(x\) is the parameter being specified as having a log normal distribution.

  **Note:** \(\log()\) is the natural logarithm function. For parameter xxx in the example in “Specifying Parameter Distributions Using Statistics Blocks,” the process variation is according to

  \[
  \log(xxx) = \text{Normal}(\log(20000), 12)
  \]

  The nominal value for the log normal distribution is the natural log of the value of the parameter before the Monte Carlo analysis is run. If you specify a normal distribution for a parameter P1 whose value is 5000 and you specify a standard deviation of 100, the actual distribution is produced such that

  \[
  \log(P) = \text{N}(\log(5000) 100)
  \]

- **Uniform**
  This distribution is specified using dist=unif. The uniform distribution for parameter \(x\) is generated according to
\( x = \text{unif}(\text{mean}-N, \text{mean}+N) \)

such that the mean value is the nominal value of the parameter \( x \), and the parameter is varied about the mean with a range of \( \pm N \). The standard deviation is not specified for the uniform distribution, but its value can be calculated from the formula \( \text{std}=N/\sqrt{3} \). The nominal value for the uniform distribution is the value of the parameter before the Monte Carlo analysis is run. The uniform interval is specified using the parameter \( N \). For example, specifying \( \text{dist=unif N=5} \) for a parameter whose value is 200 results is a uniform distribution in the range 200\( \pm N \), that is, from 195 to 205. You can also specify \( \text{percent=yes} \), in which case, the range is 200\( \pm N \)% , that is, from 190 to 210.

Derived parameters that have their default values specified as expressions of other parameters cannot have distributions specified for them. Only parameters that have numeric values specified in their declaration can be subjected to statistical variation.

Parameters that are specified as correlated must have had an appropriate variation specified for them in the statistics block.

For example, if you have the parameters

\[ \text{XISN=XIS+XIB} \]

you cannot specify distribution for \( \text{XISN} \) or a correlation of this parameter with another.

The \textit{percent} flag indicates whether the standard deviation \textit{std} or uniform range \( N \) are specified in absolute terms (\textit{percent=no}) or as a percentage of the mean value (\textit{percent=yes}). For parameter \( \text{uuu} \) in the example in “Specifying Parameter Distributions Using Statistics Blocks,” the mean value is 200, and the variation is 200 \( \pm 10\% \)*(200), that is, 200 \( \pm 20 \). For parameter \( \text{rshsp} \), the process variation is given by \textit{Normal( 200, 12\%*(200) )} , that is, \textit{Normal( 200, 24) }. Cadence recommends that you do not use the \textit{percent=yes} with the log normal distribution.

**Truncation Factor**

The default truncation factor for Gaussian distributions (and for the Gaussian distribution underlying the log normal distribution) is 4.0 sigma. Randomly generated values that are outside the range of mean \( \pm 4.0 \) sigma are automatically rejected and regenerated until they fall inside the range. You can change the truncation factor using the \textit{truncate} statement. The following is the syntax:

\textit{truncate tr=<value>}

\textbf{Note:} The value of the truncation factor can be a constant or an expression.

\textbf{Note:} Parameter correlations can be affected by using small truncation factors.
Multiple Statistics Blocks

You can use multiple statistics blocks, which accumulate or overlay each other. Typically, process variations, mismatch variations, and correlations between process parameters are specified in a single statistics block. This statistics block can be included in a “process” include file, such as the ones shown in the example in “Process Modeling Using Inline Subcircuits” on page 101. A second statistics block can be specified in the main netlist where actual device instance correlations are specified as matched pairs.

The following statistics block can be used to specify the correlations between matched pairs of devices and probably is placed or included into the main netlist by the designer. These statistics are used in addition to those specified in the statistics block in the preceding section so that the statistics blocks “overlay” or “accumulate.”

// define correlations for "matched" devices q1 and q2
statistics {
  correlate dev=[q1 q2] param=[XISN...] cc=0.75
}

Note: You can use a single statistics block containing both sets of statements; however, it is often more convenient to keep the topology-specific information separate from the process-specific information.

Correlation Statements

There are two types of correlation statements that you can use:

- Process parameter correlation statements

  The following is the syntax of the process parameter correlation statement:

  correlate param=[list of parameters] cc=<value>

  This allows you to specify a correlation coefficient between multiple process parameters. You can specify multiple process parameter correlation statements in a statistics block to build a matrix of process parameter correlations. During a Monte Carlo analysis, process parameter values are randomly generated according to the specified distributions and correlations.

- Instance or mismatch correlation statements (matched devices)

  The following is the syntax of the instance or mismatch correlation statement:

  correlate dev=[list of subckt instances] {param=[list of parameters]} cc=<value>

  where the device or subcircuit instances to be matched are listed in list of subckt instances, and list of parameters specifies exactly which parameters with
mismatch variations are to be correlated. Use the instance mismatch correlation statement to specify correlations for particular subcircuit instances. If a subcircuit contains a device, you can effectively use the instance correlation statements to specify that certain devices are correlated (matched) and give the correlation coefficient. You can optionally specify exactly which parameters are to be correlated by giving a list of parameters (each of which must have had distributions specified for it in a mismatch block) or by specifying no parameter list, in which case all parameters with mismatch statistics specified are correlated with the given correlation coefficient. The correlation coefficients are specified in the $<value>$ field and must be between $\pm 1.0$, not including 1.0 or -1.0.

**Note:** Correlation coefficients can be constants or expressions, as can $\text{std}$ and $\text{N}$ when specifying distributions.

**Characterization and Modeling**

The following statistics blocks can be used with the example in “Process Modeling Using Inline Subcircuits” on page 101 if they are included in the main netlist, anywhere below the `main parameters` statement. These statistics blocks are meant to be used in conjunction with the modeling and characterization equations in the inline subcircuit example, for a Monte Carlo analysis only.

```plaintext
statistics {
    process {
        vary RSHSP dist=gauss std=5
        vary RSHPI dist=lnorm std=0.15
        vary SPDW dist=gauss std=0.25
        vary SNDW dist=gauss std=0.25
    }
    correlate param=[RSHSP RSHPI] cc=0.6
    mismatch {
        vary XISN dist=gauss std=1
        vary XBFN dist=gauss std=1
        vary XRSF dist=gauss std=1
    }
}
statistics {
    correlate dev=[R1 R2] cc=0.75
    correlate dev=[TNSA1 TNSA2] cc=0.75
}
```

**Special Analysis (Hot-Electron Degradation)**

The Spectre simulator lets you control the age of the circuit when simulating hot-electron degradation. This feature has the following options and restrictions:

- Only the transient analysis supports hot-electron calculations.
Hot-electron calculations are available with the following components:

- MOS—levels 1, 2, and 3
- BSIM—levels 1 and 2

You can choose between the following substrate current models for the simulation:

- The Toshiba model
- The BERT model

To simulate the hot-electron degradation of a circuit, you need to run at least two transient analyses. In the first analysis, you establish baseline values for a new circuit. To create this baseline, you set the `circuitage` parameter to zero (`circuitage=0`). (`circuitage=0` is also the default value if you leave this parameter unspecified.)

For each subsequent transient analysis, you reset the age of the circuit by changing the value of the `circuitage` parameter. You specify `circuitage` values in years. For example, `circuitage=4.0` specifies a circuit that is 4 years old.

**Hot-Electron Degradation Analysis**

To specify an analysis of hot-electron degradation, you need to provide appropriate parameter settings:

- You must be sure the `degradation` parameter is set to `degradation=yes` for the instance statement of each component you want to analyze. This parameter takes its default value from the setting in the `model` statement. (All MOS and BSIM components require `model` statements as well as instance statements.)

- You must set all necessary parameters in the `model` statements of the components you want to analyze. To determine which model parameters you must specify, look at the parameter listings for each component in the Spectre online help (`spectre -h`). Examine the following subsections:
  - Degradation-related parameters
  - The Spectre simulator stress parameters (for Toshiba analysis)
  - BERT stress parameters (for BERT analysis)
Caution

Be careful to set the following parameters correctly: dvthc, dythe, duoc, and duoe. These are degradation parameters, which are usually obtained from DC stress measurements. Errors in setting these parameters can lead to inaccurate results.

You must set the circuitage parameter correctly in the transient analyses. In the following example, the Spectre simulator makes a baseline measurement and then examines changes in circuit behavior after 1, 5, and 10 years.

```
new tran 1n 100n circuitage=0
year1 tran 1n 100n circuitage=1
year5 tran 1n 100n circuitage=5
year10 tran 1n 100n circuitage=10
```

Note: After you establish the baseline value, you can enter subsequent circuitage parameters in any order.

Output Options for Hot-Electron Degradation Analysis

After the transient analyses are completed, the Spectre simulator produces a sorted table of device lifetimes. Two device lifetimes are calculated for each device that receives hot-electron degradation analysis. The first lifetime is based on the threshold voltage, and the second lifetime is based on the low mobility shift.

For each transient analysis in which circuitage is greater than 0, the Spectre simulator generates a second sorted table. This table lists all the degradations of the threshold voltage ($V_{th}$), the mobility factor ($U_o$), the transconductance ($g_m$), and the drain current ($I_{ds}$) of each device for the period specified in the circuitage parameter.

The degradation of $V_{th}$ is given in absolute values. The degradations of $U_o$, $g_m$, and $I_{ds}$ are expressed as a percentage of their original values. All four degradations are compared to failure criteria that you can specify with the model statement parameters crivth, criuo, crigm, and criids. If any degradation is greater than its corresponding failure criterion, the device is listed as having failed.

In the transient analysis, you can also save waveforms of the following quantities for any device:

- Voltage
- Any operating-point parameter
- Stress
You create these output files with the `save` statement. You use a waveform display tool, such as the Cadence® Analog Waveform Display (AWD) tool, to view the waveforms.

**Example of Hot-Electron Degradation**

This netlist of an 11-stage ring oscillator demonstrates the statements required to simulate hot-electron degradation in the Spectre simulator. You must properly characterize and extract the degradation parameters `dvthc`, `dvthe`, `duoc`, and `duoe`. These are usually obtained from DC stress measurements. Because device degradations are power functions of the substrate current, if you do not accurately determine the degradation parameters, the predicted device lifetimes and degradations might lead to false conclusions.

* Hot-Electron Degradation Circuit: Spectre degradation model

```
subckt inv ( 1 2 3)
c1 3 0 0.1p
m1 3 2 1 1 pmen l=1.5u w=30u ad=120p as=75p pd=36u ps=6u
m2 3 2 0 0 nmen l=1.5u w=15u ad=60p as=37.5p pd=23u ps=6u
ends
x1 21 2 3 inv
x2 1 3 4 inv
x3 1 4 5 inv
x4 1 5 6 inv
x5 1 6 7 inv
x6 1 7 8 inv
x7 1 8 9 inv
x8 1 9 10 inv
x9 1 10 11 inv
x10 1 11 12 inv
x11 1 12 2 inv
vdd1 1 0 vsrce dc=5.0
vdd 21 0 pulse( 0 5.0 0 0.1m 1n 500n )
.tran 0.1n 10n circuitage=0.0
.tran 0.1n 10n circuitage=4.0

.model nmen nmos level=3 vto=0.6876 gamma=0.5512 kappa=5.0
+kfp=0.8675e-4 tox=0.206e-7 nsub=0.197e17 vmax=4e5 theta=0.2
+cbs=10f cbd=10f degramod=spectre degradation=yes
+strc=1.5e-6 sube=1 duoc=0.9 duoe=0.9 wnom=15u lnom=2u
.model pmen pmos level=3 vto=-0.6893 gamma=0.4411 kappa=5.0
+ld = 0.45e-6 kp=0.3269e-4 tox=0.206e-7 nsub=0.197e17 +vmax=4e5 theta=0.2 cbs=10f
cbd=10f degramod=spectre +degradation=yes strc=3.0e-6 sube=1 duoc=0.9
+duoe=0.9 wnom=30u lnom=2u
save x1.ml:isub x1.ml:stress x1.ml:age
.end
```
The `save` statement in the previous netlist saves the waveforms of the substrate current, the stress, and the effective device age for transistor $m_1$ in the $x_1$ subcircuit. These waveforms are shown in Figure 6-1 on page 179, Figure 6-2 on page 179, and Figure 6-3 on page 180. The ring oscillator output waveform is shown before and after degradation in Figure 6-4 on page 180.

**Figure 6-1 Transistor m1 Substrate Current**

![Substrate Current Waveform](image)

**Figure 6-2 Transistor m1 Stress**

![Stress Waveform](image)
The Spectre simulator also creates two files called `moslev3_dgt_0` and `moslev3_dgt_4.0`. These files are the sorted tables that contain the results of hot-electron analysis. The filenames show that the outputs are for level-3 MOSFETs. The numbers 0 and 4.0 in the filenames further specify that the degradations are for `circuitage = 0` and `circuitage = 4.0`. 
circuitage = 4.0, the values requested in the .tran statements. The contents of these two files are shown in the following tables.

### Device Lifetime (Years) in Ascending Order

<table>
<thead>
<tr>
<th>Device</th>
<th>$V_{th}$-lifetime</th>
<th>$U_0$-lifetime</th>
<th>Age (sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>x7.m1</td>
<td>4.5372</td>
<td>3.9493</td>
<td>6.9888e-18</td>
</tr>
<tr>
<td>x3.m1</td>
<td>4.563</td>
<td>3.9718</td>
<td>6.9493e-18</td>
</tr>
<tr>
<td>x9.m1</td>
<td>4.5865</td>
<td>3.9922</td>
<td>6.9138e-18</td>
</tr>
<tr>
<td>x2.m1</td>
<td>4.5935</td>
<td>3.9983</td>
<td>6.9032e-18</td>
</tr>
<tr>
<td>x11.m1</td>
<td>4.6045</td>
<td>4.0078</td>
<td>6.8867e-18</td>
</tr>
<tr>
<td>x5.m1</td>
<td>4.6269</td>
<td>4.0273</td>
<td>6.8534e-18</td>
</tr>
<tr>
<td>x2.m2</td>
<td>5.6758</td>
<td>4.9404</td>
<td>5.5868e-18</td>
</tr>
<tr>
<td>x3.m2</td>
<td>6.5803</td>
<td>5.7277</td>
<td>4.8189e-18</td>
</tr>
<tr>
<td>x6.m2</td>
<td>6.6639</td>
<td>5.8004</td>
<td>4.7585e-18</td>
</tr>
<tr>
<td>x1.m2</td>
<td>6.6998</td>
<td>5.8317</td>
<td>4.733e-18</td>
</tr>
<tr>
<td>x10.m2</td>
<td>6.7037</td>
<td>5.8351</td>
<td>4.7302e-18</td>
</tr>
<tr>
<td>x4.m2</td>
<td>6.7768</td>
<td>5.8986</td>
<td>4.6792e-18</td>
</tr>
<tr>
<td>x8.m2</td>
<td>6.8111</td>
<td>5.9285</td>
<td>4.6556e-18</td>
</tr>
<tr>
<td>x4.m1</td>
<td>7.2344</td>
<td>6.297</td>
<td>4.3832e-18</td>
</tr>
<tr>
<td>x1.m1</td>
<td>8.1518</td>
<td>7.0956</td>
<td>3.8899e-18</td>
</tr>
<tr>
<td>x10.m1</td>
<td>9.0831</td>
<td>7.9062</td>
<td>3.4911e-18</td>
</tr>
<tr>
<td>x6.m1</td>
<td>9.0909</td>
<td>7.9129</td>
<td>3.4881e-18</td>
</tr>
<tr>
<td>x8.m1</td>
<td>9.3002</td>
<td>8.0952</td>
<td>3.4096e-18</td>
</tr>
<tr>
<td>x9.m2</td>
<td>12.981</td>
<td>11.299</td>
<td>2.4428e-18</td>
</tr>
<tr>
<td>x5.m2</td>
<td>13.019</td>
<td>11.332</td>
<td>2.4356e-18</td>
</tr>
<tr>
<td>x7.m2</td>
<td>13.235</td>
<td>11.52</td>
<td>2.3959e-18</td>
</tr>
<tr>
<td>x11.m2</td>
<td>13.291</td>
<td>11.569</td>
<td>2.3858e-18</td>
</tr>
</tbody>
</table>
### Device Degradations after Four Years in Descending Order

<table>
<thead>
<tr>
<th>Device</th>
<th>Delta $V_{th}$ (V)</th>
<th>Delta $U_0$ (%)</th>
<th>Delta $G_m$ (%)</th>
<th>Delta $I_d$ (%)</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>x7.m1</td>
<td>0.08816</td>
<td>10.115</td>
<td>10.112</td>
<td>10.094</td>
<td>Failed</td>
</tr>
<tr>
<td>x3.m1</td>
<td>0.087661</td>
<td>10.064</td>
<td>10.06</td>
<td>10.042</td>
<td>Failed</td>
</tr>
<tr>
<td>x9.m1</td>
<td>0.087213</td>
<td>10.018</td>
<td>10.014</td>
<td>9.9962</td>
<td>Failed</td>
</tr>
<tr>
<td>x2.m1</td>
<td>0.087079</td>
<td>10.004</td>
<td>10</td>
<td>9.9824</td>
<td>Failed</td>
</tr>
<tr>
<td>x11.m1</td>
<td>0.086872</td>
<td>9.9824</td>
<td>9.9789</td>
<td>9.961</td>
<td>Passed</td>
</tr>
<tr>
<td>x5.m1</td>
<td>0.086451</td>
<td>9.9389</td>
<td>9.9354</td>
<td>9.9176</td>
<td>Passed</td>
</tr>
<tr>
<td>x2.m2</td>
<td>0.070474</td>
<td>8.2693</td>
<td>8.265</td>
<td>8.243</td>
<td>Passed</td>
</tr>
<tr>
<td>x3.m2</td>
<td>0.060787</td>
<td>7.2389</td>
<td>7.2351</td>
<td>7.2157</td>
<td>Passed</td>
</tr>
<tr>
<td>x6.m2</td>
<td>0.060025</td>
<td>7.1572</td>
<td>7.1534</td>
<td>7.1342</td>
<td>Passed</td>
</tr>
<tr>
<td>x1.m2</td>
<td>0.059703</td>
<td>7.1226</td>
<td>7.1189</td>
<td>7.0997</td>
<td>Passed</td>
</tr>
<tr>
<td>x10.m2</td>
<td>0.059668</td>
<td>7.1189</td>
<td>7.1151</td>
<td>7.096</td>
<td>Passed</td>
</tr>
<tr>
<td>x4.m2</td>
<td>0.059025</td>
<td>7.0498</td>
<td>7.0461</td>
<td>7.0271</td>
<td>Passed</td>
</tr>
<tr>
<td>x8.m2</td>
<td>0.058728</td>
<td>7.0178</td>
<td>7.0141</td>
<td>6.9952</td>
<td>Passed</td>
</tr>
<tr>
<td>x4.m1</td>
<td>0.055291</td>
<td>6.6471</td>
<td>6.6447</td>
<td>6.6324</td>
<td>Passed</td>
</tr>
<tr>
<td>x1.m1</td>
<td>0.049069</td>
<td>5.9699</td>
<td>5.9677</td>
<td>5.9566</td>
<td>Passed</td>
</tr>
<tr>
<td>x10.m1</td>
<td>0.044038</td>
<td>5.4161</td>
<td>5.4141</td>
<td>5.4039</td>
<td>Passed</td>
</tr>
<tr>
<td>x6.m1</td>
<td>0.044</td>
<td>5.4119</td>
<td>5.41</td>
<td>5.3998</td>
<td>Passed</td>
</tr>
<tr>
<td>x8.m1</td>
<td>0.04301</td>
<td>5.3021</td>
<td>5.3002</td>
<td>5.2902</td>
<td>Passed</td>
</tr>
<tr>
<td>x9.m2</td>
<td>0.030815</td>
<td>3.9276</td>
<td>3.9255</td>
<td>3.9145</td>
<td>Passed</td>
</tr>
<tr>
<td>x5.m2</td>
<td>0.030724</td>
<td>3.9171</td>
<td>3.915</td>
<td>3.9041</td>
<td>Passed</td>
</tr>
<tr>
<td>x7.m2</td>
<td>0.030223</td>
<td>3.8597</td>
<td>3.8576</td>
<td>3.8468</td>
<td>Passed</td>
</tr>
<tr>
<td>x11.m2</td>
<td>0.030096</td>
<td>3.845</td>
<td>3.8429</td>
<td>3.8322</td>
<td>Passed</td>
</tr>
</tbody>
</table>
Control Statements

The Spectre® circuit simulator lets you place a sequence of control statements in the netlist. You can use the same control statement more than once. Different Spectre control statements are discussed throughout this manual. The following are control statements:

- The alter and altergroup Statements on page 183
- The check Statement on page 186
- The ic and nodeset Statements on page 187
- The info Statement on page 192
- The options Statement on page 198
- The paramset Statement on page 200
- The save Statement on page 201
- The set Statement on page 210
- The shell Statement on page 211
- The statistics Statement on page 211

The alter and altergroup Statements

You modify individual parameters for devices, models, circuit, and subcircuit parameters during a simulation with the alter statement. The modifications apply to all analyses that follow the alter statement in your netlist until you request another parameter modification. You also use the alter statement to change the following options statement temperature parameters and scaling factors:

- temp
- tnom
- scale
You can use the `altergroup` statement to respecify device, model, and circuit parameter statements that you want to change for subsequent analyses. You can also change subcircuits if you do not change the topology.

### Changing Parameter Values for Components

To change a parameter value for a component device or model, you specify the device or model name, the parameter name, and the new parameter value in the `alter` statement. You can modify only one parameter with each `alter` statement, but you can put any number of `alter` statements in a netlist. The following example demonstrates `alter` statement syntax:

SetMag alter dev=Vt1 param=mag value=1

- `SetMag` is the unique netlist name for this `alter` statement. (Like many Spectre statements, each `alter` statement must have a unique name.)
- The keyword `alter` is the primitive name for the `alter` statement.
- `dev=Vt1` identifies `Vt1` as the netlist name for the component statement you want to modify. You identify an instance statement with `dev` and a model statement with `mod`. When you use the `alter` statement to modify a circuit parameter, you leave both `dev` and `mod` unspecified.
- `param=mag` identifies `mag` as the parameter you are modifying. If you omit this parameter, the Spectre simulator uses the first parameter listed for each component in the Spectre online help as the default.
- `value=1` identifies `1` as the new value for the `mag` parameter. If you leave `value` unspecified, it is set to the default for the parameter.

### Changing Parameter Values for Models

To change a parameter value for model files with the `altergroup` statement, you list the device, model, and circuit parameter statements as you would in the main netlist. Within an `alter` group, each model is first defaulted and then the model parameters are updated. You cannot nest `alter` groups. You cannot change from a model to a model group and vice versa. The following example demonstrates `altergroup` statement syntax:

ag1 altergroup {
  parameters p1=1
  model myres resistor r1=1e3 af=p1
  model mybsim bsim3v3 lmax=p1 lmin=3.5e-7
}
The following example shows the full replacement of models using the `altergroup` statement:

```
ff_25 altergroup {
    include "./models/corner_ff"
}
```

For each model or device being altered, the parameters are first defaulted and then set to the new values. The parameter dependencies are updated and maintained.

You can include files into the alter group and can use the `simulator lang=spice` command to switch language mode. For more details on the `include` command, see the Spectre online help (`spectre -h include`). A model defined in the netlist has to have the same model name and primitive type (such as `bsim2`, `bsim3`, or `bjt`) in the alter group. For model groups, you can change the number of models in the group. You cannot change from a model to a model group and vice versa. For details on model groups, see the Spectre online help (`spectre -h bsim3v3`).

**Further Examples of Changing Component Parameter Values**

This example changes the `is` parameter of a model named `SH3` to the value `1e-15`:

```
modify2 alter mod=SH3 param=is value=1e-15
```

The following examples show how to use the `param default` in an `alter` statement. The first parameter listed for resistors in the Spectre online help is the default. For resistors, this is the resistance parameter `r`.

Consequently, if `R1` is a resistor, the following two `alter` statements are equivalent:

```
change1 alter dev=R1 param=r value=50
change1 alter dev=R1 value=50
```

**Changing Parameter Values for Circuits**

When you change a circuit parameter, you use the same syntax as when you change a device or model parameter except that you do not enter a `dev` or a `mod` parameter.

This example changes the ambient temperature to 0°C:

```
change2 alter param=temp value=0
```
The following table describes the circuit parameters you can change with the alter statement:

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>temp</td>
<td>Ambient temperature</td>
</tr>
<tr>
<td>tnom</td>
<td>Default measurement temperature for component parameters</td>
</tr>
<tr>
<td>scalem</td>
<td>Component model scaling factor</td>
</tr>
<tr>
<td>scale</td>
<td>Component instance scaling factor</td>
</tr>
</tbody>
</table>

Note: If you change temp or tnom using an alter statement, all expressions with temp or tnom are reevaluated.

The check Statement

You can perform a check analysis at any point in a simulation to be sure that the values of component parameters are reasonable. You can perform checks on input, output, or operating-point parameters. The Spectre simulator checks parameter values against parameter soft limits. To use the check analysis, you must also enter the +param command line argument with the spectre command to specify a file that contains the soft limits.

The following example illustrates the syntax of the check statement. It tells the Spectre simulator to check the parameter values for instance statements.

```
ParamChk check what=inst
```

- ParamChk is your unique name for this check statement.
- The keyword check is the component keyword for the statement.
- The what parameter tells the Spectre simulator which parameters to check.

The what parameter of the check statement gives you the following options:

<table>
<thead>
<tr>
<th>Option</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>none</td>
<td>Disables parameter checking</td>
</tr>
<tr>
<td>models</td>
<td>Checks input parameters for all models only</td>
</tr>
<tr>
<td>inst</td>
<td>Checks input parameters for all instances only</td>
</tr>
<tr>
<td>input</td>
<td>Checks input parameters for all models and all instances</td>
</tr>
</tbody>
</table>
The ic and nodeset Statements

The Spectre simulator lets you provide state information to the DC and transient analyses. You can specify two kinds of state information:

- Initial conditions

  The `ic` statement lets you specify values for the starting point of a transient analysis. The values you can specify are voltages on nodes and capacitors, and currents on inductors.

- Nodesets

  Nodesets are estimates of the solution you provide for the DC or transient analyses. Unlike initial conditions, their values have no effect on the final results. Nodesets usually act only as aids in speeding convergence, but if a circuit has more than one solution, as with a latch, nodesets can bias the solution to the one closest to the nodeset values.

Setting Initial Conditions for All Transient Analyses

You can specify initial conditions that apply to all transient analyses in a simulation or to a single transient analysis. The `ic` statement and the `ic` parameter described in this section set initial conditions for all transient analyses in the netlist. In general, you use the `ic` parameter of individual components to specify initial conditions for those components, and you use the `ic` statement to specify initial conditions for nodes. You can specify initial conditions for inductors with either method. Specifying `cmin` for a transient analysis does not satisfy the condition that a node has a capacitive path to ground.

**Note**: Do not confuse the `ic` parameter for individual components with the `ic` parameter of the transient analysis. The latter lets you select from among different initial conditions specifications for a given transient analysis.

Specifying Initial Conditions for Components

You can specify initial conditions in the instance statements of capacitors, inductors, and windings for magnetic cores. The `ic` parameter specifies initial voltage values for capacitors
and current values for inductors and windings. In the following example, the initial condition voltage on capacitor Cap13 is set to two volts:

```
cap13 11 9 capacitor c=10n ic=2
```

### Specifying Initial Conditions for Nodes

You use the `ic` statement to specify initial conditions for nodes or initial currents for inductors. The nodes can be inside a subcircuit or internal nodes to a component.

The following is the format for the `ic` statement:

```
ic <signalName=value> ...
```

The format for specifying signals with the `ic` statement is similar to that used by the `save` statement. This method is described in detail in “Saving Main Circuit Signals” on page 213. Consult this discussion if you need further clarification about the following example.

```
ic Voff=0 X3.7=2.5 M1:int_d=3.5 L1:1=1u
```

This example sets the following initial conditions:

- The voltage of node `Voff` is set to 0.
- Node 7 of subcircuit X3 is set to 2.5 V.
- The internal drain node of component M1 is set to 3.5 V. (See the following table for more information about specifying internal nodes.)
- The current for inductor `L1` is set to 1µ.

Specifying initial node voltages requires some additional discussion. The following table tells you the internal voltages you can specify with different components.

<table>
<thead>
<tr>
<th>Component</th>
<th>Internal Node Specifications</th>
</tr>
</thead>
<tbody>
<tr>
<td>BJT</td>
<td><code>int_c, int_b, int_e</code></td>
</tr>
<tr>
<td>BSIM</td>
<td><code>int_d, int_s</code></td>
</tr>
<tr>
<td>MOSFET</td>
<td><code>int_d, int_s</code></td>
</tr>
<tr>
<td>GaAs MESFET</td>
<td><code>int_d, int_s, int_g</code></td>
</tr>
<tr>
<td>JFET</td>
<td><code>int_d, int_s, int_g, int_b</code></td>
</tr>
<tr>
<td>Winding for Magnetic Core</td>
<td><code>int_Rw</code></td>
</tr>
<tr>
<td>Magnetic Core with Hysteresis</td>
<td><code>flux</code></td>
</tr>
</tbody>
</table>
Supplying Solution Estimates to Increase Speed

You use the `nodeset` statement to supply estimates of solutions that aid convergence or bias the simulation towards a given solution. You can use nodesets for all DC and initial transient analysis solutions in the netlist. The `nodeset` statement has the following format:

```plaintext
nodeset <signalName=value>…
```

Values you can supply with the `nodeset` statement include voltages on topological nodes, including internal nodes, and currents through voltage sources, inductors, switches, transformers, N-ports, and transmission lines.

The format for specifying signals with the `nodeset` statement is similar to that used by the `save` statement. This method is described in detail in “Saving Main Circuit Signals” on page 213. Consult this discussion if you need further clarification about the following example.

```plaintext
nodeset Voff=0 X3.7=2.5 M1:int_d=3.5 L1:1=1u
```

This example sets the following solution estimates:

- The voltage of node `Voff` is set to 0.
- Node 7 of subcircuit `X3` is set to 2.5 V.
- The internal drain node of component `M1` is set to 3.5 V. (See the table in the `ic statements` section of this chapter for more information about specifying internal nodes.)
- The current for inductor `L1` is set to 1µ.

Specifying State Information for Individual Analyses

You can specify state information for individual analyses in two ways:

- You can use the `ic` parameter of the transient analysis to choose which previous specifications are used.
- You can create a state file that is read by an individual analysis.
Choosing Which Initial Conditions Specifications Are Used for a Transient Analysis

The `ic` parameter in the transient analysis lets you select among several options for which initial conditions to use. You can choose the following settings:

<table>
<thead>
<tr>
<th>Parameter Setting</th>
<th>Action Taken</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>dc</code></td>
<td>Initial conditions specifiers are ignored, and the existing DC solution is used.</td>
</tr>
<tr>
<td><code>node</code></td>
<td>The <code>ic</code> statements are used, and the <code>ic</code> parameter settings on the capacitors and inductors are ignored.</td>
</tr>
<tr>
<td><code>dev</code></td>
<td>The <code>ic</code> parameter settings on the capacitors and inductors are used, and the <code>ic</code> statements are ignored.</td>
</tr>
<tr>
<td><code>all</code></td>
<td>Both the <code>ic</code> statements and the <code>ic</code> parameters are used. If specifications conflict, <code>ic</code> parameters override <code>ic</code> statements.</td>
</tr>
</tbody>
</table>

Specifying State Information with State Files

You can also specify initial conditions and estimate solutions by creating a state file that is read by the appropriate analysis. You can create a state file in two ways:

- You can instruct the Spectre simulator to create a state file in a previous analysis for future use.
- You can create a state file manually in a text editor.

Telling the Spectre Simulator to Create a State File

You can instruct the Spectre simulator to create a state file from either the initial point or the final point in an analysis. To write a state file from the initial point in an analysis, use the `write` parameter. To write a state file from the final point, use the `writefinal` parameter. Each of the following two examples writes a state file named `ua741.dc`. The first example writes the state file from the initial point in the DC sweep, and the second example writes the state file from the final point in the DC sweep.

```plaintext
Drift dc param=temp start=0 stop=50.0 step=1 \
readns="ua741.dc" write="ua741.dc"

Drift dc param=temp start=0 stop=50.0 step=1 \
readns="ua741.dc" writefinal="ua741.dc"
```
Creating a State File Manually

The syntax for creating a state file in a text editor is simple. Each line contains a signal name and a signal value. Anything after a pound sign (#) is ignored as a comment. The following is an example of a simple state file:

```
# State file generated by Spectre from circuit file 'wilson'
# during 'stepresponse' at 5:39:38 PM, Jan 21, 1992.
1  0.588793510612534
2  1.17406247989272
3  14.9900516233357
pwr  15
vcc:p  -9.9483766642647e-06
```

Reading State Files

To read a state file as an initial condition, use the `read` transient analysis parameter. To read a state file as a nodeset, use the `readns` parameter. This example reads the file `intCond` as initial conditions:

```
DoTran_z12 tran start=0 stop=0.003 \
step=0.00015 maxstep=6e-06 read="intCond"
```

This second example reads the file `soluEst` as a nodeset.

```
DoTran_z12 tran start=0 stop=0.003 \
step=0.00015 maxstep=6e-06 readns="soluEst"
```

Special Uses for State Files

State files can be useful for the following reasons:

- You can save state files and use them in later simulations. For example, you can save the solution at the final point of a transient analysis and then continue the analysis in a later simulation by using the state file as the starting point for another transient analysis.
- You can use state files to create automatic updates of initial conditions and nodesets.

The following example demonstrates the usefulness of state files:

```
altTemp alter param=temp value=0
Drift dc param=temp start=0 stop=50.0 step=1 \
readns="ua741.dc0" write="ua741.dc0"
XferVsTemp xf param=temp start=0 stop=50 step=1 \nprobe=Rload freq=1kHz readns="ua741.dc0"
```

The first analysis computes the DC solution at T=0C, saves it to a file called `ua741.dc0`, and then sweeps the temperature to T=50C. The transfer function analysis (`xf`) resets the temperature to zero. Because of the temperature change, the DC solution must be recomputed. Without the use of state files, this computation might slow the simulation.
because the only available estimate of the DC solution would be that computed at T=50C, the final point in the DC sweep. However, by using a state file to preserve the initial DC solution at T=0C, you can enable the Spectre simulator to compute the new DC solution quickly. The computation is fast because the Spectre simulator can use the DC solution computed at T=0C to estimate the new solution. You can also make future simulations of this circuit start quickly by using the state file to estimate the DC solution. Even if you have altered a circuit, it is usually faster to start the DC analysis from a previous solution than to start from the beginning.

The info Statement

You can generate lists of component parameter values with the info statement. With this statement, you can access the values of input, output, and operating-point parameters, and print the node capacitance table. These parameter types are defined as follows:

- **Input parameters**
  
  Input parameters are those you specify in the netlist, such as the given length of a MOSFET or the saturation current of a bipolar resistor.

- **Output parameters**
  
  Output parameters are those the simulator computes, such as temperature-dependent parameters and the effective length of a MOSFET after scaling.

- **Operating-point parameters**
  
  Operating-point parameters are those that depend on the operating point.

- **Node Capacitance Table**
  
  The node capacitance table displays the capacitance between the nodes of a circuit.

You can also list the minimum and maximum values for the input, output, and operating-point parameters, along with the names of the components that have those values.

<table>
<thead>
<tr>
<th>Parameter Setting</th>
<th>Action Taken</th>
</tr>
</thead>
<tbody>
<tr>
<td>what=oppoint</td>
<td>Prints the oppoint parameters. Other possible values are none, inst, input, output, nodes, all, terminals, oppoint, models, and captab.</td>
</tr>
<tr>
<td>where=logfile</td>
<td>Prints the parameters to the logfile. Other possible values are nowhere, screen, file, and rawfile.</td>
</tr>
</tbody>
</table>
Specifying the Parameters You Want to Save

You specify parameters you want to save with the `info` statement `what` parameter. You can give this parameter the following settings:

<table>
<thead>
<tr>
<th>Setting</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>none</td>
<td>Lists no parameters</td>
</tr>
<tr>
<td>inst</td>
<td>Lists input parameters for instances of all components</td>
</tr>
<tr>
<td>models</td>
<td>Lists input parameters for models of all components</td>
</tr>
<tr>
<td>input</td>
<td>Lists input parameters for instances and models of all components</td>
</tr>
<tr>
<td>output</td>
<td>Lists effective and temperature-dependent parameter values</td>
</tr>
<tr>
<td>all</td>
<td>Lists input and output parameter values</td>
</tr>
<tr>
<td>oppoint</td>
<td>Lists operating-point parameters</td>
</tr>
<tr>
<td>terminals</td>
<td>The output is a node-to-terminal map</td>
</tr>
<tr>
<td>nodes</td>
<td>The output is a terminal-to-node map</td>
</tr>
<tr>
<td>captab</td>
<td>Prints node-to-node capacitance.</td>
</tr>
</tbody>
</table>

The `info` statement gives you some additional options. You can use the `save` parameter of the `info` statement to specify groups of signals whose values you want to list. For more information about `save` parameter options, consult “Saving Groups of Signals” on page 219. Finally, you can generate a summary of maximum and minimum parameter values with the `extremes` option.
Specifying the Output Destination

You can choose among several output destination options for the parameters you list with the `info` statement. With the `info` statement `where` parameter, you can

- Display the parameters on a screen
- Send the parameters to a log file, to the raw file, or to a file you create

When the `info` analysis is called from a transient analysis or used inside of a sweep, the name of the `info` analysis is prepended by the parent analysis. If the `file` option is used to save the results, use the `%A` percent code (described in “Description of Spectre Predefined Percent Codes” on page 248) in the filename to prevent the file from being overwritten.

For example, the following `info` statement

```
tempSweep sweep param=temp start=27 stop=127 step=10 {dc1 dc dcInfo info what=oppoint where=file file="infodata.%A"}
```

Produces:
```
infodata.tempSweep-000_dcInfo
infodata.tempSweep-001_dcInfo
infodata.tempSweep-002_dcInfo
infodata.tempSweep-003_dcInfo
```

and so on...

Examples of the info Statement

You format the `info` statement as follows:

```
StatementName info parameter=value
```

The following example tells the Spectre simulator to send the maximum and minimum input parameters for all models to a log file:
```
Inparams info what=models where=logfile extremes=only
```

For a complete description of the parameters available with the `info` statement, consult the lists of analysis and control statement parameters in the Spectre online help (`spectre -h`).

Printing the Node Capacitance Table

The Spectre simulator allows you to print node capacitance to an output file. This can help you in identifying possible causes of circuit performance problems due to capacitive loading.

The capacitance between nodes \( x \) and \( y \) is defined as
where $q_x$ is the sum of all charges in the terminal connected to node $x$, and $v_y$ is the voltage at node $y$.

The total capacitance at node $x$ is defined as

$$C_{xx} = \frac{\partial q_x}{\partial v_x}$$

where charge $q_x$ and voltage $v_x$ are at the same node $x$.

You may use the captab analysis to display the capacitance between the nodes in your circuit. This is an option in the info statement. Here is an example of the info settings you would set to perform a captab analysis:

<table>
<thead>
<tr>
<th>Parameter Setting</th>
<th>Action Taken</th>
</tr>
</thead>
<tbody>
<tr>
<td>what=captab</td>
<td>Performs captab analysis. The default value is oppoint.</td>
</tr>
<tr>
<td>where=logfile</td>
<td>Prints the parameters to a logfile. Other possible values are nowhere, screen, and file. The value rawfile is not supported for node capacitance.</td>
</tr>
<tr>
<td>title=captab</td>
<td>Prints captab as the title of the analysis in the output file.</td>
</tr>
<tr>
<td>threshold=0</td>
<td>Specifies the threshold capacitance value. The nodes for which the total node capacitance is below the threshold value will not be printed in the output table.</td>
</tr>
<tr>
<td>detail=node</td>
<td>Displays the total node capacitances. Other possible values are nodetoground and nodetonode.</td>
</tr>
<tr>
<td>sort=value</td>
<td>Sorts the captab output according to value. The other possible value is name.</td>
</tr>
</tbody>
</table>

For a complete list of captab parameters and values, consult the Spectre online help (spectre -h).
You may use the `infotimes` option of the transient analysis when you bind the `captab` analysis to a transient analysis. This runs the `captab` analysis at specified time intervals. The syntax for the `infotimes` option is

```
infolines=[x1, x2...]```

where `x1` and `x2` are time points for which the `info` analysis should be performed. The following is an example of binding a `captab` analysis to a transient analysis.

```
tran1 tran stop=1µ infolines=[0.1µ 0.5µ] infoname=capInfo
capInfo info what=captab where=file file='capNodes'
detail=nodetonode
```

**Output Table**

The output for the `captab` analysis is printed in the following format:

- The first column displays the names of the two nodes (From_node:To_Node).
- The second column displays the fixed (linear) capacitance between the two nodes.
- The third column is the variable (non-linear) capacitance between the two nodes.
- The last column displays the total capacitance between the nodes.

Table 7-1 displays the output for the circuit below when `detail=nodetonode`:
In this circuit, the total capacitance at node 2 (n2:n2 in the table) is:
C_1+C_2+C_3=5p

The total capacitance between node 2 and node 1 (n2:n1) is:
C_2(Linear)+C_3(Non-Linear)=3p

**Table 7-1 Node Capacitance Table Sorted by Value**

<table>
<thead>
<tr>
<th></th>
<th>Fixed</th>
<th>Variable</th>
<th>Sum</th>
</tr>
</thead>
<tbody>
<tr>
<td>n2:n2</td>
<td>2p</td>
<td>3p</td>
<td>5p</td>
</tr>
<tr>
<td>n2:n1</td>
<td>2p</td>
<td>1p</td>
<td>3p</td>
</tr>
<tr>
<td>n1:n1</td>
<td>3.5p</td>
<td>1p</td>
<td>4.5p</td>
</tr>
<tr>
<td>n1:0</td>
<td>1p</td>
<td>0</td>
<td>1p</td>
</tr>
<tr>
<td>n1:n2</td>
<td>2p</td>
<td>1p</td>
<td>3p</td>
</tr>
<tr>
<td>n1:n3</td>
<td>0.5p</td>
<td>0</td>
<td>0.5p</td>
</tr>
<tr>
<td>n3:n3</td>
<td>1.5p</td>
<td>0</td>
<td>1.5p</td>
</tr>
<tr>
<td>n3:0</td>
<td>1p</td>
<td>0</td>
<td>1p</td>
</tr>
<tr>
<td>n3:n1</td>
<td>0.5p</td>
<td>0</td>
<td>0.5p</td>
</tr>
</tbody>
</table>

The total node capacitance at nodes 1, 2, and 3 is represented by the rows n1:n1, n2:n2, and n3:n3 respectively.

There is no entry for n3:n2, which means there is no capacitance between these two nodes.

Table 7-1 is sorted by value. The rows are first grouped according to node names - the rows with the same From_Node are kept in a group. The row depicting the total capacitance at each node is always displayed first in the group, and the row displaying the node-to-ground capacitance is second. The remaining rows within each group are sorted in descending order of the Sum value.

**Note:** If the threshold is set to 2p (thresh=2p), the row n1:n3 will not be printed since the capacitance between them is less than the threshold. The row n1:0 will be printed since the node-to-ground capacitance is always printed. The group n3:n3, n3:0, and n3:n1 will not be printed because the total node capacitance at node 3 (n3:n3) is less than the threshold.

If you sort the table by name (sort=name), it would look as follows:
Table 7-2  Node Capacitance Table Sorted by Name

<table>
<thead>
<tr>
<th>Node</th>
<th>Fixed</th>
<th>Variable</th>
<th>Sum</th>
</tr>
</thead>
<tbody>
<tr>
<td>n1:n1</td>
<td>3.5p</td>
<td>1p</td>
<td>4.5p</td>
</tr>
<tr>
<td>n1:0</td>
<td>1p</td>
<td>0</td>
<td>1p</td>
</tr>
<tr>
<td>n1:n2</td>
<td>2p</td>
<td>1p</td>
<td>3p</td>
</tr>
<tr>
<td>n1:n3</td>
<td>0.5p</td>
<td>0</td>
<td>0.5p</td>
</tr>
<tr>
<td>n2:n2</td>
<td>2p</td>
<td>3p</td>
<td>5p</td>
</tr>
<tr>
<td>n2:0</td>
<td>0</td>
<td>2p</td>
<td>2p</td>
</tr>
<tr>
<td>n2:n1</td>
<td>2p</td>
<td>1p</td>
<td>3p</td>
</tr>
<tr>
<td>n3:n3</td>
<td>1.5p</td>
<td>0</td>
<td>1.5p</td>
</tr>
<tr>
<td>n3:n0</td>
<td>1p</td>
<td>0</td>
<td>1p</td>
</tr>
<tr>
<td>n3:n1</td>
<td>0.5p</td>
<td>0</td>
<td>0.5p</td>
</tr>
</tbody>
</table>

In this case, the From_Node:To_Node column is sorted alpha-numerically. However, the row depicting the total capacitance at each node is always displayed first in the group, and the row displaying the node-to-ground capacitance is second.

The options Statement

To enter initial parameters for your simulation that you do not specify in your environment variables or on your command line, you use the options statement. You can control parameters in a number of areas with the options statement:

- Parameters that specify tolerances for accuracy
- Parameters that control temperature
- Parameters that select output data
- Parameters that help solve convergence difficulties
- Parameters that control error handling and annotation

For a complete list of the parameters you can set with the options statement, consult the Spectre online help (spectre -h).
options Statement Form at

The options statement format is as follows:

```
Name options parameter=value...
```

*Name* is a unique name you give to the options statement. The Spectre simulator uses this name to identify this statement in error or annotation messages.

*options* is the keyword options, the primitive name for this control statement.

*Parameter=value* is an options statement parameter followed by an equal sign and the value you choose for the parameter. You can enter any number of parameter specifications with a single options statement.

options Statement Example

The following is a correctly formatted options statement:

```
Examp options rawfmt=psfbin audit=brief temp=30 \
  save=lvlpub nestlvl=3 rawfile=%C:r.raw useprobes=no
```

The example sets the *rawfmt*, *audit*, *temp*, *save*, *nestlvl*, *rawfile*, and *useprobes* parameters for an options statement named Examp. The backslash (\) at the end of the first line is a line continuation character. Nonnumerical parameter values are chosen from the possible values listed in the Spectre online help (spectre -h).

Setting Tolerances

You need to set tolerances if the Spectre simulator’s default settings do not suit your needs. This section tells you how to make the needed adjustments. If you need to examine default tolerances for any Spectre parameters, you can find them in the Spectre online help (spectre -h).

Setting Tolerances with the options Statement

The following options statement parameters control error tolerances:

```
reltol
```

One of the Spectre simulator’s convergence criteria is that the difference between solutions in the last two iterations for a given time must be sufficiently small. With reltol, you set the maximum relative tolerance for values computed in the last two iterations. The default for reltol is 0.001.
**iabstol and vabstol**

These parameters set absolute, as opposed to relative, tolerances for differences in the computed values of voltages and currents in the last two iterations. These parameter values are added to the tolerances specified by `reltol`. They let the Spectre simulator converge when the differences accepted by `reltol` approach zero. You can also set these values with the `quantity` statement.

### Additional options Statement Settings You Might Need to Adjust

This section provides some explanation of commonly used `options` statement parameters. It is not a complete listing of `options` statement parameters. For a complete list, consult the Spectre online help (`spectre -h`).

#### tempeffects

This parameter defines how temperature affects the built-in primitive components. It takes the following three values:

- **vt**—Only the thermal voltage $V_t = \frac{kT}{q}$ is allowed to vary with temperature.
- **tc**—The component temperature coefficient parameters (parameters that start with `tc`, such as `tc1`, and `tc2`) are active as well as the thermal voltage. You use this setting when you want to disable the temperature effects for nonlinear devices.
- **all**—All built-in temperature models are enabled.

#### compatible

This parameter changes some of the device models to be more consistent with the models in other simulators. See Chapter 3, “Component Statements,” of the *Spectre Circuit Simulator Reference* manual and the `options` statement parameter listings in the Spectre online help (`spectre -h`) for more information.

### The paramset Statement

For the `sweep` analysis only, the `paramset` statement allows you to specify a list of parameters and their values. This can be referred by a `sweep` analysis to sweep the set of parameters over the values specified. For each iteration of the sweep, the netlist parameters are set to the values specified by a row. The values have to be numbers, and the parameters’ names have to be defined in the input file (netlist) before they are used. The `paramset` statement is allowed only in the top level of the input file.

The following is the syntax for the `paramset` statement:
Name paramset {
    <list of netlist parameters>
    <list of values foreach netlist parameter>
    <list of values foreach netlist parameter> ...
}

Here is an example of the paramset statement:
parameters p1=1 p2=2 p3=3
data paramset {
    p1 p2 p3
    5 5 5
    4 3 2
}

Combining the paramset statement with the sweep analysis allows you to sweep multiple parameters simultaneously, for example, power supply voltage and temperature.

The save Statement

You can save signals for individual nodes and components or save groups of signals.

Saving Signals for Individual Nodes and Components

You can include signals for individual nodes and components in the save list by placing save statements (not to be confused with the save parameter) in your netlist. When you specify signals in a save statement, the Spectre simulator sends these signals to the output raw file, as long as the nestlv1 setting does not filter them. In this section, you will learn the following:

n How to save voltages for individual nodes
n How to save all signals for an individual component
n How to save selected signals for an individual component

The syntax for the save statement varies slightly, depending on whether the requested data is from the main circuit or a subcircuit. You will learn about the syntax for main circuit statements first, and then you will learn how to modify the save statement to store signals from subcircuits.

Saving Main Circuit Signals

The save statement general syntax has the following arguments. You can specify more than one argument with a single save statement, and you can mix the types of arguments in a single statement.
save <signalName>…
save <compName>…
save <compName:modifier>…
save <subcircuitName:terminalIndex> …

- **signalName** is generally the netlist name of a node whose voltage you want to save.
- **compName** is the netlist name of a component whose signals you want to save.
- **modifier** specifies signals you want to save for a particular component. It can have the following types of values:
  - A terminal name
    
    Terminal names for components are the names for nodes in component instance definitions. You can find instance definitions for each component in the component parameter listings in the Spectre online help (`spectre -h`). For example, the following is the instance definition of a microstrip line. The terminal names are t1, b1, t2, and b2.
    
    ```
    Name t1 b1 t2 b2 msline parameter=value...
    ```
  - A terminal index
    
    The terminal index is a number that indicates where a terminal is in the instance definition. You give the first terminal a terminal index of 1, the second a terminal index of 2, and so on. In this example, the terminal indexes are 1 for sink and 2 for src.
    
    ```
    Name sink src isource parameter=value...
    ```
  - A name of an operating-point parameter (from the lists of parameters for each component in the Spectre online help)
  - The name of a SpectreHDL/Verilog©-A internal variable
  - One of the following keywords
    - **currents** To save all currents of the device
    - **static** To save resistive currents of the device
    - **displacement** To save capacitive currents of the device
    - **dynamic** To save charge or flux of the device
    - **oppoint** To save the operating points of the device
    - **probe** To measure current of the device with a probe
subcircuitName is the instance name of a subcircuit call. Saving terminal currents for subcircuit calls is the same as saving terminal currents for other instance statements except that you must identify individual terminal currents you want to save by the terminal index.

**Note:** To save all terminal currents for subcircuit calls, you use a `save` statement or specify the `subcktprobelvl` parameter in an `options` statement. The `currents=all` option of the `options` statement saves currents only for devices.

### Saving Subcircuit Signals

To save a subcircuit signal with the `save` statement, modify the main circuit syntax as follows:

- Give a full path to the subcircuit name. Start with the highest level subcircuit and identify the signals you want to save at the end of the path. Separate each name with a period.

### Examples of the `save` Statement

The following table shows you examples of `save` statement syntax. When you specify node names, the Spectre simulator saves node voltages. Currents are identified by the terminal node name or the index number.

**Exception:** Currents through probes take the name of the probe.

<table>
<thead>
<tr>
<th>save Statement</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>save 7</td>
<td>Saves voltage for a node named 7.</td>
</tr>
<tr>
<td>save Q4:currents</td>
<td>Saves all terminal currents associated with component Q4.</td>
</tr>
<tr>
<td>save Q4:static</td>
<td>Saves resistive terminal currents associated with component Q4.</td>
</tr>
<tr>
<td>save D8:cap</td>
<td>Saves the junction capacitance for component D8. (Assumes D8 is a diode, and, therefore, cap is an operating-point parameter.)</td>
</tr>
<tr>
<td>save Q5 D9:oppoint</td>
<td>Saves all signal information for component Q5 and the operating-point parameters for component D9.</td>
</tr>
</tbody>
</table>
### Saving Individual Currents with Current Probes

A current probe is a component that measures the current passing between two nodes. Its effect is like placing an amp meter on two points of a circuit. It creates a new branch in the circuit between the two nodes, forces the voltages on the two nodes to be equal, and then measures the flow of current.

### When to Use Current Probes

Sometimes it is preferable or necessary to use a current probe rather than a `save` statement. You use a probe instead of a `save` statement under the following circumstances:

- **If you want increased flexibility for giving currents descriptive names**
  
  With a current probe, you can name the current anything you want. With a `save` statement, the name of the current must have a `:name` suffix.

- **If you are saving measurements for current-controlled components**

- **If you are saving currents for an AC analysis**

- **If you are saving measurements for a current that passes between two parts of a circuit but not through a terminal**

---

<table>
<thead>
<tr>
<th><code>save Statement</code></th>
<th><code>Action</code></th>
</tr>
</thead>
<tbody>
<tr>
<td><code>save Q1:c</code></td>
<td>Saves the collector current for component <code>Q1</code>. (Example assumes <code>Q1</code> is a BJT, and, therefore, <code>c</code> is a terminal name.)</td>
</tr>
<tr>
<td><code>save Q1:1</code></td>
<td>Same effect as the previous statement. Saves the collector current for component <code>Q1</code>. Identifies the terminal with its terminal index instead of its terminal name.</td>
</tr>
<tr>
<td><code>save M2:d:displacement</code></td>
<td>Saves capacitive current associated with the drain terminal of component <code>M2</code>. (Example assumes <code>M2</code> is a MOSFET, so <code>d</code> is a terminal name.)</td>
</tr>
<tr>
<td><code>save Q3:currents M1:all</code></td>
<td>Saves all currents for component <code>Q3</code> and all signals for component <code>M1</code>.</td>
</tr>
<tr>
<td><code>save F4.S1.BJT3:oppoint</code></td>
<td>Saves operating-point parameters for terminal <code>BJT3</code>. <code>BJT3</code> is in subcircuit <code>S1</code>. Subcircuit <code>S1</code> is nested within subcircuit <code>F4</code>.</td>
</tr>
</tbody>
</table>
In the following example, you insert a current probe to measure the current flowing between A and B. Because there is no component between A and B, there is no other way to measure this current except to insert a current probe that has an identical current to the one you want to measure.

![Diagram of current probe inserted to measure current from A to B]

**Probe Statement Example**

To specify a current probe, place a statement with the following syntax in your netlist:

```
<Name> <in> <out> iprobe
```

- `<Name>` is the unique netlist name for the current probe component. The measured current also receives this name.
- `<in>` is the name you choose for the input node of the probe.
- `<out>` is the name you choose for the output node of the probe.
- `iprobe` is the primitive name of the component.

In the following example, the current probe measures the current between nodes `src` and `in`, names the measured current `Iin`, and saves `Iin` to the raw file.

```
Iin src in iprobe
```

**Note:** You can also direct the Spectre simulator to save currents with probes with a `save` statement option. For further information, see the description of `save statement keywords` in this chapter.

**Saving Power**

To save power dissipated on a circuit, subcircuit, or device, you use the `pwr` parameter. Power is calculated only during DC and transient analyses. The results are saved as a waveform, representing the instantaneous power dissipated in the circuit, subcircuit, or device.
Formatting the pwr Parameter

The syntax for the pwr parameter is illustrated by the following examples.

To save the power dissipated on a device or instance of a subcircuit, the syntax is
\[ \text{save instance\_name:pwr} \]

To save the total power, the syntax is
\[ \text{save :pwr} \]

You can explicitly save particular power variables. For example:
\[ \text{save :pwr } x1:pwr \text{ x1.x2.m1:pwr} \]

This statement saves three power signals:

- the total power dissipated (\(:pwr\))
- the power dissipated in the \(x1\) subcircuit instance (\(x1:pwr\))
- the power dissipated in the \(x1.x2.m1\) MOSFET

Power Options

The pwr parameter in the options statement can also be used to save power. The following table shows the five possible settings for the pwr option:

<table>
<thead>
<tr>
<th>Setting</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>all</td>
<td>The total power, the power dissipated in each subcircuit, and the power dissipated in each device is saved.</td>
</tr>
<tr>
<td>subckts</td>
<td>The total power and the power dissipated in each subcircuit is saved.</td>
</tr>
<tr>
<td>devices</td>
<td>The total power and the power dissipated in each device is saved.</td>
</tr>
<tr>
<td>total</td>
<td>The total power dissipated in the circuit is calculated and saved.</td>
</tr>
<tr>
<td>none</td>
<td>No power variable is calculated or saved. This is the default setting.</td>
</tr>
</tbody>
</table>

For example:

- opts options pwr=total
- save x1:pwr

This creates two power signals, \(:pwr\) (generated by the options statement) and \(x1:pwr\) (generated by the save statement).
Saving Groups of Signals

To save groups of signals as results, you use the `save` and `nestlvl` parameters. You specify which signals you want to save with the `save` parameter. You use the `nestlvl` parameter when you save signals in subcircuits. The `nestlvl` parameter specifies how many levels deep into the subcircuit hierarchy you want to save signals.

You can set these parameters as follows:

- **In `options` statements or `set` statements**
  
  If you set the `save` and `nestlvl` parameters with an `options` or a `set` statement, the setting applies to signal data from all analyses that follow that statement in the netlist.

- **In most analysis statements**
  
  If you set the `save` and `nestlvl` parameters with an analysis statement, the setting applies to that analysis only. It overrides any previous `save` or `nestlvl` settings.

Formatting the `save` and `nestlvl` Parameters

The syntax for both the `save` and `nestlvl` parameters is illustrated by the following `options` statement:

```plaintext
setting1 options save=lvlpub nestlvl=2
```

The `save` Parameter Options

The following table shows the possible settings for the `save` parameter:

<table>
<thead>
<tr>
<th>Setting</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>none</td>
<td>Does not save any data (currently does save one node chosen at random).</td>
</tr>
<tr>
<td>selected</td>
<td>Saves only signals specified with <code>save</code> statements. This is the default setting.</td>
</tr>
<tr>
<td>lvlpub</td>
<td>Saves all signals that are normally useful up to <code>nestlvl</code> deep in the subcircuit hierarchy. This option is equivalent to <code>allpub</code> for subcircuits. Normally useful signals include shared node voltages and currents through voltage sources and iprobes.</td>
</tr>
<tr>
<td>lvl</td>
<td>Saves all signals up to <code>nestlvl</code> deep in the subcircuit hierarchy. This option is relevant for subcircuits.</td>
</tr>
</tbody>
</table>
Use `lvl` or `all` (instead of `lvlpub` or `allpub`) to include internal node voltages and currents through other components that compute current.

Use `lvlpub` or `allpub` to exclude signals at internal nodes on devices (the internal collector, base, emitter on a BJT, the internal drain and source on a FET, etc). `lvlpub` and `allpub` also exclude the currents through inductors, controlled sources, transmission lines, transformers, etc.

**Note:** Setting the `save` parameter value to `selected` without any `save` statements in the netlist is not equivalent to specifying no output. Currently, the Spectre simulator saves all circuit nodes and branch currents with this combination of settings. This might change in future releases of the Spectre simulator.

### Saving Subcircuit Signals

To save groups of signals for subcircuits, you must adjust two parameter settings:

- Set the `save` parameter to either `lvl` or `lvlpub`.
- Set the `nestlvl` parameter to the number of levels in the hierarchy you want to save. The default setting for `nestlvl` is infinity, which saves all levels.

### Saving Groups of Currents

The `currents` parameter of the `options` statement computes and saves terminal currents. You use it to create settings for currents that apply to all terminals in the netlist.

For two-terminal components, the Spectre simulator saves only the first terminal (entering) currents. You must use a `save` statement or use the global `redundant_currents` parameter of the `options` statement to save data for the second terminal of a two-terminal component. For more information about the `save` statement, see “Saving Signals for Individual Nodes and Components” on page 201.
Setting the currents Parameter

The `currents` parameter has the following options:

<table>
<thead>
<tr>
<th>Setting</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>selected</code></td>
<td>Saves only currents that you specifically request with <code>save</code> statements or <code>save</code> parameters. Also saves naturally computed “branch” currents (currents through current probes, voltage sources, and inductors). This is the default setting.</td>
</tr>
<tr>
<td><code>nonlinear</code></td>
<td>Saves all terminal currents for nonlinear devices, naturally computed “branch” currents (currents through current probes, voltage sources, and inductors), and currents you specify with <code>save</code> statements. Can significantly increase simulation time.</td>
</tr>
<tr>
<td><code>all</code></td>
<td>Saves all terminal currents and currents available from <code>selected</code> settings to the raw file. Can significantly increase simulation time.</td>
</tr>
</tbody>
</table>

**Note:** Currently, if you set the `currents` parameter value to `nonlinear` or `all` and do not specify a `save` parameter value in an `options` statement, the Spectre simulator saves circuit nodes as well as the currents you requested. This might change in future releases of the Spectre simulator.

Examples of the currents Parameter

You use the following syntax for the `currents` parameter in the `options` statement. For more information about the `options` statement, see the parameter listings in the Spectre online help (`spectre -h`).

- The Spectre simulator saves all terminal currents for nonlinear components, currents specified with the `save` statement, and routinely computed currents.
  ```
  opt1 options currents=nonlinear
  ```

- The Spectre simulator saves all terminal currents.
  ```
  opt2 options currents=all
  ```

Setting Multiple Current Probes

Sometimes you might need to set a large number of current probes. This could happen, for example, if you need to save a number of ACs. (Current probes can find such small signal currents when they are not normally computed.) You can specify that all currents be calculated with current probes by placing `useprobes=yes` in an `options` statement.
Setting multiple current probes can greatly increase the DC and transient analysis simulation times. Consequently, this method is typically used only for small circuits and AC analysis.

**Important**

Adding probes to circuits that are sensitive to numerical noise might affect the solution. In such cases, an accurate solution might be obtained by reducing `reltol`.

### Saving Subcircuit Terminal Currents

You use the `subcktprobelvl` parameter to control the calculation of terminal currents for subcircuits. Current probes are added to the terminals of each subcircuit, up to `subcktprobelvl` deep. You can then save these terminal currents by setting the `save` parameter. The `nestlvl` parameter controls how many levels are returned.

### The set Statement

Except for temperature parameters and scaling factors, you use the `set` statement to modify any `options` statement parameters you set at the beginning of the netlist. The new settings apply to all analyses that follow the `set` statement in the netlist.

You can change the initial settings for the state of the simulator by placing a `set` statement in the netlist. The `set` statement is similar to the `options` statement that sets the state of the simulator, but it is queued with the analysis statements in the order you place them in the netlist.

You use the `set` statement to change previous `options` or `set` statement specifications. The modifications apply to all analyses that follow the `set` statement in the netlist until you request another parameter modification. The `set` and `options` statements have many identical parameters, but the `set` statement cannot modify all `options` statement parameters. The parameter listings in the Spectre online help tell you which parameters you can reset with the `set` statement.

The following example demonstrates the `set` statement syntax. This example turns off several annotation parameters.

```
Quiet set narrate=no error=no info=no
```

- **Quiet** is the unique name you give to the `set` statement.
- **The keyword** `set` is the primitive name for the `set` statement.
- **narrate, error, and info** are the parameters you are changing.
Note: If you want to change temp or tnom, use the alter statement.

The shell Statement

The shell analysis passes a command to the operating system command interpreter given in the SHELL environment variable. The command behaves as if it were typed into the Command Interpreter Window, except that any %X codes in the command are expanded first.

The default action of the shell analysis is to terminate the simulation.

The following is the syntax for the shell statement:

```
Name shell parameter=value ...
```

The statistics Statement

The statistics blocks allow you to specify batch-to-batch (process) and per-instance (mismatch) variations for netlist parameters. These statistically varying netlist parameters can be referenced by models or instances in the main netlist and can represent IC manufacturing process variation or component variations for board-level designs. For more information about the statistics statement, see “Specifying Parameter Distributions Using Statistics Blocks” on page 170.
Specifying Output Options

This chapter discusses the following topics:

- Signals as Output
- Saving Signals for Individual Nodes and Components
- Saving Groups of Signals
- Listing Parameter Values as Output
- Preparing Output for Viewing
- Accessing Output Files

Signals as Output

Signals are quantities that the simulator must determine to solve the network equations formulated to represent the circuit. The signals must be known before other output data can be computed.

Signals are mainly the physically meaningful quantities of interest to the user, such as the voltages on the topological nodes and naturally computed branch currents (such as those for inductors and voltage sources).

Other examples of signals are the voltages at the internal nodes of components and the terminal currents computed by using current probes at device or subcircuit terminals.

**Note:** If there are more than four terminals on a device (such as vbic, hbt, or bta_soi), the fifth and higher terminals do not return actual currents but return 0.0.

You can save signals and include them as simulation results. Signals you can save include the following:

- Voltages at topological nodes
- All currents
Other quantities the Spectre® circuit simulator computes to determine the operating point and other analysis data

Saving Signals for Individual Nodes and Components

You can include signals for individual nodes and components in the save list by placing `save` statements (not to be confused with the `save` parameter) in your netlist. When you specify signals in a `save` statement, Spectre sends these signals to the output raw file, as long as the `nestlvl` setting does not filter them. In this section, you will learn the following:

- How to save voltages for individual nodes
- How to save all signals for an individual component
- How to save selected signals for an individual component

The syntax for the `save` statement varies slightly, depending on whether the requested data is from the main circuit or a subcircuit. You will learn about the syntax for main circuit statements first, and then you will learn how to modify the `save` statement to store signals from subcircuits.

Saving Main Circuit Signals

The `save` statement general syntax has the following arguments. You can specify more than one argument with a single `save` statement, and you can mix the types of arguments in a single statement.

```markdown
save <signalName>...
save <compName>...
save <compName:modifier>...
save <subcircuitName:terminalIndex> ...
```

- `signalName` is generally the netlist name of a node whose voltage you want to save.
- `compName` is the netlist name of a component whose signals you want to save.
- `modifier` specifies signals you want to save for a particular component. It can have the following types of values:
  - A terminal name

  Terminal names for components are the names for nodes in component instance definitions. You can find instance definitions for each component in the component parameter listings in the Spectre online help (`spectre -h`). For example, the
following is the instance definition of a microstrip line. The terminal names are t1, b1, t2, and b2.

    Name t1 b1 t2 b2 msline parameter=value...

q A terminal index

The terminal index is a number that indicates where a terminal is in the instance
definition. You give the first terminal a terminal index of 1, the second a terminal
index of 2, and so on. In this example, the terminal indexes are 1 for sink and 2 for src.

    Name sink src isource parameter=value...

q A name of an operating-point parameter (from the lists of parameters for each
component in the Spectre online help)

q The name of a SpectreHDL/Verilog©-A internal variable

q One of the following keywords:

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>currents</td>
<td>To save all currents of the device</td>
</tr>
<tr>
<td>static</td>
<td>To save resistive currents of the device</td>
</tr>
<tr>
<td>displacement</td>
<td>To save capacitive currents of the device</td>
</tr>
<tr>
<td>dynamic</td>
<td>To save charge or flux of the device</td>
</tr>
<tr>
<td>oppoint</td>
<td>To save the operating points of the device</td>
</tr>
<tr>
<td>probe</td>
<td>To measure current of the device with a probe</td>
</tr>
<tr>
<td>pwr</td>
<td>To save power dissipated on a circuit, subcircuit, or device</td>
</tr>
<tr>
<td>all</td>
<td>To save all signals of the device</td>
</tr>
</tbody>
</table>

subcircuitName is the instance name of a subcircuit call. Saving terminal currents
for subcircuit calls is the same as saving terminal currents for other instance statements
except that you must identify individual terminal currents you want to save by the terminal
index.

Note: To save all terminal currents for subcircuit calls, you use a save statement or
specify the subcktprobelvl parameter in an options statement. The
currents=all option of the options statement saves currents only for devices.
## Saving Subcircuit Signals

To save a subcircuit signal with the `save` statement, modify the main circuit syntax as follows:

- Give a full path to the subcircuit name. Start with the highest level subcircuit and identify the signals you want to save at the end of the path. Separate each name with a period.

### Examples of the `save` Statement

The following table shows you examples of `save` statement syntax. When you specify node names, the Spectre simulator saves node voltages. Currents are identified by the terminal node name or the index number.

**Exception:** Currents through probes take the name of the probe.

<table>
<thead>
<tr>
<th><code>save</code> Statement</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>save 7</code></td>
<td>Saves voltage for a node named 7.</td>
</tr>
<tr>
<td><code>save Q4:currents</code></td>
<td>Saves all terminal currents associated with component Q4.</td>
</tr>
<tr>
<td><code>save Q4:c:static</code></td>
<td>Saves resistive terminal currents associated with component Q4.</td>
</tr>
<tr>
<td><code>save D8:cap</code></td>
<td>Saves the junction capacitance for component D8. (Assumes D8 is a diode, and, therefore, cap is an operating-point parameter.)</td>
</tr>
<tr>
<td><code>save Q5 D9:oppoint</code></td>
<td>Saves all signal information for component Q5 and the operating-point parameters for component D9.</td>
</tr>
<tr>
<td><code>save Q1:c</code></td>
<td>Saves the collector current for component Q1. (Example assumes Q1 is a BJT, and, therefore, c is a terminal name.)</td>
</tr>
<tr>
<td><code>save Q1:1</code></td>
<td>Same effect as the previous statement. Saves the collector current for component Q1. Identifies the terminal with its terminal index instead of its terminal name.</td>
</tr>
<tr>
<td><code>save M2:d:displacement</code></td>
<td>Saves capacitive current associated with the drain terminal of component M2. (Example assumes M2 is a MOSFET, so d is a terminal name.)</td>
</tr>
<tr>
<td><code>save Q3:currents M1:all</code></td>
<td>Saves all currents for component Q3 and all signals for component M1.</td>
</tr>
</tbody>
</table>
Saving Individual Currents with Current Probes

A current probe is a component that measures the current passing between two nodes. Its effect is like placing an amp meter on two points of a circuit. It creates a new branch in the circuit between the two nodes, forces the voltages on the two nodes to be equal, and then measures the flow of current.

Important

Adding probes to circuits that are sensitive to numerical noise might affect the solution. In such cases, an accurate solution might be obtained by reducing reltol.

When to Use Current Probes

Sometimes it is preferable or necessary to use a current probe rather than a `save` statement. You use a probe instead of a `save` statement under the following circumstances:

- If you want increased flexibility for giving currents descriptive names
  
  With a current probe, you can name the current anything you want. With a `save` statement, the name of the current must have a :name suffix.

- If you are saving measurements for current-controlled components

- If you are saving currents for an AC analysis

- If you are saving measurements for a current that passes between two parts of a circuit but not through a terminal

  In the following example, you insert a current probe to measure the current flowing between A and B. Because there is no component between A and B, there is no other
way to measure this current except to insert a current probe that has an identical current to the one you want to measure.

![Diagram of current probe from A to B]

**Probe Statement Example**

To specify a current probe, place a statement with the following syntax in your netlist:

```
<Name> <in> <out> iprobe
```

- `<Name>` is the unique netlist name for the current probe component. The measured current also receives this name.
- `<in>` is the name you choose for the input node of the probe.
- `<out>` is the name you choose for the output node of the probe.
- The keyword `iprobe` is the primitive name of the component.

In the following example, the current probe measures the current between nodes `src` and `in`, names the measured current `Iin`, and saves `Iin` to the raw file.

```
Iin src in iprobe
```

**Note:** You can also direct the Spectre simulator to save currents with probes with a `save` statement option. For further information, see the description of `save statement keywords` in this chapter.

**Saving Power**

To save power dissipated on a circuit, subcircuit, or device, you use the `pwr` parameter. Power is calculated only during DC and transient analyses. The results are saved as a waveform, representing the instantaneous power dissipated in the circuit, subcircuit, or device.

**Formatting the pwr Parameter**

The syntax for the `pwr` parameter is illustrated by the following examples.
To save the power dissipated on a device or instance of a subcircuit, the syntax is

\[ \text{save } \text{instance} \text{\textunderscore name}:\text{pwr} \]

To save the total power, the syntax is

\[ \text{save :pwr} \]

You can explicitly save particular power variables. For example:

\[ \text{save :pwr } x1:pwr x1.x2.m1:pwr \]

This statement saves three power signals:

- The total power dissipated (:pwr)
- The power dissipated in the x1 subcircuit instance (x1:pwr)
- The power dissipated in the x1.x2.m1 MOSFET

**Power Options**

The \text{pwr} parameter in the \text{options} statement can also be used to save power. The following table shows the five possible settings for the \text{pwr} option:

<table>
<thead>
<tr>
<th>Setting</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>all</td>
<td>The total power, the power dissipated in each subcircuit, and the power dissipated in each device is saved.</td>
</tr>
<tr>
<td>subckts</td>
<td>The total power and the power dissipated in each subcircuit is saved.</td>
</tr>
<tr>
<td>devices</td>
<td>The total power and the power dissipated in each device is saved.</td>
</tr>
<tr>
<td>total</td>
<td>The total power dissipated in the circuit is calculated and saved.</td>
</tr>
<tr>
<td>none</td>
<td>No power variable is calculated or saved. This is the default setting.</td>
</tr>
</tbody>
</table>

For example:

```plaintext
opts options pwr=total
save x1:pwr
```

This creates two power signals, \text{:pwr} (generated by the \text{options} statement) and \text{x1:pwr} (generated by the \text{save} statement).
Saving Groups of Signals

To save groups of signals as results, you use the save and nestlvl parameters. You specify which signals you want to save with the save parameter. You use the nestlvl parameter when you save signals in subcircuits. The nestlvl parameter specifies how many levels deep into the subcircuit hierarchy you want to save signals.

You can set these parameters as follows:

- In options statements or set statements
  - If you set the save and nestlvl parameters with an options or a set statement, the setting applies to signal data from all analyses that follow that statement in the netlist.

- In most analysis statements
  - If you set the save and nestlvl parameters with an analysis statement, the setting applies to that analysis only. It overrides any previous save or nestlvl settings.

Formatting the save and nestlvl Parameters

The syntax for both the save and nestlvl parameters is illustrated by the following options statement:

```
setting1 options save=lvlpub nestlvl=2
```

The save Parameter Options

The following table shows the possible settings for the save parameter:

<table>
<thead>
<tr>
<th>Setting</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>none</td>
<td>Does not save any data (currently does save one node chosen at random).</td>
</tr>
<tr>
<td>selected</td>
<td>Saves only signals specified with save statements. This is the default setting.</td>
</tr>
<tr>
<td>lvlpub</td>
<td>Saves all signals that are normally useful up to nestlvl deep in the subcircuit hierarchy. This option is equivalent to allpub for subcircuits. Normally useful signals include shared node voltages and currents through voltage sources and iprobes.</td>
</tr>
<tr>
<td>lvl</td>
<td>Saves all signals up to nestlvl deep in the subcircuit hierarchy. This option is relevant for subcircuits.</td>
</tr>
</tbody>
</table>
Use `lvl` or `all` (instead of `lvlpub` or `allpub`) to include internal node voltages and currents through other components that compute current.

Use `lvlpub` or `allpub` to exclude signals at internal nodes on devices (the internal collector, base, emitter on a BJT, the internal drain and source on a FET, etc). `lvlpub` and `allpub` also exclude the currents through inductors, controlled sources, transmission lines, transformers, etc.

**Note:** Setting the `save` parameter value to `selected` without any `save` statements in the netlist is not equivalent to specifying no output. Currently, the Spectre simulator saves all circuit nodes and branch currents with this combination of settings. This might change in future releases of the Spectre simulator.

## Saving Subcircuit Signals

To save groups of signals for subcircuits, you must adjust two parameter settings:

- Set the `save` parameter to either `lvl` or `lvlpub`.
- Set the `nestlvl` parameter to the number of levels in the hierarchy you want to save. The default setting for `nestlvl` is infinity, which saves all levels.

## Saving Groups of Currents

The `currents` parameter of the `options` statement computes and saves terminal currents. You use it to create settings for currents that apply to all terminals in the netlist.

For two-terminal components, the Spectre simulator saves only the first terminal (entering) currents. You must use a `save` statement or use the global `redundant_currents` parameter of the `options` statement to save data for the second terminal of a two-terminal component. For more information about the `save` statement, see “Saving Signals for Individual Nodes and Components” on page 213.

<table>
<thead>
<tr>
<th>Setting</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>allpub</code></td>
<td>Saves only signals that are normally useful. Normally useful signals include shared node voltages and currents through voltage sources and iprobes.</td>
</tr>
<tr>
<td><code>all</code></td>
<td>Saves all signals.</td>
</tr>
</tbody>
</table>
Setting the currents Parameter

The `currents` parameter has the following options:

<table>
<thead>
<tr>
<th>Setting</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>selected</td>
<td>Saves only currents that you specifically request with <code>save</code> statements or <code>save</code> parameters. Also saves naturally computed “branch” currents (currents through current probes, voltage sources, and inductors). This is the default setting.</td>
</tr>
<tr>
<td>nonlinear</td>
<td>Saves all terminal currents for nonlinear devices, naturally computed “branch” currents (currents through current probes, voltage sources, and inductors), and currents you specify with <code>save</code> statements. Can significantly increase simulation time.</td>
</tr>
<tr>
<td>all</td>
<td>Saves all terminal currents and currents available from <code>selected</code> settings to the raw file. Can significantly increase simulation time.</td>
</tr>
</tbody>
</table>

**Note:** Currently, if you set the `currents` parameter value to `nonlinear` or `all` and do not specify a `save` parameter value in an `options` statement, the Spectre simulator saves circuit nodes as well as the currents you requested. This might change in future releases of the Spectre simulator.

Examples of the currents Parameter

You use the following syntax for the `currents` parameter in the `options` statement. For more information about the `options` statement, see the parameter listings in the Spectre online help (`spectre -h`).

- The Spectre simulator saves all terminal currents for nonlinear components, currents specified with the `save` statement, and routinely computed currents.
  
  `opt1 options currents=nonlinear`

- The Spectre simulator saves all terminal currents.
  
  `opt2 options currents=all`

Setting Multiple Current Probes

Sometimes you might need to set a large number of current probes. This could happen, for example, if you need to save a number of ACs. (Current probes can find such small signal currents when they are not normally computed.) You can specify that all currents be calculated with current probes by placing `useprobes=yes` in an `options` statement.
Setting multiple current probes can greatly increase the DC and transient analysis simulation times. Consequently, this method is typically used only for small circuits and AC analysis.

**Important**

Adding probes to circuits that are sensitive to numerical noise might affect the solution. In such cases, an accurate solution might be obtained by reducing `reltol`.

**Saving Subcircuit Terminal Currents**

You use the `subcktprobelvl` parameter to control the calculation of terminal currents for subcircuits. Current probes are added to the terminals of each subcircuit, up to `subcktprobelvl` deep. You can then save these terminal currents by setting the `save` parameter. The `nestlvl` parameter controls how many levels are returned.

**Saving All AHDL Variables**

If you want to save all the ahdl variables belonging to all the ahdl instances in the design, set the `saveahdlvars` option to `all` using a Spectre `options` command. For example:

```
Saveahdl options saveahdlvars=all
```

**Listing Parameter Values as Output**

You can generate lists of component parameter values with the `info` statement. With this statement, you can access the values of input, output, and operating-point parameters. These parameter types are defined as follows:

- **Input parameters**
  
  Input parameters are those you specify in the netlist, such as the given length of a MOSFET or the saturation current of a bipolar resistor.

- **Output parameters**
  
  Output parameters are those the simulator computes, such as temperature-dependent parameters and the effective length of a MOSFET after scaling.

- **Operating-point parameters**
  
  Operating-point parameters are those that depend on the operating point.
You can also list the minimum and maximum values for the input, output, and operating-point parameters, along with the names of the components that have those values.

**Specifying the Parameters You Want to Save**

You specify parameters you want to save with the `info` statement `what` parameter. You can give this parameter the following settings:

<table>
<thead>
<tr>
<th>Setting</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>none</td>
<td>Lists no parameters.</td>
</tr>
<tr>
<td>inst</td>
<td>Lists input parameters for instances of all components.</td>
</tr>
<tr>
<td>models</td>
<td>Lists input parameters for models of all components.</td>
</tr>
<tr>
<td>input</td>
<td>Lists input parameters for instances and models of all components.</td>
</tr>
<tr>
<td>output</td>
<td>Lists effective and temperature-dependent parameter values.</td>
</tr>
<tr>
<td>all</td>
<td>Lists input and output parameter values.</td>
</tr>
<tr>
<td>oppoint</td>
<td>Lists operating-point parameters.</td>
</tr>
<tr>
<td>terminals</td>
<td>The output is a node-to-terminal map.</td>
</tr>
<tr>
<td>nodes</td>
<td>The output is a terminal-to-node map.</td>
</tr>
</tbody>
</table>

The `info` statement gives you some additional options. You can use the `save` parameter of the `info` statement to specify groups of signals whose values you want to list. For more information about `save` parameter options, consult “Saving Groups of Signals” on page 219. Finally, you can generate a summary of maximum and minimum parameter values with the `extremes` option.

**Specifying the Output Destination**

You can choose among several output destination options for the parameters you list with the `info` statement. With the `info` statement `where` parameter, you can

- Display the parameters on a screen
- Send the parameters to a log file, to the raw file, or to a file you create
Examples of the info Statement

You format the info statement as follows:

\(<\text{StatementName}\> \text{ info } <\text{parameter}=\text{value}...>\)

The following example tells the Spectre simulator to send the maximum and minimum input parameters for all models to a log file:

Inparams info what=models where=logfile extremes=only

For a complete description of the parameters available with the info statement, consult the lists of analysis and control statement parameters in the Spectre online help (spectre -h).

Preparing Output for Viewing

In this section, you will learn how to format output data files so you can view them with a postprocessor.

Output Formats Supported by the Spectre Simulator

You can choose from among seven format settings for output data files or directories. The default setting is psfbin.

<table>
<thead>
<tr>
<th>Option</th>
<th>Format</th>
</tr>
</thead>
<tbody>
<tr>
<td>wsfbin</td>
<td>Cadence® waveform storage format (WSF)—binary</td>
</tr>
<tr>
<td>wsfascii</td>
<td>Cadence waveform storage format (WSF)—ASCII</td>
</tr>
<tr>
<td>psfbin</td>
<td>Cadence parameter storage format (PSF)—binary</td>
</tr>
<tr>
<td>psfascii</td>
<td>Cadence parameter storage format (PSF)—ASCII</td>
</tr>
<tr>
<td>nutbin</td>
<td>Nutmeg—binary (SPICE3 standard output)</td>
</tr>
<tr>
<td>nutascii</td>
<td>Nutmeg—ASCII (SPICE3 standard output)</td>
</tr>
<tr>
<td>sst2</td>
<td>Cadence Signal Scan format</td>
</tr>
</tbody>
</table>

In the Nutmeg format, the Spectre simulator writes output data to a raw file. For further information about the Nutmeg format, consult the Nutmeg Users’ Manual (available from the University of California, Berkeley). In the other formats, the Spectre simulator creates and writes output to a directory of files. The Spectre simulator overwrites existing files and
directories. If the Spectre simulator cannot open files or create the necessary directories, it stops.

**Defining Output File Formats**

You can redefine the output file format in two ways:

- With a `spectre` command you type from the command line or place in an environment variable
- With an `options` statement you put in the netlist

You use the `options` statement in the netlist to override an environment default setting, and you use the `spectre` command at run time to override any settings in the netlist. The parameter values you enter are the same for either method.

**Example Using the `spectre` Command**

The following example shows you how to set format options with the `spectre` command. This statement, which you type at the command line or place in an environment variable, directs the Spectre simulator to run a simulation on a circuit named `circuitfile` and format the results in binary Nutmeg.

```
spectre -format nutbin circuitfile
```

**Example Using the `options` Statement**

You set format options with the `rawfmt` parameter of the `options` statement as shown in the following example. This statement, which you place in the netlist, instructs the Spectre simulator to create an output file in binary Nutmeg. For more information about the `options` statement, see the parameter listings in the Spectre online help (`spectre -h`).

```
Settings options rawfmt=nutbin
```

**Accessing Output Files**

After you run a simulation, you will want access to results of the various analyses you specified. To access these results, you need to know the names of the files and directories where the Spectre simulator stores these results. In this section, you will learn how the Spectre simulator names its output directories and files and how you can change these naming conventions to fit your needs. The information in this section applies to `psf` and `wsf` formats.
How the Spectre Simulator Creates Names for Output Directories and Files

When you create a netlist, you give the netlist a filename. When you simulate the circuit, the Spectre simulator adds the suffix `.raw` to this filename to create the name of the output directory for the simulation. For example, results from the simulation of a file named `test` are stored in a directory named `test.raw`.

In the output directory, results from each analysis you specify are stored in separate files. The root of each filename is the name you gave the analysis in the netlist, and the suffix for each filename is the type of analysis you specified. For example, if you run the following analysis, the Spectre simulator stores the results in a file named `Sparams.sp`:

```
Sparams sp start=100M stop=100G dec=100
```

For the `sweep` and `montecarlo` analyses, the names of the filenames are a concatenation of the parent analysis name, the iteration number, and the child analysis name. For example, if you run:

```
sweep1 sweep param=temp values=[-25 50]{
  dc0p dc
}
```

creates `sweep1_000_dc0p.dc` and `sweep1_001_dc0p.dc`.

The following example contains a number of analysis statements from a netlist. It shows the name of the output file the Spectre simulator creates for each analysis. In some cases, the Spectre simulator creates more than one output file for an analysis. This is because the analysis statement contains a parameter that specifies that certain output information be sent to a file.
**Results File** | **Description**
--- | ---
logFile | Log file (identifies output file format)
OpPoint.op | Nonlinear device DC operating-point file
OpPoint.dc | Node voltages at the operating point
Drift.dc | DC sweep
XferVsTemp.xf | Transfer function versus temperature
OpenLoop.ac | AC analysis
XferVsFreq.xf | Transfer function versus frequency
StepResponse.tran | Transient analysis
StepResponse.op | Operating-point information for transient analysis
SineResponse.tran | Transient analysis
Filenames for SPICE Input Files

If you specify analyses with standard SPICE syntax, the name of the output file for the first instance of each type of analysis is the same as the name shown in the following table:

<table>
<thead>
<tr>
<th>SPICE Analysis</th>
<th>Output Filename</th>
</tr>
</thead>
<tbody>
<tr>
<td>.OP</td>
<td>opBegin.dc</td>
</tr>
<tr>
<td>.AC</td>
<td>frequencySweep.ac</td>
</tr>
<tr>
<td>.DC</td>
<td>srcSweep.dc</td>
</tr>
<tr>
<td>.TRAN</td>
<td>timeSweep.tran</td>
</tr>
</tbody>
</table>

Note: These names have special significance to the Cadence analog design environment.

If you request multiple unnamed analyses using SPICE syntax, the names are constructed by appending a sequence integer to the name of the analysis type and further adding the dot extension that is appropriate for the analysis. For example, multiple AC analyses would generate this sequence of files: frequencySweep.ac, ac2.ac, ac3.ac, ac4.ac, and so on.

Specifying Your Own Names for Directories

You might want to specify a name for an output directory that is different from the name of your netlist. (For example, if you use the same netlist in more than one simulation, you probably want different names for the output files.) You can specify names in two ways:

- You can specify your directory name from the command line or in an environment variable with the `spectre` command `-raw` option. For example, if you want your output directory to be named `test.circuit.raw`, you start your simulation as follows:

```
spectre -raw test.circuit <inputFilename>
```

- You can set the `rawfile` parameter of the `options` statement. For example, the following `options` statement creates an output directory named `test.circuit.raw`:

```
Setup options rawfile="test.circuit"
```

Note: The Spectre simulator has some additional features that help you manage data by letting you systematically specify or modify filenames. For more information about these features, see Chapter 11, “Managing Files.”
Running a Simulation

This chapter discusses the following topics:

- Starting Simulations on page 229
- Checking Simulation Status on page 230
- Interrupting a Simulation on page 231
- Recovering from Transient Analysis Terminations on page 231
- Controlling Command Line Defaults on page 233

Starting Simulations

To start a simulation, you type the `spectre` command at the command line with the following syntax:

```
spectre <options> <inputfile>
```

The `spectre` command starts a simulation of `inputfile`. The simulation includes any options you request. For a given simulation, the `spectre` command options override any settings in default environment variables or `options` statement specifications.

The simplest example is to simulate a circuit with no run options. The following example starts a simulation of the input file `test1`.

```
spectre test1
```

Specifying Simulation Options

Many simulation runs require more complicated instructions than the previous example. Spectre® circuit-simulator-run options can be specified in two ways. Which method you use depends on the run option.
You specify some Spectre options by typing a minus (–) in front of the option. The (–V) in the following example specifies that version information be printed for the simulation of circuit test1.

```
spectre -V test1
```

You activate some Spectre options by typing a plus (+) before the option. You deactivate these options by typing a minus (–) before the option. For example, the following `spectre` command starts a simulation run for circuit test1. In this simulation, the Spectre simulator sets checkpoints but does not print error messages:

```
spectre +checkpoint -error test1
```

Some Spectre options have abbreviations. You can find these abbreviations in Chapter 2, “Spectre Command Options,” of the Spectre Circuit Simulator Reference manual. For example, you can type the previous command as follows:

```
spectre +cp -error test1
```

Specific `spectre` command options are discussed throughout this guide. For a complete list of options and formats, consult the Spectre Circuit Simulator Reference manual.

**Determining Whether a Simulation Was Successful**

When the Spectre simulator finishes a simulation, it sets the shell status variable to one of the following values:

- **0**: If the Spectre simulator completed the simulation normally
- **1**: If the Spectre simulator stopped any analysis because of an error
- **2**: If the Spectre simulator stopped the simulation early because of a Spectre error condition
- **3**: If a Spectre simulation was stopped by you or by the operating system

**Checking Simulation Status**

If you want to check the status of a simulation during a run, type the following UNIX command:

```
% kill -USR1 <PID>
```

`PID` is the Spectre process identification number, which you can find by activating the UNIX `ps` utility.

The Spectre simulator displays the status information on the screen or sends it to standard output if it cannot write to the screen. If you check the status from a remote terminal, the...
Spectre Circuit Simulator User Guide
Running a Simulation

Spectre simulator also writes the status to the SpectreStatus file in the directory from which the Spectre simulator was called. The Spectre simulator deletes this file at termination of the run.

**Note:** You can also give Spectre netlist instructions to display some status information. For more information, consult the Spectre online help about the sweep and steps options for the annotate parameter. You can set the annotate parameter for most Spectre analyses.

**Interrupting a Simulation**

If you want to stop the Spectre simulator while a simulation is running, do one of the following:

- Send an INT signal with your interrupt character.
  
  The interrupt character is usually Control-c. You can get information about the interrupt character for your system with the UNIX stty utility.

- Send the INT signal with a `kill(1)` command.

When you use either of these commands, the Spectre simulator prepares the incomplete output data file for reading by the postprocessor and then stops the simulation.

**Caution**

*Do not stop the Spectre simulator with a `kill -9` command. This command stops the simulation before the Spectre simulator can prepare the output files for reading by the postprocessor.*

**Recovering from Transient Analysis Terminations**

If a transient analysis ends before a successful conclusion, you can recover the work that is completed and restart the analysis. The Spectre simulator needs checkpoint files to perform this recovery. This section tells you about the following ways to create checkpoint files:

- Automatically, with defaults and options statement settings
- From the command line during a simulation
- For specific analyses with netlist instructions

This section also tells you how to restart a simulation after a transient analysis termination.
Customizing Automatic Recovery

By default, the Spectre simulator creates checkpoint files every 30 minutes during a transient analysis. The Spectre simulator deletes these checkpoint files when the simulation ends successfully.

Reactivating Automatic Recovery for a Single Simulation

If you have deactivated the default setting (by putting the -checkpoint setting in an environment variable), you can reactivate the default value for a given simulation run with the following procedure:

➤ Type +checkpoint as a command line argument to the spectre command that starts the simulation.

Determining How Often the Spectre Simulator Creates Recovery Files

If you want to change how often the Spectre simulator creates checkpoint files for a particular simulation, or if you want your checkpoint files saved after a successful completion, you should set the ckptclock parameter of the options statement. For more information about the options statement, consult the parameter listings in the Spectre online help (spectre -h).

The following options statement tells the Spectre simulator to create checkpoint files every 3 1/2 minutes for all transient analyses in a simulation. (You indicate parameters for ckptclock in seconds.)

SetCkptInterval options ckptclock=210

Creating Recovery Files from the Command Line

You can create a checkpoint file when a transient analysis is running with a UNIX interrupt signal from the command line.

➤ To create a checkpoint file from the command line, send a USR2 signal with a UNIX kill command.

```
kill -USR2 <PID>
```

(You find the necessary process identification number by running the UNIX ps utility.)

The Spectre simulator also attempts to write a checkpoint file after QUIT, TERM, INT, or HUP interrupts. After other interrupt signals, the Spectre simulator might be unable to write a checkpoint file.
Setting Recovery File Specifications for a Single Analysis

When you specify a transient analysis, you can also create periodic checkpoint files for that analysis.

➤ To create periodic checkpoint files for a transient analysis, set the `ckptperiod` parameter in the transient analysis statement.

The following example creates a checkpoint every 20 seconds during the transient analysis `SineResponse`:

```
SineResponse tran stop=150u ckptperiod=20
```

Restarting a Transient Analysis

➤ To restart a transient analysis from the last checkpoint file, start the simulation again with the `spectre` command. Make sure to enter the `+recover` command line argument.

Controlling Command Line Defaults

There are many run options you can specify with either the `spectre` command or the `options` statement. The Spectre simulator provides defaults for many of these options, so you can avoid the inconvenience of specifying many options for each simulation. Spectre defaults are satisfactory for most situations, but, if you have specialized needs, you can also set your own defaults. Spectre command line defaults control the following general areas:

- Messages from the Spectre simulator
- Destination and format of results
- Default values for the C preprocessor
- Default values for percent codes
- Creating checkpoints and initiating recoveries
- Screen display
- Name of the simulator
- Simulation environment (such as `opus`, `edge`, and so on)
Examining the Spectre Simulator Defaults

You can identify the various Spectre defaults by consulting the detailed description of `spectre` command options in the *Spectre Circuit Simulator Reference* manual.

Setting Your Own Defaults

You can set your own defaults by setting the UNIX environment variables `%S_DEFAULTS` or `SPECTRE_DEFAULTS`. In `%S_DEFAULTS`, `%S` is replaced by the name of the simulator, so this variable is typically `SPECTRE_DEFAULTS`. However, the name created by the `%S` substitution is different if you move the executable to a file with a different name or if you call the program with a symbolic or hard link with a different name. Consequently, you can create multiple sets of defaults, which you identify with different `%S` substitutions. Initially, the Spectre simulator looks for defaults settings in `%S_DEFAULTS`. If this variable does not exist, it looks for default settings in `SPECTRE_DEFAULTS`.

To set these environment variables, use the following procedure.

1. In an appropriate file, such as `.cshrc` or `.login`, use the appropriate UNIX command to create settings for the environment variables `%S_DEFAULTS` or `SPECTRE_DEFAULTS`. Format the new default settings like `spectre` command line arguments and place them in quotation marks.

   The following example changes the default output format from `psfbin` to `wsfbin`. It also sets an option that is normally deactivated. It sends all messages from the Spectre simulator to a `%C:r.log` file.

   For *csh*:
   ```
   setenv SPECTRE_DEFAULTS " -format wsfbin +log %C:r.log "
   ```

   For *sh* or *ksh*:
   ```
   SPECTRE_DEFAULTS=" -format wsfbin +log %C:r.log "
   EXPORT SPECTRE_DEFAULTS
   ```

   This second example, a typical use of the `SPECTRE_DEFAULTS` environment variable, tells the Spectre simulator to do the following:

   - Write a log file named `cktfile.out`, where `cktfile` is the name of the input file minus any dot extension.
   - Use the parameter soft limits file in the Cadence® software hierarchy.

   ```
   setenv SPECTRE_DEFAULTS "+log %C:r.out \
   +param /cds/etc/spectre/range.lmts"
   ```
You can use the default settings to specify alternative conditions for running the Spectre simulator. Suppose you create the following environment variables:

setenv SPECTRE_DEFAULTS "+param range.lmts +log %C:r.o -E"
setenv SPECSIM_DEFAULTS "+param corner.lmts =log %C:r.log -f psfbin"

If `spectre` and `specsim` are both links to the Spectre executable, and you run the executable as `spectre`, the Spectre simulator does the following:

- Reads the file `range.lmts` for the parameter limits
- Directs all messages to the screen and to a log file named after the circuit file with `.o` appended (see Chapter 11, “Managing Files,” for more information about specifying filenames)
- Runs the C preprocessor

Running the executable as `specsim` causes the Spectre simulator to select a different set of defaults and to do the following:

- Read the range limits from the file `corner.lmts`
- Direct log messages to a file named after the circuit file with `.log` appended (the `=log` specification suppresses the log output to the screen)
- Format output files in binary parameter storage format (PSF)

**References for Additional Information about Specific Defaults**

In some cases, you need to consult other sections of this book before you can set defaults.

- If you want to set default limits for warning messages about parameter values, consult Chapter 12, “Identifying Problems and Troubleshooting,” to find information about installing Cadence defaults or creating your own range limits file if you need to customize defaults.

- You can find additional information about percent code defaults in “Description of Spectre Predefined Percent Codes” on page 248.

**Overriding Defaults**

You can override defaults in the UNIX environment variables for a given simulation with either `spectre` command line arguments or `options` statement specifications. The `spectre` command line arguments also override `options` statement specifications.
Time-Saving Techniques

In this chapter, you will learn about different methods to reduce simulation time. This chapter discusses the following topics:

- Specifying Efficient Starting Points on page 236
- Reducing the Number of Simulation Runs on page 236
- Adjusting Speed and Accuracy on page 237
- Saving Time by Starting Analyses from Previous Solutions on page 237
- Saving Time by Specifying State Information on page 237
- Saving Time by Modifying Parameters during a Simulation on page 242
- Saving Time by Selecting a Continuation Method on page 245

Specifying Efficient Starting Points

The Spectre® circuit simulator arrives at a solution for a simulation by calculating successively more accurate estimates of the final result. You can increase simulation speed by providing information that the Spectre simulator uses to increase the accuracy of the initial solution estimate. There are three ways to provide a good starting point for a simulation:

- Start analyses from previous solutions
- Specify initial conditions for components and nodes
- Specify solution estimates with nodesets

Reducing the Number of Simulation Runs

With the Spectre simulator, you can run many analyses (including analyses of the same type) in a single simulation. With other SPICE-like simulators, you might require multiple simulations to complete the same tasks. In a single simulation run, you can run a set of
Spectre analyses; modify the component, temperature, or options parameters; and then run additional analyses with the new parameters.

**Adjusting Speed and Accuracy**

You can use the `errpreset` parameter to increase the speed of transient analyses, but this speed increase requires some sacrifice of accuracy.

**Saving Time by Starting Analyses from Previous Solutions**

A solution for one analysis can be an appropriate starting point for the next analysis. For example, if a DC analysis precedes a transient analysis, you can use the DC solution as the first guess for the initial point in the transient analysis solution. There are two Spectre analysis parameters that let you start analyses from previous solutions. They are available for most Spectre analyses.

- **The `restart` parameter**

  If you set this parameter to `restart=no` in an analysis statement, the Spectre simulator uses the DC solution of the previous analysis as an initial guess for the following analysis.

- **The `prevoppoint` parameter**

  If you set this parameter to `prevoppoint=yes` in an analysis statement, the Spectre simulator does not compute or recompute the operating point. Instead, it uses the operating point computed by the previous analysis.

**Saving Time by Specifying State Information**

The Spectre simulator lets you provide state information (the current or last-known status or condition of a process, transaction, or setting) to the DC and transient analyses. You can specify two kinds of state information:

- **Initial conditions**

  The `ic` statement lets you specify values for the starting point of a transient analysis. The values you can specify are voltages on nodes and capacitors, and currents on inductors.

- **Nodesets**
Nodesets are estimates of the solution you provide for the DC or transient analyses. Unlike initial conditions, their values have no effect on the final results. Nodesets usually act only as aids in speeding convergence, but if a circuit has more than one solution, as with a latch, nodesets can bias the solution to the one closest to the nodeset values.

**Setting Initial Conditions for All Transient Analyses**

You can specify initial conditions that apply to all transient analyses in a simulation or to a single transient analysis. The ic statement and the ic parameter described in this section set initial conditions for all transient analyses in the netlist. In general, you use the ic parameter of individual components to specify initial conditions for those components, and you use the ic statement to specify initial conditions for nodes. You can specify initial conditions for inductors with either method. Specifying cmin for a transient analysis does not satisfy the condition that a node has a capacitive path to ground.

**Note:** Do not confuse the ic parameter for individual components with the ic parameter of the transient analysis. The latter lets you select from among different initial conditions specifications for a given transient analysis.

**Specifying Initial Conditions for Components**

You can specify initial conditions in the instance statements of capacitors, inductors, and windings for magnetic cores. The ic parameter specifies initial voltage values for capacitors and current values for inductors and windings. In the following example, the initial condition voltage on capacitor Cap13 is set to two volts:

```
Cap13 11 9 capacitor c=10n ic=2
```

**Specifying Initial Conditions for Nodes**

You use the ic statement to specify initial conditions for nodes or initial currents for inductors. The nodes can be inside a subcircuit or internal nodes to a component.

The following is the format for the ic statement:

```
ic <signalName=value> …
```

The format for specifying signals with the ic statement is similar to that used by the save statement. This method is described in detail in “Saving Main Circuit Signals” on page 213. Consult this discussion if you need further clarification about the following example.

```
ic Voff=0 X3.7=2.5 M1:int_d=3.5 L1:1=1u
```

This example sets the following initial conditions:
The voltage of node $V_{off}$ is set to 0.

Node 7 of subcircuit X3 is set to 2.5 V.

The internal drain node of component M1 is set to 3.5 V. (See the following table for more information about specifying internal nodes.)

The current for inductor $L_1$ is set to 1$\mu$.

Specifying initial node voltages requires some additional discussion. The following table tells you the internal voltages you can specify with different components.

<table>
<thead>
<tr>
<th>Component</th>
<th>Internal Node Specifications</th>
</tr>
</thead>
<tbody>
<tr>
<td>BJT</td>
<td>int_c, int_b, int_e</td>
</tr>
<tr>
<td>BSIM</td>
<td>int_d, int_s</td>
</tr>
<tr>
<td>MOSFET</td>
<td>int_d, int_s</td>
</tr>
<tr>
<td>GaAs MESFET</td>
<td>int_d, int_s, int_g</td>
</tr>
<tr>
<td>JFET</td>
<td>int_d, int_s, int_g, int_b</td>
</tr>
<tr>
<td>Winding for Magnetic Core</td>
<td>int_{Rw}</td>
</tr>
<tr>
<td>Magnetic Core with Hysteresis</td>
<td>flux</td>
</tr>
</tbody>
</table>

**Supplying Solution Estimates to Increase Speed**

You use the `nodeset` statement to supply estimates of solutions that aid convergence or bias the simulation towards a given solution. You can use nodesets for all DC and initial transient analysis solutions in the netlist. The `nodeset` statement has the following format:

```
nodeset <signalName=value>...
```

Values you can supply with the `nodeset` statement include voltages on topological nodes, including internal nodes, and currents through voltage sources, inductors, switches, transformers, N-ports, and transmission lines.

The format for specifying signals with the `nodeset` statement is similar to that used by the `save` statement. This method is described in detail in “Saving Main Circuit Signals” on page 213. Consult this discussion if you need further clarification about the following example.

```
nodeset Voff=0 X3.7=2.5 M1:int_d=3.5 L1:1=1u
```

This example sets the following solution estimates:
The voltage of node $V_{off}$ is set to 0.

Node 7 of subcircuit X3 is set to 2.5 V.

The internal drain node of component M1 is set to 3.5 V. (See the table in the ic statements section of this chapter for more information about specifying internal nodes.)

The current for inductor $L_1$ is set to 1 $\mu$A.

**Specifying State Information for Individual Analyses**

You can specify state information for individual analyses in two ways:

- You can use the ic parameter of the transient analysis to choose which previous specifications are used.
- You can create a state file that is read by an individual analysis.

**Choosing Which Initial Conditions Specifications Are Used for a Transient Analysis**

The ic parameter in the transient analysis lets you select among several options for which initial conditions to use. You can choose the following settings:

<table>
<thead>
<tr>
<th>Parameter Setting</th>
<th>Action Taken</th>
</tr>
</thead>
<tbody>
<tr>
<td>dc</td>
<td>Initial conditions specifiers are ignored, and the existing DC solution is used.</td>
</tr>
<tr>
<td>node</td>
<td>The ic statements are used, and the ic parameter settings on the capacitors and inductors are ignored.</td>
</tr>
<tr>
<td>dev</td>
<td>The ic parameter settings on the capacitors and inductors are used, and the ic statements are ignored.</td>
</tr>
<tr>
<td>all</td>
<td>Both the ic statements and the ic parameters are used. If specifications conflict, ic parameters override ic statements.</td>
</tr>
</tbody>
</table>

**Specifying State Information with State Files**

You can also specify initial conditions and estimate solutions by creating a state file that is read by the appropriate analysis. You can create a state file in two ways:

- You can instruct the Spectre simulator to create a state file in a previous analysis for future use.
You can create a state file manually in a text editor.

**Telling the Spectre Simulator to Create a State File**

You can instruct the Spectre simulator to create a state file from either the initial point or the final point in an analysis. To write a state file from the initial point in an analysis, use the `write` parameter. To write a state file from the final point, use the `writefinal` parameter. Each of the following two examples writes a state file named `ua741.dc`. The first example writes the state file from the initial point in the DC sweep, and the second example writes the state file from the final point in the DC sweep.

```
Drift dc param=temp start=0 stop=50.0 step=1 \
   readns="ua741.dc" write="ua741.dc"
Drift dc param=temp start=0 stop=50.0 step=1 \
   readns="ua741.dc" writefinal="ua741.dc"
```

**Creating a State File Manually**

The syntax for creating a state file in a text editor is simple. Each line contains a signal name and a signal value. Anything after a pound sign (`#`) is ignored as a comment. The following is an example of a simple state file:

```
# State file generated by Spectre from circuit file 'wilson'
# during 'stepresponse' at 5:39:38 PM, jan 21, 1992.
1  .588793510612534
2  1.17406247989272
3  14.9900516233357
pwr  15
vcc:p -9.9483766642647e-06
```

**Reading State Files**

To read a state file as an initial condition, use the `read` transient analysis parameter. To read a state file as a nodeset, use the `readns` parameter. This example reads the file `intCond` as initial conditions:

```
DoTran_z12 tran start=0 stop=0.003 \
   step=0.00015 maxstep=6e-06 read="intCond"
```

This second example reads the file `soruEst` as a nodeset.

```
DoTran_z12 tran start=0 stop=0.003 \
   step=0.00015 maxstep=6e-06 readns="soruEst"
```

**Special Uses for State Files**

State files can be useful for the following reasons:
You can save state files and use them in later simulations. For example, you can save the solution at the final point of a transient analysis and then continue the analysis in a later simulation by using the state file as the starting point for another transient analysis.

You can use state files to create automatic updates of initial conditions and nodesets.

The following example demonstrates the usefulness of state files:

```
altTemp alter param=temp value=0
Drift dc param=temp start=0 stop=50.0 step=1 \  
  readns="ua741.dc0" write="ua741.dc0"
XferVsTemp xf param=temp start=0 stop=50 step=1 \  
  probe=Rload freq=1kHz readns="ua741.dc0"
```

The first analysis computes the DC solution at T=0°C, saves it to a file called ua741.dc0, and then sweeps the temperature to T=50°C. The transfer function analysis (xf) resets the temperature to zero. Because of the temperature change, the DC solution must be recomputed. Without the use of state files, this computation might slow the simulation because the only available estimate of the DC solution would be that computed at T=50°C, the final point in the DC sweep. However, by using a state file to preserve the initial DC solution at T=0°C, you can enable the Spectre simulator to compute the new DC solution quickly. The computation is fast because the Spectre simulator can use the DC solution computed at T=0°C to estimate the new solution. You can also make future simulations of this circuit start quickly by using the state file to estimate the DC solution. Even if you have altered a circuit, it is usually faster to start the DC analysis from a previous solution than to start from the beginning.

### Saving Time by Modifying Parameters during a Simulation

The Spectre simulator lets you place specifications in the netlist to modify parameters and then resimulate. This lets you accomplish tasks with a single Spectre run that might require multiple runs with another simulator. To change parameter settings during a run, you use the following Spectre control statements:

- **The `alter` statement**

  You use this statement to change the parameters of circuits, subcircuits, and individual models or components. You also use it to change the following options statement temperature parameters and scaling factors:

  - `temp`
  - `tnom`
  - `scale`
You can use the `altergroup` statement to specify device, model, and netlist parameter statements that you want to change the values of with analyses.

The `set` statement

Except for temperature parameters and scaling factors, you use the `set` statement to modify any `options` statement parameters you set at the beginning of the netlist. The new settings apply to all analyses that follow the `set` statement in the netlist.

The `alter` and `set` statements are queued up with analysis statements and are processed in order.

### Changing Circuit or Component Parameter Values

You modify parameters for devices, models, circuit, and subcircuit parameters during a simulation with the `alter` statement. The modifications apply to all analyses that follow the `alter` statement in your netlist until you request another parameter modification.

### Changing Parameter Values for Components

To change a parameter value for a component device or model, you specify the device or model name, the parameter name, and the new parameter value in the `alter` statement. You can modify only one parameter with each `alter` statement, but you can put any number of `alter` statements in a netlist. The following example demonstrates `alter` statement syntax:

```
SetMag alter dev=Vt1 param=mag value=1
```

- `SetMag` is the unique netlist name for this `alter` statement. (Like many Spectre statements, each `alter` statement must have a unique name.)
- The keyword `alter` is the primitive name for the `alter` statement.
- `dev=Vt1` identifies `Vt1` as the netlist name for the component statement you want to modify. You identify an instance statement with `dev` and a model statement with `mod`. When you use the `alter` statement to modify a circuit parameter, you leave both `dev` and `mod` unspecified.
- `param=mag` identifies `mag` as the parameter you are modifying. If you omit this parameter, the Spectre simulator uses the first parameter listed for each component in the Spectre online help as the default.
- `value=1` identifies `1` as the new value for the `mag` parameter. If you leave `value` unspecified, it is set to the default for the parameter.
Changing Parameter Values for Models

To change a parameter value for model files with the `altergroup` statement, you list the device, model, and circuit parameter statements as you do in the main netlist. Within an alter group, each model is first defaulted and then the model parameters are updated. You cannot nest alter groups. You cannot change from a model to a model group and vice versa. The following example demonstrates `altergroup` statement syntax:

```plaintext
ag1 altergroup {
    parameters p1=1
    model myres resistor r1=1e3 af=p1
    model mybsim bsim3v3 lmax=p1 lmin=3.5e-7
}
```

Further Examples of Changing Component Parameter Values

This example changes the `is` parameter of a model named `SH3` to the value `1e-15`:

```plaintext
modify2 alter mod=SH3 param=is value=1e-15
```

The following examples show how to use the `param default` in an `alter` statement. The first parameter listed for resistors in the Spectre online help (`spectre -h`) is the default. For resistors, this is the resistance parameter `r`.

Consequently, if `R1` is a resistor, the following two `alter` statements are equivalent:

```plaintext
change1 alter dev=R1 param=r value=50
change1 alter dev=R1 value=50
```

Changing Parameter Values for Circuits

When you change a circuit parameter, you use the same syntax as when you change a device or model parameter except that you do not enter a `dev` or a `mod` parameter.

This example changes the ambient temperature to 0°C:

```plaintext
change2 alter param=temp value=0
```

The following table describes the circuit parameters you can change with the `alter` statement:

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>temp</code></td>
<td>Ambient temperature</td>
</tr>
<tr>
<td><code>tnom</code></td>
<td>Default measurement temperature for component parameters</td>
</tr>
</tbody>
</table>
Note: If you change temp or tnom using an alter statement, all expressions with temp or tnom are reevaluated.

Modifying Initial Settings of the State of the Simulator

You can change the initial settings for the state of the simulator by placing a set statement in the netlist. The set statement is similar to the options statement that sets the state of the simulator, but it is queued with the analysis statements in the order you place them in the netlist.

You use the set statement to change previous options or set statement specifications. The modifications apply to all analyses that follow the set statement in the netlist until you request another parameter modification. The set and options statements have many identical parameters, but the set statement cannot modify all options statement parameters. The parameter listings in the Spectre online help (spectre -h) tell you which parameters you can reset with the set statement.

Formatting the set Statement

The following example demonstrates the set statement syntax. This example turns off several annotation parameters.

```
Quiet set narrate=no error=no info=no
```

■ Quiet is the unique name you give to the set statement.
■ The keyword set is the primitive name for the set statement.
■ narrate, error, and info are the parameters you are changing.

Note: If you want to change temp or tnom, use the alter statement.

Saving Time by Selecting a Continuation Method

The Spectre simulator normally starts with an initial estimate and then tries to find the solution for a circuit using the Newton-Raphson method. If this attempt fails, the Spectre simulator automatically tries several continuation methods to find a solution and tells you which method
was successful. Continuation methods modify the circuit so that the solution is easy to compute and then gradually change the circuit back to its original form. Continuation methods are robust, but they are slower than the Newton-Raphson method.

If you need to modify and resimulate a circuit that was solved with a continuation method, you probably want to save simulation time by directly selecting the continuation method you know was previously successful.

You select the continuation method with the homotopy parameter of the set or options statements. In addition to the default setting, all, five settings are possible for this parameter: gmin stepping (gmin), source stepping (source), the pseudotransient method (ptran), and the damped pseudotransient method (dptran). You can also prevent the use of continuation methods by setting the homotopy parameter to none.
Managing Files

This chapter discusses the following topics:

- About Spectre Filename Specification on page 247
- Creating Filenames That Help You Manage Data on page 247

About Spectre Filename Specification

Many analysis statements require a filename as a parameter value for the input or output of data. It is often easier to keep track of output files if these filename parameter values are related to some other filename, typically the input filename.

The Spectre® circuit simulator’s filename specification features help you manage your data by letting you systematically specify or modify filenames. With the Spectre simulator, you can easily identify data from multiple simulation runs or from single runs containing repeated similar analyses. You can modify input filenames so that you can easily identify the output file from a specific simulation or analysis. You can also construct output filenames in ways that prevent accidental overwriting of data.

Creating Filenames That Help You Manage Data

The Spectre simulator helps you keep track of simulation data by letting you create filenames that are variants of input filenames. For example, with Spectre, you can

- Identify simulation data by date, time, process ID, or other defining characteristics in the results filenames
- Keep multiple circuits in a single directory without having subsequent simulations overwrite previous results

  To do this, you set environment variables so that output filenames are automatically different variants of input filenames.

- Construct filenames at run time
This is convenient if your input data comes from several files. For example, you can use an `include` statement to insert several different circuit files into main input files that each contain analyses. Each circuit file can also be used with several stimulus files. To prevent confusion, you can create filenames at run time for the stimulus files that associate them with the appropriate main input files.

In this section, you will learn how to use the various Spectre features for creating filenames.

### Creating Filenames by Modifying Input Filenames

The Spectre simulator gives you predefined percent codes you can put in your filenames. These predefined codes let you construct filenames that add defining characteristics, such as date or time, to input filenames. You specify predefined percent codes with a percent character (%) followed by an uppercase letter. The uppercase letter tells the Spectre simulator how to construct the filename. You can use percent codes in environment variables, in `spectre` command parameters, or in your netlist—wherever you need to specify filenames for simulation results.

For example, `%C` is the predefined percent code for the name of the input circuit file. If your circuit file is named `opamp1` and you place the following `-raw` setting for your UNIX environment variable

```
setenv SPECTRE_DEFAULTS "-raw %C.raw"
```

the Spectre simulator sends simulation results to a file named `opamp1.raw`.

### Description of Spectre Predefined Percent Codes

These are the percent code options that can help you organize your simulation data:

- `%A`  
  %A is replaced by the name of the current analysis that is running.  
  If it is specified in a device statement, it is expanded to a blank string because there is no current analysis.

- `%C`  
  %C is replaced by the input circuit filename, as it is used in the command line. If the circuit filename is `opamp1`, the specification `%C.raw` generates a file named `opamp1.raw`.

  When the Spectre simulator does not know the name of the input file, as when the circuit is read from the standard input (from a pipe or from a redirected file), the Spectre simulator substitutes the name `stdin` for `%C`. 
The predefined percent codes feature does not perform recursive substitutions. For example, if you have an input file named A%SD.xyz and you create an output file with the percent code designation %C.xyz, the Spectre simulator creates the output file A%SDxyz.xyz. The Spectre simulator does not substitute the simulator name for %S in this case.

**Customizing Percent Codes**

You can define your own percent codes or redefine existing codes with the +%(X) option of the spectre command. Names of customized percent codes can be any single uppercase or lowercase letter. You can define percent codes in two ways:

- **%D** %D is replaced by the date when the program started. For example, the specification %D.opamp1 might generate a file named 94-09-19.opamp1. The date is in year-month-day format. All leading zeros are included. This format generates filenames that you can sort alphabetically into chronological order.

- **%H** %H is replaced by the host name (network name) of the system on which the Spectre simulator is running.

- **%M** %M is replaced by the current CMIVersion.

- **%P** %P is replaced by the process ID. The process ID is a unique integer assigned to the Spectre process by the operating system.

- **%S** %S is replaced by the simulator name. For example, the specification %S.opamp1 might generate a file named spectre.opamp1. If you use a different name or a symbolic link with a different name to access a copy of the executable program, the new name becomes the program name.

- **%T** %T is replaced by the time when the program started. For example, the specification %T.opamp1 might generate a file named 14:44:07.opamp1. Time is in 24-hour format, and all leading zeros are included. This format generates filenames that sort alphabetically into chronological order.

- **%V** %V is replaced by the simulator version string. For example, the specification out_%V.raw might generate a file named out_1.0.2.raw.

- **%%** This specifies the % character by itself. This option lets you use percent characters in filenames. Two percent characters (%%) in a filename specification produce a single percent character in the filename, which is not interpreted as a percent code indicator.
You can define percent codes for a single simulation by typing this option into the command line with the `spectre` command at the start of the simulation.

You can specify the customized percent code as a default by typing the `spectre` command into the `SPECTRE_DEFAULTS` environment variables.

For example, if you type in the following instruction at the command line

```
spectre +%E opamp1 test3
```

the Spectre simulator runs a simulation for circuit `test3`. During this simulation, the Spectre simulator substitutes the name `opamp1` for any `%E` it finds in results filename specifications.

You undefine customized percent codes with the `-%<X>` `spectre` command option. For example, if you customize percent codes with the `SPECTRE_DEFAULTS` environment variable, you might want to undefine them for a given simulation run. To do this, you include the `-%<X>` command line option in the `spectre` command that starts the simulation. Undefined percent codes return to predefined values. If an undefined percent code has no predefined value, it is treated like an empty string.

### Enabling the Spectre Simulator to Recognize the Input Names of Piped or Redirected Files

One practical application for customized percent codes is to enable the Spectre simulator to recognize the input filenames of piped or redirected files. If the Spectre simulator reads the circuit from standard input, as it does when it reads from a pipe, the Spectre simulator cannot determine the name of the original input file. If you want the results filename to be a variant of the input filename, you can enable the Spectre simulator to recognize the input filename by redefining the Spectre simulator’s predefined percent codes.

In the following two examples, the circuit file is passed through `sed(1)`. The resulting file, `cktfile`, is then piped to the Spectre simulator as input. The first example shows the problem created if you want the results filename to be a modification of the input filename, and the second example shows how you can correct this difficulty.

In the first example, the Spectre simulator cannot identify the name `cktfile` of the file that is piped to the `spectre` command. As a default action, it puts the output simulation data in the directory `stdin.raw`.

```bash
setenv SPECTRE_DEFAULTS "-raw %C.raw"
sed -e 's/\$/\\$/g' cktfile | spectre
```

In the second example, redefining the `%C` percent code causes the Spectre simulator to base the output filename on the input filename. The `spectre` command in the environment variable redefines the normal predefined `%C` code. The Spectre simulator substitutes the
name cktfile for all %C specifications and puts the output simulation data in the directory cktfile.raw.

setenv SPECTRE_DEFAULTS "-raw %C.raw"

sed -e 's/$/\\$/' cktfile | spectre +%C cktfile

**Note:** For more information about Spectre defaults, see “Selecting Limits for Parameter Value Warning Messages” on page 263 and the *Spectre Circuit Simulator Reference manual*.

---

### Creating Filenames from Parts of Input Filenames

Colon modifiers (:x) create filenames from parts of input filenames. You can use colon modifiers with all percent codes except the %% code.

For example, if you apply %C:r.raw to the input filename opamp.ckt

- %C is the input filename (opamp.ckt)
- :r is the root of the input filename (opamp)
- .raw is the new filename extension

The result is the output filename opamp.raw. In this example, (:r) is the colon modifier.

---

### Definitions of Colon Modifiers

The Spectre simulator recognizes the following colon modifiers:

<table>
<thead>
<tr>
<th>Modifier</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>:r</td>
<td>Signifies the root (base name) of the given path for the file</td>
</tr>
<tr>
<td>:e</td>
<td>Signifies the extension for the given path of the file</td>
</tr>
<tr>
<td>:h</td>
<td>Signifies the head of the given path for any portion of the file before the last /</td>
</tr>
<tr>
<td>:t</td>
<td>Signifies the tail of the given path for any portion of the file after the last /</td>
</tr>
<tr>
<td>::</td>
<td>Signifies the (:) character itself; use two consecutive colons (::) to place a single colon in an output filename that is not read as a percent code modifier</td>
</tr>
</tbody>
</table>

Any character except a modifier after a colon (:) signals the end of modifications. The Spectre simulator appends both the character and the colon to the filename.
The Spectre simulator applies a chain of colon modifiers in the sequence you specify them. For example, if you apply %C:e.%C:r:t to the input filename /circuits/opamp.ckt

- %C:e is the extension (ckt) of the input filename
- %C:r is the root (opamp.ckt) of the input filename
- :t is the tail of the input filename after the last / (opamp)

The result is the output filename ckt.opamp.

**Note:** If the input filename does not contain a slash and a period, the modifiers :t and :r return the whole filename, and the modifier :h returns a period.

### Examples of Colon Modifier Use

The following table shows the various filenames you can generate from an input filename (%C) of /users/maxwell/circuits/opamp.ckt:

<table>
<thead>
<tr>
<th>Colon Modifier</th>
<th>Comments</th>
<th>Output Filename Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>%C</td>
<td>Input filename</td>
<td>/users/maxwell/circuits/opamp.ckt</td>
</tr>
<tr>
<td>%C:r</td>
<td>Root of the input filename</td>
<td>/users/maxwell/circuits/opamp</td>
</tr>
<tr>
<td>%C:e</td>
<td>Extension of the input filename</td>
<td>ckt</td>
</tr>
<tr>
<td>%C:h</td>
<td>Head of the input filename</td>
<td>/users/maxwell/circuits</td>
</tr>
<tr>
<td>%C:t</td>
<td>Tail of the input filename</td>
<td>opamp.ckt</td>
</tr>
<tr>
<td>%C::</td>
<td>Second colon is appended to the input filename and the end of the modification</td>
<td>/users/maxwell/circuits/opamp.ckt:</td>
</tr>
<tr>
<td>%C:h:h</td>
<td>The head of %C:h (such a recursive use of :h might be useful if you want to direct your output to a different directory from that of the input file)</td>
<td>/users/maxwell</td>
</tr>
<tr>
<td>%C:t:r</td>
<td>The root of %C:t</td>
<td>opamp</td>
</tr>
<tr>
<td>%C:r:t</td>
<td>The tail of %C:r</td>
<td>opamp</td>
</tr>
</tbody>
</table>
### Managing Files

<table>
<thead>
<tr>
<th>Colon Modifier</th>
<th>Comments</th>
<th>Output Filename Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>/tmp%C:t:r.raw</td>
<td>The suffix .raw is appended to the root of %C:t, and the full path is altered to put opamp.raw in the /tmp file.</td>
<td>/tmp/opamp.raw</td>
</tr>
<tr>
<td>%C:e.%C:r:t</td>
<td>Extension of %C followed by the tail of %C:r</td>
<td>ckt.opamp</td>
</tr>
</tbody>
</table>
Identifying Problems and Troubleshooting

This chapter discusses the following topics:

- **Error Conditions** on page 254
- **Spectre Warning Messages** on page 257
- **Customizing Error and Warning Messages** on page 262
- **Controlling Program-Generated Messages** on page 273
- **Correcting Convergence Problems** on page 274
- **Correcting Accuracy Problems** on page 277

**Error Conditions**

Error conditions terminate a Spectre run. If you receive any of the messages described in this section, you must fix the problem and rerun the simulation.

**Invalid Parameter Values That Terminate the Program**

If you enter a parameter that causes the Spectre simulator to stop or puts a model in an invalid region, such as giving $z_0=0$ to a transmission line, the Spectre simulator sends you a message like this one and exits.

```
Error from spectre during hierarchy flattening.
t11: Value of 'z0' should be nonzero.
spectre terminated prematurely due to fatal error.
```

To run the simulation, you must change the parameter to an acceptable value.
Singular Matrices

If you receive an error message that says a matrix is singular, your netlist contains either a floating node which is causing the problem or a loop of zero resistance branches, for example, a loop of voltage sources or inductors. The following procedures might help you find the problem:

- Check the options statement in your netlist to ensure that topcheck=yes in at least one statement. The topology checker normally helps you identify singular matrix problems, but it cannot do so if it is disabled.

- If the error message appears only for particular components or circuit parameters or only for particular voltages or currents, try one of the following procedures:
  - Set gmin=1e-12 (the default value).
  - If you are working with simplified semiconductor models, try using more complex models.

A CMOS (complementary metal oxide semiconductor) inverter whose model parameters have infinite output impedance in saturation demonstrates the usefulness of these techniques. When either the N- or P-type device is in the ohmic region, the solution is unique. However, when both devices are saturated, there is a range of output voltages that all satisfy Kirchhoff’s Current Law. In this situation, the Newton-Raphson method forms a linearized circuit that is singular for that iteration.

- Check ideal transformers, N-ports, or transmission lines for floating nodes or loops of zero-resistance branches and modify the circuit to eliminate them.

For example, consider this center-tapped transformer:

```
subckt ct_xfmr (t1 b1 t2 m2 b2)
  Tt t1 m1 t2 m2 transformer n1=2
  Tb m1 b1 m2 b2 transformer n1=2
end ct_xfmr
```
If you use this transformer and leave the center-tap terminal \((m2)\) floating, the Spectre simulator notifies you of a singular matrix. Because both \(m1\) and \(m2\) are floating, the DC solution is not unique.

If you choose a different topology for the transformer, like the one in the following example, you can avoid the problem.

```
subckt ct_xfmr (t1 b1 t2 m2 b2)
  Tt t1 b1 t2 m2 transformer n1=2
  Tb t1 b1 m2 b2 transformer n1=2
end ct_xfmr
```

Circuits that contain ideal transformers, N-ports, or transmission lines can have floating nodes or loops of zero-resistance branches because the topology checker cannot adequately verify these components. Finding these currents is difficult because all these components act like ideal transformers at DC. When you look into one port of a transformer, you can see either a short or an open circuit, depending on what you see looking out of the other port.

**Internal Error Messages**

If the Spectre simulator detects an internal error, it displays a message like one of the following:

- Error detected in file 'file.c' at line 101.
  - Arithmetic exception.

Cadence can help you find solutions to these problems. If you get one of these messages, call Cadence Customer Support or contact a Cadence application engineer.
Time Is Not Strictly Increasing

PWL takes a wave parameter that accepts time/value pairs. If the time value does not increase, the Spectre simulator displays the following message:

```
Error found in spectre during initial setup.
   v10:time is not strictly increasing in waveform.
```

Check the PWL component to fix this error.

Spectre Warning Messages

Warning messages tell you about conditions that might cause invalid results. Unlike error messages, warnings do not stop a simulation. When you receive a warning message, you must decide whether the particular condition creates a problem for your simulation. This section describes some common Spectre warning messages. It also tells you how to modify parameters to correct conditions that might produce invalid simulation results.

The Spectre simulator often prints warnings and notices that are eventually determined to be “uninteresting,” and there is a natural tendency after a while to ignore them. We recommend that you carefully study them the first few times you simulate a particular circuit and whenever the simulator gives you unexpected results.

P-N Junction Warning Messages

Almost every semiconductor device includes at least one p-n junction. Normally, these p-n junctions are biased in a particular operating region. Three types of warning messages are available for each p-n junction, one for exceeding a maximum current, one for exceeding a melting current, and one for exceeding a breakdown voltage.

Explosion Region Warnings

```
Warning from spectre at dc = 191 mA during DC analysis 'srcSweep'.
mos_mod: The bulk-drain junction current exceeds `imelt'.
The results computed by Spectre are now incorrect because the junction current model has been linearized.
```

Or

```
xram.d3247: The junction is melting (increase imax)
```

The Spectre simulator provides two parameters, imax and imelt, that limit the current across a PN junction. These parameters aid convergence and prevent numerical overflow.
The junction characteristics of the device are assumed to be accurately modeled for current up to $i_{\text{max}}$. If $i_{\text{max}}$ is exceeded during iterations, the linear model is substituted until the current drops below $i_{\text{max}}$ or until convergence is achieved. If convergence is achieved with the current exceeding $i_{\text{max}}$, the results are inaccurate, and Spectre prints a warning similar to the first one above.

The $i_{\text{melt}}$ parameter is used as a limit warning for the junction current. This parameter can be set to the maximum current rating of the device. By default it is set to the value of $i_{\text{max}}$. When any component of the junction current exceeds $i_{\text{melt}}$, Spectre issues a warning and again the results become inaccurate. The junction current is linearized above the value of $i_{\text{melt}}$ to prevent arithmetic exceptions.

Both these parameters have current density counterparts, $j_{\text{max}}$ and $j_{\text{melt}}$, that you can specify if you want the absolute current values to depend on the device area. For more information, see Spectre Circuit Simulator Known Problems and Solutions.

**Melting Current Warnings**

A separate model parameter, $i_{\text{melt}}$, is used as a limit warning for the junction current. This parameter can be set to the maximum current rating of the device. When any component of the junction current exceeds $i_{\text{melt}}$, Spectre issues a warning and the results become inaccurate. The junction current is linearized above the value of $i_{\text{melt}}$ to prevent arithmetic exception, with the exponential term replaced by a linear equation at $i_{\text{melt}}$.

**Breakdown Region Warnings**

Messages like the following are breakdown region warnings:

D2: Breakdown voltage exceeded.
Q1: The collector-substrate voltage exceeded breakdown voltage.

The warning message identifies the relevant component name (D2 and Q1) and the affected junction.

The Spectre simulator issues breakdown region warnings only when you specify conditions for them. For information on setting parameters to identify a breakdown region, see “Customizing Error and Warning Messages” on page 262.

**Missing Diode Would Be Forward-Biased**

Warning from spectre at time = 501.778 ns during transient analysis tran_to_1u.

i1.q0: Missing collector-substrate diode would be forward biased.

Notice from spectre at time = 510.1688 ns during transient analysis tran_to_1u.
Missing collector-substrate diode returns to normal bias condition.

Most p-n junctions in semiconductor models include both a resistive (the diode) and a capacitive (the junction capacitance) model. If the diode reverse saturation current is set to zero, the resistive part of the junction is turned off, and the Spectre simulator assumes that the resistive portion of the junction does not exist (but the junction capacitance may still be present). In this case, if the voltage across the missing diode is larger than 10 * Vt (where Vt is the thermal voltage), Spectre will issue a warning message telling you that the junction, which is missing, will be forward biased. A follow-up notice is issued if, and when, the device returns to a normal bias condition.

Parameter Is Unusually Large or Small

The Spectre simulator checks the parameter values to see if they are within a normal range of expected values. This check can catch data entry errors or identify situations that can cause the Spectre simulator to have difficulties simulating the circuit.

The “Parameter is unusually large or small” message issues a notice about a parameter value. The message looks like one of the following:

- NPNbjt: 'rb' has the unusually small value of 1mOhms.
- PNPbjt: 'tf' has the unusually large value of 1Gs.
- OA1.Q16 of ua741: 'region' has the unusual value of rev.

If you receive such a message, check the parameter. If the unusual parameter value is correct, you can ignore this message.

The limits settings that generate these warning messages are soft limits, as opposed to hard limits settings. Hard limits stop a simulation if they are violated. the Spectre simulator has automatic soft limits on a few parameter values. However, you can override these limits or specify your own limits for parameters that do not have automatic limits. For more information, see “Customizing Error and Warning Messages” on page 262.

gmin Is Large Enough to Noticeably Affect the DC Solution

Warning detected by spectre during DC analysis oppint.
Gmin=1pS is large enough to noticeably affect the DC solution.

By default, the Spectre simulator (and SPICE) adds a very small conductance of $10^{-12}$ siemens called gmin across nonlinear devices. This conductance prevents nodes from floating if the nonlinear devices are turned off. By default, GMIN=1e-12 Siemens. The gmin parameter usually has a minimal effect on circuit behavior. However, some circuits, such as charge storage circuits are very sensitive to the small currents that flow through gmin.
You see a message such as the one given above if the current flowing through the \textit{gmin} conductors, when treated as an error current, does not meet the \textit{gmin} criteria. That is, the message is displayed if the current that enters any node from all attached \textit{gmin} conductors is larger than either \textit{iabstol} or \textit{reltol} multiplied by the sum of the absolute value of the individual currents that enter the node.

If your circuit is not sensitive to small leakage currents, you can ignore this message. If your circuit is sensitive to these currents, reduce the \textit{gmin} value or set it to zero.

**Minimum Timestep Used**

If this problem occurs, the Spectre circuit simulator discards the timepoint that does not meet the convergence criteria and cuts the timestep.

1. Make sure devices have junction and overlap capacitance specified.
2. Increase \textit{maxiters}, but do not go higher than 200.
3. Change to the \textit{gear2} or \textit{gear2only} method of integration.
4. Reduce other occurrences of the local truncation error cutting the timestep. Increase \textit{lteratio} and increase the absolute error tolerances \textit{vabstol} and \textit{iabstol}. Do not go too high with any of these.
5. Combine 2, 3, and 4 and set \textit{cmin} to prevent instantaneous change at every node in the circuit.
6. Relax \textit{reltol} in combination with 5.

**Syntax Errors**

Warning from spectre in \texttt{indab_7} during circuit read-in:
\texttt{na300.scs" 27: `c11': Encountered statement in Spectre format while in Spice language mode. This will not be supported in a future release.}

The Spectre parser is dual mode and accepts both the Spectre native language and documented Spice2G6. The default for the Spectre simulator is SPICE. If you include a file written in Spectre native syntax, you must either specify simulator \texttt{lang=spectre} or name the file with a \texttt{.scs} suffix. After Spectre processes the file, it reverts to the default mode. It is illegal to include Spectre syntax in the SPICE mode or vice versa.

For the MOS instance line given below:
\texttt{M1 1 2 0 0 NCH W= 10u L= 2u}

Spectre displays the following warning:
ml: Encountered statement in Spectre format while in SPICE language mode. This will not be supported in a future release.

Since the instance statement is valid in both languages, you can ignore this warning.

**Topology Messages**

Notice from spectre during topology check.
Only one connection to the following node:
4
No DC path from node `4' to ground, Gmin installed to provide path.

The Spectre topology checker identifies floating nodes and automatically inserts a gmin resistor (1e12 Ohms) to prevent a non-isolated solution. The Spectre simulator then displays a message telling you what it did.

**Model Parameter Values Clamped**

Warning from spectre during initial setup.
n: The value of `vj(pb)' at T = 25 C is 50e-03 V, which is too small.
Clamped to 0.1 V.
n: The value of `vj(pb)' at T = 27 C is 41.7617e-03 V, which is too small.
Clamped to 0.1 V.

The Spectre simulator clamps model parameter values to prevent numerical difficulties during simulation. When clamping is completed, Spectre displays a message indicating that it is using clamped values. There is no way to disable these clamps.

**Invalid Parameter Warnings**

Warning from spectre during circuit read-in.
`pchmod': `tox' is not a valid parameter for `bsim4' models.
`pchmod': `nch' is not a valid parameter for `bsim4' models.

This type of warning is issued any time you specify an invalid parameter in a model definition. The models included with Spectre have predefined model parameters. For more information, see spectre -h.

Only these predefined parameters can be used within a model definition. The Spectre circuit simulator issues similar warnings for invalid instance and subcircuit parameters.

**Redefine Primitives Messages**

Warning from spectre in `q2' during circuit read-in.
"redefPrim.scs" 6: `q2.resistor' redefines the primitive named `resistor

Spectre displays this message if you define a model or subcircuit with the same name as a built-in primitive device.

The following message tells you that the local definition will override the built-in definition:

model resistor bjt
q1 1 2 3 resistor

q1 is considered a bjt device rather than a resistor.

Initial Condition Messages

Notice from spectre during IC analysis, during transient analysis 'tran1'.
Initial condition computed for node 2 is in error by 755.152 uV. To reduce error in computed initial conditions, decrease `rforce'. However, setting rforce too small may result in convergence difficulties or in the matrix becoming singular.

The Spectre simulator sets initial conditions on a node by attaching a voltage source through a resistor. The default value of this resistor is 1, but you can control the value through the options parameter rforce. This notice indicates that the initial condition calculated for this node is about 755uV from the value specified in the netlist. You can lower the value of rforce to bring the voltage values into agreement in one of the following ways:

- Through the Analog Options window in the Analog Design Environment.
- Inserting an options statement in the netlist. An example is given below:

  myOptions options rforce=1m

Output Messages

Notice from spectre during transient analysis 'tran1'.
No outputs found. Loosening output filter criterion to 'lvlpub'.

If you set save=selected, the Spectre simulator saves the voltages in the save statement. If the save statement does not contain any voltage values, Spectre issues the above warning and changes the save option default to lvlpub. This saves all node voltages.

Customizing Error and Warning Messages

You can customize the Spectre error and warning messages to some extent to fit the needs of a simulation. This section tells you about these customization options.
Selecting Limits for Parameter Value Warning Messages

You can accept Cadence default soft limits that determine when you receive warning messages about parameter values, or you can enter your own limits. You can also control which parameters the Spectre simulator checks. This section gives you instructions for all.

Accepting Cadence Range Limits Defaults

The most convenient option for deciding which warning messages you receive is to accept Cadence Range Limits Defaults. The Cadence defaults are located in

`your_install_dir/tools/dfII/etc/spectre/range.lmts`, and you can examine them to see if they meet your needs. You can enter Cadence defaults with the `SPECTRE_DEFAULTS` environment variable in your shell initialization file (such as `.profile` or `.cshrc`). The entry in your shell initialization file looks like the following:

```
setenv SPECTRE_DEFAULTS "+param $HOME/tools/dfII/etc/
spectre/range.lmts"
```

With this entry in the shell initialization file, the Spectre simulator reads parameter limits from `your_install_dir/tools/dfII/etc/spectre/range.lmts`.

You can override a `SPECTRE_DEFAULTS` setting with the `param` option of the `spectre` command. Specifying `+param` as a command line argument overrides `+param` in `SPECTRE_DEFAULTS` and tells the Spectre simulator to read range limits from the file you specify. Specifying `-param` tells the Spectre simulator to ignore the `+param` given in `SPECTRE_DEFAULTS` without giving the Spectre simulator a new location to find range limits.

Note: For more information about Spectre defaults, see the *Spectre Circuit Simulator Reference* manual and “Customizing Percent Codes” on page 249.

Creating a Parameter Range Limits File

In some circumstances, you might want to set your own parameter limits for warning messages. This might be the case, for example, if you are maintaining your own sets of model libraries. If you want to choose your own parameter limits for warnings, you must use a text editor to create a parameter range limits file.

A parameter range limits file requires the following syntax. Fields enclosed by single brackets ([ ]) are optional.

```
[ComponentKeyword] [model] [LowerLimit <[=]]
 [Param[|] <[=] UpperLimit]
```

Observe the following syntax rules for a parameter limits file:
You can specify limits for input, output, or operating-point parameters for either component instances or models. You can also specify limits for analysis parameters.

You must specify the limits for each parameter on a single line.

You can specify open bounds using angle brackets (<) or closed bounds using an angle bracket with an equal sign (<=). If you specify closed bounds, there can be no space between < and =.

You can specify inclusive or exclusive ranges. If you specify exclusive ranges, the upper limit must be smaller than the lower limit.

The diagram on the following pages shows you the proper formats for range specifications.
### Examples of Range Limits Specifications

<table>
<thead>
<tr>
<th>Type</th>
<th>Specification</th>
<th>Diagram</th>
</tr>
</thead>
<tbody>
<tr>
<td>Single boundary</td>
<td>p &lt; 2</td>
<td><img src="image1.png" alt="Diagram" /></td>
</tr>
<tr>
<td></td>
<td>p &lt;= 2</td>
<td><img src="image2.png" alt="Diagram" /></td>
</tr>
<tr>
<td></td>
<td>1 &lt; p</td>
<td><img src="image3.png" alt="Diagram" /></td>
</tr>
<tr>
<td></td>
<td>1 &lt;= p</td>
<td><img src="image4.png" alt="Diagram" /></td>
</tr>
<tr>
<td>Inclusive boundaries</td>
<td>1 &lt; p &lt; 2</td>
<td><img src="image5.png" alt="Diagram" /></td>
</tr>
<tr>
<td></td>
<td>1 &lt;= p &lt; 2</td>
<td><img src="image6.png" alt="Diagram" /></td>
</tr>
<tr>
<td></td>
<td>1 &lt; p &lt;= 2</td>
<td><img src="image7.png" alt="Diagram" /></td>
</tr>
<tr>
<td></td>
<td>1 &lt;= p &lt;= 2</td>
<td><img src="image8.png" alt="Diagram" /></td>
</tr>
<tr>
<td>Exclusive boundaries</td>
<td>2 &lt; p &lt; 1</td>
<td><img src="image9.png" alt="Diagram" /></td>
</tr>
<tr>
<td></td>
<td>2 &lt;= p &lt; 1</td>
<td><img src="image10.png" alt="Diagram" /></td>
</tr>
</tbody>
</table>
Examples of Range Limits Specifications *(continued)*

- The component keyword must be a Spectre name, not a name used for SPICE compatibility. For example, use `mos3` rather than `mos`.

- If you specify more than one parameter limit for a component, you need to specify the component keyword only once. The Spectre simulator assumes the keyword is unchanged from the previous parameter unless you specify a new component keyword.

- If you give a parameter limit more than once, your last instructions override previous limits.

- If you mention a parameter but give it no limits, all limits are disabled for that parameter.

- You can specify limits for integer, real, or enumerated parameters. Enumerated parameters are those that take only predefined values (such as `yes` or `no` and `all` or `none`).

To specify limits on enumerated parameters, use the index of the enumeration in the limits declaration for that parameter. To find the index of a parameter of component name, see the parameter listings for the component name in the Spectre online help (`spectre -h`) and count the enumerations in the limits declaration starting from zero.

For example, to specify that the BJT operating-point parameter `region` should not be `rev` (reversed), look for the `region` parameter in the parameter listings for the BJT component. The `region` parameter is described as follows:

```
region=fwd
```

*Estimated operating region. Possible values are off, fwd, rev, or sat.*
For this parameter, \texttt{off} has index 0, \texttt{fwd} has index 1, \texttt{rev} has index 2, and \texttt{sat} has index 3. To specify a limit that notifies you if any BJT is reversed, use either of the following specifications:

\begin{itemize}
  \item $2 < \text{region} < 2$
  \item $3 \leq \text{region} \leq 1$
\end{itemize}

- You must give the keyword \texttt{model} when you place limits on model parameters. If you do not give the keyword \texttt{model}, the limits are applied to instance parameters.

- You can indicate upper or lower limits for the absolute value of a parameter with the vertical line character ($|vto|$).

For example,

\begin{itemize}
  \item \texttt{resistor} $0.1 < |r| < 1\text{M}$
\end{itemize}

specifies that the absolute value of $r$ should be greater than 0.1 ohm and less than 1 megohm. There can be no spaces between the absolute value symbols and the parameter name.

- You currently cannot place limits on vector parameters.

- You can write parameter limits using Spectre native-mode scale factors. For example, you can write the limit

\begin{itemize}
  \item $f \leq 1.0e6$
\end{itemize}

as

\begin{itemize}
  \item $f \leq 1\text{M}$
\end{itemize}

\section*{Example of a Parameter Range Limits File}

This example shows a parameter limits file with correct syntax.

\begin{verbatim}
mos3  0.5u <= l <= 100u
     0.5u <= w
     0 < as <= 1e-8
     0 < ad <= 1e
model |vto| <= 3
\end{verbatim}

You can find the parameter names ($l, w, as, ad, vto$) and component keywords ($\texttt{mos3}$) in the parameter listings in the Spectre online help (\texttt{spectre -h}). This example instructs the Spectre simulator to accept without warnings $\texttt{mos3}$ components for these conditions:

- If channel length is more than or equal to 0.5 $\mu$m, or less than or equal to 100 $\mu$m
- If channel width is greater than or equal to 0.5 $\mu$m
If the area of source diffusion is greater than 0, or less or equal to 1e-8 m²

If the area of drain diffusion is greater than 0, or less or equal to 1e-8 m²

If the mos3 model parameter vto (the threshold voltage at zero body bias) has an absolute value less than or equal to 3

**Entering a Parameter Range Limits File**

You can enter a parameter range limits file in two ways:

- Type the `+param <filename>` option of the `spectre` command from the command line or place it in an environment variable. `<filename>` is the name of the parameter range limits file. In the following example, `limits3` is the range limits file for this simulation of `test.circuit`.

  ```
spectre +param limits3 test.circuit
  ```

- Read the parameter limits file from within another file by putting an `include` statement with a syntax like the following example in your netlist.

  ```
  include "filename"
  ```

  `filename` is the name you give to the range limits file.

You can nest `include` statements. The only limit on depth is that imposed by the operating system on the number of files that can be open simultaneously in the Spectre simulator.

Paths you specify in filenames refer to the directory that contains the current file, not to the directory in which the Spectre simulator was started. For example, suppose your directory tree is set up as follows

- `design1/ckt1`
- `design1/param.lmts`
- `design1/resistor.lmts`
- `design2/ckt2`
- `design2/param.lmts`
- `design2/resistor.lmts`

and you run the Spectre simulator in `design1` with the following `spectre` command:

```
spectre +param ../design2/param.lmts ckt1
  ```

If the file `design1/param.lmts` contains the line

```
include "resistor.lmts"
```

the Spectre simulator reads in the `design2/resistor.lmts` file, but not the `design1/resistor.lmts` file.
Requesting Breakdown Region Warnings for Transistors

If you want warning messages about the breakdown regions of transistors, you must set the appropriate parameters for each component when you identify the component with an instance or model statement. For most transistors, you set the \textit{bvj} parameter.

For BJTs, you must set three parameters: \textit{bvbe}, \textit{bvbc}, and \textit{bvsub}. These are breakdown parameters for the base-emitter, the base-collector, and the substrate junctions.

Diodes are also exceptions because you can set both the \textit{bvj} and \textit{bv} parameters. You need two different parameters for the diode breakdown voltage because of the Zener breakdown model in the diode. When you use the diode as a Zener diode, it is purposely biased in the breakdown region, and you do not want to be warned about the Zener breakdown. By specifying the \textit{bv} parameter, you tell the Spectre simulator to implement the Zener diode model at \textit{bv}.

Telling Spectre to Perform Additional Checks of Parameter Values

You can perform a \textit{check} analysis at any point in a simulation to be sure that the values of component parameters are reasonable. You can perform checks on input, output, or operating-point parameters. The Spectre simulator checks parameter values against parameter soft limits. To use the \textit{check} analysis, you must also enter the \texttt{+param} command line argument with the \texttt{spectre} command to specify a file that contains the soft limits.

The following example illustrates the syntax of the \textit{check} statement. It tells the Spectre simulator to check the parameter values for instance statements.

\begin{verbatim}
ParamChk check what=inst
\end{verbatim}

- \texttt{ParamChk} is your unique name for this \textit{check} statement.
- The keyword \textit{check} is the component keyword for the statement.
- The \textit{what} parameter tells the Spectre simulator which parameters to check.

The \textit{what} parameter of the \textit{check} statement gives you the following options:

<table>
<thead>
<tr>
<th>Option</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>none</td>
<td>Disables parameter checking.</td>
</tr>
<tr>
<td>models</td>
<td>Checks input parameters for all models only.</td>
</tr>
<tr>
<td>inst</td>
<td>Checks input parameters for all instances only.</td>
</tr>
<tr>
<td>input</td>
<td>Checks input parameters for all models and all instances.</td>
</tr>
</tbody>
</table>
Selecting Limits for Operating Region Warnings

The Spectre simulator lets you specify forbidden operating regions for transistors. If a transistor operates in a forbidden operating region, the Spectre simulator sends you a warning message. This feature is available for BJTs, MOSFETs, JFETs, and GaAs MESFETs.

Specifying Forbidden Operating Regions for Transistors

You specify a forbidden operating region in a transistor with the `alarm` parameter. The `alarm` parameter gives you the following options:

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>none</td>
<td>The default condition with no warnings issued.</td>
</tr>
<tr>
<td>off</td>
<td>Warns if the transistor is turned off.</td>
</tr>
<tr>
<td>triode</td>
<td>Warns if the transistor operates in the triode region (available for MOSFETs).</td>
</tr>
<tr>
<td>sat</td>
<td>Warns if the transistor operates in the saturation region.</td>
</tr>
<tr>
<td>subth</td>
<td>Warns if the transistor operates in the subthreshold region (available for MOSFETs).</td>
</tr>
<tr>
<td>fwd</td>
<td>Warns if the transistor is forward-biased (available for BJTs).</td>
</tr>
<tr>
<td>rev</td>
<td>Warns if the transistor is reverse-biased.</td>
</tr>
</tbody>
</table>

For example, to be sure that a group of MOSFETs always operates in the saturation region, you enter this `model` statement:

```plaintext
model mos_example nmos alarm=off alarm=triode
    alarm=subth ..... 
```
Each of the three alarm parameters in this example identifies a forbidden operating condition. Operating the device anywhere except in the saturation region triggers a warning. The warning looks like the following:

Warning detected by spectre during transient analysis 'timesweep'.
M1: Device operated in the triode region.

**Defining BJT Operating Regions**

The Spectre simulator provides two parameters, $v_{befwd}$ and $v_{bcfwd}$, that let you specify the boundaries between BJT operating regions. The default value for each parameter is 0.2 volts.

The following table shows you the criteria the Spectre simulator uses to determine BJT operating regions.

<table>
<thead>
<tr>
<th>Region</th>
<th>Bias Conditions</th>
</tr>
</thead>
<tbody>
<tr>
<td>off</td>
<td>$V_{be} \leq v_{befwd}$ and $V_{bc} \leq v_{bcfwd}$</td>
</tr>
<tr>
<td>saturation</td>
<td>$V_{be} &gt; v_{befwd}$ and $V_{bc} &gt; v_{bcfwd}$</td>
</tr>
<tr>
<td>forward</td>
<td>$V_{be} &gt; v_{befwd}$ and $V_{bc} \leq v_{bcfwd}$</td>
</tr>
<tr>
<td>reverse</td>
<td>$V_{be} \leq v_{befwd}$ and $V_{bc} &gt; v_{bcfwd}$</td>
</tr>
</tbody>
</table>

**Range Checking on Subcircuit Parameters**

You can test the value of subcircuit parameters with the **paramtest** component. If the parameters meet your testing criteria, you can print an informational message, print a warning, or print an error message and terminate the program.

**Formatting the paramtest Component**

The **paramtest** component has the following format:

```
Name paramtest parameter=value
```

- **Name** is your unique name for this **paramtest** component.
- The keyword **paramtest** is the component keyword for the component.
- The parameters specify the tests that are applied to the parameters, the action taken if parameters satisfy the test conditions, and the text of the message that is printed when parameters satisfy the test conditions.
Rules and Guidelines to Remember

- If you specify more than one test, the conditional action is taken if any test passes.
- If you use the `paramtest` component without specifying test conditions, the specified actions are taken, and the message is printed unconditionally. This option is useful for using the `paramtest` component with the `if` statement. The `paramtest` instruction can be followed whenever a given `if` statement option is executed.

The `paramtest` Options

The following table explains the possible `paramtest` options.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>printf</code></td>
<td>Informational message is printed if test condition is satisfied.</td>
</tr>
<tr>
<td><code>warnif</code></td>
<td>Warning is printed if test condition is satisfied.</td>
</tr>
<tr>
<td><code>errorif</code></td>
<td>Program quits and error message is printed if testing condition is satisfied.</td>
</tr>
<tr>
<td><code>message</code></td>
<td>Parameter value is the message text.</td>
</tr>
<tr>
<td><code>severity</code></td>
<td>You set this parameter when you use <code>paramtest</code> without a test condition. It specifies the type of message printed and the action to be taken, if any. The possible values are <code>debug</code>, <code>status</code>, <code>warning</code>, <code>error</code>, and <code>fatal</code>. If you specify <code>error</code>, the current analysis quits with an error message. If you specify <code>fatal</code>, the whole simulation stops.</td>
</tr>
</tbody>
</table>

A `paramtest` Example

This example uses three consecutive `paramtest` statements to check the values of four parameters—`l`, `w`, `ls`, and `ld`. If a parameter value satisfies a test condition, one of three different warning messages is printed:

```plaintext
TooShort paramtest warnif=(l < 1um) \ message="Channel length for nmos must be greater than 1u."
TooThin paramtest warnif=(w < 1um) \ message="Channel width for nmos must be greater than 1u."
TooNarrow paramtest warnif=(ls < 1um) warnif=(ld < 1um) \ message="Strip width for nmos must be greater than 1u."
```
Controlling Program-Generated Messages

The Spectre simulator normally sends error, warning, and informational messages to the screen. To prevent confusion, the Spectre simulator limits the amount of material it sends to the screen. You can, however, get a more complete printout of messages if you send the messages to a log file that you can generate with the `spectre` command or in a `SPECTRE_DEFAULTS` environment variable.

Specifying Log File Options

You can choose from among the following command line options:

+log <file>  The Spectre simulator sends all messages to a log file as well as printing to the screen. You specify the name for the file. You can use +l as an abbreviation of +log.

=log <file>  The Spectre simulator sends all messages to a log file but does not print to the screen. You specify the name for the file. You can use =l as an abbreviation of =log.

-log       The Spectre simulator does not create a log file. You can use -l as an abbreviation of -log. This is the default option.

Command Line Example

The following entry on the command line runs a simulation for circuit smps.circuit and sends all messages to a log file named smps.logfile:

```
spectre =log smps.logfile smps.circuit
```

Setting Environment Variables

If you specify log file options in a `SPECTRE_DEFAULTS` environment variable, you might want to name log files according to some system that helps you keep track of log files from different simulations. Spectre predefined percent codes are useful for this. The following example uses the predefined percent code `%C` to create log filenames based on the input filename. If you run a simulation for `smps.circuit`, the Spectre simulator creates a log file named `smps.circuit.logfile`. You place the `SPECTRE_DEFAULTS` environment variable in the `.cshrc` or `.profile` files.

```
setenv SPECTRE_DEFAULTS "=log %C.logfile"
```
For more information about predefined percent codes and the SPECTRE_DEFAULTS environment variable, see Chapter 11, “Managing Files.”

Suppressing Messages

There are also spectre command options that let you print or suppress error, warning, or informational messages:

- `+error` Prints error messages
- `-error` Does not print error messages
- `+warning` Prints warning messages
- `-warning` Does not print warning messages
- `+info` Prints informational messages
- `-info` Does not print informational messages

As a default, the Spectre simulator prints all these messages.

Correcting Convergence Problems

In this section, you will learn about procedures that can help you if a simulation does not converge.

Correcting DC Convergence Problems

If you have DC convergence problems, these suggestions might help you. Simple solutions generally precede more radical or complex measures in the list.

- Evaluate and resolve any warning or error messages.
- Check for circuit connection errors. Check to see that the polarity and value are correct for independent sources. Check to see if the polarity and multiplier are correct for controlled sources.
- Try all of the homotopy methods (`gmin`, `source`, `ptran`, and `dptran`). These are tried by default.
- Check for an incorrect estimated operating region. The default estimated operating region in the Spectre simulator is on in the forward region for all devices. If there are a
reasonable number of devices that are really off, set them off in the schematic. For a large number of devices, this might not be practical.

- Check for extremely high gain circuits with nonlinearities and feedback. The convergence criteria are applied to all nodes in the circuit. The output of the gain block meets Kirchhoff’s Current Law and delta \( (\text{reltol} \times V) \) about 1 part in \( 10^3 \) by default. Because of the gain, the input must move by this value divided by the gain. If the gain is high (for example, \( 10^8 \)), the input must move less than 1 part in \( 10^{11} \). This is an extremely small motion from iteration to iteration that might not be achievable. If the gain is even higher, the numerical resolution of the machine might be approached. About 15 digits of resolution is available in a 64-bit floating-point number. In this case, the gain needs to be reduced.

**Note:** In this case, one node (the output) controls convergence, and all the other nodes are more accurate than the convergence criteria by itself would predict. This is typical for most circuits.

- Enable the topology checker (set `topcheck=full` on the `options` statement) and pay attention to any warnings.

- Increase `maxiters` for the DC analysis.

- If you have convergence problems during a DC sweep, reduce the step size.

- Check for unusual parameter values using the parameter range checker (add `+param param-limits-file` to the `spectre` command line arguments) and pay attention to any warnings.

  Print out the minimum and maximum parameter values by placing an `info` statement in the netlist. Make sure that the values for the instance, model, output, temperature-dependent, and (if possible) operating-point parameters are reasonable.

- Avoid using very small floating resistors, particularly small parasitic resistors in semiconductors. Use voltage sources or `iprobes` to measure currents instead. Small floating resistors connected to high impedance nodes can cause convergence difficulties. `rbm` in the bipolar model is especially troublesome.

- If the `minr` model parameter is set, make certain it is set to 1 mOhm or larger.

- Use realistic device models. Make sure that all component parameters are reasonable, particularly nonlinear device model parameters.

- Increase the value of `gmin` with the `options` statement.

- Loosen tolerances, particularly absolute tolerances such as `iabstol` (on the `options` statement).
Simplify the nonlinear component models. Try to avoid regions in the model that might cause convergence problems.

When you have a solution, write it to a nodeset file using the write parameter. When you run the simulation again, read the solution back in using the readns parameter in the dc statement.

If this is not the first analysis, the solution from the previous analysis might be an inadequate solution estimate because it differs too much from the solution for the current analysis. If this is so, set restart=yes.

If you have an estimate of the solution, use nodeset statements or a nodeset file to set as many nodes as possible.

If using nodesets or initial conditions causes convergence difficulties, try increasing rforce with the options statement.

If you are simulating a bipolar analog circuit, make sure the region parameters on all transistors and diodes are set correctly.

If the analysis fails at an extreme temperature but succeeds at room temperature, try adding a DC analysis that sweeps temperature. Start at room temperature, sweep to the extreme temperature, and write the last solution to a nodeset file.

Use numeric pivoting in the sparse matrix factorization. Set pivotdc=yes with the options statement. Sometimes you must also increase the pivot threshold to between 0.1 to 0.5 by resetting the pivrel parameter with the options statement.

Divide the circuit into pieces and simulate them individually. Make sure that results for a part alone are close to results for that part combined with the rest of the circuit. Use the results to create nodesets for the whole circuit.

Try replacing the DC analysis with a transient analysis. Modify all the independent sources to start at zero and ramp to the independent source DC values. Run the transient analysis well beyond the time when all the sources have reached their final values. Write the final point to a nodeset file.

You can make this transient analysis more efficient with one of the following procedures:

- Set the integration method to backward-Euler (method=euler).
- Loosen the local truncation error criteria by increasing iteratio to 50 or more.

Occasionally, an oscillator in the circuit causes the transient analysis to terminate or work very slowly.
Correcting Transient Analysis Convergence Problems

You can use two approaches to eliminate transient analysis convergence problems. The first strategy is to reduce the effect of discontinuities in nonlinear capacitors. The second method is to eliminate discontinuous jumps in the solution. Try the following suggestions if you have difficulty with transient analysis convergence:

- Use a complete set of parasitic capacitors on nonlinear devices to avoid jumps in the solution waveforms. Specify nonzero source and drain areas on MOS models.
- Use the $c_{\text{min}}$ parameter to install a small capacitor from every node in the circuit to ground. This usually eliminates any jumps in the solution.
- If you can identify a nonlinear capacitance that might have a discontinuity, simplify the nonlinear capacitor model. If you cannot actually simplify the model, modifying it might help convergence.
- As a last resort, relax the tolerance values for the $l_{\text{iteratio}}$ or $r_{\text{eltol}}$ parameters and widen transitions in the stimulus waveforms.

Correcting Accuracy Problems

If you need greater accuracy from a Spectre simulation, the most common solution is to tighten the $r_{\text{eltol}}$ parameter of the options or set statements. In addition, be sure that the absolute tolerance parameters, $v_{\text{abstol}}$ and $i_{\text{abstol}}$, are set to appropriate values. If tightening $r_{\text{eltol}}$ does not help or if it greatly slows the simulation, try the additional suggestions in the following sections.

Suggestions for Improving DC Analysis Accuracy

- Be sure there are no errors in the circuit. Use the computed DC solution and the operating point to debug the circuit. Check the topology, the component parameters, the models, and the power supplies.
- Be sure you are using appropriate models and that the model parameters are consistent and correct.
- If the circuit might have more than one solution, use nodeset statements to influence the Spectre simulator to compute the solution you want.
- Be sure that $g_{\text{min}}$ is not influencing the solution. If possible, set $g_{\text{min}}$ to 0 (in an options or set statement).
Suggestions for Improving Transient Analysis Accuracy

- Verify that the circuit biased up properly. If it did not, there might be a problem in the topology, the models, or the power supplies.

- Be sure you are using appropriate models and that the model parameters are consistent and correct. Check the operating point of each device.

- Set the transient analysis parameter errpreset to conservative.

- If there is a charge conservation problem, use only charge-conserving models if you are not already doing so. Then tighten reltol to increase accuracy. (With the Spectre simulator, only customer-installed models might not be charge conserving.)

- Be sure that gmin is not influencing the solution. If possible, set gmin to 0 (in an options or set statement).

- If a solution exhibits point-to-point ringing, set the integration method in the transient analysis to Gear’s second-order backward-difference formula (method=gear2only).

- If a low-loss resonator exhibits too much loss, set the integration method in the transient analysis to the trapezoidal rule (method=traponly).

- If the initial conditions used by the Spectre simulator are not the same as the ones you specified, decrease the rforce parameter in the options or set statements until the initial conditions are correct.

- If the Spectre simulator does not accurately follow the turn-on transient of an oscillator, set the maxstep parameter of the transient analysis to one-tenth the size of the expected period of oscillation or less.
Example Circuits

This appendix contains example netlists for testing the BSIM3v3 standard model.

- Notes on the BSIM3v3 Model on page 280
- Spectre Syntax on page 280
- SPICE BSIM 3v3 Model on page 280
- Spectre BSIM 3v3 Model on page 281
- Ring Oscillator Spectre Deck for Inverter Ring with No Fanouts (inverter_ring.sp) on page 281
- Ring Oscillator Spectre Deck for Two-Input NAND Ring with No Fanouts (nand2_ring.sp) on page 283
- Ring Oscillator Spectre Deck for Three-Input NAND Ring with No Fanouts (nand3_ring.sp) on page 284
- Ring Oscillator Spectre Deck for Two-Input NOR Ring with No Fanouts (nor2_ring.sp) on page 286
- Ring Oscillator Spectre Deck for Three-Input NOR Ring with No Fanouts (nor3_ring.sp) on page 287
- Opamp Circuit (opamp.cir) on page 289
- Opamp Circuit 2 (opamp1.cir) on page 289
- Original Open-Loop Opamp (openloop.sp) on page 289
- Modified Open-Loop Opamp (openloop1.sp) on page 290
- Example Model Directory (q35d4h5.modsp) on page 290
Notes on the BSIM3v3 Model

The Spectre® circuit simulator supports the standard BSIM 3v3 MOS model (both BSIM 3v3.1 and BSIM 3v3.2) as published by the University of California at Berkeley. Further information about this model can be obtained by using Spectre’s online help by typing `spectre -h bsim3v3` at the command line or by consulting the BSIM3 home page at [http://www-device.eecs.berkeley.edu/~bsim3/index.html](http://www-device.eecs.berkeley.edu/~bsim3/index.html).

Spectre does not add proprietary parameters to its implementation of the standard model.

Spectre Syntax

Notes explaining Spectre syntax are included as comments throughout the example netlists.

The Spectre circuit simulator reads Spice2G6 input along with its own native format. The model card can therefore be specified in either format. Below is an example of each. Note that the valid parameter list does not change, only the primitive name, level designation, and version/type parameters.

Beginning with the 4.4.3 release, the Spectre simulator is compatible with SPICE input language beyond documented SPICE2G6. Contact your local Cadence representative for more details.

SPICE BSIM 3v3 Model

```
*model = bsim3v3
*Berkeley Spice Compatibility
*Lmin= .35 Lmax= 20 Wmin= .6 Wmax= 20
.model N1 NMOS
+Level=11
+Tnom=27.0
+Nch=2.498E+17  Tox=9E-09  Xj=1.00000E-07
+Lint=9.36e-8  Wint=1.47e-7
+Vth0=.6322  K1=.756  K2=-3.83e-2  K3=-2.612
+Dvt0=2.812  Dvt1=0.462  Dvt2=-9.17e-2
+Nlx=3.52291E-08  W0= 1.163e-6  K3b= 2.233
+Vsat=86301.58  Ua=6.47e-9  Ub=4.23e-18  Uc=-4.706281E-11
+Rds=650  U0=388.3203  wr=1
+A0=.3496967  Ags=.1
+B0=0.546  B1= 1
+Dwg=-6.0E-09  Dwb=-3.56E-09  Prwb=-.213
+Keta=-3.605872E-02  A1=2.778747E-02  A2=.9
+Voff=-6.735529E-02  NFactor=1.139926  Cit=1.622527E-04
+Cdsc=-2.147181E-05  Cdscb= 0
+Dvto=0  Dvt1w=0  Dvt2w=0
+Cdsc=0  Prwg=0
+Eta0=1.0281729E-02  Etab=-5.042203E-03
+Dsub=.31871233
```
+Pvag=0  delta=0.01
+Wl=0  Ww=-1.420242E-09  Wwl = 0
+Wln=0  Wwn=2.613948  Ll=1.300902E-10
+Lw=0  Lwl=0  Lln=3.163948  Lwn=0
+kl1=-.3  kt2=-.051
+At=22400
+Ute=-1.48
+Ual=3.31E-10  Ubl=2.61E-19  Uc1=-3.42e-10
+Kt1l=0  Prt=764.3

Spectre BSIM 3v3 Model

*Berkeley Spice Compatibility
*Lmmin = .35  Lmax= 20  Wmin= .6  Wmax= 20
simulator lang=spectre
model nch bsim3v3
+version=3.1
+type=n
+tnom=27.0
+nch=2.498E+17  tox=9E-09  xj=1.000000E-07
+lint=9.36e-8  wint=1.47e-7
+vth0=6.322  k1=7.756  k2=-3.83e-2  k3=-2.612
+dvt0=2.812  dvt1=0.462  dvt2=-9.17e-2
+nln=3.52291E-08  w0=1.163e-6  k3b=2.233
+Vsat=86301.58  ua=6.47e-9  ub=4.23e-18  uc=-4.706281e-11
+rdsw=650  u0=388.3203  wr=1
+a0=.3496967  ags=.1
+b0=0.546  b1= 1
+dwg=-6.0e-09  dwb=-3.56e-09  prwb=-.213
+keta=-3.605872e-02  a1=2.778747e-02  a2=.9
+Voff=-6.735529e-02  nfactor=1.139926  cit=1.622527e-04
+cdsc=-2.147181e-05  cdscb= 0
+dvt0w=0  dvt1w=0  dvt2w=0
+cdscd=0  prwg=0
+eta0=1.0281729e-02  etab=-5.042203e-03
+dssub=3.1871233
+Pclm=1.114846  pdiblc1=2.45357e-03  pdiblc2=6.406289e-03
+Drout=.31871233  Pscbe1=5000000  Pscbe2=5E-09  Pdiblc=-.234
+Pvag=0  delta=0.01
+Wl=0  Ww=-1.420242E-09  Wwl = 0
+Wln=0  Wwn=2.613948  Ll=1.300902E-10
+Lw=0  Lwl=0  Lln=3.163948  Lwn=0
+kt1=-.3  kt2=-.051
+At=22400
+Ute=-1.48
+Ual=3.31E-10  Ubl=2.61E-19  Uc1=-3.42e-10
+Kt1l=0  Prt=764.3

Ring Oscillator Spectre Deck for Inverter Ring with No Fanouts (inverter_ring.sp)

This example uses Spectre syntax.
// Ring oscillator Spectre deck for INVERTER ring with no fanouts.
simulator lang=spectre
global 0 gnd vdd vss
aliasGnd ( gnd 0 ) vsource type=dc dc=0
// Spectre options to be used
SetOption1 options  iabstol=1.00n audit=full temp=25
MyAcct1 info what=inst extremes=yes
MyAcct2 info what=models extremes=yes
MyAcct3 info what=input extremes=yes
MyAcct5 info what=terminals extremes=yes
MyAcct6 info what=oppoint extremes=yes

// Next section is the subckt for inv
subckt inv ( nq a )
m1 ( nq a vdd vdd ) p l=0.35u w=2.60u ad=1.90p pd=6.66u as=1.90p ps=6.66u
m2 ( vss a nq vss ) n l=0.35u w=1.10u ad=0.80p pd=3.66u as=0.80p ps=3.66u
// Interconnect Caps for inv
c0 ( a vdd ) capacitor c=1.0824323e-15
c1 ( a nq ) capacitor c=3.0044e-16
c2 ( nq vss ) capacitor c=5.00186e-16
c3 ( nq vdd ) capacitor c=6.913993e-16
c4 ( a vss ) capacitor c=8.5372566e-16
ends

// Begin top level circuit definition
xinv1 ( 1 90 ) inv
xinv2 ( 2 1 ) inv
xinv3 ( 3 2 ) inv
xinv4 ( 4 3 ) inv
xinv5 ( 5 4 ) inv
xinv6 ( 6 5 ) inv
xinv7 ( 7 6 ) inv
xinv8 ( 8 7 ) inv
xinv9 ( 9 8 ) inv
xinv10 ( 10 9 ) inv
xinv11 ( 11 10 ) inv
xinv12 ( 12 11 ) inv
xinv13 ( 13 12 ) inv
xinv14 ( 14 13 ) inv
xinv15 ( 15 14 ) inv
xinv16 ( 16 15 ) inv
xinv17 ( 90 16 ) inv

// Next couple of lines sets variables for vdd and vss.
parameters vdd_S1=3.3
parameters vss_S1=0.0
vdd_I1 ( vdd gnd ) vsource dc=vdd_S1
vss_I1 ( vss gnd ) vsource dc=vss_S1

// Set initial conditions:
ic 2=0 4=0 6=0 8=0 10=0
// Next line makes the call to the model
// NOTE: The user may utilize the '.lib' syntax with Spectre's +spp
// command line option if they are using Spectre 4.43 or greater.
// There is also a Spectre native syntax for equivalent
// functionality. It is shown in q35d4h5.modsp.
include "q35d4h5.modsp" section=tt
// Analysis Statement
tempOption options temp=25
typ_tran tran step=0.010n stop=35n
alter_ss altergroup {
    include "q35d4h5.modsp" section=ss
    parameters vdd_S1=3.0
}
alterTempTo100 alter param=temp value=100
ss_tran tran step=0.010n stop=35n
alter_ff altergroup {
    include "q35d4h5.modsp" section=ff
    parameters vdd_S1=3.3
}
alterTempTo0 alter param=temp value=0
ff_tran tran step=0.010n stop=35n

Ring Oscillator Spectre Deck for Two-Input NAND Ring with No Fanouts (nand2_ring.sp)

This example uses Spectre syntax.

// Ring oscillator Spectre deck for 2-Input NAND ring with no fanouts.
simulator lang=spectre
global 0 gnd vdd vss
aliasGnd ( gnd 0 ) vsrce type=dc dc=0

// Spectre options to be used
SetOption1 options  iabstol=1.00n audit=full
    MyAcct1 info what=inst extremes=yes
    MyAcct2 info what=models extremes=yes
    MyAcct3 info what=input extremes=yes
    MyAcct5 info what=terminals extremes=yes
    MyAcct6 info what=oppoint extremes=yes
// Next section is the subckt for na2  *****
subckt na2 ( nq a )
m1 ( nq a vdd vdd ) p l=0.35u w=2.70u ad=1.03p pd=3.46u as=1.98p ps=6.86u
m2 ( nq vdd vdd vdd ) p l=0.35u w=2.70u ad=1.03p pd=3.46u as=1.98p ps=6.86u
m3 ( vss a 6 vss ) n l=0.35u w=1.70u ad=1.25p pd=4.86u as=0.18p ps=1.91u
m4 ( nq vdd 6 vss ) n l=0.35u w=1.70u ad=1.25p pd=4.86u as=0.18p ps=1.91u
c0 ( a vdd ) capacitor c=1.0512057e-15
c1 ( a nq ) capacitor c=7.308e-17
c2 ( vdd vss ) capacitor c=6.12359e-16
c3 ( nq vss ) capacitor c=5.175377e-16
c4 ( nq vdd ) capacitor c=1.1668172e-15
c5 ( a vss ) capacitor c=9.530671e-16
ends
// Begin top-level circuit definition
xna21 ( 1 90 ) na2
xna22 ( 2 1 ) na2
xna23 ( 3 2 ) na2
xna24 ( 4 3 ) na2
xna25 ( 5 4 ) na2
Ring Oscillator Spectre Deck for Three-Input NAND Ring with No Fanouts (nand3_ring.sp)

This example uses Spectre syntax.

// Ring oscillator Spectre deck for 3-Input NAND ring with no fanouts.
simulator lang=spectre
global 0 gnd vdd vss
aliasGnd ( gnd 0 ) vsource type=dc dc=0

// Simulator options to use
SetOption1 options  iabstol=1.00n audit=full
MyAcct1 info what=inst extremes=yes
MyAcct2 info what=models extremes=yes
Spectre Circuit Simulator User Guide
Example Circuits

MyAcct3 info what=input extremes=yes
MyAcct5 info what=terminals extremes=yes
MyAcct6 info what=oppoint extremes=yes

// Next section is the subckt for na3p1
subckt na3p1 ( nq a )
m1 ( nq a vdd vdd ) p l=0.35u w=2.90u ad=1.10p pd=3.66u as=2.12p ps=7.26u
m2 ( nq vdd vdd vdd ) p l=0.35u w=2.90u ad=1.10p pd=3.66u as=1.10p ps=3.66u
m3 ( nq vdd vdd vdd ) p l=0.35u w=2.90u ad=2.12p pd=7.26u as=1.10p ps=3.66u
m4 ( vss a 7 vss ) n l=0.35u w=2.40u ad=1.75p pd=6.26u as=0.25p ps=2.61u
m5 ( 7 vdd 8 vss ) n l=0.35u w=2.40u ad=0.25p pd=2.61u as=0.25p ps=2.61u
m6 ( nq vdd 8 vss ) n l=0.35u w=2.40u ad=1.75p pd=6.26u as=0.25p ps=2.61u
m7 ( 8 vdd ) capacitor c=1.341e-17
c1 ( vdd vss ) capacitor c=9.945302e-16
c2 ( nq vss ) capacitor c=6.287e-16
c3 ( nq vdd ) capacitor c=2.0719818e-15
c4 ( a vss ) capacitor c=5.3760487e-16
c5 ( a vdd ) capacitor c=1.2446956e-15
c6 ( a nq ) capacitor c=7.308e-17
c7 ( 7 vdd ) capacitor c=2.0115e-17
ends

// Begin top level circuit definition
xna3p11 ( 1 90 ) na3p1
xna3p12 ( 2 1 ) na3p1
xna3p13 ( 3 2 ) na3p1
xna3p14 ( 4 3 ) na3p1
xna3p15 ( 5 4 ) na3p1
xna3p16 ( 6 5 ) na3p1
xna3p17 ( 7 6 ) na3p1
xna3p18 ( 8 7 ) na3p1
xna3p19 ( 9 8 ) na3p1
xna3p110 ( 10 9 ) na3p1
xna3p111 ( 11 10 ) na3p1
xna3p112 ( 12 11 ) na3p1
xna3p113 ( 13 12 ) na3p1
xna3p114 ( 14 13 ) na3p1
xna3p115 ( 15 14 ) na3p1
xna3p116 ( 16 15 ) na3p1
xna3p117 ( 90 16 ) na3p1

// Next couple of lines sets variables for vdd and vss.
parameters vdd_S1=3.3
parameters vss_S1=0.0
vdd_I1 ( vdd gnd ) vsource dc=vdd_S1
vss_I1 ( vss gnd ) vsource dc=vss_S1

// Next line initializes nodes within ring
ic 2=0 4=0 6=0 8=0 10=0
include "q35d4h5.modsp" section=tt

// Transient analysis card
tempOption options temp=25
typ_tran tran step=0.010n stop=35n
alter_ss altergroup {
   include "q35d4h5.modsp" section=ss
   parameters vdd_S1=3.0
}
alterTempTo100 alter param=temp value=100
ss_tran tran step=0.010n stop=35n
myAlter2 altergroup {
include "q35d4h5.modsp" section=ff
parameters vdd_S1=3.3
}
alterTempTo0 alter param=temp value=0
ff_tran tran step=0.010n stop=35n

Ring Oscillator Spectre Deck for Two-Input NOR Ring with No Fanouts (nor2_ring.sp)

This example uses Spectre syntax.

// Ring oscillator Spectre deck for 2-Input NOR ring with no fanouts.
simulator lang=spectre
global 0 gnd vdd vss
aliasGnd ( gnd 0 ) vsourcetype=dc dc=0
// Spectre options
SetOption1 options iabstol=1.00n audit=full
MyAcct1 info what=inst extremes=yes
MyAcct2 info what=models extremes=yes
MyAcct3 info what=input extremes=yes
MyAcct5 info what=terminals extremes=yes
MyAcct6 info what=oppoint extremes=yes
// Next section is the subckt for no2
subckt no2 ( nq a )
m1 ( vdd a 6 vdd ) p l=0.35u w=4.80u ad=3.50p pd=11.06u as=0.50p ps=5.01u
m2 ( nq vss 6 vdd ) p l=0.35u w=4.80u ad=3.50p pd=11.06u as=0.50p ps=5.01u
m3 ( vss a nq vss ) n l=0.35u w=1.20u ad=0.88p pd=3.86u as=0.46p ps=1.96u
m4 ( vss vss nq vss ) n l=0.35u w=1.20u ad=0.88p pd=3.86u as=0.46p ps=1.96u
c0 ( a vdd ) capacitor c=6.3676066e-16

c1 ( a nq ) capacitor c=5.3592e-17

c2 ( vdd vss ) capacitor c=5.39538e-16

c3 ( nq vss ) capacitor c=8.780327e-16

c4 ( nq vdd ) capacitor c=5.577428e-16

c5 ( a vss ) capacitor c=1.1100392e-15
ends
// begin top level circuit definition
xno21 ( 1 90 ) no2
xno22 ( 2 1 ) no2
xno23 ( 3 2 ) no2
xno24 ( 4 3 ) no2
xno25 ( 5 4 ) no2
xno26 ( 6 5 ) no2
xno27 ( 7 6 ) no2
xno28 ( 8 7 ) no2
xno29 ( 9 8 ) no2
xno210 ( 10 9 ) no2
xno211 ( 11 10 ) no2
xno212 ( 12 11 ) no2
xno213 ( 13 12 ) no2
xno214 ( 14 13 ) no2
Ring Oscillator Spectre Deck for Three-Input NOR Ring with No Fanouts (nor3_ring.sp)

This example uses Spectre syntax.

// Ring oscillator spectre deck for 3-Input NOR ring with no fanouts.
simulator lang=spectre
global 0 gnd vdd vss
aliasGnd ( gnd 0 ) vsource type=dc dc=0

// Spectre options
SetOption1 options iabstol=1.00n audit=full rforce=1 temp=25
MyAcct1 info what=inst extremes=yes
MyAcct2 info what=models extremes=yes
MyAcct3 info what=input extremes=yes
MyAcct5 info what=terminals extremes=yes
MyAcct6 info what=oppoint extremes=yes

// Next section is the subckt for no3
subckt no3 ( nq a )
m1 ( vdd a 7 vdd ) p l=0.35u w=3.60u ad=2.63p pd=8.66u as=0.38p ps=3.81u
m2 ( 7 vss 8 vdd ) p l=0.35u w=3.60u ad=0.38p pd=3.81u as=0.38p ps=3.81u
m3 ( nq vss 8 vdd ) p l=0.35u w=3.60u ad=1.37p pd=4.36u as=0.38p ps=3.81u
m4 ( nq vss 9 vdd ) p l=0.35u w=3.60u ad=1.37p pd=4.36u as=0.38p ps=3.81u
m5 ( 9 vss 10 vdd ) p l=0.35u w=3.60u ad=0.38p pd=3.81u as=0.38p ps=3.81u
m6 ( vdd a 10 vdd ) p l=0.35u w=3.60u ad=2.63p pd=8.66u as=0.38p ps=3.81u
m7 ( vss a nq vss ) n l=0.35u w=1.40u ad=1.02p pd=4.26u as=0.53p ps=2.16u
m8 ( vss vss nq vss ) n l=0.35u w=1.40u ad=0.53p pd=2.16u as=0.53p ps=2.16u
m9 ( vss vss nq vss ) n l=0.35u w=1.40u ad=0.53p pd=2.16u as=1.02p ps=4.26u
c1 ( 9 nq ) capacitor c=3.7995e-17
c2 ( vdd vss ) capacitor c=1.3996057e-15
c3 ( nq vss ) capacitor c=3.1546797e-15
c4 ( nq vdd ) capacitor c=5.5551875e-16
c5 ( a vss ) capacitor c=1.2907233e-15
c6 ( a vdd ) capacitor c=2.1779808e-15
c7 ( 10 nq ) capacitor c=2.0115e-17
c8 ( a nq ) capacitor c=5.233362e-16
ends

// Begin top-level circuit definition
xno31 ( 1 90 ) no3
xno32 ( 2 1 ) no3
xno33 ( 3 2 ) no3
xno34 ( 4 3 ) no3
xno35 ( 5 4 ) no3
xno36 ( 6 5 ) no3
xno37 ( 7 6 ) no3
xno38 ( 8 7 ) no3
xno39 ( 9 8 ) no3
xno310 ( 10 9 ) no3
xno311 ( 11 10 ) no3
xno312 ( 12 11 ) no3
xno313 ( 13 12 ) no3
xno314 ( 14 13 ) no3
xno315 ( 15 14 ) no3
xno316 ( 16 15 ) no3
xno317 ( 90 16 ) no3

// Next couple of lines sets variables for vdd and vss.
parameters vdd_S1=3.3
parameters vss_S1=0.0
vdd_I1 ( vdd gnd ) vsource dc=vdd_S1
vss_I1 ( vss gnd ) vsource dc=vss_S1

// Next line initializes nodes within ring
ic 2=0 4=0 6=0 8=0 10=0
include "q35d4h5.modsp" section=tt

// Analysis
tempOption options temp=25
typ_trt tran step=0.010n stop=35n
alter_ss altergroup {
include "q35d4h5.modsp" section=ss
parameters vdd_S1=3.0
}
alterTempTo100 alter param=temp value=100
ss_trt tran step=0.010n stop=35n
alter_ff altergroup {
include "q35d4h5.modsp" section=ff
parameters vdd_S1=3.3
}
alterTempTo0 alter param=temp value=0
ff_trt tran step=0.010n stop=35n
Opamp Circuit (opamp.cir)

This example uses Spectre’s SPICE syntax.

```
.subckt opamp 1 2 6 8 9
m1 4 2 3 3 nch w=43u l=10u ad=0.3n as=0.3n pd=50u ps=50u
m2 5 1 3 3 nch w=43u l=10u ad=0.3n as=0.3n pd=50u ps=50u
m3 4 4 8 8 pch w=10u l=10u ad=0.3n as=0.3n pd=20u ps=20u
m4 5 4 8 8 pch w=10u l=10u ad=0.3n as=0.3n pd=20u ps=20u
m5 3 7 9 9 nch w=38u l=10u ad=0.3n as=0.3n pd=40u ps=40u
m6 6 5 8 8 pch w=344u l=10u ad=1.3n as=1.3n pd=400u ps=400u
m7 6 7 9 9 nch w=652u l=10u ad=2.3n as=2.3n pd=660u ps=660u
m8 7 7 9 9 nch w=38u l=10u ad=0.3n as=0.3n pd=40u ps=40u
cc 5 6 4.4p
ibias 8 7 8.8u
.ends opamp
```

Opamp Circuit 2 (opamp1.cir)

This example uses Spectre’s SPICE syntax.

```
.subckt opamp 1 2 6 8 9
m1 4 2 3 3 nch w=20u l=0.5u ad=0.3n as=0.3n pd=50u ps=50u
m2 5 1 3 3 nch w=20u l=0.5u ad=0.3n as=0.3n pd=50u ps=50u
m3 4 4 8 8 pch w=20u l=0.5u ad=0.3n as=0.3n pd=20u ps=20u
m4 5 4 8 8 pch w=20u l=0.5u ad=0.3n as=0.3n pd=20u ps=20u
m5 3 7 9 9 nch w=20u l=0.5u ad=0.3n as=0.3n pd=40u ps=40u
m6 6 5 8 8 pch w=20u l=0.5u ad=1.3n as=1.3n pd=350u ps=350u
m7 6 7 9 9 nch w=652u l=10u ad=2.3n as=2.3n pd=660u ps=660u
m8 7 7 9 9 nch w=38u l=10u ad=0.3n as=0.3n pd=40u ps=40u
cc 5 6 4.4p
ibias 8 7 8.8u
.ends opamp
```

Original Open-Loop Opamp (openloop.sp)

```
* Allen & Holmberg, p. 438 - Original Open Loop OpAmp Configuration
vinp 1 0 dc 0 ac 1.0
vdd 4 0 dc 5.0
vss 0 5 dc 5.0
vinm 2 0 dc 0
c1 3 0 20p
x1 1 2 3 4 5 opamp
** Bring in opamp subcircuit
include "opamp.cir"
** Bring in models here
.model nch bsim3v3
.model pch bsim3v3 type=p
.op
simulator lang = spectre
tf (3 0) xf save=lv1pub nest1lv1=1 start=1 stop=1K dec=20
simulator lang = spice
.dc vinp -0.005 0.005 100u
.print dc v(3)
```
.ac dec 10 1 10MEG
.print ac vdb(3) vp(3)
.end

**Modified Open-Loop Opamp (openloop1.sp)**

* Allen & Holmberg, p. 438 - Modified Open Loop OpAmp Configuration

vinp 1 0 dc 0 ac 1.0
vdd 4 0 dc 5.0
vss 0 5 dc 5.0
vinm 2 0 dc 0
cl 3 0 20p
x1 1 2 3 4 5 opamp
** Bring in opamp subcircuit
include "opamp1.cir"
** Bring in models here
/model nch bsim3v3
/model pch bsim3v3 type=p
.op
simulator lang = spectre
tf (3 0) xf save=lvlpub nestlvl=1 start=1 stop=1K dec=20
simulator lang = spice
.dc vinp -0.10 0.10 10u
.print dc v(3)
.ac dec 10 1 10MEG
.print ac vdb(3) vp(3)
.end

**Example Model Directory (q35d4h5.modsp)**

//example model directory
simulator lang = spectre
library mosmodels
section tt
/model n bsim3v3 tox=1.194e-08
/model p bsim3v3 type=p tox=7.4e-09
endsection
section ss
/model n bsim3v3 tox=1.242e-08
/model p bsim3v3 type=p tox=7.724e-09
endsection
section ff
/model n bsim3v3 tox=1.1544e-08
/model p bsim3v3 type=p tox=7.148e-09
endsection
endlibrary
Dynamic Loading

The Spectre® circuit simulator supports dynamic loading of device models. This feature allows you to dynamically load device primitives (stored in shared objects) at run time. This is useful for developing and distributing models.

Configuration File

The Spectre circuit simulator can be configured to load a specific set of shared objects based on the content of a set of configuration files. The default CMI (compiled-model interface) configuration file is shown below.

; The default search path is
; $CDS_ROOT/tools/spectre/lib/cmi/%M
; This file is automatically generated.
; Any changes made to it will not be saved.

load libnortel.so
load libphilips.so
load libsiemens.so
load libstmodels.so

The CMI file allows legal UNIX file paths and Spectre predefined percent codes. For more information on predefined percent codes, see “Description of Spectre Predefined Percent Codes” on page 248.

Configuration File Format

The following commands can be used in the configuration file:

<table>
<thead>
<tr>
<th>Command</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>setpath</td>
<td>Specifies the search path.</td>
</tr>
</tbody>
</table>
The following examples show the syntax for these commands.

**To specify a search path:**

```
setpath path [path2 ... path N]
```

Example 1:
```
setpath $HOME/cds/4.4.6/tools.$CDS_SYSNAME/spectre/lib/cmi/%M
```

Example 2:
```
setpath ($HOME/myLib/cmi/%M $HOME/cds/4.4.6/tools.$CDS_SYSNAME/spectre/lib/cmi/%M)
```

**To prepend a path:**

```
prepend path [path 2 ... path N]
```

Example 1:
```
prepend $HOME/myLib/cmi/%M
```

Example 2:
```
prepend ($HOME/myLib/cmi/%M $HOME/cds/4.4.6/tools.$CDS_SYSNAME/spectre/lib/cmi/%M)
```

**To append a path:**

```
append path [path2 ... path N]
```

Example 1:
```
append $HOME/myLib/cmi/%M
```

Example 2:
```
append ($HOME/myLib/cmi/%M $HOME/cds/4.4.6/tools.$CDS_SYSNAME/spectre/lib/cmi/%M)
```

The default search path is the path to the directory that contains Spectre shared objects:
```
$CDS_ROOT/tools/spectre/lib/cmi/CMIVersion
```
To load a shared object:
load [path/] soname.ext

Example 1:
load libnortel.so

Example 2:
load $HOME/myLib/cmi/%M/libmydevice.so

To unload a shared object:
unload [path/] soname.ext.version

Example 1:
unload libsiemens.so.1

Example 2:
unload $HOME/myLib/cmi/%M/libmydevice.so

The name of the shared object file includes an extension and can also have a version number. The path to the shared object is optional. If you do not specify the path, the Spectre simulator uses the search path from the current configuration file.

A line that begins with a semicolon is a comment and is ignored. Empty lines are allowed and are ignored.

Precedence for the CMI Configuration File

The Spectre simulator reads configuration files in the following order:

- The default Cadence CMI configuration file
- The configuration file specified by the value of the $CMI_CONFIG environment variable
- The file $HOME/.cmiconfig, if it exists
- The CMI configuration file specified in the -cmiconfig argument

Each configuration file modifies the previous configuration.
Dynamic Loading

Configuration File Example

This section contains examples that show how configuration files can be used to customize the list of shared objects that the Spectre circuit simulator loads at run time. The default configuration file includes `libnortel.so`, `libstmodels.so`, `libphilips.so`, and `libsiemens.so`.

If you need only the ST models, you can create a configuration file called `site_cmi_config` that loads only `libstmodels.so` by unloading the other three shared objects:

```
;default search path is $CDS_ROOT/tools/spectre/lib/cmi/%M
;only libstmodels for this site
;this file is called site_cmi_config
unload libnortel.so
unload libphilips.so
unload libsiemens.so
```

When the environment variable `$CMI_CONFIG` is set to `site_cmi_config`, only `libstmodels.so` is loaded.

A model developer can create a file `$HOME/myLib/libmybjt.so` consisting of the BJT model under development. To check the results of the BJT under development in
libmybjt.so with the BJT503 models in libphilips.so, the developer can create a CMI configuration file in the home directory as follows:

;this is $HOME/.cmiconfig file
    ;I want to include libphilips.so released by Cadence so that
    ;I can check my BJT with BJT503.
    load libphilips.so
    ;I also want to include my BJT model from libmybjt.so
    append $HOME/myLib
    load libmybjt.so

CMI Versioning

The version format for CMI is major.minor. The value of major is increased when there are major changes that require CMI developers to recompile their components.

Type spectre -cmiversion to display the current CMI version.

The Spectre circuit simulator checks for CMI version compatibility for each shared object as well as for each primitive. This ensures that

- A shared object is compiled with the latest version of CMI
- The source code for each device primitive is up to date

Note: A primitive can be installed only once. Different versions of the same primitive cannot be used.
Index

Symbols

% codes 233
  predefined 248
% pre-defined percent code 249
... in syntax 15
//, in Spectre syntax 59
:: colon modifier 251
[] in syntax 14
[] in vectors 78
\, in Spectre syntax 59
{} in syntax 15
| in syntax 14

A

%A pre-defined percent code 248
absolute error tolerance 142
abstol parameter 74, 142
AC analysis 139
AC analysis, brief description of 129
accuracy
correcting problems 277
improving 18
user control of tolerances 18
AHDL variables, saving 222
AHDL, definition 20
alarm parameter 270
algebraic functions 86
all
  as homotopy parameter option 246
  as info statement parameter 193, 223
  as pwr option 206, 218
  as save parameter option 208, 220
allglobal (relative error parameter) 141, 143
alllocal (relative error parameter) 141, 142
allpub, as save parameter option 208, 220
alter statement 67, 186, 244, 245
  changing parameter values with example 184, 243
  formatting 184, 243
  use with circuit parameters 185, 244
  example of use in netlist 109
altergroup statement
  changing parameter values with 184, 244
  using 67
Analog Artist
  Calculator 34
  case sensitivity of scale factors 81
  use in mixed-signal simulation 23
  use of with Spectre 20
Waveform Window 36
Analog Waveform Display (AWD)
  brief description of 14, 20
  definition of 24
  using 32
analog workbench design system, use of with Spectre 23
analyses
  AC 139
  basic rules 65
  brief description of types 129
  DC 138
  envelope following 22, 130
  example of analysis statement 30
  formatting 64, 65
  fourier 150
  handling of prerequisite 66
  Monte Carlo, performing 161
  no default analysis 66
  order performed 66
  parameter sweep examples 160
  parameter value defaults 133
  rules for naming 66
  sens 147
  setting probes 133
  specifying 30, 64, 133
  statistical 161
  subcircuits composed of
calling 137
  formatting example 137
analysis order, netlist example of effective use 134
analysis statement 64
  examples 30, 65
  formatting 64
annotate parameter
  in Monte Carlo analysis 168
colons (:), in Spectre syntax 59
Command Interpreter Window (CIW) 34
comments, formatting for 59
compatible parameter 200
compiled model interface (CMI),
   description 19
components
   naming 28, 59
   rules for names 59
   specifying initial conditions for 187, 238
Composer-to-Spectre direct simulation
   environment 20
conditional if statements
   example of use in netlist 112
   formatting example 106
conditional instantiation 107
   specifying 106
   warning 108
configuration file
   default 291
   example 294
   format 291
   precedence 293
connections, inherited 75
constants, built-in, table of 87
continuation characters 50, 51
continuation methods 245
control statements 66, 67
   definition 29
   formatting 67
   paramset 200
   statistics 211
   use of in instance statement 29
controls, specifying 66
conventions, typographic and syntax 14
convergence problems 274
   backward Euler integration method to
      correct 276
   correcting with current probes 275
   correcting with DC sweeps 276
   dividing the circuit into parts to
      correct 276
   floating resistors causing 275
   gmin parameter, increasing 275
   loosening iabstol parameter to
      correct 275
   loosening truncation error criteria to
      correct 276
   iteratio parameter 276
maxiters parameter 275
nodesets to correct 276
nonlinear component models, simplifying 276
numeric pivoting in the sparse matrix
   factorization 276
options statement settings to
   prevent 275
oscillators causing 276
pivotdc parameter, settings to
   correct 276
pivrel parameter, resetting to
   correct 276
realistic device models needed 275
reduced in Spectre 18
region parameter settings in bipolar analog circuits causing 276
region parameters of transistors and diodes 276
replacing DC analysis with transient analysis to correct 276
restarting to correct 276
rforce parameter, increasing with
   nodesets 276
step size 275
system messages helpful 274
temperature sweeps to correct 276
topcheck parameter enabling transient analysis 277
unusual parameter values, checking for 275
corners, example 111
correlation coefficients 175
correlation statements 174
CPP (C preprocessor)
defaults 233
   using the include statement with 70
crigm parameter 177
crids parameter 177
criuo parameter 177
crivth parameter 177
.cshrc file
   and environment defaults 234
   Cadence range limits 263
current probes
   correcting convergence problems with 275
   probe statement 205, 217
   saving individual currents with 204, 216
   setting multiple 209, 221
   setting with the save statement 205,
The Spectre Circuit Simulator User Guide

- When to use: 204, 216
- Currents parameter:
  - Example: 209, 221
  - Formatting: 209, 221
- Not used in subcircuit calls: 203, 214
- Options statement: 209, 221
- Customer service, contacting: 19, 256
- Customizing:
  - Error and warning messages: 262
  - Percent codes: 249

D

- %D predefined percent code: 249
- Damped pseudotransient method: 245
- Data compression: 146
- DataDir command line option: 32
- DC analysis: 138
  - Brief description of: 129
  - Correcting accuracy problems: 277
  - Maxiters parameter, and convergence problems: 275
  - Oppoint parameter: 157
  - Transfer curves: 157
- DC convergence problems, correcting: 274
- Dec parameter: 160
- Defaults:
  - Analyses: 66
  - Analysis parameters: 133
  - C preprocessor (CPP): 233
  - Cadence range limits for warnings: 263
  +Checkpoint spectre command option: 232
  - Command line: 233
  - Controlling destination and format of Spectre results: 233
  - Controlling system-generated messages: 233
  - Creating checkpoints and initiating recovery: 233
  - Measurement units of parameters: 116
  - Name of the simulator: 233
  - Overriding in UNIX environment variables: 235
  - Percent codes: 233
  - Screen display: 233
  - Setting for environment with SPECTRE_DEFAULTS: 263
  - Simulation environment: 233

spectre command:
  - Changing: 234
  - Examining: 234
  - Defining a library: 73
  - Degradation parameters (for hot-electron degradation simulation): 177, 178
- Design Framework II, use of with Spectre: 23
- Designer’s Guide to SPICE and Spectre: 14
- Dev parameter: 158
- Devices:
  - As pwr option: 206, 218
  - Diagnosis mode, use of: 20
  - Differential amplifier, example of input file: 109
- Direction, associated reference: 58
- Directory names:
  - Generating: 226
  - Specifying your own: 228
- Displaying a waveform with awd: 36
- Donominal parameter, Monte Carlo analysis: 167
- Double slash (//), in Spectre syntax: 59
- Dptran, as homotopy parameter option: 246
- Dracula, use of with Spectre: 23
- Duoc parameter: 177, 178
- Duoe parameter: 177, 178
- Dvthc parameter: 177, 178
- Dvthe parameter: 177, 178

E

- :e colon modifier: 251
- Electrical current in Amperes: 74
- Electrical potential in Volts: 74
- End: 50
- Environments, setting parameter limits with: 79
- Envelope following analysis: 22, 130
- Environment variables, changing defaults in: 234
- Environments in which Spectre can be used: 23
- Envlp: 22, 130
- Equal sign (=), in Spectre syntax: 59
- Error conditions (terminate simulation): internal errors: 256
  - Invalid parameter values: 254
  - Singular Jacobian or matrix: 255
error messages caused by invalid parameters in subcircuit calls
specifying conditions for in subcircuit calls
error options of spectre command
error tolerances
abstol parameter (error tolerance parameter)
erp established parameter
iteratio (error tolerance parameter)
refig (error tolerance parameter)
reltol parameter (error tolerance parameter)
erp established parameter
escaping special characters
estimating solutions with the nodeset statement
euler (integration method parameter)
example in netlist statement conditional if statement
parameterized inline subcircuit
paramtest statement
specifying N-ports
estimating solutions with the nodeset statement
exit codes, Spectre
explosion region warnings
expressions as model parameters
in subcircuit parameters
extending statements beyond a single line, formatting for

F

failure criteria parameters (for hot-electron degradation simulation)
field separators (input language syntax)
file formats, defining output
filenames
creating by modifying input filenames
creating from parts of input filenames
extension

S-parameter files
S-parameter state, reading
firstrun parameter, Monte Carlo analysis

using floating resistors, and convergence problems
forbidden operating regions for transistors, specifying
formatting
alarm parameter
analyses, composed of subcircuits
analysis parameters
analysis statements
check statement
colon modifiers
comment lines
conditional if statement
control statements
currents parameter
customized percent codes
ic statement
identical components in parallel
identical subcircuits in parallel
include statements
info statement
instance statements
line extension
model statements
models for multiple components
nestlvl parameter
nodeset statement
options statement
parameter range limits file
parameter sweeps
paramtest specification
probe statement
pwr parameter
rules for names
save parameter
save statement
scaling parameter values
scaling physical dimensions
sens command
set statement
S-parameter files
S-parameter specifying parameter values
spectre command
starting a simulation
state files
write parameter
writefinal parameter
subcircuit calls
subcircuit definitions
component synopsis 151
model definition 151
Fourier analysis improvements in 17
fourier analysis 150
freq parameter 158
functions
  algebraic 86
  hyperbolic 86
  trigonometric 86
functions, user-defined 89

G
GaAs traveling wave amplifier, sample input file 92
gauss parameter 172
Gaussian distribution 172
gear2 (integration method parameter) 141, 143
gear2only (integration method parameter) 141, 143
gmin parameter
  increasing value with convergence problems 275
  resetting to correct error conditions 255
  setting to correct accuracy problems 277, 278
  warning about size of 259
  gmin stepping 245
  gmin, as homotopy parameter option 246
  ground, definition of 58

H
:h colon modifier 251
%H predefined percent code 249
hard limit, definition 78
homotopy parameter 246
hot-electron degradation, simulating 175
circuitage parameter 176
degradation parameters 176, 177
example 178
failure criteria parameters 177
options and restrictions 175
results files 177
specifying the analysis 176

HPMNS format, translating 120
huge parameter 74
hyperbolic functions 86

I
i 74
iabstol parameter 200
  convergence problems and 275
  setting to correct accuracy problems 277
ic parameter, in transient analyses 187, 238
ic statement example 188, 238
  formatting 188, 238
  specifying initial conditions 187, 238
if statements 106
imelt 258
#include statements examples 70, 71
  formatting 70
include statements 70
  examples 71
    formatting 70
    of use in netlist 109
    rules for using 70
    initial settings of the state of the simulator, modifying 186, 245
    info (+/-), spectre command option 274
    info statement 192, 222
      choosing the output destination 194, 223
      example 194, 224
      formatting 194, 224
      options 193, 223
    informational messages, specifying conditions for in subcircuit calls 95
    infotime 144
    inheritance 75
    inherited connections 75
    initial conditions, specifying 187, 237
      examples 187, 188, 238
      individual analyses 190, 240
      transient analyses 187, 238
    initial settings of the state of the simulator, modifying 186, 245
    inline model statements, inside of inline subcircuits 100
    inline subcircuits containing inline model statements 100
maxdelta parameter 74
maximum and minimum parameter values, listing 192, 223
maxiters parameter 275
maxstep parameter 141, 144
setting to correct accuracy problems 278
MCNC benchmark suite 19
measuring
differential signals, example in netlist 109
loop gain, example in netlist 109, 134
output resistance, netlist example discussed 135
power supply rejection, netlist example discussed 135
melting current warnings 258
message control
specifying the destination of 273
suppressing 274
method parameter 141
euler setting 143
gear2 setting 143
gear2only setting 143
setting to correct accuracy problems 278
trap setting 143
trapgear2 (integration method parameter) 143
traponly setting 143
MHARM format, translating 120
minimum timestep used, fixing warning 260
mismatch block 170
mismatch correlation statement 174
missing diode would be forward-biased (warning message) 258
mixed-signal simulation 23
MMF 74
mod parameter 158
model
definition (fourier) 151
model binning rules 108
model scaling factor, changing with alter statement 186, 245
model statement 29
examples 29, 69
formatting 68
inline 100
using 68
modeling
identical components in parallel 63, 71
identical subcircuits in parallel 64
multidisciplinary 73
N-ports 117
parasitics, in inline subcircuits 97
process, using inline subcircuits 101
models
as info statement parameter 193, 223
charge conservation of 17
formatting for multiple components 62
selecting automatically 105
use of expressions in 70
Monte Carlo analysis
brief description of 133
characterization 175
elements of 17
examples 169
modeling 175
parameters, table of 163
performing 161
specifying statistics 174
specifying the first iteration number 168
MOS models, advantages of Spectre
version 17
MOS0 model, brief description 17
multidisciplinary modeling 73
multiple analyses, example in netlist 109
multiple components, creating models for 68
multiple directories, accessing data from 40
multiplication factor 63

N
terminals (formatting in analysis statements) 65
names, rules for
components 59
nodes 59
old netlists 61
nestlvl parameter 207, 219
netlist
canventions 58
definition 26
elements of 27
introduction to 27
Spectre example 26
netlist parameters, predefined 90
netlist statements 57
use of in example 28
newlink keywords 202, 214
newlink save statement 201, 213
Newton-Raphson iteration 245, 255
node capacitance table 194
nodes 187, 237
as info statement parameter 193, 223
definition of 58
naming 59
rules for names 59
specifying initial conditions with 188, 238
nodeset statement 189, 239
nodesets 187, 237
convergence problems and 276
specifying for individual analyses 190, 240
node-to-terminal map 193, 223
noise analysis, brief description of 130
none
as homotopy parameter option 246
as info statement parameter 193, 223
as pwr option 206, 218
as save parameter option 207, 219
nonlinear component models, and convergence problems 276
notifications, responding to 31
nport component, and S-parameter files 120
nport statement, example in netlist 117
N-ports example 117
modeling 117
numerical error, improved control of 17
numruns parameter, Monte Carlo analysis 163
nutascii (output format) 224
nutbin (output format) 224
Nutmeg format 224

O
online help 20
open loop gain, in netlist example of measuring 134
operating regions determining for BJTs 271
selecting limits for warnings about specifying forbidden regions for
transistors 270
operating-point parameters listing 192, 222
saving with save statement 202, 214
operational amplifier, example of characterization in one simulation run 134
oppoint as DC analysis parameter 157
as info statement parameter 193, 223
options statement ckptclock option example 232
correcting convergence problems 275
defining output file formats 225
element 199
formatting 199
generating output directory names 228
overriding environment defaults 235
parameters
compatible 200
iabstol 200
rawfile 228
rawfmt 225
reltol 199
tempeffects 200
topcheck 255, 275
vabstol 200
resetting options 245
setting tolerances with 199
OR-bars in syntax 14
order of analyses, netlist example of effective use 134
output
as info statement parameter 193, 223
data, controlling the amount of
outputstart parameter 144
skipping 144
strobing 144
directory names
how Spectre generates 226
specifying your own 228
file format options 224, 225
filenames, how Spectre generates 226
parameters, listing 192, 222
viewing 32
output resistance, netlist example of
measurement discussed 135
outputstart parameter 144
overriding defaults 235
%P predefined percent code 249
p terminals (formatting in analysis statements) 65
pac analysis, brief description of 131
param parameter 158
+param, spectre command option 263
parameter checking, stopping 186, 269
parameter range checker
convergence problems 275
parameter range limits file
creating 263
absolute value specifications 267
exclusive boundaries in 264
inclusive boundaries in 264
model specifications 267
entering 268
example 267
parameter storage format (PSF) 224
formatting output files 235
parameter sweeps
dev parameter 158, 159, 200
examples
linear sweep 160
logarithmic sweep 160
sweeping temperature 160
with vector of values 160
formatting 160
freq parameter 158, 159, 200
mod parameter 158, 159, 200
param parameter 158, 159, 200
setting sweep limits 159
specifying 156
temp parameter 158
with vector of values 161
parameter values
changing for components 184, 243
changing for models 184, 244
checking 186, 187, 269, 270
all parameter values 187, 270
in instance statements 186, 269
in model statements 186, 269
in subcircuit calls 95
operating-point parameters 187, 270
stopping 186, 269
controlling the number of checks on 269
example of altering in netlist 109
general rules for specifying 82
in instance statements 29
invalid 254
listing 192, 222
permissible types 77
scaling 81
testing 271
vector 78
with arithmetic operators 84
with Boolean operators 84
with built-in constants 87
with user-defined functions 89
parameterized inline subcircuits
example of use in netlist 112
parameterized models, using inline subcircuits 99
parameters 82
circuit 82
Monte Carlo analysis, table of 163
predefined netlist 90
subcircuit 82
parameters statement 82
paramfile parameter, Monte Carlo analysis 164
paramset 158
paramset statement 200
paramset statement, using 159, 200
paramtest statement
checking subcircuit parameter values with 95
checking values of subcircuit parameters 271
errorif option 272
example 272
example of use in netlist 112
formatting 271
message option 272
printif option 272
severity option 272
warnif option 272
parasitic elements
in parallel with device terminals 99
in series with device terminals 99
parasitics, modeling in inline subcircuits 97
parentheses (()), in Spectre syntax 59
pdisto analysis, brief description of 22, 131
percent codes 233
customizing 249
example 250
formatting 250
removing customized settings 250
predefined quantities 74
predefined netlist parameters 90
predefined percent code
% character 249
CMIVersion 249
date 249
described 248
host name (network name) 249
input circuit file name 249
process ID 249
program starting time 249
simulator name 249
substitution in 249
version string 249

redirected files 250
use with piped files 250
percent parameter, Monte Carlo
analysis 171, 173
periodic AC analysis (pac), brief description of 131
periodic distortion analysis (pdisto), brief
description of 131
periodic noise analysis (pnoise), brief
description of 131
periodic small-signal analyses,
description 21
periodic steady-state analysis (pss), brief
description of 21, 131
periodic transfer function analysis (pxf), brief
description of 131
physical dimensions of components,
scaling 115
PID (process identification number) 230
use of 163
piecewise linear (PWL) vector values,
reading 72
piped files, and percent codes 250
P-N junction warnings 257
breakdown region 258
explosion region 257
missing diode would be forward-biased 258
pnoise analysis, brief description of 131
pointlocal (relative error parameter) 142
power in Watts 74
power supply rejection, netlist example of
measurement discussed 135
power, saving 205, 217
ps utility, UNIX 230
psfascii (output format) 224
psfbin (output format) 224
pss analysis, brief description of 131
ptran, as homotopy parameter option 246
punctuation characters, in Spectre
syntax 59
PWL, definition 72
Pwr 74
pwr 203, 205, 214, 217
pwr parameter 206, 217
pxf analysis, brief description of 131

Q
quantity statement 74
abstol parameter 74
Spectre Circuit Simulator User Guide

blowup parameter 74
example 75
huge parameter 74
maxdelta parameter 74
predefined quantities 74

R
:r colon modifier 251
range checking for subcircuit parameters 271
reading state files 191, 241
+recover, spectre command option 233
recovery from transient analysis terminations
automatic recovery, customizing 232
restarting a transient analysis 233
setting recovery (checkpoint) file specifications for a single analysis 233
recursive inclusion 71
redirected files, and percent codes 250
reducing simulation time with previous solutions 237
reference direction, associated 58
reference to a library 73
relative error tolerance 142
relref parameter 141, 142
allglobal setting 143
alllocal setting 142
pointlocal setting 142
sigglobal setting 143
reltol parameter 141, 142, 199
setting to correct accuracy problems 277
reserved words, Spectre 59
restart parameter, in analyses 237
results
controlling destination and format of 233
displaying 32
Results Browser 36
RF capabilities 21
rforce parameter
and convergence problems 276
setting to correct accuracy problems 278
run terminations. See terminations of Spectre 230
running a simulation, introductory

example 30

S
%S predefined percent code 249
%S_DEFAULTS 234
sample input file
differential amplifier 109
GaAs traveling wave amplifier 92
process file 112
two port test circuit 117
uA741 operational amplifier 134
save parameter
example 206, 207, 217, 219
formatting 207, 219
options 207, 219
setting with an analysis statement 207, 219
setting with the options statement 207, 219
save statement 201
examples 203, 215
formatting 201, 213
keywords 202, 203, 214
operating-point parameters of individual components, saving 202, 214
saving signals
by keywords 202, 214
by terminal index 202, 214
by terminal name 202, 213
of individual components 202, 213
of individual nodes 202, 213
of individual subcircuits 203, 215
voltages of individual nodes, saving 202, 213
saveahdlvars 222
savefamilyplots parameter, Monte Carlo analysis 168
saveprocessparams parameter, Monte Carlo analysis 164
saveprocessvec parameter, Monte Carlo analysis 166
saving
all ahdl variables 222
currents for AC analysis 204, 216
groups of currents
with the currents parameter 208, 220
with the save and nestlvl parameters 207, 219
groups of signals
  in main circuits with the save parameter 207, 219
  in subcircuits 208, 220
  in subcircuits with the nestlvl parameter 207, 219
 individual currents
  with current probes 204, 216
  with the save statement 201, 213
 individual signals of subcircuits 203, 215
 main circuit signals. See save statement 201, 213
measurements for current-controlled components 204, 216
parameter values 192, 222
signals from subcircuit calls 203, 214
saving power 205, 217
scalarfile parameter, Monte Carlo analysis 164
scale factors 81
scale parameter
devices scaled by 115
how affected by default units of components 116
in alter statement 186, 245
scalem parameter
devices scaled by 115
how affected by default units of component 116
in alter statement 186, 245
scaling
parameter values
in Spectre 81
in SPICE 81
physical dimensions of components 115
schematic 25
screen printout, example 31
second-order Gear integration method 143
seed parameter, Monte Carlo analysis 163
selected, as save parameter option 207, 219
sens analysis 147
sens command
  example 149
  formatting 148
sensitivity analysis 147
set statement, to modify options statement settings 245
setting tolerances with the quantity statement 74
shell initialization file, and Cadence range limits 263
shell status variable 230
sigglobal (relative error parameter) 141, 143
Signal Scan output format 224
signals, defined 212
simulation
  checking status 230
  following the progress of 31
  improved accuracy 18
  interrupting 231
  introductory Spectre run 30
  sample narration of progress 31
  specifying options 229
  starting 229
  termination. See terminations of Spectre 230
viewing output 32
simulation language, use of in example netlist 28
simulator lang command
default 53
singular Jacobian or matrix (error message) 255
skipping data points
  skipcount parameter 145
  skipstart parameter 145
  skipstop parameter 145
skipping time points
  example 146
  explanation 144
soft limits
  definition 78
  use of 19
source stepping 245
source, as homotopy parameter option 246
span parameter 159
S-parameter analysis (sp), brief description of 130
S-parameter file format translator 120
S-parameter files
  creating manually 118
db-deg 118
db-rad 118
example 119
formatting 118
generated by Spectre 118
mag-deg 118
and writefinal parameter  190, 241
creating manually  191, 241
reading  191, 241
special uses for  191, 241
using to specify initial conditions and estimate solutions  190, 240
write parameter  190, 241
state information, specifying
  initial conditions  187, 237
  nodesets  187, 237
state of the simulator, modifying initial settings  186, 245
statements
  analysis  64
    formatting  64, 65
  control  66
  instance  61
  library  72
  model  68
    examples  69
    formatting  68
  netlist  57
  paramset  200
  save  201
  statistics  211
statistical analysis, performing 161
statistics blocks  161
  multiple, Monte Carlo analysis  174
  specifying parameter distributions  170
statistics statement  211
status, checking simulation 230
step parameter  144, 160
step size for DC sweep, and convergence problems  275
stop parameter  159
stopping a simulation 231
string parameters, specifying 117
strobing  144
  example  146
  skipstart parameter  145
  skipstop parameter  145
  strobedelay parameter  145
  strobeperiod parameter  145
stty utility, UNIX  231
sub parameter  158
subcircuit calls
  checking for invalid parameter values in  95
  example  94
  formatting  94
  names unrestricted  95
saving signals  203, 214
subcircuit parameters  82
subcircuits
  calling  90, 94
  composed of analyses
    calling  137
    example  137
    formatting  137
  defining  90
    example  91, 92
    formatting  90
  expressions within  93
  inline  96
  multiplication factor in  64
parameters  82
  range checking for parameter values  271
  saving groups of signals  208, 220
  saving individual signals  203, 215
subckts
  as pwr option  206, 218
  suppressing messages  274
  sweep limits, setting  159
  sweeping parameters. See parameter sweeps 156
  swept periodic steady-state (SPSS) analysis, brief description of 21
synopsis
  (fourier)  151
syntax conventions  58
system-generated messages, controlling  233

T

:t colon modifier  251
%T predefined percent code  249
tdr analysis, brief description of  130
Temp  74
temp parameter  90, 158
  in alter statement  186, 244
tempeffects parameter  200
temperature
  changing with alter statement  186, 244
  in Celsius  74
terminal index
  subcircuit calls  203, 214
  use in save statement  202, 214
terminal name, use in save statement  202, 213
terminals, as info statement
parameter 193, 223
terminal-to-node map 193, 223
terminations of Spectre
because of a Spectre error
condition 230
by the operating system 230
manual 231
normal 230
recovering from transient analysis 231
with an error in an analysis 230
testing values of subcircuit parameters 271
time not increasing, error message 257
time-domain reflectometer (tdr) analysis,
brief description of 130
time-step adjustment 144
time-step control algorithm, advantages of 17
timestep used, minimum 260
title card 27
title line 27, 51
title parameter, in Monte Carlo
analysis 168
tnom parameter 90
in alter statement 186, 244
tolerances, setting, with the options
statement 199
Toshiba model (for hot-electron degradation
simulation) 176
total
as pwr option 206, 218
tran analysis, brief description of 130
transfer curves 157
transfer function analysis (xf), brief
description of 130
transient analysis
accuracy 143
brief description of 130
ckptperiod parameter example 277
convergence problems 277
correcting accuracy problems 278
error tolerances. See error
tolerances 142
ic parameter 187, 238
improving convergence 144
integration method 143
read parameter for state files 191, 241
readns parameter for state files 191, 241
restarting 233
specifying initial conditions in 187, 238
speed 143
terminations, recovering from 231
transistors
requesting breakdown region warnings
for 269
specifying forbidden operating regions
for 270
trap (integration method parameter) 143
trapezoidal rule integration method 143
traponly (integration method
parameter) 141, 143
trigonometric functions 86
truncate statement 173
truncation factor 173
tutorial for using Spectre 24
two-port test circuit, sample input file 117
typographical conventions 14

U
U 75
uA741 operational amplifier, sample input
file 134
uniform parameter 172
uniform distribution 172
unitless 75
UNIX commands, and environment
defaults 234
unload shared object 293
unusual parameter size warnings 259
usability features 19
user-defined functions 89

V
V 74
%V predefined percent code 249
vabstol parameter 200
setting to correct accuracy
problems 277
values parameter 160
variations parameter, Monte Carlo
analysis 166
vbcfwd parameter, and determining
operating regions for BJTs 271
vbefwd parameter, and determining
operating regions for BJTs 271
VCO, definition of 21
vector parameter values 78
vector values, piecewise linear 72
Verilog, coupling with Spectre 23
Verilog-A 13
vertical bars in syntax 14
viewing output 32

W

warning messages 19
about size of gmin 259
caused by invalid parameters in subcircuit calls 95
customizing 262
generating about transistor operating regions 270
options of spectre command 274
parameter is unusually large or small 259
P-N junction warnings 257
requesting breakdown region warnings for transistors 269
responding to 31
selecting limits
operating region warnings 270
parameter values 262
specifying conditions for in subcircuit calls 95
warnings
melting current 258
waveform display 36
waveform storage format (WSF) 224
Waveform Window 36
Wb 74
what parameter, of check statement 186, 269
write parameter, and creating state files 190, 241
writefinal parameter, and creating state files 190, 241
WSF (waveform storage format) 224
wsfascii (output format) 224
wsfbin (output format) 224

X

xf analysis, brief description of 130