WRspice Reference Manual

Whiteley Research Incorporated
456 Flora Vista Avenue
Sunnyvale, CA 94086

Release 2.2.64
4 September, 2006

© Whiteley Research Incorporated, 2006.

WRspice is part of the *XicTools* software package for integrated circuit design from Whiteley Research Inc. *WRspice* was authored by S. R. Whiteley, with extensive adaptation of the Berkeley SPICE3 program. This manual was prepared by Whiteley Research Inc., acknowledging the material originally authored by the developers of SPICE3 in the Electrical Engineering and Computer Sciences Department of the University of California, Berkeley. Technical support for registered users of *WRspice* and other *XicTools* software is available via electronic mail at `support@wrcad.com`, and by telephone at (408) 735-8973.

WRspice and subsidiary programs and utilities are offered as-is, and the suitability of these programs for any purpose or application must be established by the user, as neither Whiteley Research Inc., or the University of California can imply or guarantee such suitability.

This page intentionally left blank.

Contents

1 Introduction to *WRspice* **1**

1.1 History of *WRspice* . . . 1

1.2 *WRspice* Overview . . . 4

1.3 Types of Analysis . . . 6

1.4 Circuit Input and Program Control . . . 11

1.5 Post-Processing and Run Control . . . 11

1.6 Introduction to Interactive Simulation . . . 12

2 *WRspice* Input Format **19**

2.1 Input Format . . . 19

2.2 Circuit Description . . . 20

2.3 Title, Comments, and Inclusions . . . 21

2.3.1 Title Line . . . 21

2.3.2 `.title` Line . . . 21

2.3.3 `.end` Line . . . 22

2.3.4 Comments . . . 22

2.3.5 `.include` Line . . . 22

2.3.6 `.lib` Line . . . 23

2.4 Initialization . . . 23

2.4.1 `.global` Line . . . 23

2.4.2 `.ic` Line . . . 24

2.4.3 `.nodeset` Line . . . 24

2.4.4 `.options` Line . . . 25

2.4.5 `.table` Line . . . 26

2.5 Parameters and Expressions . . . 27

2.5.1 Single-Quoted Expressions . . . 27
2.5.2 .param Line . . . 27
2.5.3 .if, .elif, .else, and .endif Lines . . . 28
2.6 Subcircuits . . . 29
2.6.1 .subckt Line . . . 29
2.6.2 .ends Line . . . 30
2.6.3 Subcircuit Calls . . . 30
2.7 Analysis Specification . . . 31
2.7.1 .ac Line . . . 32
2.7.2 .dc Line . . . 32
2.7.3 .disto Line . . . 33
2.7.4 .noise Line . . . 35
2.7.5 .op Line . . . 36
2.7.6 .pz Line . . . 36
2.7.7 .sens Line . . . 37
2.7.8 .tf Line . . . 37
2.7.9 .tran Line . . . 38
2.8 Output Generation . . . 39
2.8.1 .save Line . . . 39
2.8.2 .print Line . . . 40
2.8.3 .plot Line . . . 40
2.8.4 .four Line . . . 40
2.8.5 .width Line . . . 41
2.9 Parameter Measurement . . . 41
2.9.1 .measure Line . . . 41
2.9.1.1 Interval Specification . . . 41
2.9.1.2 Measurements . . . 42
2.10 Control Script Execution . . . 45
2.10.1 .exec, .control, and .endc Lines . . . 45
2.10.2 .check, .checkall, .monte, and .noexec Lines . . . 46
2.11 Verilog Interface . . . 47
2.11.1 .verilog, .endv Lines . . . 47
2.11.2 .adc Line . . . 48

2.12 Voltage and Current Sources 49
2.12.1 Source Expressions 50
2.12.2 POLY Expressions 52
2.12.3 Tran Functions 53
2.12.3.1 Exponential 55
2.12.3.2 Gaussian Random 55
2.12.3.3 Interpolation 56
2.12.3.4 Pulse 56
2.12.3.5 Piecewise Linear 57
2.12.3.6 Single-Frequency FM 58
2.12.3.7 Sinusoidal 58
2.12.3.8 Sinusoidal Pulse 59
2.12.3.9 Table Reference 59
2.12.4 Dependent Sources 60
2.12.4.1 Voltage-Controlled Current Sources 60
2.12.4.2 Voltage-Controlled Voltage Sources 61
2.12.4.3 Current-Controlled Current Sources 62
2.12.4.4 Current-Controlled Voltage Sources 63
2.13 Element Lines and the Device Library 63
2.14 Passive Element Lines 65
2.14.1 Capacitors 65
2.14.2 Inductors 65
2.14.3 Coupled (Mutual) Inductors 65
2.14.4 Resistors 66
2.14.5 Switches 66
2.14.6 Transmission Lines 67
2.14.6.1 Model Level 67
2.14.6.2 Electrical Characteristics 68
2.14.6.3 Initial Conditions 69
2.14.6.4 Timestep and Breakpoint Control 69
2.14.6.5 History List 71
2.14.7 Uniform RC Line 71
2.15 Semiconductor Devices 72

2.15.1 Junction Diodes . . . 72
2.15.2 Bipolar Junction Transistors (BJTs) . . . 73
2.15.3 Junction Field-Effect Transistors (JFETs) . . . 73
2.15.4 MESFETs . . . 73
2.15.5 MOSFETs . . . 74
2.16 Superconductor Devices . . . 75
2.16.1 Josephson Junctions . . . 75
2.17 Device Models . . . 75
2.17.1 Analysis at Different Temperatures . . . 76
2.17.2 Capacitor Model . . . 78
2.17.3 Resistor Model . . . 78
2.17.4 Switch Model . . . 79
2.17.5 Transmission Line Model . . . 79
2.17.6 Uniform Distributed RC Model . . . 80
2.17.7 Diode Model . . . 80
2.17.8 BJT Models (both NPN and PNP) . . . 81
2.17.9 JFET Models (both N and P Channel) . . . 83
2.17.10 MESFET Models (both N and P Channel) . . . 84
2.17.11 MOSFET Models (both N and P channel) . . . 85
2.17.11.1 HSPICE MOS Level 49 Compatibility in *WRspice* . . . 86
2.17.11.1.1 General parameters . . . 86
2.17.11.1.2 Length and Width . . . 86
2.17.11.1.3 MOS Diode Model Parameters . . . 87
2.17.11.1.4 Capacitance Parameters . . . 87
2.17.11.1.5 Impact Ionization . . . 88
2.17.11.1.6 V3.2 parameters . . . 88
2.17.11.2 SPICE3 MOS Models . . . 88
2.17.12 Josephson Junction Model . . . 93

3 The *WRspice* User Interface 95

3.1 Starting *WRspice* . . . 95
3.2 Environment Variables . . . 97
3.2.1 Unix/Linux . . . 97
3.2.2 Microsoft Windows . . . 98

3.2.3 *WRspice* Environment Variables 98
3.3 Initialization Files 100
3.4 The Tool Control Window 101
3.4.1 The tbsetup Command 105
3.4.2 The update Command 105
3.4.3 The setfont Command 105
3.4.4 The Variable Setting Tools 106
3.4.5 The Circuits Tool 107
3.4.6 The Files Tool 107
3.4.7 The Fonts Tool 108
3.4.8 The Plots Tool 108
3.4.9 The Trace Tool 109
3.4.10 The Variables Tool 109
3.4.11 The Vectors Tool 109
3.5 The Mail Client 110
3.6 The File Manager 110
3.7 The Text Editor 112
3.8 The Plot Panel 115
3.9 The Mplot Panel 118
3.10 The Print Control Panel 118
3.10.1 Print Drivers 120
3.10.2 wr_lp 122
3.11 The *WRspice* Help System 122
3.11.1 The HTML Viewer 124
3.11.2 The Help Database 126
3.11.3 Help System Forms Processing 126
3.11.4 Help System Initialization File 127
3.12 The *WRspice* Shell 127
3.12.1 Command Line Editing 127
3.12.2 Command Completion 128
3.12.3 History Substitution 128
3.12.4 Alias Substitution 129
3.12.5 Global Substitution 129

3.12.6 Quoting . . . 130
3.12.7 I/O Redirection . . . 130
3.12.8 Semicolon Termination . . . 130
3.12.9 Variables and Variable Substitution . . . 130
3.12.10 Commands and Scripts . . . 132
3.13 Plots, Vectors and Expressions . . . 133
3.13.1 The `constants` Plot . . . 136
3.13.2 Operators in Expressions . . . 136
3.13.3 Math Functions . . . 138
3.13.4 Expression Lists . . . 142
3.13.5 Set and Let . . . 143
3.14 Control Structures . . . 146
3.15 Scripts and Batch Mode . . . 148
3.16 The *WRspice* Daemon and Remote SPICE Runs . . . 150

4 *WRspice* Commands 151

4.1 Shell Commands . . . 151
4.1.1 alias . . . 152
4.1.2 cd . . . 152
4.1.3 echo . . . 152
4.1.4 history . . . 152
4.1.5 mapkey . . . 153
4.1.6 pause . . . 154
4.1.7 rehash . . . 154
4.1.8 set . . . 154
4.1.9 shell . . . 155
4.1.10 shift . . . 155
4.1.11 unalias . . . 155
4.1.12 unset . . . 155
4.1.13 usrset . . . 156
4.2 Simulation Control Commands . . . 156
4.2.1 ac . . . 157
4.2.2 alter . . . 158
4.2.3 aspice . . . 158

4.2.4 check . . . 158
4.2.5 dc . . . 163
4.2.6 delete . . . 163
4.2.7 destroy . . . 164
4.2.8 disto . . . 164
4.2.9 dump . . . 164
4.2.10 free . . . 165
4.2.11 listing . . . 165
4.2.12 loop . . . 165
4.2.13 noise . . . 166
4.2.14 op . . . 166
4.2.15 pz . . . 166
4.2.16 reset . . . 166
4.2.17 resume . . . 166
4.2.18 rhost . . . 167
4.2.19 rspice . . . 167
4.2.20 run . . . 167
4.2.21 save . . . 168
4.2.22 sens . . . 169
4.2.23 setcirc . . . 169
4.2.24 show . . . 169
4.2.25 state . . . 170
4.2.26 status . . . 170
4.2.27 step . . . 171
4.2.28 stop . . . 171
4.2.29 tf . . . 171
4.2.30 trace . . . 172
4.2.31 tran . . . 172
4.2.32 version . . . 172
4.3 Data Manipulation Commands . . . 173
4.3.1 compose . . . 173
4.3.2 cross . . . 174
4.3.3 define . . . 174

4.3.4 deftype . . . 174
4.3.5 diff . . . 175
4.3.6 display . . . 175
4.3.7 fourier . . . 175
4.3.8 let . . . 176
4.3.9 linearize . . . 176
4.3.10 pick . . . 177
4.3.11 print . . . 177
4.3.12 seed . . . 178
4.3.13 setplot . . . 178
4.3.14 setscale . . . 179
4.3.15 settype . . . 179
4.3.16 spec . . . 180
4.3.17 undefine . . . 181
4.3.18 unlet . . . 181
4.4 Input and Output Commands . . . 181
4.4.1 codeblock . . . 182
4.4.2 edit . . . 183
4.4.3 load . . . 183
4.4.4 sced . . . 184
4.4.5 source . . . 184
4.4.6 write . . . 184
4.4.7 xeditor . . . 185
4.5 Graphical Output Commands . . . 185
4.5.1 asciiplot . . . 186
4.5.2 combine . . . 186
4.5.3 hardcopy . . . 187
4.5.4 iplot . . . 188
4.5.5 mplot . . . 188
4.5.6 plot . . . 190
4.5.7 xgraph . . . 191
4.6 Miscellaneous Commands . . . 192
4.6.1 bug . . . 192

4.6.2 jobs . . . 192
4.6.3 help . . . 192
4.6.4 qhelp . . . 193
4.6.5 quit . . . 193
4.6.6 rusage . . . 193
4.6.7 setrdb . . . 194
4.6.8 strcmp . . . 195
4.6.9 where . . . 195
4.7 Variables . . . 196
4.7.1 Shell Variables . . . 196
4.7.2 Command Specific Variables . . . 198
4.7.3 Plot Variables . . . 201
4.7.4 Simulation Option Variables . . . 207
4.7.5 Debugging Variables . . . 212

5 Margin Analysis 213
5.1 Operating Range Analysis . . . 213
5.2 Operating Range Analysis File Format . . . 218
5.2.1 Initializing Header . . . 218
5.2.2 Control Statements . . . 220
5.2.3 Circuit Description . . . 220
5.3 Example Operating Range Analysis Control File . . . 221
5.4 Monte Carlo Analysis . . . 225
5.5 Example Monte Carlo Analysis Control File . . . 226
5.6 Circuit Margin Optimization . . . 231

A File Formats 233
A.1 Rawfile Format . . . 233
A.2 Help Database Files . . . 235
A.2.1 Anchor Text . . . 239
A.2.2 `.mozyrc` File . . . 240
A.3 Example Data Files . . . 243
A.3.1 Circuit 1: Simple Differential Pair . . . 243
A.3.2 Circuit 2: MOS Output Characteristics . . . 243

A.3.3 Circuit 3: Simple RTL Inverter . . . 243
A.3.4 Circuit 4: Four-Bit Adder . . . 244
A.3.5 Circuit 5: Transmission Line Inverter . . . 245
A.3.6 Circuit 6: Function and Table Demo . . . 246
A.3.7 Circuit 7: MOS Convergence Test . . . 246
A.3.8 Circuit 8: Verilog Pseudo-Random Sequence . . . 248
A.3.9 Circuit 9: Josephson Junction I-V Curve . . . 249
A.3.10 Circuit 10: Josephson Gap Potential Modulation . . . 250

B Utility Programs **251**

B.1 The `multidec` Utility: Coupled Lossy Transmission Lines . . . 251
B.2 The `printtoraw` Utility: Print to Rawfile Conversion . . . 252
B.3 The `proc2mod` Utility: BSIM1 Model Generation . . . 252
B.4 The `wrspiced` Daemon: Remote SPICE Controller . . . 252
B.4.1 SPICE Server Configuration . . . 252
B.4.2 Client Configuration . . . 254

Chapter 1

Introduction to *WRspice*

1.1 History of *WRspice*

In the early days of radio, before the term "electronics" came into use, experimenters and scientists (engineers designed bridges then) built circuits on whatever could be found that was appropriate. To the dismay of otherwise supportive wives and mothers, one popular substrate was the wooden breadboard found in most kitchens. The breadboard could be used to secure the tube sockets and other appendages through use of screws. Thus, the term "breadboard" as a substrate for the building of electronic circuits was born.

Breadboards, in one form or another, were used for the construction of all electronic prototype circuits until the integrated circuit was invented in 1957. Even well afterward, the integrated circuit was only another component on the breadboard. A new circuit could be easily (well, in principle) debugged, modified, enhanced, or otherwise engineered toward perfection, as all points of the circuit were accessible for testing and measurement.

After it became possible to put more than a small number of devices on an integrated circuit chip, design of such chips became quite challenging. Obviously there was no opportunity to solder in new components, or even probe the internal connections. The circuit had to be perfect as designed, or it wouldn't work properly. This is almost never the case for any but the simplest design for any but those engineers with godlike intelligence (or luck). Clearly new tools and methods were needed.

The rescue came in the early 1970's with the distribution of a computer program called SPICE (Simulation Program for Integrated Circuit Engineering), developed at the University of California, Berkeley. SPICE was a big (for its day) Fortran computer program, requiring the power of a mainframe computer. A circuit was described in a certain syntax, which was punched into IBM computer cards. The terminology this inspired persists to this day, as a line of SPICE input is often referred to as a "card", and a complete circuit description as a "deck". In the bad old days, the engineer would laboriously punch the cards, deliver them to the Computer Center, and receive the line printer output a few hours later. It would generally take several iterations before the first page of output would not say "Run Aborted...". Old-time engineers abhorred this activity, which seemed suitable for office wimps and students only. After a few years, and the advent of Tektronix direct view storage terminals, SPICE became the preeminent means of designing not just integrated circuits, but the old-fashioned kind as well.

SPICE is the progenitor of most circuit simulators currently in use. As the source code was available

for next to nothing, major organizations customized it for their own needs, and smoothed over some of the rougher edges. It remains a numerically intensive program which can tax even the largest of today's computers.

Looking back, one can identify several important milestones in the accessibility of computer power to the technically inclined masses. The first example was the ability to run the BASIC programming language on a desktop computer, introduced in the late 1970's with the Tektronix 4051. This 8-bit machine with a built-in direct view storage tube for the first time allowed engineers and scientists to have directly applicable computer power at their desk or laboratory bench. Also at about this time, the UNIX operating system running on a VAX minicomputer became popular, largely supplanting the ahead-of-their-time desktop computers. The VAX, although it was a mainframe, was very cost effective as compared to the competition, and UNIX was a much more desirable operating system for scientific and engineering purposes than others available. As with the 4051, it allowed computer accessibility at the point of need, through use of a terminal. Although one could compile and run SPICE on a VAX, it would tend to radically hog the VAX's resources, bringing the system to an annoying unresponsive state. As a consequence, many system managers forbade the use of SPICE on their machines, thus SPICE users were still faced with interfacing to the company CDC or IBM or Cray, and the attendant CPU time charges and batch mode operation. This situation persisted until the advent of the UNIX workstation in the early 1980's.

The original workstations were able to run SPICE, however the execution speed was quite slow by modern standards. Still, they were often faster than a heavily loaded central mainframe, and although expensive, were cost effective over time as compared to CPU charges for a mainframe. Many more engineers were able to take advantage of the convenience of running their simulations on local workstations, but due to the expense of the early workstations, the vast majority still had to slug it out with the mainframe.

In the early 1980's, the IBM personal computer came on the scene, and the real computer revolution was at hand. However, these micros had severe limitations which prevented them from even loading a program as large as SPICE, thus they offered no solution to engineers and scientists needing major number crunching ability. They did, however, provide the ability to run BASIC (thanks to Bill Gates), thus the personal computer began to displace the VAX in many instances, which had in turn displaced the earliest desktop BASIC machines. As the number of machines grew, and thanks to the ingenuity and manufacturing skills resident in the Far East, the cost of these desktop computers dropped rapidly. Intel, designer and purveyor of the microprocessor which was at the heart of the IBM compatible PC, made tons of money, which it wisely reinvested in newer and more capable models. However, the original operating system, DOS, which ran universally on PC's, was designed for and significantly incorporated the limitations of the earliest microprocessors used in PC's. These limitations have echos even today in certain Microsoft products.

Thus, it was not with a bang but with a whimper that the first inexpensive desktop computers which were capable of running SPICE and similar applications appeared. The Intel 386 microprocessor made this possible. Unlike its predecessors, the 386 was a true 32 bit architecture, more powerful than a room full of VAX hardware. It could easily sit on a desk, and cost, even in the early days, less than ten thousand dollars for a fully equipped system. Alas, however, the 386 PC was like a Ferrari which was deliberately equipped to emulate a Volkswagen. In order for the 386 to be compatible with its relatively brain-dead predecessors, Intel included an emulation mode in the instruction set. In this mode, called "real mode" by Intel, the 386 would behave as a somewhat faster version of earlier microprocessors, and thus be compatible with DOS, and all of the software written for DOS. So thorough was the emulation that even though the 386 could access 4 gigabits of memory directly, it was prevented from doing so under DOS, so that games and other archaic programs which could possibly make use of the memory address wrap around at 1 megabyte would perform as on earlier chips. To this day under DOS, the 386 and its descendants operate in this emulation mode, leaving the high-power native mode unused.

Intel assumed that a software vendor, meaning Microsoft, would soon produce a successor to DOS that would unleash the full power of its new chips. Alas, Microsoft responded by ignoring this potential in its own products, and appeared unsupportive of the exploitation of this power by other software vendors. The size of the DOS market evidently influenced this business decision. However, thankfully, a few small companies saw the potential. The first was Phar Lap, which by working directly with Intel delivered a product known as a DOS extender. Intel and Phar Lap, and others, created a specification under which extended DOS applications could coexist with regular DOS programs to a large extent. The DOS extender allowed programs of arbitrary size to be compiled and executed in the 32-bit native mode, which Intel termed "protected mode". The term derives from the memory mapping which allows all applications to have their own address space, and not clobber one another as they can under DOS. At last, with this product and the right compiler, it was possible to run SPICE on a desktop PC, without worrying about the memory limitations of DOS.

Microsoft eventually began to exploit some aspects of protected mode in the Windows product, however, Windows was completely incompatible with extended DOS software at the time. Microsoft's objective was to influence developers of large applications to port to Windows, for which they made available a $500.00 software development kit. Unfortunately Windows at that time had a memory management system which most judged inadequate for applications such as SPICE, plus the support for earlier versions of the Intel microprocessors in Windows added rather severe performance penalties. Thus, in spite of the lack of Windows compatibility, the DOS extender market greatly expanded, and several large commercial applications made use of this technology. DOS extenders were eventually being included with many advanced compilers for "free".

The DOS extenders extended the life span of DOS, however many limitations remained. One such limitation was the lack of multi-tasking. Although some products such as Quarterdeck's DesqView provided some crude multitasking, clearly the time had arrived that a completely new operating system was in order. These new operating systems began arriving in 1992. Vendors of advanced workstations, such as Sun and Next, released versions of their UNIX-derived operating systems for Intel machines. IBM introduced OS2 2.0, which was a 32-bit version of the OS2 operating system. Microsoft has released the NT operating system, the first version of Windows that "really" used protected mode.

The contender that will probably most interest engineers and scientists is UNIX. This operating system has a multi-decade history of use and improvement, with features and performance other operating systems are striving to emulate. Proprietary operating systems based on UNIX are available from several vendors, unfortunately, the cost, still quite high due to the licensing fee extracted by the copyright holder, is a deterrent. However, UNIX clones, which operate the same but use non-copyrighted software, are now widely available. The most popular of these are Linux and FreeBSD, both of which are available on the Internet, and on CD for a small fee. Both provide the advanced user with a state-of-the-art operating system, capable of running the plethora of applications available on the Internet. Linux has become quite popular with the general technically-inclined public, while FreeBSD has found a large niche as an Internet server, due to its reliability and speed. The running of massive applications on a desktop computer (or even a laptop) is now a reality. A properly equipped Intel-compatible computer running FreeBSD can be considered in every respect a ''Unix workstation".

The original Fortran version of SPICE became widespread in the industry, and the creators of SPICE dispersed and went on to new projects. As SPICE became more widely used on modern hardware, its age began to show. Thus, the Berkeley groups set about to rewrite SPICE in a modern programming language (C), and added new features and functionality. The result was SPICE3. Unlike the previous versions of SPICE, SPICE3 was designed for interactive use. Furthermore, there were built-in features for plotting output on-screen, as well as enhanced control over the run in progress. SPICE3 has to date not received the widespread acceptance of its predecessor, mainly because is lacks the long history of use. Early releases did have bugs, and certain features were lacking. The new code, being written in a structured form, is relatively easy to modify, thus SPICE3 should become a standard in time, however

it is also competing with commercial versions of SPICE with many of the same features. The original SPICE is still being shipped "as is", and although it is robust and stable, there are parts of the code that seemingly nobody understands.

In the early 1980's, IBM had a large project to introduce a computer based on Josephson junction logic. IBM used internal software to model these circuits, as SPICE was not designed to support Josephson junctions. To provide a generally available software simulation tool for the analysis of Josephson circuits, the Cryoelectronics group at Berkeley modified SPICE to include Josephson junctions. This version of SPICE, JSPICE, was distributed by the Cryoelectronics group to the handful of interested researchers. It quickly became the dominant tool for the simulation of these circuits (outside of IBM).

However, being based in SPICE, there were many aspects of the program which could stand improvement. Making these improvements would entail a complete rewriting of the SPICE code, a rather formidable task with the Fortran source. Nevertheless, this was done in large measure for internal use at Hypres, a small company engaged in superconductive electronics. The Hyspice program contained a numerical core written in C, which was supported by much of the original SPICE Fortran. The algorithms were modified to increase execution speed, while providing full support (rather than a tacked on appendage) for Josephson junctions. The execution speed for Josephson circuits increased by about a factor of 3–5 over the older JSPICE. Hyspice, however, was basically a batch mode program similar to JSPICE, although it was designed to work with a graphics post-processor, which was a separate application.

While entering a consulting and contract design practice, the author of Hyspice required a simulation tool in order to pursue design activities. Hyspice was proprietary to a former employer, and was obsolete anyway. JSPICE was even more obsolete. Thus, the author decided to modify SPICE3 to incorporate a Josephson model. Further modifications were anticipated so as to provide uncompromising capability and flexibility in the simulation of circuits using Josephson devices (without affecting the ability to simulate conventional circuits). The resulting program, named JSPICE3, also had to be compatible with the author's 386 computer, yet be portable to other computers and operating systems should the need arise. JSPICE3 evolved for several years. Along the way, new features were added, including a schematic capture front-end. The program, still available from Whiteley Research Inc., currently runs on most UNIX platforms, and is still being used in several industrial and educational facilities.

The author, once again getting restless, founded Whiteley Research Inc., in Sunnyvale, CA in 1996. The company was to develop a new successor to JSPICE3, and other tools for schematic capture and mask layout editing. After about one solid year of development, the *XicTools* toolset was announced. The electrical circuit simulator, part of the core of the *XicTools* and known as *WRspice*, is a descendent of the JSPICE3 program, and maintains full compatibility. The new program was migrated to the C++ programming language, for improved maintainability. Compatibility with the older SPICE program was improved, as support for certain capabilities in SPICE, such as the POLY directive, that were missing from SPICE3 were incorporated in *WRspice*. *WRspice* is more network-aware than its predecessors, and in fact can control the dispatching of jobs to remote machines, so that repetitive operations, such as Monte Carlo analysis, can exploit all of the machines in the user's workgroup in parallel. Incidently, Monte Carlo analysis is a new feature, too. Most of *WRspice* can be controlled graphically using point-and-click, yet is prompt-line compatible with its predecessors.

1.2 *WRspice* Overview

WRspice is a general-purpose circuit simulation program based on the venerable Berkeley SPICE (Simulation Program for Integrated Circuit Engineering). Although completely compatible with modern implementations of Berkeley SPICE, and partially compatible with many commercial extensions, *WR-*

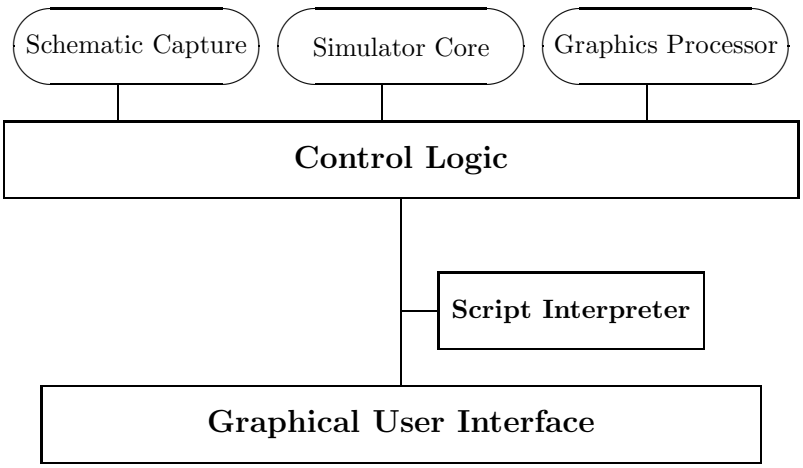

Figure 1.1: Block diagram of the *WRspice* interactive circuit simulation tool.

spice is an entirely new simulator written in the C++ programming language for ease of development and maintenance with high performance. *WRspice* includes a built-in Verilog parser/simulator for mixed analog/digital simulations. Verilog is a popular IEEE standard hardware description language used to model digital logic circuits.

The overall structure of *WRspice* is shown in Figure 1.1. The core of the program is the numerical analysis kernel, which actually solves the nonlinear circuit equations. This engine is controlled by a large block of logic, which in turn is controlled through interaction with the keyboard and mouse, or under control of a script file, or even under the control of another program. Repeated analyses, or analyses dependent upon the outcome of simulating variables, can be set up through use of control language scripts. Input can be entered as description files, or graphically from a schematic representation. Output can be plotted on the screen, with powerful ability to manipulate and transform the data. The basic user interface is very similar to a UNIX shell, with automatic command completion, a history mechanism, and other features known to UNIX users.

WRspice uses the same basic algorithm to solve the nonlinear circuit equations as the original version of SPICE. This is a modified nodal analysis, where a matrix $\mathbf{A}$ is determined, and a solution vector $\mathbf{X}$ is obtained from an excitation vector $\mathbf{B}$ by inverting the expression $\mathbf{AX} = \mathbf{B}$. In *WRspice*, the coefficients of $\mathbf{X}$ are node voltages, and branch currents of voltage sources and inductors. The coefficients of $\mathbf{B}$ are independent source currents, plus terms which are added during the linearization process. The coefficients of $\mathbf{A}$ are the small-signal admittance parameters of each device, plus factors which relate branch currents to other circuit currents.

When the input description is submitted, *WRspice* sets the coefficients of the $\mathbf{A}$ matrix corresponding to each device in the circuit. The $\mathbf{B}$ vector is also defined from knowledge of the source values. The solution vector $\mathbf{X}$ is then obtained through an in-place LU decomposition of $\mathbf{A}$. If all circuit elements are linear, then $\mathbf{X}$ represents the output vector at the initial time point. However, in general, the circuits contain nonlinear elements, and $\mathbf{X}$ at this point can be considered only an approximation to the correct solution. This is because the $\mathbf{A}$ matrix contains only first order terms, and approximations of the contributions of higher order terms have been added to the $\mathbf{B}$ vector. Initially, these "predictor" terms represent an educated guess, however after solving for $\mathbf{X}$, one can obtain more accurate estimates.

These better estimates are then incorporated into a new **B** vector, a new **A** matrix is obtained, and the LU solution repeated. This iterative process continues until the predictor terms converge to a stable value within an error tolerance. This process is known as Newton's method.

A transient analysis solves the equation set at increments of time over the range specified by the user. The time increment is determined by an algorithm which predicts the maximum allowable time step given past behavior. Clearly, to simulate as rapidly as possible, the number of time steps and iterations should be minimized. There are a number of variables which can be set in *WRspice* which affect this behavior, and it is difficult to generalize from one circuit to another which are the best conditions. For example, one can lengthen the average time step, however this will generally require more iterations at each time step, which may lead to slower execution time. Also, one can reduce the number of iterations by increasing the error tolerance, however this may result in excessive errors in the output.

Figure 1.2 below shows a flow diagram of the solution algorithm for transient analysis.

Some distributions of *WRspice* separate the device models from the program, placing them into a dynamically linked library, which is loaded at run-time. In some cases, the user has control of this library and can add or delete devices at will. All device-related information in this manual pertains to the library supplied by Whiteley Research Inc. with the *WRspice* product. Local system administrators should be consulted for information on locally-added devices.

The default device library contains the devices familiar from SPICE2 and SPICE3, including resistors, capacitors, inductors, mutual inductors, independent and dependent voltage and current sources, lossy and lossless transmission lines, switches, and the five most common semiconductor devices: diodes, BJTs, JFETs, MESFETs, and MOSFETs, plus Josephson junctions, similar to the RSJ model first included in SPICE2 by Jewett[11]. The original device models from SPICE3 are provided, along with a number of third-party models, particularly for MOS transistors.

WRspice is based on JSPICE3, which in turn was derived from SPICE3F4, which developed from SPICE2G.6. While *WRspice* is being developed to include new features, it will continue to support those capabilities and models which remain in extensive use in the SPICE community.

WRspice is part of the *XicTools* design system from Whiteley Research, Inc. These tools are designed to be modular, yet interactive. In particular, *WRspice* will work seamlessly with the *Xic* graphical front-end for schematic capture, if the *Xic* program is present. Otherwise *WRspice* can be utilized in a stand-alone mode. From the *Xic* graphical editor, *WRspice* can be called upon to perform simulations, if *WRspice* is present. In this case, since it is used in a background mode, the *WRspice* binary can exist on a remote machine.

The *XicTools* package has been developed primarily under BSD-4.4 Unix (FreeBSD), which is the reference operating system. The tools have been ported to many other UNIX-type operating systems, including Linux, Sun Solaris and SunOS 4.1.x, HPUX, and DEC Alpha-OSF. The tools are also now available for Microsoft Windows.

Unix/Linux releases of *WRspice* use the gtk+ toolkit running on the X window system for the graphical user interface. If X is not available, or if the user so chooses, *WRspice* will run without graphics (other than crude ascii-mode plots).

1.3 Types of Analysis

Like its predecessors, *WRspice* supports various forms of nonlinear dc, nonlinear transient, and linear ac analyses.

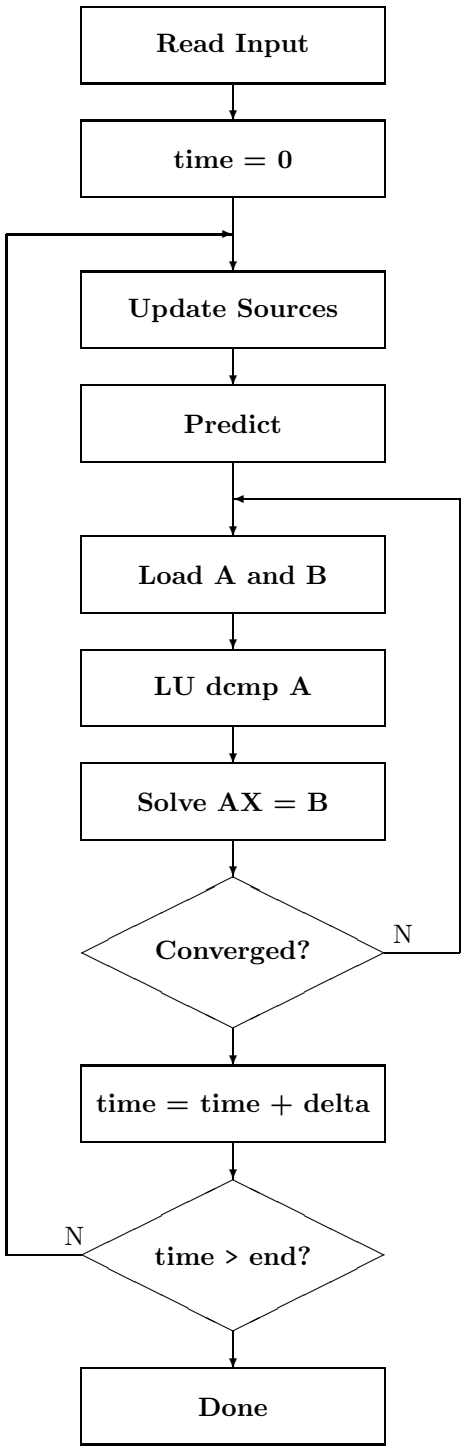

Figure 1.2: Flow diagram of the algorithm used by *WRspice*.

DC Analysis

The dc analysis portion of *WRspice* determines the dc operating point of the circuit with inductors shorted and capacitors opened. A dc analysis is automatically performed prior to a transient analysis to determine the transient initial conditions, and prior to an ac small-signal analysis to determine the linearized, small-signal models for nonlinear devices. The dc analysis can also be used to generate dc transfer curves: a specified independent voltage or current source is stepped over a user-specified range and the dc output variables are stored for each sequential source value. In *WRspice*, dc analysis can be combined with other analysis types to generate a family of analysis results representing data from each point of the dc analysis. The dc analysis is not available if Josephson junctions are present in the circuit.

AC Analysis

The ac small-signal portion of *WRspice* computes the ac output variables as a function of frequency. The program first computes the dc operating point of the circuit and determines linearized, small-signal models for all of the nonlinear devices in the circuit. The resultant linear circuit is then analyzed over a user-specified range of frequencies. The desired output of an ac small-signal analysis is usually a transfer function (voltage gain, transimpedance, etc). If the circuit has only one ac input, it is convenient to set that input to unity and zero phase, so that output variables have the same value as the transfer function of the output variable with respect to the input. The ac analysis can be combined with a dc sweep so that ac analysis is performed at each point over a range of bias conditions. The ac analysis is not available on circuits containing Josephson junctions.

Transient Analysis

The transient analysis portion of *WRspice* computes the transient output variables as a function of time over a user-specified time interval. The initial conditions can be automatically determined by a dc analysis. All sources which are not time dependent (for example, power supplies) are set to their dc value. If Josephson junctions are present, or if the `uic` option is given, initial conditions are assumed at the start of analysis rather than the result of the dc operating point analysis. With Josephson junctions, all sources should start with zero output. Transient analysis can be combined with a dc sweep so that the transient simulation is performed at each point over a range of bias conditions.

Transfer Function Analysis

The transfer analysis portion of *WRspice* computes the dc or ac small signal transfer function, input impedance, and output impedance of a network. For ac analysis, the dc operating point is automatically determined through an operating point analysis. The transfer analysis can be combined with a dc sweep so that the transfer function is computed at each point over a range of bias conditions.

Pole-Zero Analysis

The pole-zero analysis portion of *WRspice* computes the poles and/or zeros in the small-signal ac transfer function. The program first computes the dc operating point and then determines the linearized, small-signal models for all the nonlinear devices in the circuit. This circuit is then used to find the poles and zeros. Two types of transfer functions are allowed: one of the form

(*output_voltage*)/(*input_voltage*)

and the other of the form

(output_voltage)/(input_current).

These two types of transfer functions cover all the cases and one can find the poles/zeros of functions like input/output impedance and voltage gain. The pole-zero analysis works with resistors,

capacitors, inductors, linear-controlled sources, independent sources, BJTs, MOSFETs, JFETs and diodes. Transmission lines and Josephson junctions are not supported.

Distortion Analysis

The distortion analysis portion of *WRspice* computes steady-state harmonic and intermodulation products for small input signal magnitudes. If signals of a single frequency are specified as the input to the circuit, the complex values of the second and third harmonics are determined at every point in the circuit. If there are signals of two frequencies input to the circuit, the analysis finds the complex values of the circuit variables at the sum and difference of the input frequencies, and at the difference of the smaller frequency from the second harmonic of the larger frequency. Distortion analysis can be combined with a dc sweep so that distortion is analyzed at each point over a range of bias conditions.

Distortion analysis is supported for the following nonlinear devices: diodes, bipolar transistors, JFETs, MOS1-4, MESFETs. All linear devices are automatically supported by distortion analysis. If there are switches present in the circuit, the analysis continues to be accurate provided the switches do not change state under the small excitations used for distortion analysis.

Sensitivity Analysis

WRspice will calculate either the DC operating-point sensitivity or the AC small-signal sensitivity of an output variable with respect to all circuit variables, including model parameters. *WRspice* calculates the difference in an output variable (either a node voltage or a branch current) by perturbing each parameter of each device independently. Since the method is a numerical approximation, the results may demonstrate second-order effects in highly sensitive parameters, or may fail to show very low but non-zero sensitivity. Further, since each variable is perturbed by a small fraction of its value, zero-valued parameters are not analyzed (this has the benefit of reducing what is usually a very large amount of data). Sensitivity analysis can be combined with a dc sweep so that sensitivity can be analyzed at each point over a range of bias conditions.

Noise Analysis

The noise analysis portion of *WRspice* performs analysis of device-generated noise for the given circuit. When provided with an input source and an output node, the analysis calculates the noise contributions of each device (and each noise generator within the device) to the output node voltage. It also calculates the level of input noise from the specified input source to generate the equivalent output noise. This is done for every frequency point in a specified range — the calculated value of the noise corresponds to the spectral density of the circuit variable viewed as a stationary Gaussian stochastic process. Noise analysis can be combined with a dc sweep so that noise can be computed at each point over a range of bias conditions.

After calculating the spectral densities, noise analysis integrates these values over the specified frequency range to arrive at the total noise voltage/current (over this frequency range). This calculated value corresponds to the variance of the circuit variable viewed as a stationary Gaussian process.

Operating Range Analysis

WRspice has an integrated two-dimensional operating range analysis capability. The operating range analysis mode is used in conjunction with the other analysis types, such as transient or ac. A suitably configured source file and circuit description is evaluated over a one or two dimensional area of parameter space, producing (optionally) an output file describing the results at each evaluated point, or vectors giving the minimum and maximum values of the varying parameters for operation. Results can be viewed graphically during or after simulation.

Monte Carlo Analysis

WRspice has a built-in facility for performing Monte Carlo analysis, where one or more circuit

variables are set according to a random distribution, and the circuit analyzed for functionality. The file format and operation is very similar to operating range analysis.

Automated Looping
In *WRspice*, any analysis can be automatically repeated while stepping over a one or two dimensional area of parameter space. Any circuit parameter may be varied.

Both dc and transient solutions are obtained by an iterative process which is terminated when both of the following conditions hold:

1. The nonlinear branch currents converge to within a tolerance of 0.1 percent or 1 picoamp (1.0E-12 Amp), whichever is larger.

2. The node voltages converge to within a tolerance of 0.1 percent or 1 microvolt (1.0E-6 Volt), whichever is larger.

Although the algorithm used in *WRspice* has been found to be very reliable, in some cases it will fail to converge to a solution. When this failure occurs, the program will terminate the job.

Failure to converge in dc analysis is usually due to an error in specifying circuit connections, element values, or model parameter values. Regenerative switching circuits or circuits with positive feedback probably will not converge in the dc analysis unless the `off` option is used for some of the devices in the feedback path, or the `.nodeset` card is used to force the circuit to converge to the desired state.

WRspice runs can consume quite a bit of virtual memory, and it is possible to exceed machine limits on many systems. The main consumer of memory is the data arrays from simulation runs. Each point is a double precision number requiring 8 bytes. Typically, all nodes and branch currents are saved, though this can be changed with the save command. One set of values is retained for each output increment. For example, a circuit with 100 saved vectors running `tran 1p 1n` requires roughly 8 X 100 X 1000 bytes per run. This is allocated to the plot structure. By default, all plots are saved, so memory usage increases with each run.

The maximum memory that can be used for plot data storage for a single run is set by the `maxdata` variable. The Tool Control window displays memory statistics, and can be used to keep track of memory in use.

The vectors are copied when a plot is produced (including iplots), thus this additional memory must be available for plots to be displayed. In addition, iplots with a large number of data points (more than about 10000) can noticeably slow the simulation run.

The free and destroy commands can be used to delete existing plots, making the memory available for other purposes. The rusage command displays memory usage and memory limits. Note that once *WRspice* obtains memory from the operating system, on many systems this memory is never returned. Thus, the free command can make more memory available for *WRspice*, but not for other programs which may also be running.

Exceeding virtual memory limits is not in general a fatal error, depending on when the error occurs. Plots and iplots allocate all memory needed at the beginning of the operation, so an out of memory condition will usually abort the operation and return the command prompt. It is possible, though, for further errors to be generated by a memory failure which may cause a segmentation fault.

1.4 Circuit Input and Program Control

WRspice is intended for use as an interactive tool, though various batch-mode features are supported. Circuit input is provided in the form of files which are loaded into *WRspice*. These files can be generated by the user with a text editor, or be generated by a graphical editor program such as *Xic*. Once loaded into *WRspice*, a circuit is subject to the many types of analysis and post-processing operations available through *WRspice* commands. These commands can be given interactively through the text-mode interface provided by *WRspice*, or in many cases through graphical operations.

The most common way directives are provided to *WRspice* is through the text-mode command line interface. The command line interface behaves very much like a UNIX shell, through which commands are entered, variables set, and output is printed. The shell provides most of the mechanisms familiar from UNIX shells, including aliasing, history substitution, and command completion.

The command shell is normally established on the input terminal or terminal emulation window from which *WRspice* was executed. *WRspice* takes control of this terminal, that is, all input typed will be directed to *WRspice*, however the operating system job control commands can be used to place *WRspice* in the background.

The *WRspice* shell provides a command language, which enables scripts containing commands to be executed. Writing scripts enables automation of repetitive or complicated tasks. Control commands can be added to circuit files, and in fact a unified input processing system handles both type of input. Input files are loaded into *WRspice* with the source command. The "`source`" is in fact optional. If the file name does not conflict with the name of a *WRspice* command, simply typing the name of the file will perform the source operation.

When graphics is available, *WRspice* provides a small "Tool Control" window which contains menus. The menus contain buttons which in turn bring up graphical tools which control most of *WRspice*. All of the operations of these tools have analogous command line commands, though many users find the graphical interface preferable.

WRspice contains a complete HTML-based help system, available with the help command. The help windows provide an extensively cross-linked reference on the various commands and features. In addition, the help windows can be used to view arbitrary HTML content on the Internet or on the user's local machine.

1.5 Post-Processing and Run Control

WRspice in interactive mode provides a powerful plotting capability for simulation output. Plots can be generated on the fly while simulating, or after simulation is complete. The command language provides interactive data manipulation and generation capability on output prior to plotting. Any number of plots can be shown on-screen at a given time, and traces can be copied between plot windows via mouse operations for easy comparison.

A trace command allows the value of expressions involving circuit variables to be printed as simulation progresses. Simulation can be paused by typing the interrupt character (**Ctrl-C**), and can be resumed later. Simulation can also be paused after a certain number of data points, or when a logical expression involving circuit variables becomes true.

The Verilog capability can be used to provide automata for control and monitoring of a circuit during simulation. Verilog modules are defined within the circuit description, and are evaluated as simulation (transient analysis only) progresses. This is clearly useful for mixed analog/digital systems, but has

additional utility for implementing event or error counters, etc., for output statistical analysis.

WRspice provides a measurement capability for providing timing or other information from a circuit simulation. Any number of measurements can be included in a circuit description. This can be particularly useful in optimization scripts.

All of this capability is tied together in the *WRspice* shell, which provides command processing in interactive mode, but also provides a scripting capability. Scripts can be written to automate complicated analyses and data manipulation. All circuit output data, device and circuit parameters, and shell variables and vectors are available in a rich programming environment.

1.6 Introduction to Interactive Simulation

WRspice is an interactive circuit simulation program. One can find details about preparing *WRspice* input files in Chapter 2.1 — this section assumes some familiarity with this syntax, which is basically that of SPICE2. This section is intended to be a quick introduction to the use of *WRspice*, and its capabilities. The remaining chapters provide details and in-depth explanations of the various modes, functions, and features.

If *Xic* is being run, circuits can be entered graphically, making the command line interface described here somewhat unnecessary (there are users, however, who prefer the command line interface). However, in order to use the full spectrum of capabilities, the command line interface is required.

To start *WRspice*, one can type

`wrspice` *input_file*

where *input_file* is the name of the *WRspice* circuit description file to run. Alternately, one can simply type

```
wrspice
```

in which case *WRspice* will start up without loading a circuit. A new circuit description file can be loaded into *WRspice* by typing the command

`source` *input_file*

If the name *input_file* is different from any internal or external *WRspice* commands, the source command can be eliminated, and the *WRspice* input file is loaded simply by typing the file name at the command prompt. The file is parsed into an internal circuit representation, which is held in memory until explicitly deleted. The list of circuits is shown in the panel brought up by the **Circuits** button in the **Tools** menu of the Tool Control window. One can switch the "current circuit" with the setcirc command.

When *WRspice* starts, it normally displays a Tool Control window containing command menus. The menu buttons bring up panels which control and display information. Most of the typed commands have analogues within the display panels. These panels can be arranged on the screen, and the configuration saved, so that when *WRspice* is started subsequently, the user's screen arrangement will be presented.

In addition, *WRspice* "takes over" the text window from which it was launched. The command interface is very much like a shell, and in fact it can be configured to run operating system commands in a manner very similar to the C-shell.

If there are any analysis lines in the input file and the user wishes to run the analyses as given there, one simply enters

```
run
```

WRspice will run the requested analyses and will prompt the user again when finished and the output data are available. If the user wishes to perform an analysis that is not specified by a line in the input file, one can type the analysis line just as it would appear in the input file, without the dot. For example, the command

```
ac lin 20 0.99 1.01
```

will initiate an ac analysis with 20 frequency values between 0.99 and 1.01.

After the analysis is complete and the *WRspice* prompt is displayed, the values of the nodes are available. Similarly, one can load the results from previous *WRspice* sessions using the load command. In either case, to plot the voltage on nodes 4 and 5, for example, one could then issue the command

```
plot v(4) v(5).
```

To display a list of the circuit variables available for plotting, type `display` at the command prompt. The output data items displayed are vectors, with a length generally equal to the number of analysis or output points in the simulation.

To plot vectors with the plot package, one types

`plot` *varlist*

where *varlist* is a list of outputs (such as `v(3)`) or expressions (such as `v(3)*time`). *WRspice* will plot a graph of the outputs on the screen. When finished plotting, *WRspice* will issue a prompt. Under windowing systems such as X, the plot will be drawn in a newly created window somewhere on the screen. This window will remain open until explicitly dismissed by the user, however the execution returns to *WRspice* immediately, so that any number of plots can be on-screen simultaneously.

One can also specify combinations of outputs and functions of them, as in

```
plot v(1) + 2 * v(2)
```

or

```
plot log(v(1)) sin(cos(v(2))).
```

Notice that the vector name `v(1)` is not a function, but rather denotes the voltage at the node named 1. One can use most algebraic functions, including trig functions, `log()` - (base 10), `ln()` - (base e), and functions such as `mag()`, the magnitude of the complex number, `phase()`, the phase, `real()`, the real part, and `imag()`, the imaginary part. These all operate on real or complex values. A complete list of functions and operations available can be found in 3.13. Generally, any command which expects a vector as an argument will accept an expression.

The notation

`plot` *something* `vs` *something_else*

means to plot *something* with *something_else* on the X-axis.

The plot style can be modified through buttons and features found on the plot window. These and other features can have default behavior changed through setting of shell variables. Shell variables are set with the set command from the command line, and any alphanumeric variable can be set to a value or a string. There are a number of such variables that are predefined to affect plotting. These can also be altered graphically from the panel brought up by the **Plot Defs** button in the **Tools** menu. The panel brought up by the **Variables** button in the **Tools** menu shows a listing of the shell variables currently set.

For example, one can modify the plot command to plot a subset of the data available. The command

```
set ylimit = "1 2"
plot v(1) v(2)
```

will plot the two vectors when the values are between 1 and 2, and

```
set xlimit = "1 2"
plot v(1) v(2)
```

will plot them when the scale (time or frequency) is between 1 and 2. The variables will remain set until they are unset, using

```
unset xlimit ylimit
```

or the corresponding buttons in the **Plot Defs** panel are made inactive.

The command

```
set xcompress 5
plot v(1) v(2)
```

plots only every fifth point, and

```
unset xcompress
set xindices 20 30
plot v(1) v(2)
```

plots the values between the 20th time point and the 30th. Any of these variables may be used together, and they are also available in the asciiplot command (which produces an ascii representation for use with text-only printers).

Typing let without arguments is synonymous with the display command. The let command is different from the set command, which sets non-vector shell variables, which may control various aspects of *WRspice* operation. The let command is used to assign a new vector, for example

```
let aa = v(1)
```

will assign a new vector `aa` with all components equal to `v(1)`. If no arguments are given, a listing of output vectors from the most recent simulation is shown. This listing is also shown in the panel brought up by the **Vectors** button in the **Tools** menu.

One can print the values of vectors with the print command:

```
print time
```

The command

```
print all
```

will print the values of all the data available. Incidentally, one can also use the keyword `all` with any of the other commands that take vector names, like plot. There are also alias and history mechanisms available (see 3.12 for details), and a shell command, which passes its arguments to the operating system shell, or starts a subshell.

If the user wishes to save the output values in a data file known as a rawfile, one can then type

`write` *filename* `v(4) v(5)`

to put the values of `v(4)`, and `v(5)` into *filename*. If the user wishes to save everything, one can type

`write` *filename*`.`

There are also many commands for tracing the analysis — one can print the values at a node for each time point or cause *WRspice* to stop whenever a value gets to a certain point. Descriptions of the commands stop, trace, and step can be found in 4.2. A listing of these commands that are currently in force is available in the panel brought up with the **Trace** button in the **Tools** menu.

After the user is done with the values obtained from the simulation run, one can change the circuit and re-run the analysis. If it is desired to edit the circuit itself, one can use the command edit — it will bring up an internal text editor (or a favorite external editor) and allow changes to the circuit in whatever way is necessary, and then when the editor is exited, *WRspice* can re-load the circuit and be ready to run it again. Under UNIX with X windows, a default internal editor is provided. This editor is also available as the "`xeditor`" command from the UNIX shell. Also, one can give another analysis (ac, dc, tran, ...) command after the first one completes. If the analysis is not finished, i.e. the user typed an interrupt (**Ctrl-C**), or the run stopped under the stop command, then one must type `reset` in order to re-run an analysis from the beginning.

Each separate analysis that is performed will create one or more sets of values. Such a set of values is called a plot — if several analyses have been performed, and the user wishes to switch from the results of one to the results of another, the setplot command will inform the user as to which analysis results are available and let the user choose one. The plots are displayed in the panel brought up by the **Plots** button in the **Tools** menu.

To see what other commands are available, skip to Chapter 4, or type **Ctrl-D** in *WRspice*. For information about a particular command, type `help` *command*, where *command* is one of those listed by **Ctrl-D**. This introduction should be enough to get started.

Here is a sample *WRspice* run:

```
csh% wrspice
wrspice 10 -> source xtal.in
Circuit:  crystal filter
wrspice 11 -> listing
```

(listing of the circuit is printed)

```
wrspice 12 -> run
wrspice 15 -> display
Here are the vectors currently active:
Title:  crystal filter
Plotname:  AC analysis curves.
Date:  Thu Sep 26 12:16:34 PDT 1985

     FREQ    : frequency (complex, 20 long) [scale]
     V(4)    : voltage (complex, 20 long)
     V(6)    : voltage (complex, 20 long)
     V(5)    : voltage (complex, 20 long)
```

(and so on...)

```
wrspice 16 -> plot v(4)
```

(plot takes place)

```
wrspice 17 -> write outfile freq v(4)
wrspice 18 -> ac lin 30 1 2
wrspice 19 -> display
Here are the vectors currently active:
Title:  crystal filter
Plotname:  AC analysis curves.
Date:  Thu Sep 26 12:16:34 PDT 1985

     FREQ    : frequency (complex, 30 long) [scale]
     V(4)    : voltage (complex, 30 long)
     V(6)    : voltage (complex, 30 long)
     V(5)    : voltage (complex, 30 long)
```

(and so on...)

```
wrspice 20 -> print v(4) > tempfile
```

(print to tempfile takes place)

```
wrspice 21 -> shell lpr tempfile
```

(a printout is made of the results)

```
wrspice 22 -> load testh
Title:  SPICE 3-C raw output test heading
Name:  Transient analysis.
Date:  08/19/84 03:17:11
wrspice 23 -> display
Here are the variables currently active:
Title:  SPICE 3-C raw output test heading
Plotname:  Transient analysis.
Date:  Sun Dec 1 11:18:25 PST 1985
```

```
TIME    : time (real, 152 long) [scale]
v(1)    : voltage (real, 152 long)
v(2)    : voltage (real, 152 long)
v(3)    : voltage (real, 152 long)
v(4)    : voltage (real, 152 long)
v(5)    : voltage (real, 152 long)
```

```
wrspice 24 -> print v(1)
```

(prints v(1) values)

```
wrspice 25 -> plot v(1)
```

(plot takes place)

```
wrspice 26 -> let xxx = log(v(1))
wrspice 27 -> plot xxx
```

(plot takes place)

```
wrspice 28 -> plot v(1) v(2) v(3) + 1 vs TIME * 2
```

(plot takes place)

```
wrspice 30 -> asciiplot v(1) v(2) TIME + 2 > File
wrspice 31 -> shell lpr File
```

(Pick up ascii plot ...)

```
wrspice 33 -> quit
Warning:  the following plot hasn't been saved:
crystal filter, AC analysis curves.
Are you sure you want to quit?  y
csh%
```

Note that *WRspice* will issue a warning if there is work in progress that has not been saved.

This page intentionally left blank.

Chapter 2

WRspice Input Format

In this document, text which is provided in `typewriter` font represents verbatim input to or output from the program. Text enclosed in square brackets ([text]) is optional in the given example, as in optional command arguments, whereas other text should be provided as indicated. Text which is *italicized* should be replaced with the necessary input, as described in the accompanying text.

2.1 Input Format

The input format for *WRspice* is of the free format type. Fields on a line are separated by one or more blanks, a comma, an equal (=) sign, or left or right parenthesis; extra white space is ignored.

A line may be continued to the following line(s) in two ways. If the last character on a line is a backslash character (\), the "newline" is effectively hidden, and the text on the following line will be appended to the current line, however leading white space is stripped from the continuing line. The traditional SPICE line continuation is also available, whereby a line may be continued by entering a + (plus) as the first non-white space character of the following line, *WRspice* will continue reading beginning with the character that follows.

Device and model names and parameters are case-insensitive, however, unlike other SPICE programs, node names *are* case sensitive. One should consider this when importing old SPICE files that use textual node names.

Files read by *WRspice* as circuit descriptions or control directives are text-mode files, in either the UNIX or DOS line termination format. If a character is found in the first (title) line with the most-significant bit set, the read is aborted, as this is taken as binary input, and attempting to read a binary file would generate a cascade or errors or crash the program. Binary characters can exist elsewhere in the file, if necessary for whatever reason.

A name field must begin with a letter and cannot contain any delimiters.

A number field may be an integer field (12, -44), a floating point field (3.14159), either an integer or floating point number followed by an integer exponent (1E-14, 2.65E3), or either an integer or a floating point number followed by one of the following scale factors (case insensitive):

t	1e12
g	1e9
meg	1e6
k	1e3
mil	25.4e-6
m	1e-3
u	1e-6
n	1e-9
p	1e-12
f	1e-15

Letters immediately following a number that are not scale factors are ignored, and letters immediately following a scale factor are ignored. Hence, 10, 10v, 10volts, and 10hz all represent the same number, and m, ma, msec, and mmhos all represent the same scale factor. Thus 1000, 1000.0, 1000hz, 1E3, 1.0e3, 1KHz, and 1k all represent the same number.

2.2 Circuit Description

The circuit to be analyzed is described to *WRspice* by a set of element lines, which define the circuit topology and element values, and a set of control lines, which define the model parameters and the run controls. The first line in the input file must be a title line. The order of the remaining circuit definition and control lines is arbitrary, except that continuation lines must immediately follow the line being continued, and certain constructs are expected to contain blocks of lines, the order of which is generally arbitrary within the block except for shell commands which of course must be ordered. Certain special input file formats are recognized, such as operating range analysis control files, and files generated by the schematic capture front end. Exceptions to the rule of arbitrary line placement will be described in the sections describing these files.

As the file is being read, any shell variable references are expanded. The *WRspice* shell is described in 3.12, but at this point the reader should be aware that just about any text within the circuit description can be specified through the shell substitution mechanism.

Each element in the circuit is specified by an element line that contains the element name, the circuit nodes to which the element is connected, and the values of the parameters that determine the electrical characteristics of the element. The first letter of the element name specifies the element type (case insensitive). For example, a resistor name must begin with the letter '`R`' or '`r`' and can contain one or more characters. Hence, `R`, `r1`, `Rse`, `ROUT`, and `r3ac2zY` are valid resistor names.

In the descriptions that follow, data fields that are enclosed in square brackets '`[`', '`]`' are optional. All indicated punctuation (parentheses, equal signs, etc.) is optional and merely indicates the presence of any delimiter. A consistent style such as that shown here will make the input easier to understand. With respect to branch voltages and currents, *WRspice* uniformly uses the associated reference convention (current flows in the direction of voltage drop).

Node names must begin with an alphanumeric character, but may have trailing or embedded punctuation (but this is generally not recommended). In some cases, "strange" node names must be quoted when referenced in *WRspice* as `v("`*nodename*`")`. The datum (ground) node must be named "0", however. Note the difference with SPICE2, where nodes are evaluated as numbers. Thus, '00' and '0' are distinct in *WRspice*, but not in SPICE2.

The circuit cannot contain a loop of voltage sources. If a dc operating point analysis is performed, which is true for all analysis except for transient analysis with the `uic` (use initial conditions) flag set, the

circuit can not contain a loop of voltage sources and/or inductors and cannot contain a cutset of current sources and/or capacitors. In transient analysis with the `uic` flag set (which is always the case when Josephson junctions are present), inductor/voltage source loops are allowed, as are capacitor/current source cut sets. However, parallel voltage sources and series current sources are not accepted. It is not strictly necessary that each node in the circuit have a dc path to ground with the `uic` flag given, however convergence problems may result. It is sometimes necessary to add a large-valued resistor to ground in these cases. In general, nodes should have at least two connections.

Text can contain references to shell variables, i.e., variable names that begin with '`$`'. These variables are evaluated as the circuit is read in, or with the reset command once the circuit is loaded. The variables must be known to the shell before the circuit is parsed, so if they are defined in the input file, the definition must occur in `.exec` blocks or `.options` lines, which are evaluated before the circuit is parsed, and not `.control` blocks, which are evaluated after the circuit is parsed. If the '`$`' is preceded with a backslash ('\'), the shell substitution is suppressed, but the construct forms a comment delimiter so that the remainder of the line is ignored.

Any text enclosed in single quotes (') will be evaluated as an expression as the file is read (before shell substitution) and the string will be replaced by the result. Since evaluation is performed before shell substitution, the expression can not contain shell variables or other '`$`' references, but it can contain vectors (which must be defined before the circuit is read).

If the first line of text in a file starts with the characters "`#!`", that line will be discarded when the file is read. This enables *WRspice* input files to be self-executing using the mechanisms of the UNIX shell. For example, if the following line is prepended to an input file

```
#!  wrspice
```

and the file is made executable, then typing the name of the file will initiate *WRspice* on the circuit contained in the file.

2.3 Title, Comments, and Inclusions

2.3.1 Title Line

The first line of the circuit description is a title line. Any text (including a blank line) can appear in the title line. The line is printed as part of generated output, but is otherwise unused by *WRspice*.

2.3.2 `.title` Line

General Form:

```
.title any text
```

Example:

```
.title This is an alternate title
```

This simply replaces the title line, which is the first line of the file, when printing output.

2.3.3 .end Line

General Form:
```
.end
```

This line is optional, but if it appears it should be the last line in the circuit description part of the input file. The `.end` line is ignored in *WRspice*.

2.3.4 Comments

In the circuit description, an asterisk ('`*`') as the first non-white space character indicates that this line is a comment line. Also, if the sequence '`\$`' appears in a line of input and is preceded by white space or is at the beginning of the line, the characters that follow are ignored. Comments may be placed anywhere in the circuit description.

General Form:
```
*any comment
any_spice_text \$ this text is ignored
```

Examples:
```
* rf=1k      gain should be 100
*beginning of amplifier description
r1 1 0 100 \$ this is a comment
```

Comment lines which begin with '`*@`' and '`*#`' in the circuit description are special: they are taken as executable statements as if included in `.exec` or `.control` blocks, respectively (see 2.10.1).

In `.exec` and `.control` blocks, comments are indicated for lines with the first non-whitespace character being '`#`', for compatibility with the shell. In Verilog blocks, The C++ commenting style ("`//`" for single line comments and "`/* ... */`" for multi-line comments) is recognized.

2.3.5 .include Line

General Form:
```
.include filename
```

Example:
```
.include mysubckt
```

The `.include` line specifies that the named file is to be read and added to the input at the location of the `.include` line. Included files can be nested arbitrarily. The `.include` lines are processed before shell variable expansion is performed.

In *WRspice*, the keyword `.spinclude` is accepted as a synonym for `.include`. This is for compatibility with *Xic*, which will replace `.include` lines with the file contents, but will pass `.spinclude` lines to SPICE, after converting "`.spinclude`" to "`.include`".

2.3.6 `.lib` Line

General Form:
`.lib` *path_to_file name*

Example:
```
.lib /usr/local/parts/mylib mos25
```

This will look in *path_to_file* for lines enclosed as follows.

```
.lib name
... lines of SPICE text
.endl
```

The lines inside the block will be read into the input deck being parsed, similar to the `.include` line.

The library file can contain any number of `.lib` blocks. The `.lib` block can itself contain `.lib` references. The text can be any valid *WRspice* input. The *name* is an arbitrary text token, which should be unique among the `.lib` blocks in a library file.

Example:
```
title line
.lib /usr/stevew/spice/stuff/mylibrary mosblock
... more lines
```

In `/usr/stevew/spice/stuff/mylibrary`:

```
.lib mosblock
m0 4 9 12 PSUB p1pvt l=0.25u w=2.4u
.endl
```

is equivalent to:

```
title line
m0 4 9 12 PSUB p1pvt l=0.25u w=2.4u
... more lines
```

In *WRspice*, the keyword `.splib` is accepted as a synonym for `.lib`. This is for compatibility with *Xic*, which will replace `.lib` lines with the block of text from the library, but will pass `.splib` lines to SPICE, after converting "`.splib`" to "`.lib`".

2.4 Initialization

2.4.1 `.global` Line

General Form:
`.global` *node1 node2* ...

The arguments are node names. These declared node names remain unaltered when subcircuits are expanded, thus the indicated nodes become accessible throughout the circuit.

For example, the substrate node of all n-channel MOSFETS in the main circuit and subcircuits can be tied to a node listed in the `.global` line. Then, substrate bias can be applied to all substrate nodes with a single source, conveniently located in the main circuit.

2.4.2 `.ic` Line

General Form:
`.ic v(`*nodname*`)=`*val* `v(`*nodname*`)=`*val* `...`

Example:
```
.ic v(11)=5 v(4)=-5 v(2)=2.2
```

This line is for setting transient initial conditions. It has two different interpretations, depending on whether the `uic` parameter is specified on the `.tran` line. Also, one should not confuse this line with the `.nodeset` line. The `.nodeset` line is only to help dc convergence, and does not affect final bias solution (except for multi-stable circuits). The two interpretations of this line are as follows:

1. When the `uic` parameter is specified on the `.tran` line, then the node voltages specified on the `.ic` line are used to compute the capacitor, diode, BJT, JFET, and MOSFET initial conditions. This is equivalent to specifying the `ic=...` parameter on each device line, but is much more convenient. The `ic=...` parameter can still be specified and will take precedence over the `.ic` values. Since no dc bias (initial transient) solution is computed before the transient analysis, one should take care to specify all dc source voltages on the `.ic` line if they are to be used to compute device initial conditions.

2. When the `uic` parameter is not specified on the `.tran` line, the dc bias (initial transient) solution will be computed before the transient analysis. In this case, the node voltages specified on the `.ic` line will be forced to the desired initial values during the bias solution. During transient analysis, the constraint on these node voltages is removed.

2.4.3 `.nodeset` Line

General Form:
`.nodeset v(`*nodname*`)=`*val* `v(`*nodname*`)=`*val* `...`

Example:
```
.nodeset v(12)=4.5 v(4)=2.23
```

This line helps the program find the dc or initial transient solution by making a preliminary pass with the specified nodes held to the given voltages. The restriction is then released and the iteration continues to the true solution. The `.nodeset` line may be necessary for convergence of bistable or astable circuits. In general, this line should not be necessary.

2.4.4 .options Line

General Form:
```
.options opt1 opt2 ...  (or opt=optval ...)
```

Example:
```
.options reltol=.005 trtol=8
```

Options which control the operation of the simulator can be entered in the *WRspice* input file following the `.options` keyword. There are a number of variables which control simulation, many familiar from SPICE2, listed below. Any variable can be set on the `.options` line, and is similar to setting the variable from the shell with the `set` command, however the variables set from the `.options` line are active only when the circuit is the current circuit. The variables set in the `.options` line are set *before* variable expansion is performed on the rest of the circuit text, so that global shell variables may be set in the `.options` line. The `.options` line is itself expanded before it is parsed, and after execution of the `.exec` lines (see 2.10.1).

The options which control simulation can also be entered from the keyboard by using the *WRspice* `set` command, or equivalently from the graphical tools available from the **Tools** menu of the Tool Control window (described in 3.4). Before a simulation starts, these options will be merged with the options from the `.options` line, unless the `noshellopts` variable is set *from the shell*. The result of the merge is that options that are booleans will be set if set in either case, and those that take values will assume the value set through the shell if conflicting definitions are given. The merge will be suppressed if the shell variable `noshellopts` is set, in which case the only options used will be those from the `.options` line, and any shell redefinitions will be ignored.

In general, variables set in the `.options` line are available for expansion in `$`*variable* references, but do not otherwise affect the workings of the shell (see 3.12). Variables set from a `.options` line are indicated with a "+" in the listing provided from the `set` command given with no arguments, or in the **Variables** tool from the **Tools** menu in the Tool Control window.

The `.options` line allows the user to reset program control and user options for specific simulation purposes. Any combination of options and arbitrary variables may be included, in any order.

Although any variable can be set from a `.options` line, the variables listed below are most likely to appear in that context due to SPICE2/3 history. The 'x' (below) represents some positive number.

option	effect
`abstol=x`	Resets the absolute current error tolerance of the program. The default value is 1 picoamp.
`acct`	Print accounting information in batch output.
`chgtol=x`	Resets the charge tolerance of the program. The default value is 1.0e-14.
`defl=x`	Resets the value for MOS channel length; the default is 100.0 micrometer.
`defw=x`	Resets the value for MOS channel width; the default is 100.0 micrometer.
`defad=x`	Resets the value for MOS drain diffusion area; the default is 0.0.
`defas=x`	Resets the value for MOS source diffusion area; the default is 0.0.
`gmin=x`	Resets the value of gmin, the minimum conductance allowed by the program. The default value is 1.0e-12.
`itl1=x`	Resets the dc iteration limit. The default is 400.
`itl2=x`	Resets the dc transfer curve iteration limit. The default is 100.
`itl4=x`	Resets the transient timepoint iteration limit. The default is 10.
`list`	Print a listing of the input file in batch output.

`node`	Print a tabulation of the operating point node voltages in batch output.
`opts`	Print a summary of the specified options in batch output.
`pivrel=x`	Resets the relative ratio between the largest column entry and an acceptable pivot value. The default value is 1.0e-3. In the numerical pivoting algorithm the allowed minimum pivot value is determined by $epsrel = max(pivrel * maxval, pivtol)$ where $maxval$ is the maximum element in the column where a pivot is sought (partial pivoting).
pivtol=x	Resets the absolute minimum value for a matrix entry to be accepted as a pivot. The default value is 1.0e-13.
`reltol=x`	Resets the relative error tolerance of the program. The default value is 0.001 (0.1 percent).
`tnom=x`	Resets the nominal temperature. The default value is 27 ° C (300K).
`trtol=x`	Resets the transient error tolerance. The default value is 7.0. This parameter is an estimate of the factor by which *WRspice* overestimates the actual truncation error.
`vntol=x`	Resets the absolute voltage error tolerance of the program. The default value is 1 microvolt.

2.4.5 `.table` Line

General Form:

`.table` *name* `[ac]` x_0 v_0 x_1 v_1 ... x_N v_N

Examples:

```
.table tab1 0 .1 1n .2 2n .4 3n .2 4n .1 5n 0
.table xgain 0 0 1 1 1 1.5 4 2
.table acvals ac 0 1.0 0, 1e3 .98 .03, ...
.table zz (0 table xgain 4 2)
.table tab1 0 1 .2 .5 .4 table txx .8 .5e-2
```

The `.table` line defines a tabulation of data points which can be referenced from other lines in the SPICE file. The data are listed in sequence with respect to the ordinate x_N. The elements are separated by white space or commas, and may optionally be surrounded by parentheses. Generally, the table construct consists of many lines, using the '+' or backslash line continuation mechanism. When a table is referenced, the data value returned is interpolated from the values in the table.

The x_i in the `.table` line are values of the independent variable (i.e., the variable given as an argument to the referencing function). The v_i entries can be numbers, or a reference to another table in the form

`table` *subtab_name*

in which case that table will be referenced to resolve the data point.

If the `ac` keyword is given, the data numbers v_i are expected to be complex values, which are expressed as two values; the real value followed by the imaginary value. Any sub-tables referenced must also have the `ac` keyword given. The `ac` tables provide data for frequency-domain analysis. Without `ac`, all values are real, and the table is intended for use in dc or transient analysis.

A non-ac table is referenced through a tran-function (see 2.12.3). Tables with the `ac` keyword given are referenced through the `ac` keyword in dependent and independent sources (see 2.12 and 2.12.4).

Let `x` be the input variable associated with the device referencing a table. The table is evaluated as follows:

`x` $< x_0$	$val = v_0(x_0)$
$x_0 <$ `x` $< x_1$	$val = v_0$(`x`) if v_0 is a table
	$val =$ interpolation of $v_0(x_0)$ and $v_1(x_1)$ if v_0 is a number
	...
`x` $> x_N$	$val = v_N$(`x`) if v_N is a table
	$val = v_N$ if v_N a number
	$val = v_{N-1}(x_N)$ if v_N is omitted

See the section A.3 for sample input files which illustrate the use of the `.table` line.

2.5 Parameters and Expressions

2.5.1 Single-Quoted Expressions

Text enclosed in single quotes (') will be evaluated as an expression as the file is read and the string will be replaced by the result. Single-quoted expressions outside of `.subckt` blocks are expanded before variable substitution and subcircuit expansion. In this case, since evaluation is performed before shell substitution, the expression can not contain shell variables or other '`$`' references, but it can contain vectors (which must be defined before the circuit is read). Single-quoted expressions that appear within `.subckt` blocks are evaluated after variable substitution during subcircuit expansion. Thus, these expressions can contain shell '`$`' constructs.

If the single-quoted expression appears to the left of an '=' sign, as an assignment, no substitution is done. In this case, the quotes are ignored.

2.5.2 `.param` Line

General Form:

`.param` *name* = *value* [*name* = *value*] ...

Example:

```
.param p1 = 1.23 p2 = '2.5*p1'
```

This assigns the text *value* to the text token *name*. If the text string *name*, delimited by space or one of `,)([]='"` but not with '`=`' to the right is found in the text, it is replaced by its value. The '`%`' concatenation character is recognized. The concatenation character is used to separate the token from the other text: for example `RES%K` allows `RES` to be identified as a token, and if `RES` is '1' the substitution would yield '1K'. The *name* token must be surrounded by non-alphanumeric characters.

Substitutions occur on a second pass, so the order of definition and reference is not important (except when used in `.if` or `.elif`, described below). Substitutions are not performed in Verilog blocks, but are performed everywhere else. The `.param` lines always have global scope, meaning that they apply to the entire circuit, whether or not they are located within `.subckt` blocks.

Values can contain parameter references, i.e., nesting is accepted.

References to parameters outside of any `.subckt` definition are evaluated before variable expansion. References within `.subckt` definitions are evaluated after variable substitution, during subcircuit expansion. Parameters defined in a `.param` line are overridden by those defined in subcircuit definitions or calls in this case.

The *value* must be a single token, or be enclosed by single or double quotes. If double quotes are used, they are stripped when the substitution is applied. Single quotes are retained in the substitution, as single quotes have significance in delimiting an expression. Parameter substitution is performed whether or not the substitution variable in part of a single or double quoted string.

Parameters can be accessed as vectors through the syntax "`@`*paramname*". The value of the vector is the numerical value of the parameter-expanded *value* string. These vectors are read-only, i.e., parameters can not be set through vectors.

2.5.3 `.if`, `.elif`, `.else`, and `.endif` Lines

General Form:
`.if` *expression* `[ = ` *expression*`]`

General Form:
`.elif` *expression* `[ = ` *expression*`]`

General Form:
`.else`

General Form:
`.endif`

Example:
```
SPICE deck title line
...
.param use_new_mod = 8
...
.if use_new_mod = 8
.model m1 nmos(level=8 ...
.elif use_new_mod = 9
.model m1 nmos(level=9 ...
.elif use_new_mod
.model m1 nmos(level=10 ...
.else
.model m1 nmos(level = 3 ...
.endif
```

The *WRspice* input file syntax supports conditional blocks, through use of these directives. The *expression* can involve constants, parameter names from a `.param` line included in the file before the reference, or vectors and shell variables defined *before* the file is read. It does not understand variables implicitly set by inclusion in the `.options` line, or parameters set in `.subckt` lines or references. The scope for these constructs is always global, meaning that they apply to lines of text in the file without regard to `.subckt` block boundaries.

If the single *expression* is nonzero, or the two expressions yield the same result, the lines following `.if` up to the matching `.elif`, `.else` or `.endif` are read, and if an `.else` block follows, those lines to the matching `.endif` are discarded. If the single *expression* if zero or the two expressions do not match, the lines following `.if` to the matching `.elif`, `.else` or `.endif` are discarded, and if a `.else` line follows, the lines following `.else` to `.endif` are retained. These blocks can be nested. The action is similar to the C preprocessor.

This filtering is performed early in the parsing of the file, so that the `.if`, etc. lines can enclose script lines, verilog blocks, etc., and not simply circuit lines. The lines not in scope are never saved in memory. However, `.include` lines are read before the filtering is performed, so that even if a `.include` line is out of scope, if the file can not be opened, an error message will appear.

2.6 Subcircuits

A subcircuit that consists of *WRspice* elements can be defined and referenced in a fashion similar to device models. The subcircuit is defined in the input file by a grouping of element lines; the program then automatically inserts the group of elements wherever the subcircuit is referenced. There is no limit on the size or complexity of subcircuits, and subcircuits may contain other subcircuits. An example of subcircuit usage is given in A.3.

2.6.1 `.subckt` Line

General Form:

`.subckt` *subnam n1* `[`*n2* `...]` `[`*param1=val1 param2=val2* `...]`

Examples:

```
.subckt opamp 1 2 3 4
.subckt stage1 3 10 2 resis=2k cap=1nf
```

A subcircuit definition begins with a `.subckt` line. The parameter *subnam* is the subcircuit name, and *n1*, *n2*, ... are the external nodes, which cannot be zero. The group of element lines which immediately follow the `.subckt` line define the subcircuit. The last line in a subcircuit definition is the `.ends` line (see below). Control lines should not appear within a subcircuit definition, however subcircuit definitions may contain anything else, including other subcircuit definitions, device models, and subcircuit calls (see below). Note that any device models or subcircuit definitions included as part of a subcircuit definition are strictly local (i.e., such models and definitions are not known outside the subcircuit definition). Also, any element nodes not included on the `.subckt` line or in `.global` lines are strictly local, with the exception of 0 (ground) which is always global.

The subcircuit declaration line can contain an optional list of *param=value* pairs. The *params* are tokens which must start with a letter, which can appear in the subcircuit lines. These are not shell variables, so there is no '`$`' or other punctuation, but the '`%`' concatenation character is recognized. The concatenation character is used to separate the token from the other text: for example `RES%K` allows `RES` to be identified as a token, and if `RES` is '1' the substitution would yield '1K'. The *param* token must be surrounded by non-alphanumeric characters. Parameters defined this way are always superseded by parameters given in the subcircuit call, but override parameters defined in `.param` lines.

WRspice can handle duplicate formal node args in `.subckt` lines. It does so by assigning a new node to one of the duplicates, then inserting a voltage source between the two nodes, which is added to the

subcircuit text. This mainly solves a problem related to files generated by Xic. If two or more subcircuit terminals are attached to the same wire net, the resulting *.subckt* line will have duplicate nodes. In the limiting case where a subcircuit consists only of a wire with two connections, the subcircuit would in addition be empty.

Example:

```
.subckt xxx 1 1
.ends
```

is converted to

```
.subckt xxx 1 _#0
v_xxx_0 1 _#0
.ends
```

during subcircuit expansion, which avoids an empty subcircuit and has the intended effect of instances shorting the two terminals together.

2.6.2 .ends Line

General Form:
`.ends` [*subnam*]

Example:
```
.ends opamp
```

This line must be the last one for any subcircuit definition. The subcircuit name, if included, indicates which subcircuit definition is being terminated; if omitted, all subcircuits being defined are terminated. The name is needed only when nested subcircuit definitions are being made.

2.6.3 Subcircuit Calls

General Form:
`x`*name n1* [*n2 n3* ...] *subnam* [*param1=val1 param2=val2* ...]

Example:
```
x1 2 4 17 3 1 multi
```

Subcircuits are used in WRspice by specifying pseudo-elements beginning with the letter '`x`' or '`X`', followed by the circuit nodes to be used in expanding the subcircuit.

Note that when a circuit is parsed, all devices and local nodes in subcircuits are renamed as *device-type*:*subcktname*:*devicename*. Nested subcircuit instances will have multiple colon-separated qualifiers. WRspice will also accept subcircuit names with components separated by periods, so long as the names do not clash with vector names specifiable as *plotname.value*.

The names and default values of the *params* are specified in the `.subckt` line. During subcircuit expansion, the *param* tokens are replaced by their corresponding *value* tokens in the text. If a list of *params* is given in the subcircuit instantiation line, those values will supersede the defaults in that subcircuit instance, and parameters set in `.param` lines.

Example:

```
.subckt resistor 1 2 resis=1k
r1 1 2 resis
.ends

x1 3 4 resistor resis = 500
x2 5 6 resistor
x3 7 8 resistor resis=2k
```

2.7 Analysis Specification

WRspice provides the analysis capabilities tabulated below. Monte Carlo and operating range analysis (described in Chpt. 5) require a special input file format, while other types of analysis can be specified in a standard input deck.

Analyses will pause if *WRspice* receives an interrupt signal, i.e., the user types **Ctrl-C** while *WRspice* has the keyboard focus. The resume command can be used to resume the analysis.

By default, the maximum size of the data produced by an analysis run is limited to 32Mb. This can be changed by setting the variable `maxdata` to the desired value in Kb, using the set command or the **Sim Defs** tool from the **Tools** menu of the Tool Control window. In transient analysis, if the `steptype` is not set to "`nousertp`", the run will abort at the beginning if the memory would exceed the limit. Otherwise, the run will end when the limit is reached.

The table below lists the basic analysis types and input file keyword.

`.ac`	AC Small-Signal Analysisa
`.dc`	DC Analysisa
`.disto`	Small-Signal Distortion Analysisa
`.noise`	Small-Signal Noise Analysisa
`.op`	Operating Pointa
`.pz`	Pole-Zero Analysisa
`.sens`	DC or Small-Signal AC Sensitivity Analysisa
`.tf`	DC or Small-Signal AC Transfer Function Analysisa
`.tran`	Transient Analysisa

WRspice has an exclusive multi-dc analysis feature. This allows ac, noise, transfer function, sensitivity, and transient analyses to have an additional dc sweep specification, resulting in the analysis being performed at each dc operating point, producing a multi-dimensional output plot.

For exampls:

Example:

```
.ac dec 10 1Hz 1Khz dc v1 0 2 .1 v2 4.5 5.5 .25
```

This will perform an ac analysis with the dc sources `v1` and `v2` stepped through the respective ranges. The resulting output vectors will have dimensions [5,21,61], as can be seen with the display command interactively. This represents 61 points of frequency data at 21 `v1` values at 5 `v2` values. Typing "`plot v(1)`" (for example) would plot all 21*5 analyses on the same scale (you probably don't want to do this). One can also type (as examples) "`plot v(1)[1]`" to plot the results for `v2` = 4.75, or "`plot v(1)[0][1]`" for `v2` = 4.5, `v1` = .1, etc. Range specifications also work, for example "`plot v(1)[2][0,2]`" plots the values for `v2` = 5.0, `v1` = 0.0, 0.1, 0.2.

Warning: The memory space required to hold the plot data can grow quite large, so be reasonable.

2.7.1 .ac Line

The ac small-signal portion of *WRspice* computes the ac output variables as a function of frequency. The program first computes the dc operating point of the circuit and determines linearized, small-signal models for all of the nonlinear devices in the circuit. The resultant linear circuit is then analyzed over a user-specified range of frequencies. The desired output of an ac small-signal analysis is usually a transfer function (voltage gain, transimpedance, etc). If the circuit has only one ac input, it is convenient to set that input to unity and zero phase, so that output variables have the same value as the transfer function of the output variable with respect to the input.

General Form:

`.ac dec|oct|lin` *np fstart fstop* `[dc` *dc_args*`]`

Examples:

```
.ac dec 10 1 10k
.ac dec 10 1k 100meg
.ac lin 100 1 100hz dc vcc 10 15 5
.ac dec 10 1meg 1g dc vdd 5 7.7 .25
```

The keyword `dec` specifies decade variation, with *np* the number of points per decade. The keyword `oct` specifies octave variation, with *np* the number of points per octave, and `lin` specifies linear variation, with *np* the number of points. The two parameters *fstart* (the starting frequency), and *fstop* (the final frequency) complete the basic analysis specification. If this line is included in the circuit file, *WRspice* will perform an ac analysis of the circuit over the specified frequency range. Note that in order for this analysis to be meaningful, at least one voltage or current source must have been specified with an `ac` value.

The optional dc sweep is a dc analysis specification which will cause the ac analysis to be performed at each point of the dc sweep. The small-signal parameters are reevaluated at every sweep point, and the output vectors will be multidimensional.

In interactive mode, the `ac` command, which takes the same arguments as the `.ac` line, can be used to initiate ac analysis.

AC analysis is not available if Josephson junctions are included in the circuit.

2.7.2 .dc Line

The dc analysis portion of *WRspice* determines the dc operating point of the circuit with inductors shorted and capacitors opened. The dc analysis is used to generate dc transfer curves: a specified voltage or current source is stepped over a user-specified range and the dc output variables are stored for each sequential source value.

General Form:

`.dc` *src1 start1* `[`*stop1* `[`*incr1*`]]` `[`*src2 start2* `[`*stop2* `[`*incr2*`]]]`

Examples:

```
.dc vin 0.25 5.0 0.25
```

```
.dc vds 0 10 .5 vgs 0 5 1
.dc vce 0 10 .25 ib 0 10u 1u
.dc vdd 5 6
```

The `.dc` line defines the dc transfer curve source and sweep limits. The variation may be in one or two dimensions, depending upon whether the second block is provided. The *src* parameter is the name of a voltage or current source to vary. The *start*, *stop*, and *incr* parameters are the starting, final, and incrementing values, respectively. If *incr* parameter is not supplied, analysis will be performed at the *start* and *stop*. If in addition the *stop* parameter is not given, analysis will be performed at *start*, i.e., the level is fixed. A parameter can be omitted only if all parameters to the right are also omitted.

A second source (*src2*) may optionally be specified with associated sweep parameters. In this case, the first source will be swept over its range for each value of the second source. This option can be useful for obtaining semiconductor device output characteristics.

The first example will cause the value of the voltage source `vin` to be swept from 0.25 volts to 5.0 volts in increments of 0.25 volts.

In *WRspice*, other types of analysis (ac, noise, transfer function, sensitivity, and transient) can be chained to a dc analysis specification. In this case, the requested analysis is performed at each successive operating point, as specified by the dc part of the analysis specification. The resulting circuit variables are saved as multi-dimensional vectors, which can subsequently be saved in a rawfile or plotted (together or as individual traces).

In interactive mode, the `dc` command, which takes the same arguments as the `.dc` line, can be used to initiate dc analysis.

DC analysis is not available if Josephson junctions are included in the circuit.

2.7.3 `.disto` Line

The distortion analysis portion of *WRspice* computes steady-state harmonic and intermodulation products for small input signal magnitudes. If signals of a single frequency are specified as the input to the circuit, the complex values of the second and third harmonics are determined at every point in the circuit. If there are signals of two frequencies input to the circuit, the analysis finds the complex values of the circuit variables at the sum and difference of the input frequencies, and at the difference of the smaller frequency from the second harmonic of the larger frequency.

Distortion analysis is supported in *WRspice* only through residual incorporation from code imported from Berkeley SPICE3. This code is particularly complex, poorly documented, and ugly. Distortion analysis has not been tested, and may not work.

Distortion analysis is included for the following nonlinear devices: diodes, bipolar transistors, JFETs, MOSFETs (levels 1, 2, 3 and BSIM1) and MESFETS. All linear devices are automatically supported by distortion analysis. If there are switches present in the circuit, the analysis will continue to be accurate provided the switches do not change state under the small excitations used for distortion calculations.

General Form:

`.disto dec|oct|lin` *np fstart fstop* [*f2overf1*] [`dc` *dc_args*]

Examples:

```
.disto dec 10 1khz 100mhz
.disto dec 10 1khz 100mhz 0.9
```

A multi-dimensional Volterra series analysis is performed using a multi-dimensional Taylor series to represent the nonlinearities at the operating point. Terms of up to third order are used in the series expansions.

If the optional parameter *f2overf1* is not specified, a harmonic analysis is performed — i.e., distortion is analyzed in the circuit using only a single input frequency *f1*, which is swept as specified by arguments of the `.disto` line exactly as in an `.ac` line. Inputs at this frequency may be present at more than one input source, and their magnitudes and phases are specified by the arguments of the `distof1` keyword in the input file lines for the input sources. The arguments of the `distof2` keyword are not relevant in this case. The analysis produces information about the ac values of all node voltages and branch currents at the harmonic frequencies $2f1$ and $3f1$, vs. the input frequency *f1* as it is swept. A value of 1 (as a complex distortion output) signifies $cos(2\pi(2f1)t)$ at $2f1$ and $cos(2\pi(3f1)t)$ at $3f1$, using the convention that 1 at the input fundamental frequency is equivalent to $cos(2\pi f1t)$.

The distortion component desired ($2f1$ or $3f1$) can be selected using commands in *WRspice*, and then printed or plotted. Normally, one is interested primarily in the magnitude of the harmonic components, so the magnitude of the ac distortion value is considered. It should be noted that these are the ac values of the actual harmonic components, and are not equal to HD2 and HD3. To obtain HD2 and HD3, one must divide by the corresponding ac values at *f1*, obtained from an `.ac` line. This division can be done using *WRspice* commands.

If the optional *f2overf1* parameter is specified, it should be a real number between (and not equal to) 0.0 and 1.0; in this case, a spectral analysis is performed. The circuit is considered with sinusoidal inputs at two different frequencies *f1* and *f2*. Frequency *f1* is swept according to the `.disto` line options exactly as for the `.ac` card. Frequency *f2* is kept fixed at a single frequency as *f1* sweeps — the value at which it is kept fixed is equal to *f2overf1*fstart*. Each voltage and current source in the circuit may potentially have two (superimposed) sinusoidal inputs for distortion, at the frequencies *f1* and *f2*. The magnitude and phase of the *f1* component are specified by the arguments of the `distof1` keyword in the source's input line, as described in 2.12; the magnitude and phase of the *f2* component are specified by the arguments of the `distof2` keyword. The analysis produces plots of all node voltages/branch currents at the intermodulation product frequencies $f1 + f2$, $f1 - f2$, and $(2f1) - f2$, vs the swept frequency *f1*. The IM product of interest may be selected using the setplot command, and displayed with the print and plot commands. As in the harmonic analysis case, the results are the actual ac voltages and currents at the intermodulation frequencies, and need to be normalized with respect to `.ac` values to obtain the IM parameters.

If the `distof1` or `distof2` keywords are missing from the description of a voltage or current source, then that source is assumed to have no input at the corresponding frequency. The default values of the magnitude and phase are 1.0 and 0.0 respectively. The phase should be specified in degrees.

It should be noted that the number *f2overf1* should ideally be an irrational number, and that since this is not possible in practice, efforts should be made to keep the denominator in its fractional representation as large as possible, certainly above 3, for accurate results. If *f2overf1* is represented as a fraction A/B, where A and B are integers with no common factors, B should be as large as possible. Note that $A < B$ because *f2overf1* is constrained to be < 1). To illustrate why, consider the cases where *f2overf1* is $49/100$ and $1/2$. In a spectral analysis, the outputs produced are at $f1 + f2$, $f1 - f2$ and $2f1 - f2$. In the latter case, $f1 - f2 = f2$, so the result at the $f1 - f2$ component is erroneous because there is the strong fundamental $f2$ component at the same frequency. Also, $f1 + f2 = 2f1 - f2$ in the latter case, and each result is erroneous individually. This problem is not seen in the case where $f2overf1 = 49/100$, because $f1 - f2 = 51/100f1$ which is not equal to $49/100f1 = f2$. In this case, there will be two very closely spaced frequency components at $f2$ and $f1 - f2$. One of the advantages of the Volterra series technique is that it computes distortions at mix frequencies expressed symbolically (i.e. n·f1 ± m·f2), therefore one is able to obtain the strengths of distortion components accurately even if the separation between them

is very small, as opposed to transient analysis for example. The disadvantage is of course that if two of the mix frequencies coincide, the results are not merged together and presented, though this could presumably be done as a postprocessing step. Currently, the interested user should keep track of the mix frequencies and add the distortions at coinciding mix frequencies together should it be necessary.

The optional dc sweep is a ¡a href=".dc"¿dc analysis¡/a¿ specification which will cause the distortion analysis to be performed at each point of the dc sweep. The small-signal parameters are reevaluated at every sweep point, and the output vectors will be multidimensional.

In interactive mode, the disto command, which takes the same arguments as the `.disto` line, can be used to initiate distortion analysis.

2.7.4 `.noise` Line

The noise analysis portion of *WRspice* performs analysis of device-generated noise for the given circuit. When provided with an input source and an output node, the analysis calculates the noise contributions of each device (and each noise generator within the device) to the output node voltage. It also calculates the level of input noise from the specified input source to generate the equivalent output noise. This is done for every frequency point in a specified range — the calculated value of the noise corresponds to the spectral density of the circuit variable viewed as a stationary Gaussian stochastic process.

After calculating the spectral densities, noise analysis integrates these values over the specified frequency range to arrive at the total noise voltage/current (over this frequency range). This calculated value corresponds to the variance of the circuit variable viewed as a stationary Gaussian process.

General Form:

`.noise v(`*out*`[,`*ref*`]) `*src* `dec|oct|lin` *pts fstart fstop* `[`*summary_pts*`] [dc` *dc_args*`]`

Examples:

```
.noise v(5) vin dec 10 1khz 100mhz
.noise v(5,3) v1 oct 8 1.0 1.0e6 1 dc vee -5 -3 1
```

This directive initiates a noise analysis of the circuit. The parameter *out* is the node at which the total output noise is desired, or if *ref* is specified, then the noise voltage v(*out*) - v(*ref*) is calculated. By default, *ref* is assumed to be ground. The parameter *src* is the name of a voltage or current source to which input noise is referred, with *pts*, *fstart* and *fstop* being the `.ac` parameters that specify the frequency range over which analysis is desired. The optional *summary_pts* is an integer; if specified, the noise contributions of each noise generator is produced every *summary_pts* frequency points.

The `.noise` analysis produces two plots — one for the Noise Spectral Density curves and one for the total Integrated Noise over the specified frequency range. All noise voltages/currents are in squared units (V^2/Hz and A^2/Hz for spectral density, V^2 and A^2 for integrated noise).

The optional dc sweep is a dc analysis specification which will cause the noise analysis to be performed at each point of the dc sweep. The small-signal parameters are reevaluated at every sweep point, and the output vectors will be multidimensional.

In interactive mode, the noise command, which takes the same arguments as the `.noise` line, can be used to initiate noise analysis.

2.7.5 .op Line

General Form:
```
.op
```

The inclusion of this line in an input file will force *WRspice* to determine the dc operating point of the circuit with inductors shorted and capacitors opened. This is done automatically prior to most other analyses, to determine the operating point of the circuit, yielding transient initial conditions or the linearized models for nonlinear devices for small-signal analysis. It will not be done in transient analysis if the `uic` keyword is given in the transient analysis specification, or if Josephson junctions are present in the circuit.

WRspice performs a dc operating point analysis if no other analyses are requested, and there are no Josephson junctions in the circuit.

In interactive mode, the `op` command can be used to compute the operating point.

In operating point analysis, any `.save` or `save` directives will be ignored. All node voltages and branch currents will be saved in an "op" plot in interactive mode.

2.7.6 .pz Line

The pole-zero analysis portion of *WRspice* computes the poles and/or zeros in the small-signal ac transfer function. The program first computes the dc operating point and then determines the linearized, small-signal models for all the nonlinear devices in the circuit. This circuit is then used to find the poles and zeros.

Two types of transfer functions are allowed: one of the form (output voltage)/(input voltage) and the other of the form (output voltage)/(input current). These two types of transfer functions cover all the cases and one can find the poles/zeros of functions like input/output impedance and voltage gain. The input and output ports are specified as two pairs of nodes.

The pole-zero analysis works with resistors, capacitors, inductors, linear controlled sources, independent sources, BJTs, MOSFETs, JFETs and diodes. Transmission lines and Josephson junctions are not supported.

General Form:
```
.pz node1 node2 node3 node4 cur|vol pol|zer|pz
```

Examples:
```
.pz 1 0 3 0 cur pol
.pz 2 3 5 0 vol zer
.pz 4 1 4 1 cur pz
```

The keyword `cur` stands for a transfer function of the type (output voltage)/(input current) while `vol` stands for a transfer function of the type (output voltage)/(input voltage). The keyword `pol` stands for pole analysis only, `zer` for zero analysis only and `pz` for both. This feature is provided mainly because if there is a nonconvergence in finding poles or zeros, then, at least the other can be found. Finally, *node1* and *node2* are the two input nodes and *node3* and *node4* are the two output nodes. Thus, there is complete freedom regarding the output and input ports and the type of transfer function.

In interactive mode, the `pz` command, which takes the same arguments as the `.pz` line, can be used to initiate pole-zero analysis.

2.7.7 .sens Line

General Form:

```
.sens outvar [ac dec|oct|lin np fstart fstop] [dc dc_args]
```

Examples:

```
.sens v(1,out)
.sens v(out) ac dec 10 100 100k
.sens i(vtest)
```

The sensitivity of *outvar* to all non-zero device parameters is calculated when the sensitivity analysis is specified. The parameter *outvar* is a circuit variable (node voltage or branch current). Without the ac specification, the analysis calculates sensitivity of the DC operating point value of *outvar*. If an ac sweep specification is included, the analysis calculates sensitivity of the ac values of *outvar*. The parameters listed for ac sensitivity are the same as in an ac analysis. The output values are in dimensions of change in output per unit change of input (as opposed to percent in output or per percent of input).

The optional dc sweep is a dc analysis specification which will cause the sensitivity analysis to be performed at each point of the dc sweep. The small-signal parameters are reevaluated at every sweep point, and the output vectors will be multidimensional.

In interactive mode, the sens command, which takes the same arguments as the .sens line, can be used to initiate sensitivity analysis.

2.7.8 .tf Line

General Form:

```
.tf outsrc | v(n1[,n2]) insrc [ac dec|oct|lin np fstart fstop] [dc dc_args]
```

Examples:

```
.tf v(5,3) vin
.tf I(vload) vin ac dec 10 1 1e12
.tf v(2) vin ac dec 10 1 1meg
.tf v(4) vx dc vcc 5 10 1
.tf v(5) vy ac dec 10 1 1meg dc 5 10 1
```

The .tf line defines the small-signal output and input for the dc or ac small-signal analysis. The first parameter is the small-signal output variable (node voltage or name of inductor or voltage source for branch current) and *insrc* is the small signal input source. If this line is included, *WRspice* computes the dc or ac small-signal value of the transfer function (output/input), input resistance or impedance, and output resistance or impedance. For the first example, *WRspice* would compute the ratio of v(5,3) to vin, the small signal input resistance at vin, and the small-signal output resistance measured across nodes 5 and 3. If the ac parameters are given, the .tf line produces output vectors representing the impedance and other parameters at each frequency point.

The optional dc sweep is a dc analysis specification which will cause the transfer function analysis to be performed at each point of the dc sweep. The small-signal parameters are reevaluated at every sweep point, and the output vectors will be multidimensional.

In interactive mode, the tf command, which takes the same arguments as the .tf line, can be used to initiate transfer function analysis.

2.7.9 .tran Line

The transient analysis portion of *WRspice* computes the transient output variables as a function of time over a user-specified time interval. The initial conditions are automatically determined by a dc analysis. All sources which are not time dependent (for example, power supplies) are set to their dc value. The transient time interval is specified on a `.tran` control line.

General Form:

```
.tran tstep1 tstop1 [tstart1 [tmax]] [tstep2 tstop2 ...  tstepN tstopN] [uic]
    [scroll | segment base delta] [dc dc_args]
```

Examples:

```
.tran 1ns 100ns
.tran 1ns 1000ns 500ns
.tran 10ns 1us uic
```

The *tstep* values are the printing or plotting increments for output, in the ranges

tstep1: *tstart* <= *time* < *tstop1*
tstep2: *tstop1* <= *time* < *tstop2*
...
tstepN: *tstopN-1* <= *time* < *tstopN*

The *tstart* is the initial time, assumed 0 if not given. The transient analysis always begins at time zero internally. In the interval [0, *tstart*), the circuit is analyzed (to reach a steady state), but no outputs are stored. Subsequently, the circuit is analyzed and outputs are stored. The parameter *tmax* is the maximum internal timestep size that *WRspice* will use. The internal timestep is computed dynamically from the circuit. The output generated at the specified *tstep* points is interpolated from the "real" internal time points. The *tmax* parameter can be used when one wishes to guarantee a computing interval which is smaller than the output increment, *tstep*.

The `uic` keyword (use initial conditions) is an optional keyword which indicates that the user does not want *WRspice* to solve for the quiescent operating point before beginning the transient analysis. If Josephson junctions are present, `uic` will always be taken as if set. If this keyword is specified, *WRspice* uses the values specified using `ic=...` on the various elements as the initial transient condition and proceeds with the analysis. If the `.ic` line has been given, then the node voltages on the `.ic` line are used to compute the initial conditions for the devices. See the description of the `.ic` line (2.4.2) for its interpretation when `uic` is not specified.

The `scroll` keyword is useful in the `tran` command in interactive mode. If the `scroll` keyword is given, the simulation will continue indefinitely, until stopped by a `stop` command or interrupt. The time range of data *tstop* - *tstart* behind the current time is retained in the plot.

If the `segment` keyword is given, along with a character string token *base* and real value *delta*, individual rawfiles are output for each range of delta as the simulation advances. The internal plot data are cleared after each segment is output. The files are named with the *base* given, as *base*.`s00`, *base*.`s01`, etc. This will not happen if a rawfile is being produced. If `scroll` is also given, it is ignored. If a dc analysis is chained, it is legitimate to pass a *delta* of zero, in which case a file is produced for each cycle. Otherwise, the *delta* should be a multiple or submultiple of `tstop`, or the files will be difficult to interpret. It is an error if *delta* is nonpositive if there is no chained dc analysis. The purpose of this feature is to facilitate extremely lengthly transient analysis runs.

The optional dc sweep is a dc analysis specification which will cause the transient analysis to be performed at each point of the dc sweep. The dc operating point is reevaluated at every sweep point,

and output vectors will be multidimensional. The optional parameters before `dc` can be omitted in this case, as the parser recognizes the "`dc`" keyword as the start of a dc sweep specification. If the `scroll` keyword is given, the dc sweep is not available.

In interactive mode, the `tran` command, which takes the same arguments as the `.tran` line, can be used to initiate transient analysis.

During transient analysis, a special vector `@delta` maps to the (most recent) internal time step. To use in a plot, it must be saved first (using a `.save` line or the `save` command). It is sometimes useful or interesting to see how the internal timestep varies in a simulation.

2.8 Output Generation

In these lines, outputs can be specified using the SPICE2 notation. The form is `v`*xx*`(`*node1*`[,`*node2*`])`, or `i`*xx*`(`*branch_device*`)`. The *xx* can be left out, indicating the basic voltage or current, or be one of the following.

`m`	Magnitude
`p`	Phase
`r`	Real part
`i`	Imaginary part
`db`	Decibel value ($20log_{10}()$)

These forms are not usually needed for other than ac analysis. The `(`*node1*`,` *node2*`)` notation indicates a voltage difference between nodes *node1* and *node2*. If *node2* and the associated comma are left out, the ground node is assumed. Output variables for noise, distortion, and some other analyses have a different general form. See the description of the analysis for the output variable names.

2.8.1 `.save` Line

General Form:

`.save [output]` *vector vector ...*

Examples:

```
.save i(vin) v(3)
.save @m1[id] vm(3,2)
```

When a rawfile is produced, the vectors listed in the `.save` line are recorded in the rawfile. The standard vector names are accepted; for the form `v`(a, b), the vectors `v`(a) and `v`(b) are saved (*not* the difference vector). The voltage vector(s) are saved for each of the forms `vm`(), `vp`(), `vr`(), `vi`(), and `vdb`(). Similarly, the branch current is saved on mention of any of the corresponding `i`() forms. A token without parenthesis is interpreted as a node name, e.g. , "1" implies `v(1)` is saved.

If no `.save` line is given or no entries are found, then all vectors produced by the analysis are saved. If `.save` lines are given, only those vectors specified are saved. The keyword "`output`" specifies that the the vector names found in all `.print`, `.plot`, and `.four` lines are to be saved, in addition to any vectors listed on the `.save` lines.

There is an analogous `save` command available within *WRspice*.

2.8.2 .print Line

General Form:
```
.print prtype ov1 [ov2 ... ov8]
```

Examples:
```
.print tran v(4) i(vin)
.print dc v(2) i(vsrc) v(23,17)
.print ac vm(4,2) vr(7) vp(8,3)
```

The `.print` line defines the contents of a tabular listing of one to eight output variables. The parameter *prtype* is the type of the analysis (dc, ac, tran, noise, etc.) for which the specified outputs are desired. Variables can take the forms tabulated above. The actual format recognized is that of the print command, which is far more general. There is no limit on the number of `.print` lines for each type of analysis.

2.8.3 .plot Line

General Form:
```
.plot pltype ov1 [ov2 ... ] [(min,max)]
```

Examples:
```
.plot dc v(4) v(5) v(1)
.plot tran v(17,5) (2,5) i(vin) v(17) (1,9)
.plot ac vm(5) vm(31,24) vdb(5) vp(5)
.plot disto hd2 hd3(R) sim2
.plot tran v(5,3) v(4) (0,5) v(7) (0,10)
```

The `.plot` line defines the contents of one plot from one or more output variables. In SPICE2, the number of variables was limited to eight, but *WRspice* has no preset limit. In SPICE2, each variable could be followed by a comma-separated pair of numbers in parentheses which indicated the plotting range. *WRspice* supports this construct only as the last argument, and it applies to all variables. The parameter *pltype* is the type of analysis (ac, dc, tran, etc.) for which the specified outputs are desired. The syntax for the *ov*N is identical to that for the `.print` line and for the plot command in the interactive mode.

This line generates ascii plots in batch mode, for compatibility with SPICE2. The overlap of two or more traces on any plot is indicated by the letter X.

When more than one output variable appears on the same plot, the first variable specified is printed as well as plotted. If a printout of all variables is desired, then a companion `.print` line should be included.

There is no limit on the number of `.plot` lines specified for each type of analysis.

2.8.4 .four Line

General Form:
```
.four freq ov1 [ov2 ov3 ...]
```

Example:
```
.four 100k v(5)
```

The `.four` line controls whether *WRspice* performs a Fourier analysis as a part of the transient analysis. The parameter *freq* is the fundamental frequency, and *ov1,...*, are the output variables for which the analysis is desired. The Fourier analysis is performed over the interval [`tstop`-period, `tstop`], where `tstop` is the final time specified in transient analysis, and `period` is one period of the fundamental frequency. The dc component and the first nine harmonics are determined. For maximum accuracy, `tmax` (see the `.tran` line, in 2.7.9) should be set to *period*/100 (or less for very high Q circuits).

2.8.5 `.width` Line

General Form:
```
.width out=wid
```

This line is ignored, except in batch mode. The *wid* is the number of columns to be used for printing output. Internally, this effectively sets the `width` variable.

2.9 Parameter Measurement

2.9.1 `.measure` Line

WRspice input files may contain `.measure` statements. These statements allow certain types of measurements to be performed during a simulation, either at a selected point or over a selected interval. The syntax is similar to the `.measure` statement in Hspice™.

General Form:
```
.measure dc|ac|tran result interval | point [measurements]
```

The `dc|ac|tran` specifies the analysis type for which the measurement is active. The *result* is a token, starting with a letter, which will be used to uniquely identify the results of the measurement. A vector with this name will be added to the current plot, if the measurement is successful.

The rest of the line can have various formats. The next part of the statement is either an *interval* specification, or a single *point* specification.

2.9.1.1 Interval Specification

The *interval* begins with the "trigger" and ends with the "target". The trigger and target are independently specified as follows:

form 1:

```
trig|targ at=value
```

form 2:

`trig|targ` *variable* `[val=]`*value* `[td=`*delay*`]` `[cross=`*crosses*`]` `[rise=`*rises*`]` `[fall=`*falls*`]`

form 3:

`trig|targ when` *expr1*`=`*expr2* `[td=`*delay*`]` `[cross=`*crosses*`]` `[rise=`*rises*`]` `[fall=`*falls*`]`

Separate specifications for `trig`/`targ` are used to specify the boundaries of the measurement interval in the `.measure` statement.

Form 1 is straightforward; the interval starts (`trig`) or ends (`targ`) at *value*. *Value* must be within the simulation range of the scale variable (e.g., time in transient analysis).

The same effect can be achieved with:

`from=`*value* `to=`*value*

The `from` keyword is equivalent to "`trig at`", the `to` keyword is equivalent to "`targ at`".

Form 2 allows the interval boundaries to be referenced to times when a variable crosses a threshold. The *variable* can be any vector whose value is available during simulation. The *value* is a constant which is used to measure crossing events. The `val=` which precedes the *value* is optional. At least one of the `rise/fall/cross` fields should be set. Their values are integers which represent the variable crossing the threshold a number of times. The `rise` indicates the variable rising through the threshold, `fall` indicates the variable decreasing from above to below the threshold, and `cross` indicates `rises + falls`. The interval boundary is set when the specified number of transitions is reached.

If the delay is specified, transition counting starts after the specified delay.

Example:

```
trig v(2) 2.5 td=0.1ns rise=2
```

This indicates that the interval begins at the second time `v(2)` rises above 2.5V after 0.1ns.

The third form is similar to the second form, except that crossings are defined when *expr1 = expr2*. These are expressions, which must be enclosed in parentheses if they contain white space or commas. A rise is defined as *expr1* going from less than to greater than *expr2*.

Note: In previous releases, it was recommended that single quotes be used to delimit complicated expressions. This will not work in the present *WRspice* release, since single quoted expressions are evaluated before simulation. Instead, the expression should be surrounded by parentheses. For example, a form such as "`'v(1) - v(2)'`" must be changed to "`(v(1) - v(2))`".

A point can be specified with a `trig` or `from` specification as for an interval. One can also use the following form:

`when` *expr1*=*expr2* `[td=`*delay*`]` `[cross=`*crosses*`]` `[rise=`*rises*`]` `[fall=`*falls*`]`

This works exactly as in "`trig when`".

2.9.1.2 Measurements

The following measurements are available when an interval has been specified.

`min` *expr*
Find the minimum value of *expr*.

`max` *expr*
Find the maximum value of *expr*.

`pp` *expr*
Find the (maximum - minimum) value of *expr*.

`avg` *expr*
Compute the average of *expr*.

`rms` *expr*
Compute the rms value of *expr*.

`pw` *expr*
This will measure the full-width half-maximum of a pulse contained in the interval. The baseline is taken as the initial or final value with the smallest difference from the peak value. The algorithm will measure the larger of a negative going or positive going pulse.

`rt` *expr*
This will measure the 10-90 percent rise or fall time of the edge contained in the interval. The reference start and final values are the values at the ends of the interval.

When a point has been specified, the only measurement form available is

`find` *expr*
Evaluate *expr* at point.

A `.measure` statement can contain any number of measurements. The created vector, which is added to the current plot, will be of length equal to the number of measurements, with the results placed in the vector in order.

The measurement scale point(s) in `.measure` statements are also saved in a vector, which is the scale for the result vector. If the measurement name is "`result`", then the scale vector is named "`result_scale`". The scale contains one or two values, depending on whether it is a point or interval measurement.

By default, nothing is printed on-screen for a `.measure` line during interactive simulation. If the keyword `print` appears in the `.measure` line, the results will be printed on the standard output. A more concise format can be obtained from the alternative keyword `print_terse`. The result vectors are created in all cases.

If the keyword `stop` appears in a `.measure` line, the analysis will be paused when *all* measurements are complete. Thus if the deck contains several `.measure` lines and `stop` is given in at least one, the analysis will pause when all of the measurements are complete, not just the one containing `stop`. The analysis can then be resumed with the resume command, or reset with the reset command.

When a `.measure` is included in an iterative analysis (Monte Carlo, loop, etc.), data are saved as follows. Before each iteration, the previous result vector and its scale are saved to the end of a "history" vector and scale, and are then deleted. The result vector and scale are recreated when the measurement is completed during the iteration. Thus, at the end of the analysis, for a measurement named "example", one would have the following vectors:

example	the result from the final trial
example_scale	the measurement interval or point in the last trial
example_hist	results from the prior trials
example_hist_scale	intervals from the prior trials

Thus, during each trial, the result vector will have the same properties as in a standard run. It can be used in the .control block of a Monte Carlo or operating range file (recall that $?*vector* can be used to query existence, and that if there is no checkPNTS vector defined, the .control block is called once at the end of each trial).

Multiple .measure lines can be "chained" in the following manner. The vector name following the from, to, trig, or targ keywords can be the name of another measure. In this case, the effective start time is the measure time of the referenced measure. The measure time is the end of the interval or the measure point. The td, rise, and other keywords can be used in the referencing measure. The td will be added to the imported time, and the other keywords operate in the normal way. If there are no keywords other than td specified, the time is the delay time plus the imported time.

Example:

```
.measure tran t1 trig v(5) val=.4m rise=3
.measure tran t2 trig v(5) val=.4m rise=4
.measure tran pw trig t1 td=20p targ t2 td=20p pw v(5) max v(5)
```

In this case, the measures t1 and t2 "frame" a period of an (assumed) repeating signal v(5). Note that no actual measurement is performed for these lines. Their purpose is to be referenced in the third line, which takes as its interval the t1–t2 interval delayed by 20 pS, and measures the pulse width and peak value.

It is possible to reference .measure results in sources. The referencing token has the same form as a circuit variable, with an optional index, i.e.

@*result*[*index*]

where the *index*, if used, is an integer that references a specific component of the result (0-based). The value is always zero for timepoints before the measurement has been performed, and a constant value afterward.

Example:

```
.measure tran peak from=50n to=150n max v(5)
.measure tran stuff trig v(4) val=4.5 rise=1 targ v(4) val=4.5 fall=2
+ min v(4) max v(4) pp v(4) avg v(4) rms v(4) print
vxx 1 0 @peak
vyy 2 0 @stuff[2]
```

In this example, during transient analysis, vxx is zero until 150 nS, where the measurement takes place, at which point it jumps to the value measured. Likewise, vyy is zero until the measurement, at which point it jumps to the third component ("pp v(4)") result. The resulting voltages can be used elsewhere in the circuit. Note that we have two implementations of a behavioral peak detector.

2.10 Control Script Execution

2.10.1 `.exec`, `.control`, and `.endc` Lines

WRspice includes a script parsing and execution facility, which uses a syntax similar to that of the UNIX C-shell and will be described in 3.12. Statements which are interpreted and executed by this facility can be included in circuit files through use of the `.exec` and `.control` tokens. These statements are enclosed in a block beginning with `.exec` or `.control` and ending with `.endc`. The `.exec`, `.control`, and `.endc` lines contain only the keyword.

General Form:

```
.exec
shell_command
...
.endc
```

Example:

```
.exec
set vmin = 2.5
.endc
```

General Form:

```
.control
shell_command
...
.endc
```

Example:

```
.control
let maxv = v(2)*v(19)
print maxv
.endc
```

The *shell_commands* are any commands which can be interpreted by the *WRspice* shell. The difference between `.exec` and `.control` is that for `.exec`, the commands are executed before the circuit is parsed, and for `.control`, the commands are executed after the circuit is parsed.

When the circuit is parsed, variable substitution is performed. Shell variable references begin with "`$`", and are replaced with the text to which the shell variable has been set, unless the character before the '`$`' is a backslash (\), which indicates a comment. The variable can be set from the shell with the `set` command, and a variable is also set if it is given in a `.options` line. Any text in a circuit description can reference a shell variable, and this offers a powerful capability for manipulating the circuit under the control of the shell. As the variables must be set before the circuit is parsed, the `set` commands which perform this action can be included in the `.exec` block of the circuit file itself, or in the `.options` line. For example, suppose one has a circuit with a large number of resistors, each the same value, but it is desired to run the circuit for several different values. The resistor lines could be specified as

```
r31 11 36 $rvalue
r32 12 35 $rvalue
etc.
```

and elsewhere in the file one would have

```
.exec
set rvalue = 50
.endc
```

The 50 can be changed to any value, avoiding the need to change the many resistor lines between simulation runs. Note that the `.exec` block must be used, if `.control` was used instead, the variables would not be set until after the circuit is read, which means that they will not be properly defined when the expansion is performed. The `.control` block is useful for initiating analysis and post-processing.

Note that there is an alternative method of parameterization using the `.param` line.

The same effect could have been obtained from the use of the `.options` line as

```
.options rvalue=50
```

and, as the `.options` lines are expanded after the `.exec` lines are executed, one could have the following contrived example:

```
.exec
set rtmp=50
.endc
.options rvalue = $rtmp
```

The shell variables set in `.exec` and `control` blocks remain set until explicitly unset, however variables set in `.options` lines are active only when the circuit is the current circuit, and cannot be unset (with the `unset` command) from the shell. A variable set in the `.options` line will be superseded by the same variable set from the shell, `.exec` or `.control` lines.

Commands can also be included using a different mechanism, which might be useful if the circuit file is to be used with other simulators. This mechanism uses comment lines to include shell commands. If a comment begins with the characters "`*@`", the remainder of the line is taken as a shell command just as if it had been enclosed in `.exec` and `.endc`. If a comment line begins with the characters "`*#`", the remainder of the line is treated as if it had been enclosed in `.control` and `.endc`. Thus, in the example above, the `.exec` block could be replaced with the line

```
*@ set rvalue = 50
```

Obviously, this facility allows the possibility that a real comment can be misconstrued as a shell command. The user is suggested to leave space after the "`*`" in intended comments, as a general rule.

If a circuit contains an `.exec` block, a temporary plot structure is created to hold any vectors defined in the `.exec` block while the circuit is parsed, after which the plot is deleted. Thus, if the circuit references vectors defined in the `.exec` block, the reference will be satisfied, and the variables will have initial values as defined in the `.exec` block.

2.10.2 `.check`, `.checkall`, `.monte`, and `.noexec` Lines

WRspice provides a built-in two-dimensional operating range analysis as well as Monte Carlo analysis. A complete description of the file formats used in these analyses is provided in Chapter 5. The analysis

is initiated with the check command described in 4.2.4, or is performed immediately if in batch mode. Files intended for operating range or Monte Carlo analysis may contain the keywords .check, checkall, or .monte. In each case, the execution of the .control block is suppressed when the circuit is read, however the .exec block is executed normally. The .noexec keyword also suppresses execution of the .control block, but does not predispose the circuit to any particular type of analysis.

The .check line specifies operating range analysis, where the contour of operation is to be determined. In the two-dimensional space of the variables being varied, the rows are evaluated from the left until a "pass" condition is found. The analysis then resumes at the far right, working left until a "pass" point is found. The area between the pass points is never evaluated. If there are islands of fail points within the pass region, they will not be found with this algorithm. The .checkall line, if used instead, will evaluate all of the points. This slows evaluation, but is more thorough.

The .monte line specifies Monte Carlo analysis. The .noexec line simply bypasses the execution of the .control lines when the file is read. It is not an error to have more than one of these lines present in the file (but this is poor practice). The .monte line has precedence, and .checkall has precedence over .check. The .noexec is assumed if any of the other lines are given.

2.11 Verilog Interface

WRspice contains a built-in Verilog parser/simulator. Verilog is a popular hardware description language for digital logic circuits. The integration of Verilog with SPICE provides a wealth of new capability:

- Direct support for analog/digital mixed-mode simulations.
- The ability to create simulation control automation in a Verilog block.
- The ability to create measurement automation in a Verilog block.
- The ability to create pulse sources that have complex output and are independent of the transient time scale.

2.11.1 .verilog, .endv Lines

General Form:
```
.verilog
```

In *WRspice*, all Verilog code is placed into a block of statements starting with a .verilog line and ending with a .endv line. The Verilog block defines the modules of a hierarchy, including a top-level "stimulus" module. The Verilog simulation is run in parallel with transient analysis, where each user time increment corresponds to one unit of Verilog time. The simulation is performed as if steptype is set to hitusertp, i.e., simulation is performed at each user time point. The Verilog simulation occurs before the SPICE iterations at the time point. Signals are passed to the Verilog block with .adc statements, and signals from the Verilog block are accessed through referencing voltage or current sources.

Output signals from the Verilog block are obtained through voltage or current sources in the circuit. The voltage/current source must refer by name to a Verilog variable in the scope of the top module, or use the Verilog "dot" path notation. The voltage/current source is set to the binary value of the variable, and has a built-in rise/fall time of one time increment. The variable reference can contain a bit or part select field.

A good primer on Verilog is: Samir Palnitkar, *Verilog HDL, A Guide to Digital Design and Synthesis*, SunSoft Press (Prentice Hall) ISBN 0-13-451675-3. The full story is in IEEE Standard 1364-1995.

An example input file that uses a `.verilog` block is given in A.3.

2.11.2 `.adc` Line

General Form:
`.adc`

This line of *WRspice* input converts a SPICE signal into a digital signal for the Verilog block. Such lines are used only as an adjunct to Verilog.

General Form:
`.adc` *digital_var node_name* `[`*offset*`]` `[`*delta*`]`

The parameters have the following interpretation:

digital_var
Qualified name of a variable in the Verilog block, can include a range specification.

node_name
Node of circuit to convert, not including any "`v()`". Current branches can be accessed as "*name*`#branch`".

offset (optional, default 0)
Real number subtracted from value before conversion.

delta (optional, default 1)
The size of an lsb for conversion.

The transfer function is:

value = *value* - *offset*
if (*value* > 0)

value = *value* + 0.5**delta*

else

value = *value* - 0.5**delta*

conversion = (integer) (*value*/*delta*)

The *offset* and *delta* arguments to the `.adc` line can be expressions. These will be evaluated once only, as the circuit is read in.

2.12 Voltage and Current Sources

In *WRspice*, the specification of an "independent" source is completely general, as the output can be governed by an arbitrary expression containing functions of other circuit variables. The syntax is a superset of the notation used in previous versions of SPICE, which separately keyed independent and dependent sources.

The leading letter "v" keys a voltage source, and "i" keys a current source. In addition, the "arbitrary source" used in SPICE3 is retained, but is keyed by "a", rather than "b" ("b" is used for Josephson junctions in *WRspice*). This is a special case of the general source specification included for backward compatibility.

General Form:

```
vname n+ n- [expr] [[dc] dcvalue] [ac [acmag [acphase]] | table(name)]
    [distof1 [f1mag [f1phase]]] [distof2 [f2mag [f2phase]]]
iname n+ n- [expr] [[dc] dcvalue] [ac [acmag [acphase]] | table(name)]
    [distof1 [f1mag [f1phase]]] [distof2 [f2mag [f2phase]]]
aname n+ n- V|I = expr [[dc] dcvalue] [ac [acmag [acphase]] | table(name)]
    [distof1 [f1mag [f1phase]]] [distof2 [f2mag [f2phase]]]
```

Examples:

```
vcc 10 0 dc 6
vin 13 2 0.001 ac 1 sin(0 1 1meg)
v2 10 1 ac table(acvals)
isrc 23 21 ac 0.333 45.0 2*sffm(0 1 10k 5 1k)
vmeas 12 9
vin 1 0 2*v(2)+v(3)
azz 2 0 v=.5*exp(v(2))
ixx 2 4 pulse(0 1 1n 10n 10n) + pulse(0 1 40n 10n 10n)
```

The *n*+ and *n*- are the positive and negative nodes, respectively. Note that voltage sources need not be grounded. Positive current is assumed to flow from the positive node, through the source, to the negative node. A current source of positive value will force current to flow in to the *n*+ node, through the source, and out of the *n*- node. Voltage sources, in addition to being used for circuit excitation, are often used as "ammeters" in *WRspice*, that is, zero valued voltage sources may be inserted into the circuit for the purpose of measuring current (in *WRspice*, an inductor can be used for this purpose as well). Zero-valued voltage sources will, of course, have no effect on circuit operation since they represent short-circuits, however they add complexity which might slightly affect simulation speed.

In transient and dc analysis, sources can in general have complex definitions which involve the dependent variable (e.g., time in transient analysis) and other circuit variables. There are built-in functions (`pulse`, `pwl`, etc.) which can be included in the *expr*.

Constant values associated with the source are specified by the following option keywords:

`dc` *dcvalue*
: This specifies a fixed dc analysis value for the source, and enables the source to be used in a dc sweep if the *expr* is given. If the *expr* is not given, the source is available for use in a dc sweep whether or not the `dc` keyword is given. If an *expr* is present without "`dc` *dcvalue*", the time=0 value of the *expr* is used for dc analysis. If the source value is zero for both dc and transient analyses, this value and the *expr* may be omitted. If the source is the same constant value in dc and transient analysis, the keyword "`dc`" and the value can be omitted.

`ac` [[*acmag* [*acphase*]] | table(*name*)]
The parameter *acmag* is the ac magnitude and *acphase* is the ac phase. The source is set to this value in ac analysis. If *acmag* is omitted following the keyword `ac`, a value of unity is assumed. If *acphase* is omitted, a value of zero is assumed. If the source is not an ac small-signal input, the keyword `ac` and the ac values are omitted. Alternatively, a table can be specified, which contains complex values at different frequency points. In ac analysis the source value will be derived from the table. The table with the given *name* should be specified in a `.table` line, with the `ac` keyword present. The values in the table are the real and imaginary components, and *not* magnitude and phase.

`distof1` and `distof2`
These are the keywords that specify that the independent source has distortion inputs at the frequencies f1 and f2 respectively for distortion analysis. The keywords may be followed by an optional magnitude and phase. The default values of the magnitude and phase are 1.0 and 0.0 respectively.

The *expr* is used to assign a time-dependent value for transient analysis and to supply a functional dependence for dc analysis. If a source is assigned a time-dependent value, the time-zero value is used for dc analysis, unless a dc value is also provided.

2.12.1 Source Expressions

Expressions used in source specifications have the same syntax as is used in vector expressions in *WRspice* shell commands. The expression can contain vectors from the current plot or the `constants` plot, and circuit parameters accessed through the `@`*device*`[`*param*`]` construct. In addition, the variable "`x`", which can appear explicitly in the expression, is defined to be the controlling variable in dependent sources, or is set to the the scale variable in the analysis (e.g., `time` for transient analysis). During a "pure" dc sweep, i.e., not part of a chained analysis, the `x` variable is set to the voltage or current of the swept source, or the first swept source if there are two.

In a chained analysis, the `x` variable will be that of core analysis. Thus, for a chained transient analysis, `x` is time, as in the unchained case. Since the functional dependence is inoperable in any kind of ac small-signal analysis (ac, noise, transfer function, pz, distortion, ac sensitivity) `x` is not set and never used. In "op" analysis, `x` is always numerically zero. The same is true in dc sensitivity analysis.

Although the syntax and most of the function names are equivalent to vector expressions used in post-processing, the mathematics subsystems are completely different. There are two main differences from ordinary vector expressions:

1. The expressions always resolve as scalars. Before evaluation, all vectors in the current plot are "scalarized" so that they temporarily have unit length with the current value as the data item.
2. All results are real values.

Although the available functions lists are similar, the internal evaluation routines are different. The shell math functions must operate on vectors of complex values, whereas the functions called in source expressions take scalar real values only. Furthermore, the source expressions must be differentiable with respect to included node voltages and branch currents, as the derivative of the expression is computed as part of the iterative process of solving the circuit matrix equations. This means that logical and conditional operators are not available in source expressions, nor is there a modulus operator or a value returned from assignment. The `sgn` function grossly violates the differentiability requirement, and many

of the functions and/or their derivatives have restricted ranges or singularities. These can easily lead to convergence problems unless some care is exercised.

As for all expressions, if an expression is enclosed in single quotes, it will be evaluated when the file is read, reducing to a constant.

In the source expression, the following math functions are available on most systems:

`abs()`	absolute value
`acos()`	arc cosine
`acosh()`	arc hyperbolic cosine
`asin()`	arc sine
`asinh()`	arc hyperbolic sine
`atan()`	arc tangent
`atanh()`	arc hyperbolic tangent
`cbrt()`	cube root
`cos()`	cosine
`cosh()`	hyperbolic cosine
`deriv()`	derivative
`erf()`	error function
`erfc()`	error function complement
`exp()`	exponential (e raised to power)
`j0()`	Bessel order 0
`j1()`	Bessel order 1
`jn()`	Bessel order n
`ln()`	log base e
`log()`	log base 10
`sgn()`	sign (+1,0,-1)
`sin()`	sine
`sinh()`	hyperbolic sine
`sqrt()`	square root
`tan()`	tangent
`tanh()`	hyperbolic tangent
`y0()`	Neumann order 0
`y1()`	Neumann order 1
`yn()`	Neumann order n

Each function takes a single argument, except for `jn()` and `yn()`, which require two arguments. For these functions, the first argument is an integer value for the order, and the second argument is the function input. The `deriv()` function will differentiate the parse tree of the argument with respect to the "`x`" variable (whether implicit or explicit). This is completely unlike the `deriv()` function for vectors, which performs a numerical differentiation with respect to some scale. Differentiating the parse tree gives an analytic result which is generally more accurate.

In addition, there are special "tran functions" described below which produce specified output in transient analysis. *WRspice* recognizes by context functions and tran functions with the same name (`exp`, `sin`, `gauss`). An unrecognized function is assumed to be a table reference (specified with a `.table line`).

During a "pure" dc sweep analysis, for "independent" sources (keyed by `v`, `i`, or `a` and not `e`, `f`, `g`, or `h`) other than the swept ones, if an expression is given, the output of the source will be the result of the expression where the input `x` is the swept voltage, rather than time as when in transient anslysis.

However, if the source line has a "dc" keyword and optional following constant value, during pure dc analysis the source will output the fixed value, or zero, if the value is omitted. However, in pure dc analysis the tran functions generally return zero. The exceptions are `pwl`, `table` and table references, and `interp`. These functions return values, but with the swept voltage (`x`) as the input (in the case of `table()` the input may be explicit anyway). For "dependent" sources (keyed by `e`, `f`, `g`, or `h`) the `x` is the controlling voltage or current as in transient analysis. Again, if a "dc" keyword appears, the output will be fixed at the given value, ignoring the controlling variable.

In addition to the built-in functions, expressions used in sources can include user defined functions, which must have been defined previously with the define command.

The following tokens are recognized in the function string:

`+,*,/`	binary: add, multiply, divide
`-`	unary or binary: negate or subtract
`^`	binary: exponentiation
`()`	association
`,`	argument separator
`x`	independent variable
number	a floating point number
string	a library function, table, or circuit vector

The independent variable `x` is context specific, and usually represents a global input variable. It is the running variable in the current analysis (time in transient analysis, for example), or the input variable in dependent source specifications (see 2.12.4).

Example expressions:

```
sqrt(v(1)*v(1) + v(3)*v(3))
ln(exp(v(3)^1.5)/3)
2*x - x*x*x
```

In expressions, if a string starts with a backslash ('\') character, it will not be replaced with a value, should the name happen to match one of the named constants, or other potential substitutions. This will be needed, for example, if a node name matches one of the predefined constant names, and you need to reference that node in a source.

For example, suppose there is a node named "`c`", which is also the name of the speed-of-light constant. A source specification like

```
vcon 1 2 5*v(c)
```

will not be accepted. The correct form is

```
vcon 1 2 5*v("\c")
```

2.12.2 POLY Expressions

In SPICE2, nonlinear polynomial dependencies are specified using a rather cumbersome syntax keyed by the word `poly`. For compatibility, this syntax is recognized by the dependent sources in *WRspice*, making possible the use of the large number of behavioral models developed for SPICE2.

There are three polynomial equations which can be specified through the `poly(`*N*`)` parameter.

`poly(1)` One-dimensional equation

`poly(2)` Two-dimensional equation

`poly(3)` Three-dimensional equation

The dimensionality refers to the number of controlling variables; one, two, or three. These parameters must immediately follow the `poly(N)` token. The inputs must correspond to the type of the source, either pairs of nodes for voltage-controlled sources, or voltage source or inductor names for current-controlled sources. Following the inputs is the list of polynomial coefficients which define the equation. These are constants, and may be in any format recognized by *WRspice*.

The simplest case is one dimension, where the coefficients `c0, c1, ...` evaluate to

$$c_0 + c_1 x + c_2 x^2 + c_3 x^3 + ...$$

The number of terms is arbitrary. If the number of terms is exactly one, it is assumed to be the linear term (`c1`) and not the constant term. The following is an example of a voltage-controlled voltage source which utilizes `poly(1)`.

```
epolysrc 1 0 poly(1) 3 2 0 2 0.25
```

The source output appears at node 1 to ground (note that *WRspice* can use arbitrary strings as node specifiers). The input is the voltage difference between nodes 3 and 2. The output voltage is twice the input voltage plus .25 times the square of the input voltage.

In the two dimensional case, the coefficients are interpreted in the following order.

$$c_0 + c_1 x + c_2 y + c_3 x^2 + c_4 xy + c_5 y^2 + c_6 x^3 + c_7 x^2 y + c_8 xy^2 + c_9 y^3 + ...$$

For example, to specify a source which produces `3.5*v(3,4) + 1.29*v(8)*v(3,4)`, one has

exx 1 0 poly(2) 3 4 8 0 0 3.5 0 0 1.29

Note that any coefficients that are unspecified are taken as zero.

The three dimensional case has a coefficient ordering interpretation given by

$$c_0 + c_1 x + c_2 y + c_3 z + c_4 x^2 + c_5 xy + c_6 xz + c_7 y^2 + c_8 yz + c_9 z^2 + c_{10} x^3 + c_{11} x^2 y + c_{12} x^2 z +$$

$$c_{13} xy^2 + c_{14} xyz + c_{15} xz^2 + c_{16} y^3 + c_{17} y^2 z + c^{18} yz^2 + c_{19} z^3 + ...$$

which is rather complex but careful examination reveals the pattern.

2.12.3 Tran Functions

There are several built-in source functions, which are based on and extend the source specifications in SPICE2. These generally produce time-dependent output for use in transient analysis. For brevity, these functions are referred to as “tran functions”.

The tran functions are listed in the table below. If parameters other than source amplitudes are omitted, default values will be assumed. The tran functions, which require multiple space or comma separated arguments in a particular order, are:

`exp()`	exponential specification
`gauss()`	gaussian specification
`interp()`	interpolation specification
`pulse()`	pulse specification
`pwl()`	piecewise-linear specification
`sffm()`	single frequency fm specification
`sin()`	sinusoidal specification
`spulse()`	sinusoidal pulse specification
`table()`	reference to a `.table` specification

Unlike the math functions, the tran functions have variable-length argument lists. If arguments are omitted, default values are assumed. In cases of a name clash between these functions and math functions (sin, exp, gauss), the argument list is examined to determine which is correct.

The tran functions implicitly use time as an independent variable, and generally return 0 in dc analysis. Exceptions are the `pwl()` and `interp()` forms, which implicitly uses the value of "`x`" which is context-specific. In dependent sources, this is the controlling value of the source rather than time. The `table()` function takes its input directly from the second argument.

The tran functions can also be used in regular vector expressions. They generate a vector corresponding to the current scale, which must exist, be real, and monotonically increasing. The length of the returned vector is equal to the length of the scale.

For example:

```
(do a tran analysis to establish a reasonable scale)
let a = pulse(0 1 10n 10n 10n 20n)
plot a      (plots a pulse waveform)
```

The construct can be used like any other token in a regular vector expression.

The tran functions (other than `table()` and `interp()`) take constant expressions as arguments. The argument list consists of comma or space separated expressions. Arguments are parsed as follows:

1. The outer parentheses, if these exist, are stripped from the list. *WRspice* can recognize most instances where parentheses are not included, since these are optional in standard SPICE syntax for the tran functions.
2. Commas that are not enclosed in parentheses or square brackets are converted to spaces.
3. Minus signs ('`-`') that are not enclosed in parentheses or square brackets, and are not followed by white space, and are first in the line or preceded by white space, are converted to "`--`" which signals a unary minus. Double minus signs are not changed.
4. The string is parsed into individual expression units, which are the arguments. The separation is determined by context.

There is no provision for a unary '`+`', thus, `func(a, +b)` is taken as `func(a+b)`. Parenthesis can be added to enforce precedence. The minus sign handling implies that `func(a, -b)` and `func(a -b)` are taken as `func((a), (-b))`, whereas `f(a-b), f(a- b), f(a - b)`, etc are taken as `func((a)-(b))`.

In addition to the built-in functions, expressions used in sources can include user defined functions, which must have been defined previously with the define command. These may be useful for encapsulating the tran functions.

Example:

```
define mypulse(delay, width) pulse(0 1 delay 1n 1n width)
...
v1 1 0 mypulse(5n, 10n)
```

Recall that a line in the deck starting with "`*@`" will be executed before the deck is parsed.

```
title line
*@ define mypulse(delay, width) pulse(0 1 delay 1n 1n width)
v1 1 0 mypulse(5n, 10n)
r1 1 0 100
.end
```

The following paragraphs describe the tran functions in detail.

2.12.3.1 Exponential

General Form:

`exp(`*v1 v2* `[`*td1 tau1 td2 tau2*`])`

Example:

```
vin 3 0 exp(-4 -1 2ns 30ns 60ns 40ns)
```

parameters	**default values**	**units**
v1 (initial value)		volts or amps
v2 (pulsed value)		volts or amps
td1 (rise delay time)	0.0	seconds
tau1 (rise time constant)	`tstep`	seconds
td2 (fall delay time)	*td1* + `tstep`	seconds
tau2 (fall time constant)	`tstep`	seconds

The shape of the waveform is described by the following table:

time	**value**
0	*v1*
td1	$v1 + (v2-v1)(1 - \exp(-(\text{time}-td1)/tau1))$
td2	$v1 + (v2-v1)(1 - \exp(-(\text{time}-td1)/tau1)) +$ $(v1-v2)(1 - \exp(-(\text{time}-td2)/tau2))$

This function applies only to transient analysis, where time is the running variable. When referring to default values, `tstep` is the printing increment and `tstop` is the final time in transient analysis, see 2.7.9 for explanation. The argument count is used to distinguish this function from the math function of the same name.

2.12.3.2 Gaussian Random

General Form:

`gauss(`*stddev mean lattice* `[`*interp*`])`

Examples:

```
v1 1 0 gauss(.5, 2, 100n, 1)
v2 1 0 gauss(.1, 0, 0)
```

parameters	default values	units
stddev (standard deviation)		none
mean (mean value)		none
lattice (sample period)		S
interp (interpolation)	0	none

The `gauss` function can be used to generate correlated random output. This function takes three or four arguments.

The parameter *lattice* is for use in transient analysis. A new random value is computed at each time increment of *lattice*. If *lattice* is 0, then no lattice is used, and an uncorrelated random value is returned for each call. The *interp* parameter, used when *lattice* is nonzero, can have value 1 or 0. If *interp* is nonzero, the value returned by the function is the (first order) interpolation of the random values at the lattice points which frame the time variable. If *interp* is 0, the function returns the lattice cell's value for any time within the lattice cell, i.e., a random step with an amplitude change at every lattice point.

The first example above provides a random signal with standard deviation of .5V and mean of 2V, based on random samples taken every 100nS.

The *lattice* value should be on the order of the user print increment `tstep` in the transient analysis. It should not be less than the maximum internal time step, since the past history is not stored, and a rejected time point may back up the time across more than one lattice cell, thus destroying the correlation.

This function applies only to transient analysis, where time is the running variable. The argument count is used to distinguish this function from the math function of the same name.

2.12.3.3 Interpolation

General Form:

`interp(`*vector*`)`

Example:

```
vin 1 0 interp(tran1.v(1))
```

The output is *vector* interpolated to the scale of the current plot. When used in a source, the output of the source is the interpolated vector, or the initial or final value for points off the ends of the original scale.

For example, say an amplifier produces vector `v(1)` (an output) in plot `tran1`. One desires to apply this as input to another circuit. This is achieved with a source specification like that shown in the example above. This works in ordinary vector expressions as well.

2.12.3.4 Pulse

General Form:

`pulse(`*v1 v2* `[`*td tr tf pw per td1 td2* `...` `])`

Examples:

```
vin 3 0 pulse(-1 1 2ns 2ns 2ns 50ns 100ns)
vin1 1 0 pulse(0 1 2n .5n .5n 1n 0 6n 10n)
v2 4 0 v(1)*pulse(0 1 5n 10n)
```

parameters	**default values**	**units**
v1 (initial value)		volts or amps
v2 (pulsed value)		volts or amps
td (delay time)	0.0	seconds
tr (rise time)	`tstep`	seconds
tf (fall time)	`tstep`	seconds
pw (pulse width)	`tstep`	seconds
per (period)	`tstop`	seconds

If any of the parameters *td1, td2, ...* appear, the period is ignored, and the pulse is repeated for the delays *td1*, etc. The output will be a superposition of these pulses. A single pulse is described by the following table:

time	**value**
0	*v1*
td	*v1*
td+tr	*v2*
td+tr+pw	*v2*
td+tr+pw+tf	*v1*
`tstop`	*v1*

Intermediate points are determined by linear interpolation. It is not an error to omit unused parameters, for example the specification

```
vxx 3 0 pulse(0 1 2n 2n)
```

describes a voltage which, starting from 0, begins rising at 2 nanoseconds, reaching 1 volt at 4 nanoseconds, and remains at that value.

This function applies only to transient analysis, where time is the running variable. When referring to default values, `tstep` is the printing increment and `tstop` is the final time in transient analysis, see 2.7.9 for explanation.

2.12.3.5 Piecewise Linear

General Form:

`pwl(`*t1 v1* `[`*t2 v2 t3 v3 t4 v4 ...*`])`

Example:

```
vclock 7 5 pwl(0 -7 10ns -7 11ns -3 17ns -3 18ns -7 50ns -7)
```

Each pair of values (*ti*, *vi*) specifies that the value of the source is *vi* (in volts or amps) at time = *ti*. The value of the source at intermediate values of time is determined by using linear interpolation on

the input values. For times before the initial time value, the return is the initial value, and for times after the final time value, the return is the final value.

In dependent sources where the controlling input is specified, a `pwl()` construct if used in the expression for the source will take as input the value of the controlling input, and not time. This is one means by which a piecewise-linear transfer function can be implemented. A similar capability exists through the `table()` function.

Example:

```
e1 1 0 2 0 pwl(-1 1 0 0 1 1)
```

The example above implements a perfect rectifier (absolute value generator) for voltages between -1 and 1V. Outside this range, the output is clipped to 1V.

2.12.3.6 Single-Frequency FM

General Form:
`sffm(`*vo va* `[`*fc mdi fs*`])`

Example:
`v1 12 0 sffm(0 1m 20k 5 1k)`

parameters	default values	units
vo (offset)		volts or amps
va (amplitude)		volts or amps
fc (carrier frequency)	1/`tstop`	Hz
mdi (modulation index)	0	
fs (signal frequency)	1/`tstop`	Hz

The shape of the waveform is described by the following equation:

$$\text{value} = vo + va{\cdot}\sin((2\pi{\cdot}fc{\cdot}\text{time}) + mdi{\cdot}\sin(2\pi{\cdot}fs{\cdot}\text{time}))$$

This function applies only to transient analysis, where time is the running variable. When referring to default values, `tstep` is the printing increment and `tstop` is the final time in transient analysis, see 2.7.9 for explanation.

2.12.3.7 Sinusoidal

General Form:
`sin(`*vo va* `[`*freq td theta*`])`

Example:
`vin 3 0 sin(0 1 100meg 1ns 1e10)`

parameters	default value	units
vo (offset)		volts or amps
va (amplitude)		volts or amps
freq (frequency)	1/`tstop`	hz
td (delay)	0.0	seconds
theta (damping factor)	0.0	1/seconds

The shape of the waveform is described by the following table:

time	value
0	*vo*
td	$vo + va \cdot \exp(-(\text{time}-td)\cdot theta) \cdot \sin(2\pi \cdot freq \cdot (\text{time}+td))$

This function applies only to transient analysis, where time is the running variable. When referring to default values, `tstep` is the printing increment and `tstop` is the final time in transient analysis, see 2.7.9 for explanation. The argument count is used to distinguish this function from the math function of the same name.

2.12.3.8 Sinusoidal Pulse

General Form:

```
spulse(vo vp [per td decay])
```

Example:

```
vin 1 0 spulse(0 1 25ns 40ns 1e8)
```

parameters	default value	units
vo (offset)		volts or amps
vp (peak amplitude)		volts or amps
per (period)	`tstop`	seconds
td (delay)	0.0	seconds
decay (decay const)	0.0	1/seconds

The shape of the waveform is described by the following table:

time	value
0	*vo*
td	$vo + 0.5\cdot(vp-vo)(\ 1 - \cos(2\pi\cdot(\text{time}-td)/per)\cdot\exp(-(\text{time}-td)\ \cdot decay)\ )$

This function applies only to transient analysis, where time is the running variable. When referring to default values, `tstep` is the printing increment and `tstop` is the final time in transient analysis, see 2.7.9 for explanation.

2.12.3.9 Table Reference

General Form:

```
table(table_name expr)
table_name(expr)      (for sources only)
```

Examples:
```
vin 1 0 table(tab1, v(2))
exx 1 0 2 0 table(tab2, x)
exx 1 0 2 0 tab2(x)
```

The table referenced must be specified in the input deck with a `.table` line. The reference to a table is in the form of a `table()` function, as above, which takes two arguments. The first argument is the name of a table defined elsewhere in the circuit file with a `.table` line. The second argument is an expression which provides input to the table. The return value is the interpolated value from the table.

Tables can also be referenced as part of the ac specification for a dependent or independent source. These references are used in ac analysis, and have a different referencing syntax.

In the expression used in voltage and current sources, dependent and independent, the second form can be used and is equivalent. The *table_name* must not conflict with another internal or user-defined function name.

The `table()` reference provides one means of implementing a piecewise-linear transfer function. This can also be accomplished by use of the `pwl()` function in dependent sources.

2.12.4 Dependent Sources

WRspice source specifications are completely general in that they allow arbitrary functional dependence upon circuit variables. However, for compatibility with previous versions of SPICE, the separate keying of independent and dependent sources is retained. *WRspice* allows circuits to contain dependent sources characterized by any of the four equations in the table below.

`VCCS`	`i = g(v)`	Voltage-Controlled Current Source
`VCVS`	`v = e(v)`	Voltage-Controlled Voltage Source
`CCCS`	`i = f(i)`	Current-Controlled Current Source
`CCVS`	`v = h(i)`	Current-Controlled Voltage Source

The functions `g`, `e`, `f`, and `h` represent transconductance, voltage gain, current gain, and transresistance, respectively.

2.12.4.1 Voltage-Controlled Current Sources

General Form:
*g**name** n+ n- nc+ nc-* [*expr*] *srcargs*
*g**name** n+ n-* `function` *expr srcargs*
*g**name** n+ n-* `poly` *poly_spec srcargs*
where *srcargs* = [`ac table(`*name*`)`]

Examples:
```
g1 2 0 5 0 0.1mmho
g2 2 0 5 0 log10(x)
g3 2 0 function log10(v(5))
```

The *n+* and *n-* are the positive and negative nodes, respectively. Current flow is from the positive node, through the source, to the negative node. The parameters *nc+* and *nc-* are the positive and negative controlling nodes, respectively. In the first form, if the *expr* is a constant, it represents the

transconductance in mhos. Otherwise, the *expr* computes the source current, where the variable "x" if used in the *expr* is taken to be the controlling voltage (v(*nc*+,*nc*-)). In this case only, the `pwl()` construct if used in the *expr* takes as its input variable the value of "x" rather than time, thus a piecewise linear transfer function can be implemented using a `pwl()` statement. The second form is similar, but "x" is not defined. The third form allows use of the SPICE2 `poly` construct.

More information on the function specification can be found in 2.12, and the `poly` specification is described in 2.12.2.

If the `ac` parameter is given and the `table` keyword follows, then the named table is taken to contain complex *transfer* coefficient data, which will be used in ac analysis (and possibly elsewhere, see below). For each frequency, the source output will be the interpolated transfer coefficient from the table multiplied by the input. The table must be specified with a `.table` line, and must have the `ac` keyword given.

If an ac table is specified, and no dc/transient transfer function or coefficient is given, then in transient analysis, the source transfer will be obtained through Fourier analysis of the table data. This is somewhat experimental, and may be prone to numerical errors.

In ac analysis, the transfer coefficient can be real or complex. If complex, the imaginary value follows the real value. Only constants or constant expressions are valid in this case. If the source function is specified in this way, the real component is used in dc and transient analysis. This will also override a table, if given.

2.12.4.2 Voltage-Controlled Voltage Sources

General Form:

e*name* *n*+ *n*- *nc*+ *nc*- [*expr*] *srcargs*
e*name* *n*+ *n*- `function` *expr srcargs*
e*name* *n*+ *n*- `poly` *poly_spec srcargs*
where *srcargs* = [`ac table(`*name*`)`]

Examples:

```
e1 2 3 14 1 2.0
e2 2 3 14 1 x+.015*x*x
e3 2 3 function v(14,1)+.015*v(14,1)*v(14,1)
```

The *n*+ is the positive node, and *n*- is the negative node. *nc*+ and *nc*- are the positive and negative controlling nodes, respectively. In the first form, if the *expr* is a constant, it represents the linear voltage gain. Otherwise, the *expr* computes the source voltage, where the variable "x" if used in the *expr* is taken to be the controlling voltage (v(*nc*+,*nc*-)). In this case only, the `pwl()` construct if used in the *expr* takes as its input variable the value of "x" rather than time, thus a piecewise linear transfer function can be implemented using a `pwl()` statement. The second form is similar, but "x" is not defined. The third form allows use of the SPICE2 `poly` construct.

More information of the function specification can be found in 2.12, and the `poly` specification is described in 2.12.2.

If the `ac` parameter is given and the `table` keyword follows, then the named table is taken to contain complex *transfer* coefficient data, which will be used in ac analysis (and possibly elsewhere, see below). For each frequency, the source output will be the interpolated transfer coefficient from the table multiplied by the input. The table must be specified with a `.table` line, and must have the `ac` keyword given.

If an ac table is specified, and no dc/transient transfer function or coefficient is given, then in transient analysis, the source transfer will be obtained through Fourier analysis of the table data. This is somewhat experimental, and may be prone to numerical errors.

In ac analysis, the transfer coefficient can be real or complex. If complex, the imaginary value follows the real value. Only constants or constant expressions are valid in this case. If the source function is specified in this way, the real component is used in dc and transient analysis. This will also override a table, if given.

2.12.4.3 Current-Controlled Current Sources

General Form:

f*name* *n*+ *n*- *vnam* *expr* *srcargs*
f*name* *n*+ *n*- `function` *expr* *srcargs*
f*name* *n*+ *n*- `poly` *poly_spec* *srcargs*
where *srcargs* = `[ac table(`*name*`)]`

Examples:

```
f1 13 5 vsens 5
f2 13 5 1-x*x ac table(acdata)
f3 13 5 function 1-i(vsens)*i(vsens)
```

The *n*+ and *n*- are the positive and negative nodes, respectively. Current flow is from the positive node, through the source, to the negative node. The parameter *vnam* is the name of a voltage source or inductor through which the controlling current flows. If *vnam* refers to a voltage source, the direction of positive controlling current flow is from the positive node, through the source, to the negative node. If *vnam* names an inductor, the current flow is from the first node specified for the inductor, through the inductor, to the second node. In the first form, if the *expr* is a constant, it represents the linear current gain. Otherwise, the *expr* computes the source current, where the variable "`x`" if used in the *expr* is taken to be the controlling current (i(*vnam*)). In this case only, the `pwl()` construct if used in the *expr* takes as its input variable the value of "`x`" rather than time, thus a piecewise linear transfer function can be implemented using a `pwl()` statement. The second form is similar, but "`x`" is not defined. The third form allows use of the SPICE2 `poly` construct.

More information of the function specification can be found in 2.12, and the `poly` specification is described in 2.12.2.

If the `ac` parameter is given and the `table` keyword follows, then the named table is taken to contain complex *transfer* coefficient data, which will be used in ac analysis (and possibly elsewhere, see below). For each frequency, the source output will be the interpolated transfer coefficient from the table multiplied by the input. The table must be specified with a `.table` line, and must have the `ac` keyword given.

If an ac table is specified, and no dc/transient transfer function or coefficient is given, then in transient analysis, the source transfer will be obtained through Fourier analysis of the table data. This is somewhat experimental, and may be prone to numerical errors.

In ac analysis, the transfer coefficient can be real or complex. If complex, the imaginary value follows the real value. Only constants or constant expressions are valid in this case. If the source function is specified in this way, the real component is used in dc and transient analysis. This will also override a table, if given.

2.12.4.4 Current-Controlled Voltage Sources

General Form:

h*name* *n*+ *n*- *vnam expr srcargs*
h*name* *n*+ *n*- `function` *expr srcargs*
h*name* *n*+ *n*- `poly` *poly_spec srcargs*
where *srcargs* = [`ac table(`*name*`)`]

Examples:

```
h1 5 17 vz 0.5k
h2 5 17 0.71,0.71
h3 5 17 vz 2.5*exp(x/2.5) ac table(myvals)
h2 5 17 function 2.5*exp(i(vz)/2.5)
```

Above, *n*+ and *n*- are the positive and negative nodes, respectively. The parameter *vnam* is the name of a voltage source or inductor through which the controlling current flows. If *nam* references a voltage source, the direction of positive controlling current flow is from the positive node, through the source, to the negative node. If *vnam* references an inductor, the controlling current flows from the first node specified for the inductor, through the inductor, to the second node. In the first form, if the *expr* is a constant, it represents the transresistance in ohms. Otherwise, the *expr* computes the source voltage, where the variable "`x`" if used in the *expr* is taken to be the controlling current (i(*vnam*)). In this case only, the `pwl()` construct if used in the *expr* takes as its input variable the value of "`x`" rather than time, thus a piecewise linear transfer function can be implemented using a `pwl()` statement. The second form is similar, but "`x`" is not defined. The third form allows use of the SPICE2 `poly` construct.

More information of the function specification can be found in 2.12, and the `poly` specification is described in 2.12.2.

If the `ac` parameter is given and the `table` keyword follows, then the named table is taken to contain complex *transfer* coefficient data, which will be used in ac analysis (and possibly elsewhere, see below). For each frequency, the source output will be the interpolated transfer coefficient from the table multiplied by the input. The table must be specified with a `.table` line, and must have the `ac` keyword given.

If an ac table is specified, and no dc/transient transfer function or coefficient is given, then in transient analysis, the source transfer will be obtained through Fourier analysis of the table data. This is somewhat experimental, and may be prone to numerical errors.

In ac analysis, the transfer coefficient can be real or complex. If complex, the imaginary value follows the real value. Only constants or constant expressions are valid in this case. If the source function is specified in this way, the real component is used in dc and transient analysis. This will also override a table, if given.

2.13 Element Lines and the Device Library

This and the following sections describe the devices available in the standard device library distributed with *WRspice*. In Unix/Linux releases, the device library is a shared library, usually installed as `/usr/local/lib/libxsdev.so.2.0`. The device library contains the element and model code for each device, as well as the parser for the element specification lines.

Most of the code for the device library (with the exception of restricted third-party semiconductor models) is available upon request from Whiteley Research Inc. In theory, users can build their own,

customized device library for use with *WRspice*. In this case, devices can be added to or deleted from the library, or modified. The only device that is mandatory is the `src` (voltage and current source) device since this model is (uniquely) accessed by name from the main *WRspice* program. This provides *all* of the dependent and independent sources.

This format for most device lines, including the key letters, number of nodes, etc., is standard for the SPICE input language, but is set entirely by the code in the device library, and hence can be abridged in a custom device library. The descriptions below pertain to the standard library.

The following is a complete list of circuit elements available in the standard *WRspice* library, and the key letter (the first letter of the device name).

Passive Elements	
Capacitor	`c`
Inductor	`l`
Mutual Inductor	`k`
Resistor	`r`
Current-Controlled Switch	`w`
Voltage-Controlled Switch	`s`
General Transmission Line	`t`
Lossy Transmission Line	`o`
Uniform RC Line	`u`
Voltage and Current Sources	
General Voltage Source	`v`
General Current Source	`i`
Arbitrary Source	`a`
Voltage-Controlled Current Source	`g`
Voltage-Controlled Voltage Source	`e`
Current-Controlled Current Source	`f`
Current-Controlled Voltage Source	`h`
Semiconductor Devices	
Junction Diode	`d`
Bipolar Junction Transistor	`q`
Junction Field-Effect Transistor	`j`
MESFET	`z`
MOSFET	`m`
Superconductor Devices	
Josephson Junction	`b`

The models for the semiconductor and some other devices require many parameter values. Often, many devices in a circuit are defined by the same set of device model parameters. For these reasons, a set of device model parameters is defined on a separate `.model` line and assigned a unique model name. The device element lines in *WRspice* then refer to the model name. This scheme alleviates the need to specify all of the model parameters on each device element line.

The show command with the `-D` option is useful for printing a list of the parameters names that can be used on a device instance line. Only the parameters not listed as "`RO`" (read-only) can be specified on the line.

2.14 Passive Element Lines

2.14.1 Capacitors

General Form:

`c`*name* *n*`+` *n*`-` [*value*] | [*modname* `l=`*length* [`w=`*width*]] [`ic=`*val*]

Examples:

```
cload 2 10 10p
cmod 3 7 cmodel l=10u w=1u
```

The *n*`+` and *n*`-` are the positive and negative element nodes, respectively, and *value* is the capacitance for a constant valued capacitor. Alternatively, a capacitor model *modname* can be specified which allows for the calculation of the actual capacitance value from strictly geometric information and the specifications of the process. If *value* is specified, it defines the capacitance. If *modname* is specified, then the capacitance is calculated from the process information in the model *modname* and the given *length* and *width*. If *value* is not specified, then *modname* and *length* must be specified. If *width* is not specified, then it will be taken from the default width given in the model. Either *value* or *modname*, *length*, and *width* may be specified, but not both sets. See the capacitor model description in 2.17.2 for more information.

The optional initial condition is the initial (time zero) voltage across the capacitor. The initial condition (if any) applies only when the `uic` option is specified in transient analysis.

2.14.2 Inductors

General Form:

`l`*name* *n*`+` *n*`-` *value* [`ic=`*val*]

Examples:

```
llink 42 69 1uh
lshunt 23 51 10u ic=15.7ma
```

The *n*`+` and *n*`-` are the positive and negative element nodes, respectively, and *value* is the inductance in henries. The optional initial condition is the initial (time-zero) value of inductor current (in amps) that flows from *n*`+`, through the inductor, to *n*`-`. The initial conditions (if any) apply only when the `uic` option is specified in transient analysis.

2.14.3 Coupled (Mutual) Inductors

General Form:

`k`*name inductor1 inductor2 value*

Examples:

```
k43 laa lbb 0.999
kxfrmr l1 l2 0.87
```

The *inductor1* and *inductor1* are the names of the two coupled inductors found elsewhere in the circuit, and *value* is the coefficient of coupling, K, which must be greater than 0 and less than or equal to 1. Using the "dot" convention, one would have a dot on the first node of each inductor.

2.14.4 Resistors

General Form:

`r`*name n1 n2* `[`*value*`] | [`*modname modopts*`]`

Examples:

```
rload 2 10 10k
rmod 3 7 rmodel l=10u w=1u
```

If a model name *modname* is given, the following *modopts* can appear.

`res=`*value* or `resistance=`*value*
: The resistance of the resistor at nominal temperature. If given, this supersedes a geometric calculation using `l`, `w`. The "`res=`" or "`resistance=`" is optional, as a lone number *value* will be interpreted as the nominal temperature resistance.

`l=`*length*
: The length of the resistor.

`w=`*width*
: The width of the resistor.

`temp=`*optemp*
: The operating temperature of the resistor.

The *n1* and *n2* are the two element nodes, and *value* is the resistance, for a constant value resistor. A resistor model *modname* can alternatively be specified and allows the modeling of temperature effects and for the calculation of the actual resistance value from strictly geometric information and the specifications of the process. If *value* is specified after *modname*, it overrides the geometric information (if any) and defines the nominal temperature resistance. If *modname* is specified, then the resistance may be calculated from the process information in the model *modname* and the given *length* and *width*. In either case, the resulting value will be adjusted for the operating temperature *optemp* if that is specified, using correction factors given in the model. If *value* is not specified, then *modname* and *length* must be specified. If *width* is not specified, then it will be taken from the default width given in the model. See the resistor model description in 2.17.3 for more information.

2.14.5 Switches

General Form:

`s`*name n+ n- nc+ nc- model* `[on | off]`
`w`*name n+ n- vnam model* `[on | off]`

Examples:

```
s1 1 2 3 4 switch1 on
s2 5 6 3 0 sm2 off
```

```
switch1 1 2 10 0 smodel1
w1 1 2 vclock switchmod1
w2 3 0 vramp sm1 on
wreset 5 6 vclck lossyswitch off
```

Nodes *n+* and *n-* are the nodes between which the switch terminals are connected. The `model` name is mandatory while the initial conditions are optional. For the voltage controlled switch, nodes *nc+* and *nc-* are the positive and negative controlling nodes respectively. For the current controlled switch, the controlling current is that through the voltage source or inductor *vnam*. The direction of positive controlling current flow is from the positive node, through the source or inductor, to the negative node. See the switch model description in 2.17.4 for more information.

2.14.6 Transmission Lines

General Form:

t*name n1 n2 n3 n4* [*model*] [*param=value ...*]
o*name n1 n2 n3 n4* [*model*] [*param=value ...*]

Examples:

```
t1 1 0 2 0 z0=50 td=10ns
tw 1 0 2 0 z0=50 f=1ghz nl=.1
tx 1 0 4 0 l=9.13e-9 c=3.65e-12 len=24
ty 2 0 4 0 l=100pH c=5pf r=1.5 len=12
tz 2 0 4 0 tranmod len=12
```

In *WRspice*, the transmission line element represents a general lossy or lossless transmission line. The simple case of the lossless line of SPICE2/3 is handled in *WRspice* by this device. The lossy line (LTRA) device of SPICE3 is also handled by this element in *WRspice*, though this is not evident to the user. Additional parameters represent an extension of the SPICE2/3 element to the lossy case. The *WRspice* transmission line is a logical combination of the SPICE3 lossless transmission line, the SPICE3 lossy (LTRA) transmission line approach of Roychowdhury and Pederson [13], and the Pade approximation lossy line approach of Lin and Kuh [14].

Above, *n1* and *n2* are the nodes at port 1, *n3* and *n4* are the nodes at port 2. Note that this element models only one propagating mode. If all four nodes are distinct in the actual circuit, then two modes may be excited. To simulate such a situation, two transmission line elements are required.

There is a fairly lengthly list of parameters which can be applied in the device line, or in a model. If a model is referenced in the element line, the element defaults to the parameters specified in the model, though any of these parameters can be overridden for the element if given new values in the element line.

2.14.6.1 Model Level

`level`

This parameter can take values 0 (the default if not given) or 1. The level indicates the treatement of a lossy element, and has no effect if the transmission line is lossless.

Level 0 handles arbitrary RLCG configurations using the Pade approximation approach. A Pade approximation is used as a rational function approximation to the transfer function in the Laplace

domain, which has a trivial inverse transformation to the time domain. Further, separability avoids the need to perform a complex convolution at each time point. The model is very fast and accurate enough for most purposes.

Level 1 handles RLC configurations using a full numerical convolution, equivalent to the LTRA model. It does not allow a G element, and is much slower than the Pade approximation approach, however it may be more accurate. Level 1 supports the following types of lines: RLC (uniform transmission line with series loss only), RC (uniform RC line), LC (lossless transmission line), and RG (distributed series resistance and parallel conductance only).

2.14.6.2 Electrical Characteristics

`len`
This provides the physical length of the transmission line in arbitrary units, though the units must match the per-length unit in the element values discussed below. If not given, the value is taken as unity, unless it is implicitly defined by other parameters.

`l`
This parameter provides the series inductance per unit length of the line. The default is 0.

`c`
This parameter provides the shunt capacitance per unit length of the line. The default is 0.

`r`
This parameter provides the series resistance per unit length of the line. The default is 0.

`g`
This parameter provides the shunt conductance per unit length of the line. The default is 0. With level 1, this cannot be nonzero if `l` or `c` is given, i.e., only `r` can be nonzero if `g` is nonzero for level = 1, as in the SPICE3 LTRA model.

`z0` or `zo`
This is the line (lossless) characteristic impedance in ohms, given by

$$Z_0 = \sqrt{L/C}$$

`td` or `delay`
This is the (lossless) phase delay of the line in seconds, given by

$$T_d = Length\sqrt{LC}$$

`nl`
This is the normalized line length at a particular frequency *f*, which must also be specified (see below). This is an alternative means for setting the line delay, where

$$T_d = nl/f$$

It is an error to give both *td* and *nl*.

`f`
This is the frequency at which the normalized line length (above) is representative.

To model a line with nonzero series inductance and shunt capacitance, a complete but non-conflicting subset of the parameters `l`, `c`, `z0`, `td`, `len`, `f`, and `nl` must be provided. The `td` parameter is the line delay in seconds, and the `z0` parameter is the impedance in ohms, for the lossless case. Specifying these two parameters is sufficient to completely specify a lossless line, or the reactive elements of a lossy line. Alternatively, one could specify `l` (inductance per length), `c` (capacitance per length) and `len` (line length). If `len` is not specified in either case, the length defaults to unity. The delay can also be specified through the `f` (frequency) and `nl` (normalized length) parameters, where the delay would be set to `nl/f`. It is an error to specify both `td` and `f`, `nl`. If `td` is specified, or both `f` and `nl` are specified, along with parameters which yield internally the L and C values, then the length is determined internally by

$$Length = T_d/\sqrt{LC}$$

One can specify `z0` and `l`, for example, which determines C. Unlike the SPICE3 (and SPICE2) lossless line devices, the delay must be specified through the parameters; there is no default.

2.14.6.3 Initial Conditions

`v1`, `i1`, `v2`, `i2`
: The (optional) initial condition specification consists of the voltage and current at each of the transmission line ports. The initial conditions (if any) apply only when the `uic` option is specified in transient analysis.

2.14.6.4 Timestep and Breakpoint Control

Internally, the transmission line models store a table of past values of the currents and voltages at the terminals, which become excitations after the delay time. As excitations, these signals can cause errors or nonconvergence if their rate of change is too large. These errors are reduced or eliminated by two mechanisms: time step truncation and breakpoint setting. Time step truncation occurs if the excitation derivative exceeds a certain threshold. A breakpoint which occurs at this time will also be rescheduled to one delay-time later. Breakpoints are set by the independent voltage and current sources at times where a slope change occurs, in piecewise linear outputs. At a breakpoint, the internal time step is cut and integration order reduced to accommodate the change in input accurately.

`truncdontcut`
: If this flag is given, no timestep cutting will be done.

`truncsl`
: If this flag is given, the device will use a slope-test timestep cutting algorithm. This is the default.

`slopetol`
: When using the slope-test timestep cutting algorithm, this is the fraction used in the slope test. The default is 0.1.

`trunclte`
: This applies to level = 1 only. When this flag is given, a local truncation error method is used for timestep control.

`truncnr`
: This applies to level = 1 only. When this flag is given, a Newton-Raphson iterative method is used for timestep control.

Time step cutting can be turned off by setting the `truncdontcut` flag. The default setting, which has a redundant flag `truncsl`, uses a slope test to limit the next time step, in conjunction with the parameter `slopetol`. The algorithm computes the difference between the quadratic extrapolation from the last three and the linear extrapolation from the last two time points, and uses this difference formula to determine the time when this error is equal to `slopetol` multiplied by the the maximum absolute value of the signal at the three time points. The `slopetol` parameter defaults to 0.1.

When using level = 1, there are two alternative timestep control options. If the `trunclte` flag is given, the timestep is reduced by one half if the computed local truncation error is larger than an error tolerance, which is given by

$$tol = trtol \cdot (reltol \cdot (abs(input1) + abs(input2)) + abstol$$

where *trtol*, *reltol* and *abstol* are the values of the SPICE options `trtol`, `reltol` and `abstol`, and *input1* and *input2* are the internally stored excitations. If the `truncnr` flag is given, a timestep is computed based on limiting the local truncation error to the tolerance given above. Only one of the trunc flags should be given. The latter two apply only to the RLC line, and if given in the LC case the default `truncsl` is applied.

NOTE: The `trunclte` and `truncnr` algorithms do not appear to work in the RLC case, except by limiting the time step to a tiny value.

`nobreaks`
: When this flag is given, there will be no breakpoint rescheduling.

`allbreaks`
: When this flag is give, all breakpoints are rescheduled.

`testbreaks`
: When this flag is given, which is the default, a test is applied and only breakpoints that pass this test are rescheduled.

`rel`
: When testing breakpoints, this is the relative tolerance value. The default is .001.

`abs`
: When testing breakpoints, this is the absolute tolerance value. The default is 1e-12.

The breakpoint setting is controlled by the three flags `nobreaks`, `allbreaks`, and `testbreaks`. Only one should be given, and the default is `testbreaks`. If `nobreaks` is set, breakpoints will not be rescheduled. If `allbreaks` is set, all breakpoints will be rescheduled to the break time plus the delay time. The default `testbreaks` will reschedule a breakpoint if a slope test is passed. This slope test makes use of the `rel` and `abs` parameters. The slopes at the last two time points are computed. The breakpoint is rescheduled if

$$abs(d1 - d2) > max(.01 \cdot rel \cdot vmax, abs)/dt$$

where *d1* and *d2* are the two slopes. The parameters `rel` and `abs` default to 1e-3 and 1e-12, respectively. The *dt* parameter is the sum of the last two time deltas, and *vmax* is a running peak detect function representing the maximum voltage applied to the line. Note that these are different defaults (and a different algorithm) from the parameters of the same name used in the SPICE3 transmission line models.

In most cases, the defaults for the timestep and breakpoint controls are sufficient. Excessive setting of breakpoints and timestep truncation increases execution time, while insufficient control can produce errors. An alternative approach is to limit the maximum internal timestep used with the `.tran` line, which can provide highly accurate results for comparison when experimenting with the control parameters.

See the description of the transmission line model (2.17.5) for more information.

2.14.6.5 History List

`lininterp`
: If this flag is set, linear interpolation is used to obtain the present value of signals in the history list.

`quadinterp`
: If this flag is set, which is the default, quadratic interpolation is used to obtain the present value of signals in the history list.

`compactrel`
: This is the relative tolerance used in history list compaction for level = 1. The default value is the same as the *WRspice* default relative tolerance (`reltol` variable).

`compactabs`
: This is the absolute tolerance used in history list compaction for level = 1. The default value is the same as the *WRspice* default absolute tolerance (`abstol` variable).

The flag `lininterp`, when specified, will use linear interpolation instead of the default quadratic interpolation for calculating delayed signals.

The parameters `compactrel` and `compactabs` control the compaction of the past history of values stored for convolution when using level = 1. Larger values of these lower accuracy but usually increase simulation speed. These are to be used with the `trytocompact` option, described in the `.options` section.

2.14.7 Uniform RC Line

General Form:

u*name n1 n2 n3 modname* `l=`*len* `[n=`*lumps*`]`

Examples:

```
u1 1 2 0 urcmod l=50u
urc2 1 12 2 umodl l=1mil n=6
```

The *n1* and *n2* are the two element nodes the RC line connects, while *n3* is the node to which the capacitances are connected. The *modname* is the model name, *len* is the length of the RC line in meters, and *lumps*, if specified, is the number of lumped segments to use in modeling the RC line. If not specified, the value will be computed as

$$N = \frac{log\left(2\pi F_{max} RC \cdot ((k-1)/k)^2\right)}{log(k)}$$

where N is the number of lumps, k is the proportionality factor, R and C are the total values for the length, and F_{max} is the maximum frequency. See the description of the uniform RC line model (2.17.6) for more information.

2.15 Semiconductor Devices

The standard *WRspice* device library contains models for the semiconductor devices listed in the table below.

Device	Name	Key
Junction Diode	`dio`	`d`
Bipolar Junction Transistor	`bjt`	`q`
Junction Field-Effect Transistor	`jfet`	`j`
MESFET	`mes`	`z`
MOSFET	`mos`	`m`

Each device element line contains the device name, the nodes to which the device is connected, and the device model name. In addition, other optional parameters may be specified for some devices: geometric factors and an initial condition.

The area factor used on the device lines determines the number of equivalent parallel devices of a specified model. The affected parameters are marked with an asterisk under the heading "area" in the model descriptions. Several geometric factors associated with the channel and the drain and source diffusions can be specified on the MOSFET device card.

Two different forms of initial conditions may be specified for some devices. The first form is included to improve the dc convergence for circuits that contain more than one stable state. If a device is specified `off`, the dc operating point is determined with the terminal voltages for that device set to zero. After convergence is obtained, the program continues to iterate to obtain the exact value for the terminal voltages. If a circuit has more than one dc stable state, the `off` option can be used to force the solution to correspond to a desired state. If a device is specified `off` when in reality the device is conducting, the program will still obtain the correct solution (assuming the solutions converge) but more iterations will be required since the program must independently converge to two separate solutions. The `.nodeset` line serves a similar purpose as the `off` option. The `.nodeset` directive is easier to apply and is the preferred means to aid convergence in this situation.

The second form of initial conditions are specified for use with transient analysis. These are true initial conditions as opposed to the convergence aids above. See the description of the `.ic` line and the `.tran` line for a detailed explanation of initial conditions.

2.15.1 Junction Diodes

General Form:

d*name* *n+* *n-* *modname* [area=*val*] [temp=*optemp*] [off] [ic=*vd*]

Examples:

```
dbridge 2 10 diode1
dclmp 3 7 dmod 3.0 ic=0.2
```

The *n+* and *n-* are the positive and negative nodes, respectively. The parameter *modname* is the model name, `area` specifies the area factor, `temp` specifies the operating temperature, and `off` indicates

an (optional) starting condition of the device for dc analysis. If the area factor is omitted, a value of 1.0 is assumed. The (optional) initial voltage specification using `ic`=*vd* is intended for use with the `uic` option in transient analysis, when a transient analysis is desired starting from other than the quiescent operating point. The `.ic` line provides another way to set transient initial conditions.

See the description of the diode model in 2.17.7 for more information.

2.15.2 Bipolar Junction Transistors (BJTs)

General Form:

q*name nc nb ne* [*ns*] *modname* [area=*val*] [temp=*optemp*] [off] [ic=*vbe*,*vce*] [icvbe=*vbe*] [icvce=*vce*]

Examples:

```
q23 10 24 13 qmod ic=0.6,5.0
q50a 11 26 4 20 mod1
```

The *nc*, *nb*, and *ne* are the collector, base, and emitter nodes, respectively, and *ns* is the (optional) substrate node. If unspecified, ground is used. The *modname* is the model name, `area` specifies the area factor, `temp` specifies the operating temperature, and `off` indicates an initial condition of the device for the dc analysis. If the area factor is omitted, a value of 1.0 is assumed. The initial conditions specified using `ic` or alternatively `icvbe` and `icvce` are intended for use with the `uic` option in transient analysis, when a transient analysis is desired starting from other than the quiescent operating point. The `.ic` line provides another way to set transient initial conditions.

More information can be found in the description of the `bjt` model in 2.17.8.

2.15.3 Junction Field-Effect Transistors (JFETs)

General Form:

j*name nd ng ns modname* [area=*val*] [temp=*optemp*] [off] [ic=*vds*,*vgs*] [icvds=*vds*] [icvgs=*vgs*]

Examples:

```
j1 7 2 3 jm1 off
j43 10 4 1 jmod2 area=2
```

The *nd*, *ng*, and *ns* are the drain, gate, and source nodes, respectively. The *modname* is the model name, `area` specifies the area factor, `temp` specifies the operating temperature, and `off` indicates an (optional) initial condition of the device for dc analysis. If the area factor is omitted, a value of 1.0 is assumed. The initial conditions specified using `ic` or alternatively `icvds` and `icvgs` are intended for use with the `uic` option in transient analysis, when a transient analysis is desired starting from other than the quiescent operating point. The `.ic` line provides another way to set initial conditions.

More information can be found in the description of the `jfet` model in 2.17.9.

2.15.4 MESFETs

General Form:

z*name nd ng ns modname* [area=*val*] [off] [ic=*vds,vgs*] [icvds=*vds*] [icvgs=*vgs*]

Examples:

```
z1 7 2 3 zm1 off
zout 21 4 16 zmod1 area=5
```

The *nd*, *ng*, and *ns* are the drain, gate, and source nodes, respectively. The *modname* is the model name, *area* specifies the area factor, and `off` indicates an initial condition of the device for dc analysis. If the area factor is omitted, a value of 1.0 is assumed. The initial condition specified using `ic` or alternatively `icvds` and `icvgs` are intended for use with the `uic` option in transient analysis, when a transient analysis is desired starting from other than the quiescent operating point. The `.ic` line provides another way to set initial conditions.

More information can be found in the description of the `mes` model in 2.17.10.

2.15.5 MOSFETs

General Form:

`m`*name nd ng ns nb modname* `[l=`*val*`] [w=`*val*`] [ad=`*val*`] [as=`*val*`] [pd=`*val*`] [ps=`*val*`]`
`[nrd=`*val*`] [nrs=`*val*`] [m=`*factor*`] [temp=`*optemp*`] [off] [ic=`*vds,vgs,vbs*`] [icvds=`*vds*`]`
`[icvgs=`*vgs*`] [icvbs=`*vbs*`]`

Examples:

```
m1 24 2 0 20 type1
m31 2 17 6 10 modm l=5u w=2u
m1 2 9 3 0 mod1 l=10u w=5u ad=100p as=100p pd=40u ps=40u
```

The parameters listed above are representative of the SPICE3 MOS models, but except for 'm' are fairly universal. Some third-party MOS models may have additional nodes and parameters. Consult the model documentation for the full listing.

The *nd*, *ng*, *ns*, and *nb* are the drain, gate, source, and bulk (substrate) nodes, respectively. The *modname* is the model name, `l` and `w` specify the channel length and width in meters, and `ad` and `as` specify the areas of the drain and source diffusions in sq-meters. Note that the suffix 'u' specifies microns (1E-6 m) and 'p' sq-microns (1E-12 sq-m). If any of `l`, `w`, `ad`, or `as` are not specified, default values are used. The use of defaults simplifies input file preparation, as well as the editing required if device geometries are to be changed. The `pd` and `ps` specify the perimeters of the drain and source junctions in meters, `nrd` and `nrs` designate the equivalent number of squares of the drain and source diffusions; these values multiply the sheet resistance `rsh` specified on the `.model` line for an accurate representation of the parasitic series drain and source resistance of each transistor. The `pd` and `ps` default to 0.0 while `nrd` and `nrs` default to 1.0. The parameter `off` indicates an initial condition of the device for dc analysis. The initial conditions specified using `ic` or alternatively `icvds`, `icvgs` amd `icvbs` are intended for use with the `uic` option in transient analysis, when a transient analysis is desired starting from other than the quiescent operating point. operating point. The `.ic` line provides another way to set initial conditions.

MOS devices using model levels 1-3 accept a real parameter "`m`" which scales all the instance capacitances, areas, and currents by the given `factor`. This can be used as a short-cut for modeling multiple devices, e.g., `m = 2` is equivalent to two identical devices in parallel. This is not available for most of the more complicated and third-party models.

More information can be found in the description of the `mos` models in 2.17.11.

2.16 Superconductor Devices

2.16.1 Josephson Junctions

General Form:

b*name* *n*+ *n*- [*np*] [*modname*] [ic=*v1*,*p1*] [vj=*v1*] [phi=*p1*] [area=*val*] [control=*name*]

Examples:

```
b1 1 0 10 jj1 area=2
bxx 2 0 type1 control=l3
b2 4 5 ybco phi=1.57
```

The *n*+ and *n*- are the positive and negative element nodes, respectively. These are followed by an optional phase node. The phase node, if specified, generally should have no other connections in the circuit, but the voltage of this node gives the phase of the junction in radians. The *modname* is the name of the Josephson junction model. If no model is specified, then a default model is used (see the description of the Josephson model in 2.17.12 for the default values). Other (optional) parameters follow in any order. The initial phase (in radians) and voltage of the junction can be specified either as a vector or independently. If not specified, the initial junction voltage and phase are taken to be zero. Transient analysis of circuits containing Josephson junctions always starts from initial conditions. The junction area (effectively the number of devices in parallel) can be specified with the `area` parameter. The control current source can be specified with the `control` parameter. The *name* in the control specification represents the name of either a voltage source or inductor which appears elsewhere in the circuit. The `control` parameter is only needed if critical current modulation is part of the circuit operation, and is only relevant to Josephson junction model types that support critical current modulation.

When Josephson junctions are present in the circuit, only transient analysis may be performed, and the transient `uic` parameter will be set implicitly. The time step is determined by junction phase change by default, in accordance with the setting of the `dphimax` option variable. If the variable `nojjtp` is set, the truncation error timestep predictor is used as is the case when Josephson junctions are not present.

See the Josephson junction model description in 2.17.12 for more information.

2.17 Device Models

Many devices reference models, which contain values for the numerous parameters describing the device, which would be cumbersome to include in each device reference. Device models are specified on a `.model` line. The model can be referenced by any number of devices of the corresponding type.

General Form:

`.model` *modname type*(*pname1=pval1 pname2=pval2 ...*)

Examples:

```
.model mod1 npn (bf=50 is=1e-13 vbf=50)
.model intercon ltra (r=0.2 l=9.13nh c=3.65pf len=5 rel=.002 compactrel=1.0E-4)
```

The `.model` line specifies a set of model parameters that will be used by one or more devices. The *modname* is the model name, which is case insensitive in matching references, and *type* is one of the following types:

c	Capacitor model
r	Resistor model
sw	Voltage-controlled switch model
csw	Current-controlled switch model
tra	General transmission line model
ltra	Lossy transmission line model
urc	Uniform RC line model
d	Diode model
npn	NPN BJT model
pnp	PNP BJT model
njf	N-channel JFET model
pjf	P-channel JFET model
nmf	N-channel MESFET model
pmf	P-channel MESFET model
nmos	N-channel MOSFET model
pmos	P-channel MOSFET model
jj	Josephson junction model

Parameter values are defined by appending the parameter name, as given for each model type, followed by an equal sign and the parameter value. Model parameters that are not given a value are generally assigned default values.

The show command with the -M option is useful for listing the parameters that can be specified to a model. Only the parameters not listed as "RO" (read-only) can appear in a .model line.

In the tables that follow, the various model parameters are listed. The "units" field of the tables provides the assumed units of measure for the parameter, which is expressed using symbols from the following table.

M	meters
cM	centimeters
μM	microns
S	seconds
Hz	hertz
F	farads
H	henries
Ω	ohms
C	degrees Celsius
$\square$	square
A	amperes
V	volts
eV	electron-volts
deg	degrees

2.17.1 Analysis at Different Temperatures

All input data for *WRspice* is assumed to have been measured at a nominal temperature of 27C, which can be changed by use of the tnom parameter on the .options control line. This value can further be overridden for any device which models temperature effects by specifying the tnom parameter on the model itself. The circuit simulation is performed at a temperature of 27C unless overridden by a temp parameter on the .options control line. Individual device instances may further override the circuit

temperature through the specification of a `temp` parameter on the instance.

Temperature dependent support is provided for resistors and semiconductor devices. The details of the temperature adjustments can be found in the description of the models. For details of the BSIM temperature adjustment, see [5] (BSIM1), [6] (BSIM2), and [8] (BSIM3).

Temperature appears explicitly in the exponential terms of the BJT and diode model equations. In addition, saturation currents have a built-in temperature dependence. The temperature dependence of the saturation current in the BJT models is determined by:

$$I_s(T_1) = I_s(T_0)\left(\frac{T_1}{T_0}\right)^{XTI} exp\left(\frac{E_g q(T_1 - T_0)}{kT_1T_0}\right)$$

where k is Boltzmann's constant, q is the electronic charge, E_g is the energy gap which is a model parameter, and XTI is the saturation current temperature exponent (also a model parameter, and usually equal to 3).

The temperature dependence of forward and reverse beta is according to the formula:

$$\beta(T_1) = \beta(T_0)\left(\frac{T_1}{T_0}\right)^{XTB}$$

where T_1 and T_0 are in Kelvin, and XTB is a user-supplied model parameter. Temperature effects on beta are carried out by appropriate adjustment to the values of β_F, I_{SE}, β_R, and I_{SC} (*WRspice* model parameters `bf, ise, br,` and `isc,` respectively.

Temperature dependence of the saturation current in the junction diode model is determined by:

$$I_S(T_1) = I_S(T_0)\left(\frac{T_1}{T_0}\right)^{\frac{XTI}{N}} exp\left(\frac{E_g q(T_1 - T_0)}{NkT_1T_0}\right)$$

where N is the emission coefficient, which is a model parameter, and the other symbols have the same meaning as above. Note that for Schottky barrier diodes, the value of the saturation current temperature exponent, XTI, is usually 2.

Temperature appears explicitly in the value of junction potential, ϕ (in *WRspice*, `phi`), for all device models. The temperature dependence is determined by:

$$\phi(T) = \frac{kT}{q} ln\left(\frac{N_a N_d}{N_i(T)^2}\right)$$

where k is Boltzmann's constant, q is the electronic charge, N_a is the acceptor impurity density, N_d is the donor impurity density, and N_i is the intrinsic carrier concentration.

Temperature appears explicitly in the value of surface mobility, μ_0 (or `u0`, for the MOSFET models. This temperature dependence is determined by:

$$\mu_0(T) = \frac{\mu_0(T_0)}{\left(\frac{T}{T_0}\right)^{1.5}}$$

The effects of temperature on resistors is modeled by the formula:

$$R(T) = R(T_0)[1 + TC_1(T - T_0) + TC_2(T - T_0)^2]$$

where T is the circuit temperature, T_0 is the nominal temperature, and TC_1 and TC_2 are the first and second-order temperature coefficients.

2.17.2 Capacitor Model

Type Name: `c`

The capacitor model contains process information that may be used to compute the capacitance from strictly geometric information.

Capacitor Model Parameters				
name	**parameter**	**units**	**default**	**example**
`cj`	junction bottom capacitance	F/M^2	-	5e-5
`cjsw`	junction sidewall capacitance	F/M	-	2e-11
`defw`	default device width	M	1e-6	2e-6
`narrow`	narrowing due to side etching	M	0.0	1e-7

The capacitor has a capacitance computed as

C = `cj·(l−narrow)(w−narrow) + 2cjsw·(l+w−2·narrow)`

2.17.3 Resistor Model

Type Name: `r`

The resistor model consists of process-related device data that allow the resistance to be calculated from geometric information and to be corrected for temperature. The parameters available are:

Resistor Model Parameters				
name	**parameter**	**units**	**default**	**example**
`tc1`	first order temperature coeff	Ω/C	0.0	-
`tc2`	second order temperature coeff	Ω/C^2	0.0	-
`rsh`	sheet resistance	$\Omega/\square$	-	50
`defw`	default width	M	1e-6	2e-6
`narrow`	narrowing due to side etching	M	0.0	1e-7
`tnom`	parameter measurement temperature	C	27	50

The sheet resistance is used with the narrowing parameter and `l` and `w` from the resistor element line to determine the nominal resistance by the formula

R = `rsh·(l - narrow)/(w - narrow)`.

The parameter `defw` is used to supply a default value for `w` if one is not specified on the device line. If either `rsh` or `l` is not specified, then the standard default resistance value of 1k is used.

After the nominal resistance is calculated, it is adjusted for temperature by the formula:

R`(temp)` = R`(tnom)·(1 + tc1·(temp−tnom) + tc2·(temp−tnom)`^{2}`)`

2.17.4 Switch Model

Type Names: `csw`, `sw`

The switch model allows an almost ideal switch to be described in *WRspice*. The switch is not quite ideal, in that the resistance can not change from 0 to infinity, but must always have a finite positive resistance. By proper selection of the on and off resistances, they can be effectively zero and infinity in comparison to other circuit elements. There are two different types of switch devices; current-controlled (keyed by `w`), and voltage-controlled (keyed by `s`). Both reference the model described below. The parameters available are:

Switch Model Parameters				
name	**parameter**	**units**	**default**	**switch**
`vt`	threshold voltage	V	0.0	S
`it`	threshold current	A	0.0	W
`vh`	hysteresis voltage	V	0.0	S
`ih`	hysteresis current	A	0.0	W
`ron`	on resistance	Ω	1.0	both
`roff`	off resistance	Ω	1/`gmin`*	both

* The `gmin` parameter, can be set on the `.options` line. Its default value results is an off resistance of 1.0e+12 ohms.

The use of an ideal element that is highly nonlinear such as a switch can cause large discontinuities to occur in the circuit node voltages. A rapid change such as that associated with a switch changing state can cause numerical roundoff or tolerance problems leading to erroneous results or timestep difficulties. The user of switches can improve the situation by taking the following steps:

First of all it is wise to set ideal switch impedances only high and low enough to be negligible with respect to other circuit elements. Using switch impedances that are close to "ideal" in all cases will aggravate the problem of discontinuities mentioned above. Of course, when modeling real devices such as MOSFETS, the on resistance should be adjusted to a realistic level depending on the size of the device being modeled.

If a wide range of on to off resistance must be used in the switches (roff/ron > 1e12), then the tolerance on errors allowed during transient analysis should be decreased by using the `.options` line and specifying `trtol` to be less than the default value of 7.0 (options can also be set from the prompt line from within *WRspice*). When switches are placed around capacitors, then the option `chgtol` should also be reduced. Suggested values for these two options are 1.0 and 1e-16 respectively. These changes inform *WRspice* to be more careful around the switch points so that no errors are made due to the rapid change in the circuit.

2.17.5 Transmission Line Model

Type Names: `ltra`, `tra`

The general transmission line model may be used in conjunction with transmission line devices, though the use is optional. The parameters that appear in the model are the same parameters that can be given on the device line (with the exception of the initial conditions). These parameters are discussed in section 2.14.6 describing the general transmission line.

Transmission line models can have either of two type names: `tra` or `ltra`. The `ltra` name is required to support the SPICE3 LTRA lossy transmission line model. If this name is used, the `level` will default

to 1. This will be overridden if the level is set explicitly. If the `tra` keyword is used, then the level will default to 0. Otherwise, the two words are interchangeable.

The parameters provided in the model will serve as defaults to the referencing device, but can be overridden if explicitly set on the device line.

2.17.6 Uniform Distributed RC Model

Type Name: `urc`

The urc model is derived from a model proposed by L. Gertzberrg in 1974. The model is generated by a subcircuit type expansion of the urc line into a network of lumped RC segments with internally generated nodes. The RC segments are in a geometric progression, increasing toward the middle of the urc line, with `k` as a proportionality constant. The number of lumped segments used, if not specified on the urc line, is determined by the following formula:

$$N = \frac{log\left(2\pi FmaxRC\left((k-1)/k\right)^2\right)}{log(k)}$$

where *Fmax* is the maximum frequency, and R and C are the total values for the given length.

The urc will be made up strictly of resistor and capacitor segments unless the `isperl` parameter is given a non-zero value, in which case the capacitors are replaced with reverse biased diodes with a zero-bias junction capacitance equivalent to the capacitance replaced, and with a saturation current taking the value given `isperl` in amps per meter of transmission line, and with an optional series resistance specified by `rsperl` in ohms per meter.

URC Model Parameters

name	parameter	units	default	example
`k`	propagation Constant	-	1.5	1.2
`fmax`	maximum frequency of interest	Hz	1.0G	6.5meg
`rperl`	resistance per unit length	Ω/M	1000	10
`cperl`	capacitance per unit length	F/M	1.0e-12	2pf
`isperl`	saturation current per unit length	A/M	0	-
`rsperl`	diode resistance per unit length	Ω/M	0	-

2.17.7 Diode Model

Type Name: `d`

The dc characteristics of the diode are determined by the parameters `is` and `n`. An ohmic resistance, `rs`, is included. Charge storage effects are modeled by a transit time, `tt`, and a nonlinear depletion layer capacitance which is determined by the parameters `cjo`, `vj`, and `m`. The temperature dependence of the saturation current is defined by the parameters `eg`, the energy, and `xti`, the saturation current temperature exponent. The nominal temperature at which these parameters were measured is `tnom`, which defaults to the value specified on the `.options` control line. Reverse breakdown is modeled by an exponential increase in the reverse diode current and is determined by the parameters `bv` and `ibv` (both of which are positive numbers).

The parameters marked with an asterisk in the **area** column scale with the `area` parameter given in the device line.

Diode Model Parameters					
name	**area**	**parameter**	**units**	**default**	**example**
is	*	saturation current	A	1.0e-14	1.0e-14
rs	*	ohmic resistance	Ω	0	10
n		emission coefficient	-	1	1.0
tt		transit-time	S	0	0.1ns
cjo	*	zero-bias junction capacitance	F	0	2PF
vj		junction potential	V	1	0.6
m		grading coefficient	-	0.5	0.5
eg		activation energy	eV	1.11	1.11 Si, 0.69 Sbd, 0.67 Ge
xti		saturation-current temperature exponent	-	3.0	3.0 junc, 2.0 Sbd
kf		flicker noise coefficient	-	0	-
af		flicker noise exponent	-	1	-
fc		coefficient for forward-bias depletion capacitance formula	-	0.5	-
bv		reverse breakdown voltage	V	infinite	40.0
ibv		current at breakdown voltage	A	1.0e-3	2.0e-3
tnom		parameter measurement temperature	C	27	50

2.17.8 BJT Models (both NPN and PNP)

Type Names: npn, pnp

The bipolar junction transistor model in *WRspice* is an adaptation of the integral charge control model of Gummel and Poon. This modified Gummel-Poon model extends the original model to include several effects at high bias levels. The model will automatically simplify to the simpler Ebers-Moll model when certain parameters are not specified. The parameter names used in the modified Gummel-Poon model have been chosen to be more easily understood by the program user, and to reflect better both physical and circuit design thinking.

The dc model is defined by the parameters `is`, `bf`, `nf`, `ise`, `ikf`, and `ne` which determine the forward current gain characteristics, `is`, `br`, `nr`, `isc`, `ikr`, and `nc` which determine the reverse current gain characteristics, and `vaf` and `var` which determine the output conductance for forward and reverse regions. Three ohmic resistances `rb`, `rc`, and `re` are included, where `rb` can be high current dependent. Base charge storage is modeled by forward and reverse transit times, `tf` and `tr`, the forward transit time `tf` being bias dependent if desired, and nonlinear depletion layer capacitances which are determined by `cje`, `vje`, and `mje` for the B-E junction, `cjc`, `vjc`, and `mjc` for the B-C junction, and `cjs`, `vjs`, and `mjs` for the C-S (Collector-Substrate) junction. The temperature dependence of the saturation current, `is`, is determined by the energy gap, `eg`, and the saturation current temperature exponent, `xti`. Additionally base current temperature dependence is modeled by the beta temperature exponent `xtb` in the new model. The values specified are assumed to have been measured at the temperature `tnom`, which can be specified on the `.options` line or overridden by a specification on the `.model` line.

The BJT parameters used in the modified Gummel-Poon model are listed below. The parameter names used in earlier versions of SPICE2 are still accepted. The parameters marked with an asterisk in the **area** column scale with the `area` parameter given in the device line.

BJT Model Parameters					
name	**area**	**parameter**	**units**	**default**	**example**
is	*	transport saturation current	A	1.0e-16	1.0e-15
bf		ideal maximum forward beta	-	100	100
nf		forward current emission coefficient	-	1.0	1
vaf		forward Early voltage	V	infinite	200
ikf	*	corner for forward beta high current roll-off	A	infinite	0.01
ise	*	B-E leakage saturation current	A	0	1.0e-13
ne		B-E leakage emission coefficient	-	1.5	2
br		ideal maximum reverse beta	-	1	0.1
nr		reverse current emission coefficient	-	1	1
var		reverse Early voltage	V	infinite	200
ikr	*	corner for reverse beta high current roll-off	A	infinite	0.01
isc	*	B-C leakage saturation current	A	0	1.0e-13
nc		B-C leakage emission coefficient	-	2	1.5
rb	*	zero bias base resistance	Ω	0	100
ikb	*	current where base resistance falls halfway to its min value	A	infinite	0.1
rbm	*	minimum base resistance at high currents	Ω	rb	10
re	*	emitter resistance	Ω	0	1
rc	*	collector resistance	Ω	0	10
cje	*	B-E zero-bias depletion capacitance	F	0	2pf
vje		B-E built-in potential	V	0.75	0.6
mje		B-E junction exponential factor	-	0.33	0.33
tf		ideal forward transit time	S	0	0.1ns
xtf		coefficient for bias dependence of tf	-	0	-
vtf		voltage describing VBC dependence of tf	V	infinite	-
itf	*	high-current parameter for effect on tf	A	0	-
ptf		excess phase at freq=1.0/(tf·2π) Hz	deg	0	-
cjc	*	B-C zero-bias depletion capacitance	F	0	2pf
vjc		B-C built-in potential	V	0.75	0.5
mjc		B-C junction exponential factor	-	0.33	0.5
xcjc		fraction of B-C depletion capacitance connected to internal base node	-	1	-
tr		ideal reverse transit time	S	0	10ns
cjs	*	zero-bias collector-substrate capacitance	F	0	2pf

vjs		substrate junction built-in potential	V	0.75	-
mjs		substrate junction exponential factor	-	0	0.5
xtb		forward and reverse beta temperature exponent	-	0	-
eg		energy gap for temperature effect on is	eV	1.11	-
xti		temperature exponent for effect on is	-	3	-
kf		flicker-noise coefficient	-	0	-
af		flicker-noise exponent	-	1	-
fc		coefficient for forward-bias depletion capacitance formula	-	0.5	-
tnom		parameter measurement temperature	C	27	50

2.17.9 JFET Models (both N and P Channel)

Type Names: njf, pjf

There are two JFET models available, selectable with the level parameter given in the list of model parameters. If level=2 is given, the Parker-Skellern JFET model from Macquarie University in Sydney, Australia will be used. Parameters given must apply to that model. Documentation for this model is available from the Whiteley Research web site, or from
http://www.elec.mq.edu.au/cnerf/spice/spice.html.

If no level parameter is given, or is set to something other than 2, the standard SPICE3 JFET model will be used. This JFET model is derived from the FET model of Shichman and Hodges. The dc characteristics are defined by the parameters vto and beta, which determine the variation of drain current with gate voltage, lambda, which determines the output conductance, and is, the saturation current of the two gate junctions. Two ohmic resistances, rd and rs, are included. Charge storage is modeled by nonlinear depletion layer capacitances for both gate junctions which vary as the -1/2 power of junction voltage and are defined by the parameters cgs, cgd, and pb. The fitting parameter b is a new addition, see[7].

The parameters marked with an asterisk in the **area** column scale with the area parameter given in the device line.

JFET Model Parameters					
name	**area**	**parameter**	**units**	**default**	**example**
vto		threshold voltage	V	-2.0	-2.0
beta	*	transconductance parameter	A/V^2	1.0e-4	1.0e-3
lambda		channel length modulation parameter	$1/V$	0	1.0e-4
rd	*	drain ohmic resistance	Ω	0	100
rs	*	source ohmic resistance	Ω	0	100
cgs	*	zero-bias G-S junction capacitance	F	0	5pf
cgd	*	zero-bias G-D junction capacitance	F	0	1pf
pb		gate junction potential	V	1	0.6
is	*	gate junction saturation current	A	1.0e-14	1.0e-14
b		doping tail parameter	-	1	1.1
kf		flicker noise coefficient	-	0	-
af		flicker noise exponent	-	1	-
fc		coefficient for forward-bias depletion capacitance formula	-	0.5	-
tnom		parameter measurement temperature	C	27	50

2.17.10 MESFET Models (both N and P Channel)

Type Names: nmf, pmf

The MESFET model is derived from the GaAs FET model of Statz et al. as described in[10]. The dc characteristics are defined by the parameters vto, b, and beta, which determine the variation of drain current with gate voltage, alpha, which determines saturation voltage, and lambda, which determines the output conductance. The formula are given by

$$Id = \frac{\beta(V_{gs} - V_T)^2}{1 + b(V_{gs} - V_T)} \left(1 - \left(1 - \alpha\frac{V_{ds}}{3}\right)^3\right)(1 + \lambda V_{ds}) \quad \text{for} \quad 0 < V_{ds} < 3/\alpha$$

$$Id = \frac{\beta(V_{gs} - V_T)^2}{1 + b(V_{gs} - V_T)}(1 + \lambda V_{ds}) \quad \text{for} \quad V_{ds} > 3/\alpha$$

Two ohmic resistances, rd and rs, are included. Charge storage is modeled by total gate charge as a function of gate-drain and gate-source voltages and is defined by the parameters cgs, cgd, and pb.

The parameters marked with an asterisk in the **area** column scale with the area parameter given in the device line.

MES Model Parameters					
name	**area**	**parameter**	**units**	**default**	**example**
vto		pinch-off voltage	V	-2.0	-2.0
beta	*	transconductance parameter	A/V^2	1.0e-4	1.0e-3
b	*	doping tail extending parameter	$1/V$	0.3	0.3
alpha	*	saturation voltage parameter	$1/V$	2	2
lambda		channel length modulation parameter	$1/V$	0	1.0e-4
rd	*	drain ohmic resistance	Ω	0	100
rs	*	source ohmic resistance	Ω	0	100
cgs	*	zero-bias G-S junction capacitance	F	0	5pf
cgd	*	zero-bias G-D junction capacitance	F	0	1pf
pb		gate junction potential	V	1	0.6
kf		flicker noise coefficient	-	0	-
af		flicker noise exponent	-	1	-
fc		coefficient for forward-bias depletion capacitance formula	-	0.5	-

2.17.11 MOSFET Models (both N and P channel)

The device library supplied with *WRspice* contains a number of MOS models supplied by various development groups. The models that are currently provided in the device library are listed below. Specific models are selected through the `level` parameter. Other parameters are specific to that model, and there is not in general a great deal of commonality of parameter names between the various models. Only the simple models provided in SPICE3 will be documented. Documentation for the third-party models is available from the Whiteley Research web site.

The user should see the help system for the most recent list of available models, since the list may have changed after the manual was printed.

Level	Name	Description
1	MOS	The SPICE3 mos1 (Shichman-Hodges) model
2	MOS	The SPICE3 mos2 model described in [1]
3	MOS	The SPICE3 mos3 semi-empirical model (see[1])
4	BSIM1	The SPICE3 bsim1 empirical model described in [2]
5	BSIM2	The SPICE3 bsimw model, successor to bsim1
6	MOS	The SPICE3 mos6 model
7, 49	BSIM3v3.2	U.C. Berkeley bsim-3.2 model
8	BSIM3v3.2.4	U.C. Berkeley bsim-3.2.4 model
9	B3SOI3.2	U.C. Berkeley bsim3-soi 3.2 SOI model
10	UFS	U. Florida SOI model release 7.5
14	BSIM4.4.0	U.C. Berkeley bsim-4.4.0
33	Soi3	Southampton Thermal Analogue (STAG-2.6) SOI model
44	EKV-2.6	EPFL (Switzerland) MOS model release 2.6
60	HISIM-1.1	Hiroshima University hisim-1.1

The level 44 model installation has not been validated by EPFL, and by agreement until such vali-

dation is performed (soon) there is no claim that this is *THE* EKV model.

Manuals for the BSIM3 and third-party MOS models are available on the Whiteley Research web site.

2.17.11.1 HSPICE MOS Level 49 Compatibility in *WRspice*

In *WRspice*, level 49 is equivalent to level 7, which is the BSIM3v3.2 model from Berkeley. However, there are differences in the parameter sets between this model and the level 49 model of HSPICE, which is based on BSIM3 and customized for HSPICE. In particular, the HSPICE extensions are not supported in *WRspice*.

This document lists the parameters that are accepted in HSPICE level 49 model parameter sets that are not supported in the Berkeley model, or have different interpretation. The parameters are listed in alphabetical order, by category. This list is possibly incomplete.

This list also applies to the level 8 (BSIM3v3.2.4) model in *WRspice*. In *WRspice* release 2.2.50 and earlier, level 49 was mapped to level 8. However, the `VERSION` parameter in level 8 *must* be "3.2.4" if given, which is not generally true in imported files. The level 7 model handles earlier versions, notably 3.1, without complaint.

2.17.11.1.1 General parameters

`BINFLAG`
: This is not used for *WRspice*.

`SCALM`, `SCALE`
: In HSPICE these are the "model scaling factor" and "element scaling factor". There are no *WRspice* equivalents.

`TREF`
: Temperature at which parameters are extracted. This is taken as an alias for the BSIM3 `TNOM` parameter in *WRspice*.

`NQSMOD`
: This is accepted by *WRspice* as an extension to BSIM3, and serves as the default for devices that use the model. If also given on the device line, that value will override.

2.17.11.1.2 Length and Width

`LREF`, `WREF`
: In HSPICE these are "channel length reference" and "channel width reference". There are no *WRspice* equivalents.

`XW`, `XL`
: In HSPICE these account for masking and etching effects. Use `WINT` and `LINT` in *WRspice*.

$$WINT_{new} = WINT_{old} \text{ - } XW/2$$
$$LINT_{new} = LINT_{old} \text{ - } XL/2$$

2.17.11.1.3 MOS Diode Model Parameters

ACM
: In HSPICE this selects the area calculation method. *WRspice* uses only one model for the bulk-to-source and bulk-to-drain diodes. It corresponds to the HSPICE equivalent of ACM=0. Do not use this parameter for *WRspice*. ACM is not needed if AS, AD, PS, and PD are specified explicitly.

CJGATE
: In HSPICE this is the zero-bias gate-edge sidewall bulk junction capacitance used with ACM=3 only. There is no *WRspice* equivalent.

HDIF, LDIF
: In HSPICE this is the "length of heavily doped diffusion" and "length of lightly doped diffusion". They are used with the HSPICE ACM=2 MOS diode models, and there are no *WRspice* equivalents. HDIF and LDIF are not needed if AS, AD, PS, and PD are specified explicitly.

N
: In HSPICE this is the "emission coefficient", and is taken as an alias for the BSIM3 NJ parameter.

RDC
: In HSPICE this is additional drain resistance due to contact resistance. If RD is specified, use

$$RD_{new} = RD_{old} + RDC$$

: If RSH is specified, then RDC should be added to RD=NRD*RSH. Since NRD is a device parameter and not a model parameter, a typical NRD value must be used.

RSC
: In HSPICE this is additional source resistance due to contact resistance. If RS is specified, use

$$RS_{new} = RS_{old} + RSC$$

: If RSH is specified, then RSC should be added to RS=NRS*RSH. Since NRS is a device parameter and not a model parameter, a typical NRS value must be used.

WMLT, LMLT
: In HSPICE these are "length of heavily doped diffusion" and "length of lightly doped diffusion" used in the ACM=1–3 models. There are no *WRspice* equivalents. WMLT and LMLT are not needed if AS, AD PS, and PD are specified explicitly.

2.17.11.1.4 Capacitance Parameters

CAPOP
: Do not use CAPOP for *WRspice*. CAPMOD is included in the BSIM3 model. CAPOP is HSPICE specific, and not included in the BSIM3 parameter set. The default BSIM3 capacitance model is CAPMOD=3.

CJM
: This is taken as an alias for the BSIM3 CJ parameter in *WRspice*.

MJO
: This is taken as an alias for the BSIM3 MJ parameter in *WRspice*.

PJ
: This is taken as an alias for the BSIM3 PB parameter in *WRspice*.

CTA, CTP
: In HSPICE these are the "junction capacitance CJ temp. coeff." and "junction sidewall capacitance CJSW temp. coeff", used with TLEVC=1. There are no *WRspice* equivalents.

PTA, PTP
: In HSPICE these are the "junction potential PB temp. coeff." and "fermi potential PHI temp. coeff", used with TLEVC=1 or 2. There are no *WRspice* equivalents.

PHP
: This is taken as an alias for the BSIM3 PBSW parameter in *WRspice*

TLEV, TLEVC
: In HSPICE this is the "temperature equation level selector" and "temperature equation level selector for junction capacitances and potentials". Do not use these parameters for *WRspice*.

2.17.11.1.5 Impact Ionization

ALPHA, LALPHA, WALPHA, VCR, LVCR, WVCR, and IIRAT
: These are impact ionization parameters in HSPICE. There are no *WRspice* equivalents. BSIM3 has its own impact ionization model which is instead used in most cases.

2.17.11.1.6 V3.2 parameters Level 49 parameter sets in *WRspice* may include BSIM3v3.2 parameters, though historically HSPICE level 49 was based on BSIM3v3.1. The following parameters are new for v3.2:

ALPHA1, ACDE, MOIN, NOFF, VOFFCV
: All except ALPHA1 are used in a new capacitance model (CAPMOD=3). ALPHA1 modifies the substrate current equation as follows:

```
Isub ~ (ALPHA0 + ALPHA1*Leff) / Leff
```

2.17.11.2 SPICE3 MOS Models

Type Names: nmos, pmos

WRspice provides the basic MOS models provided in SPICE2/3, plus third-party models from various development groups. A complete listing of the MOS models supported in the current *WRspice* release was provided earlier. This section describes only the basic SPICE2/3 models. Documentation for the third-party models is available on the Whiteley Research web site.

The level parameter specifies the model to be used.

WRspice supports MOS model selection by the L and W MOS element line. This facility is used to automatically select the proper model for a specific device dimension, from among several models which are nominally similar but optimized for a particular device size. This facility works with any of the MOS model levels.

The MOS models for the specified ranges should use the same "base" name plus an arbitrary but unique extension separated from the base name with a period. The element line should refer to the model by the base name. Each of the different models with the same base name should have ranges specified with the model parameters LMIN, LMAX, WMIN, WMAX. The model associated with an instance is the first model found such that the base name matches, and LMIN <= L <= LMAX and WMIN <= W <=

WMAX. If the MIN/MAX parameters are not found in a model line, the test is always true, i.e., if no MIN/MAX parameters are specified in a model, that model would match any L, W.

Example:

```
m1 1 2 3 4 nm l=1.5u w=1u
m2 a b c d nm l=3u w=5u
...
.model nm.1 nmos(level=8 lmin=1u lmax=2u wmin=1u wmax=2u ...)
.model nm.2 nmos(level=8 lmin=2u lmax=4u wmin=1u wmax=2u ...)
.model nm.3 nmos(level=8 lmin=1u lmax=2u wmin=2u wmax=5u ...)
.model nm.4 nmos(level=8 lmin=2u lmax=4u wmin=2u wmax=5u ...)
.model nm.5 nmos(level=8 ...)
```

In this example m1 would use model nm.1, and m2 would use nm.4. The model nm.5 is a "catch all" for elements that don't match the other models. The extension can be omitted in one of the model names.

The dc characteristics of the level 1 through level 3 MOSFETs are defined by the device parameters vto, kp, lambda, phi and gamma. These parameters are computed by *WRspice* if process parameters (nsub, tox, ...) are given, but user-specified values always override. The parameter vto is positive (negative) for enhancement mode and negative (positive) for depletion mode N-channel (P-channel) devices. Charge storage is modeled by three constant capacitors, cgso, cgdo, and cgbo which represent overlap capacitances, by the nonlinear thin-oxide capacitance which is distributed among the gate, source, drain, and bulk regions, and by the nonlinear depletion-layer capacitances for both substrate junctions divided into bottom and periphery, which vary as the mj and mjsw power of junction voltage respectively, and are determined by the parameters cbd, cbs, cj, cjsw, mj, mjsw and pb. Charge storage effects are modeled by the piecewise linear voltage dependent capacitance model proposed by Meyer. The thin-oxide charge storage effects are treated slightly differently for the level 1 model. These voltage-dependent capacitances are included only if tox is specified in the input description and they are represented using Meyer's formulation.

There is some overlap among the parameters describing the junctions, e.g., the reverse current can be input either as is (in Amps) or as js (in Amps/m^2). Whereas the first is an absolute value, the second is multiplied by ad and as to give the reverse current of the drain and source junctions respectively. This methodology has been chosen to avoid always relating junction characteristics with ad and as entered on the device line; the areas can be defaulted. The same idea applies also to the zero-bias junction capacitances cbd and cbs (in Farads) on one hand, and cj (in F/m^2) on the other. The parasitic drain and source series resistance can be expressed as either rd and rs (in ohms) or rsh (in ohms/sq.), the latter being multiplied by the number of squares nrd and nrs input on the device line.

MOS Level 1 to Level 3 Parameters

name	parameter	units	default	example
level	Model index	-	1	
vto	zero-bias threshold voltage	V	0.0	1.0
kp	transconductance parameter	A/V^2	2.0e-5	3.1e-5
gamma	bulk threshold parameter	$V^{1/2}$	0.0	0.37
phi	surface potential	V	0.6	0.65
lambda	channel-length modulation (MOS1 and MOS2 only)	$1/V$	0.0	0.02

rd	drain ohmic resistance	Ω	0.0	1.0
rs	source ohmic resistance	Ω	0.0	1.0
cbd	zero-bias B-D junction capacitance	F	0.0	20fF
cbs	zero-bias B-S junction capacitance	F	0.0	20fF
is	bulk junction saturation current	A	1.0e-14	1.0e-15
pb	bulk junction potential	A	0.8	0.87
cgso	gate-source overlap capacitance per channel width	F/M	0.0	4.0e-11
cgdo	gate-drain overlap capacitance per channel width	F/M	0.0	4.0e-11
cgbo	gate-bulk overlap capacitance per channel length	F/M	0.0	2.0e-10
rsh	drain and source diffusion sheet resistance	$\Omega/\square$	0.0	10.0
cj	zero-bias bulk junction bottom capacitance per junction area	F/M^2	0.0	2.0e-4
mj	bulk junction bottom grading coeff	-	0.5	0.5
cjsw	zero-bias bulk junction sidewall capacitance per junction perimeter	F/M	0.0	1.0e-9
mjsw	bulk junction sidewall grading coeff.	-	0.50 (level 1), 0.33 (level 2,3)	-
js	bulk junction saturation current per junction area	A/M^2	1.0e-8	-
tox	oxide thickness	M	1.0e-7	1.0e-7
nsub	substrate doping	$1/cM^3$	0.0	4.0e15
nss	surface state density	$1/cM^2$	0.0	1.0e10
nfs	fast surface state density	$1/cM^2$	0.0	1.0e10
tpg	type of gate material: +1 opp. to substrate, -1 same as substrate, 0 Al gate	-	1.0	-
xj	metallurgical junction depth	M	0.0	1u
ld	lateral diffusion	M	0.0	0.8u
uo	surface mobility	cM^2/VS	600	700
ucrit	critical field for mobility degradation (MOS2 only)	V/cM	1.0e4	1.0e4
uexp	critical field exponent in mobility degradation (MOS2 only)	-	0.0	0.1
utra	transverse field coeff (mobility) (deleted for MOS2)	-	0.0	0.3
vmax	maximum drift velocity of carriers	M/S	0.0	5.0e4
neff	total channel charge (fixed and mobile) coefficient (MOS2 only)	-	1.0	5.0
kf	flicker noise coefficient	-	0.0	1.0e-26
af	flicker noise exponent	-	1.0	1.2

`fc`	coefficient for forward-bias depletion capacitance formula	-	0.5	-
`delta`	width effect on threshold voltage (MOS2 and MOS3)	-	0.0	1.0
`theta`	mobility modulation (MOS3 only)	$1/V$	0.0	0.1
`eta`	static feedback (MOS3 only)	-	0.0	1.0
`kappa`	saturation field factor (MOS3 only)	-	0.2	0.5
`tnom`	parameter measurement temperature	C	27	50

The level 4 (BSIM1) parameters are all values obtained from process characterization, and can be generated automatically. J. Pierret[3] describes a means of generating a "process" file, and the program `proc2mod` provided with *WRspice* will convert this file into a sequence of `.model` lines suitable for inclusion in *WRspice* input. Parameters marked below with an * in the l/w column also have corresponding parameters with a length and width dependency. For example, `vfb` is the basic parameter with units of volts, and `lvfb` and `wvfb` also exist and have units of volt-μmeter. The formula

$$P = P_0 + \frac{P_L}{L_{effective}} + \frac{P_W}{W_{effective}}$$

is used to evaluate the parameter for the actual device specified with

$$L_{effective} = L_{input} - dl$$

and

$$W_{effective} = W_{input} - dw.$$

Note that unlike the other models in *WRspice*, the BSIM model is designed for use with a process characterization system that provides all the parameters, thus there are no defaults for the parameters, and leaving one out is considered an error. For an example set of parameters and the format of a process file, see the SPICE2 implementation notes[2].

BSIM (Level 4) Parameters			
name	**l/w**	**parameter**	**units**
`vfb`	*	flat-band voltage	V
`phi`	*	surface inversion potential	V
`k1`	*	body effect coefficient	$V^{1/2}$
`k2`	*	drain/source depletion charge sharing coefficient	-
`eta`	*	zero-bias drain-induced barrier lowering coefficient	-
`muz`		zero-bias mobility	cM^2/VS
`dl`		shortening of channel	μM

dw		narrowing of channel	μM
u0	*	zero-bias transverse-field mobility degradation coefficient	V^{-1}
u1	*	zero-bias velocity saturation coefficient	$\mu M/V$
x2mz	*	sens. of mobility to substrate bias at Vds = 0	cM^2/V^2S
x2e	*	sens. of drain-induced barrier lowering to substrate bias	V^{-1}
x3e	*	sens. of drain-induced barrier lowering to drain bias at Vds = Vdd	V^{-1}
x2u0	*	sens. of transverse field mobility degradation to substrate bias	V^{-2}
x2u1	*	sens. of velocity saturation effect to substrate bias	μMV^{-2}
mus		mobility at zero substrate bias and at Vds = Vdd	cM^2/V^2S
x2ms	*	sens. of mobility to substrate bias at Vds = Vdd	cM^2/V^2S
x3ms	*	sens. of mobility to drain bias at Vds = Vdd	cM^2/V^2S
x3u1	*	sens. of velocity saturation effect on drain bias at Vds = Vdd	μMV^{-2}
tox		gate oxide thickness	μM
temp		temperature at which parameters were measured	C
vdd		measurement bias range	V
cgdo		gate-drain overlap capacitance per channel width	F/M
cgso		gate-source overlap capacitance per channel width	F/M
cgbo		gate-bulk overlap capacitance per channel length	F/M
xpart		gate-oxide capacitance charge model flag	-
n0	*	zero-bias subthreshold slope coefficient	-
nb	*	sens. of subthreshold slope to substrate bias	-
nd	*	sens. of subthreshold slope to drain bias	-
rsh		drain and source diffusion sheet resistance	$\Omega/\square$
js		source drain junction current density	A/M^2
pb		built in potential of source drain junction	V
mj		grading coefficient of source drain junction	-
pbsw		built in potential of source,drain junction sidewall	V
mjsw		grading coefficient of source drain junction sidewall	-
cj		source drain junction capacitance per unit area	F/M^2
cjsw		source drain junction sidewall capacitance per unit length	F/M
wdf		source drain junction default width	M
dell		source drain junction length reduction	M

The parameter `xpart = 0` selects a 40/60 drain/source charge partition in saturation, while `xpart = 1` selects a 0/100 drain/source charge partition.

2.17.12 Josephson Junction Model

Type Name: `jj`

The Josephson junction model is an extended version of the RSJ model as used by Jewett[11].

The parameters marked with an asterisk in the **area** column scale with the `area` parameter given in the device line.

JJ Model Parameters					
name	**area**	**parameter**	**units**	**default**	**example**
`icrit`	*	Junction critical current	A	1.0e-3	1.5e-3
`cap`	*	Junction capacitance	F	1.0e-12	5.0e-13
`rn`	*	Normal state resistance	Ω	1.7	2.0
`r0`	*	Subgap resistance	Ω	30	50
`vg`		Gap voltage	V	3.0e-3	2.8e-3
`delv`		Gap voltage spread	V	1.0e-4	8.0e-5
`rtype`		Quasiparticle branch model	-	1	2
`cct`		Critical current model	-	1	3
`icon`		Critical current first zero	A	1.0e-2	2.5e-2
`icfact`		Ratio of critical to gap currents	-	$\pi/4$	0.7

The `rtype` parameter determines the type of quasiparticle branch modeling employed. Legal values are as follows:

0 The junction is completely unshunted (effectively sets `rn` and `r0` to infinity).
1 Standard piecewise-linear model.
2 Analytic exponentially-derived approximation.
3 Fifth order polynomial expansion model.
4 "Temperature" variation, allow modulation of the gap parameter.

The default is `rtype=1`. Setting `rtype=0` will disable modeling of the quasiparticle current, effectively setting the RSJ shunt resistance to infinity. Conditions with `rtype=1` and `2` are as described by Jewett, however it is not necessarily assumed that the normal resistance projects through the origin for `rtype=1`. The `icfact` parameter can be set to a value lower than the default BCS theoretical value to reflect the behavior of most real junctions. The quasiparticle resistance is approximated with a fifth order polynomial if `rtype=3`, which seems to give good results for the modeling of some NbN junctions (which tend to have gently sloping quasiparticle curves).

`Rtype=4` uses a piecewise-linear quasiparticle characteristic identical to `rtype=1`, however the gap voltage and critical current are now proportional to the absolute value of the control current set with a `control=`*src_name* entry in the device line. This is to facilitate modeling of temperature changes or nonequilibrium effects. For control current of 1 (Amp) or greater, the full gap and critical current are used, otherwise they decrease linearly to zero. If no device control source is specified, the algorithm reverts to `rtype=1`. It is expected that a nonlinear transfer function will be implemented with a controlled source, which will in turn provide the controlling current to the junction in this mode. For example, the controlling current can be translated from a circuit voltage representing temperature with an external nonlinear source. The functional dependence is in general a complicated function, but a reasonable approximation is $1-(T/T_c)^4$. See the examples for an example input file which illustrates `rtype=4`.

It is currently not possible to use other than the piecewise linear model with temperature variation. If `rtype=4`, then legal values for the critical current parameter are `cct=0` (no critical current) and `cct=1`

(fixed critical current). If another value is specified for `cct`, `cct` reverts to 0. Thus, magnetic coupling and quasiparticle injection are not simultaneously available.

In general, the `cct` variable can take on the following values:

0 No critical current.
1 Fixed critical current.
2 Sin(x)/x modulated supercurrent.
3 Symmetric linear reduction modulation.
4 Asymmetric linear reduction modulation.

The `control` parameter should be used with devices using `cct` 2,3, or 4. With `cct=2`, the first zero is equal to the value of the model parameter `icon`. For `cct=3`, the maximum critical current is at control current zero, and it reduces linearly to zero at control current = $\pm icon$. Junctions with `cct=4` have maximum critical current at control current = $-$`icon`, and linear reduction to zero at control current = $+$`icon`. If `cct` is specified as 2, 3, or 4, the area parameter, if given, is set to unity. Otherwise, the model parameters are scaled appropriately by the area before use.

Chapter 3

The *WRspice* User Interface

3.1 Starting *WRspice*

The *WRspice* simulator is invoked by typing

`wrspice` *options ... input_files ...*

All arguments are optional. There are several options which are recognized specifically by *WRspice*. These options are case insensitive — the option letters can be given in upper or lower case. In addition, there are a few additional options recognized by the graphics system.

The command line options are flagged with the '`-`' character, but this can be changed by setting the SPICE_OPTCHAR environment variable. Below, the use of the '`-`' character is assumed for simplicity.

Graphical *WRspice* requires an X server under UNIX. When using X, the DISPLAY variable should already be set, but if one wants to display graphics on a different machine than the one running *WRspice*, DISPLAY should be of the form *machine*:0. For example, if one wants the display to go to the workstation named "`crab`", for the C-shell one would enter "`setenv DISPLAY crab:0`" at the shell prompt, or equivalently for the Bourne shell one would enter "`DISPLAY=crab; export DISPLAY`" or the more compact form "`export DISPLAY=crab`" if supported. Note that this can also be supplied using the `-d` option.

Further arguments are taken to be *WRspice* input files, which are read and saved in memory. If batch mode is requested (`-b` option) then they are run immediately. *WRspice* will accept SPICE2 input files, and output ascii plots, Fourier analyses, and node printouts as specified in `.plot`, `.four`, and `.print` lines. If an `out` parameter is given on a `.width` line, the effect is the same as "`set width = ...`". Since *WRspice* ascii plots do not use multiple ranges, however, if vectors together on a `.plot` card have different ranges they will not provide as much information as they would in SPICE2. The output of *WRspice* is also much less verbose than SPICE2, in that the only data printed is that requested by the above lines.

The following option forms are accepted by *WRspice*. The option letter can be lower or upper case.

`-b`

Run in batch mode. *WRspice* will read the standard input or the specified input file and do the simulation. Note that if the standard input is not a terminal, *WRspice* will default to batch mode,

unless the `-i` option is given. In batch mode, *WRspice* generates output files for operating range and Monte Carlo analysis, otherwise if the `-r` option is used (`-r` *rawfilename*) *WRspice* generates a rawfile, or generates an ascii plot or print on standard output, as per `.plot/.print` lines, if no rawfile was specified.

`-d` [*host*]:*server*[.*screen*]
This option is applicable when running under X windows, and specifies the name of the display to use. The *host* is the hostname of the physical display, *server* specifies the display server number, and *screen* specifies the screen number. Either or both of the *host* and *screen* elements to the display specification can be omitted. If *host* is omitted, the local display is assumed. If *screen* is omitted, screen 0 is assumed (and the period is unnecessary). The colon and (display) *server* are necessary in all cases. This option can also be given as `-display` and `--display`.

`-dnone`
This is a special form of the `-d` option that when given will suppress all use of graphics. This can be desirable when running *WRspice* remotely over a slow terminal connection. This option will also work under Windows, if for some reason it is necessary to run ¡i¿WRspice¡/i¿ in text-only mode.

`-i`
Run in interactive (as opposed to batch) mode. This is useful if the standard input is not a terminal but interactive mode is desired. Command completion is not available unless the standard input is a terminal, however. Interactive mode is the default when the standard input is a terminal.

`-j`
Run in JSPICE3 compatibility mode. This applies when running interactively, and causes suppression of the Tool Control window. Also, error messages are printed in the console window, rather than in a separate pop-up window (the `noerrwin` variable is set on startup).

`-n`
Don't try to source the user's startup file (`.wrspiceinit`) upon startup. Normally *WRspice* tries to find the file in the current directory, and if it is not found then it checks in the user's home directory. In Windows, the "home directory" can be specified by setting the HOME environment variable. The global file `wrspiceinit` in the system startup directory is sourced in any case.

`-o` *outfile*
The argument *outfile* specifies a file to be used for output, rather than the standard output (terminal).

`-p`
Open *WRspice* in a mode which takes input from a UNIX port, used to establish interprocess communications as a slave process.

`-q`
Disable command completion, which saves memory and may run slightly faster. This prevents initial loading of the command completion data structures. If the variable `nocc` is set and unset, command completion will be turned on, however most internal keywords will not be present in the database.

`-r` *rawfile*
Use *rawfile* as the default file into which the results of the simulation are saved with the write command, and for data output in batch mode.

`-s`
Run in server mode. This is like batch mode, except that a temporary rawfile is used and then written to the standard output, preceded by a line with a single '`@`', after the simulation is done.

This mode is used by the SPICE daemon. In server mode, *WRspice* reads input from the standard input, and generates output, in rawfile or margin analysis file format, on the standard output. The `-r` and `-b` options are ignored.

`-t` *termname*
This specifies the name of the terminal, as known in a termcap or terminfo database. The terminal name is only needed in interactive mode when line editing is enabled, and is generally obtained from the TERM environment variable. Occasionally, this option is useful in overriding bad terminal info specifications allowing line editing to work, such as by giving a value of "`vt220`" when running in an `xterm`.

`-x`
This option, if given, will cause *WRspice* to provide its own window for text input, if *WRspice* is in interactive mode and graphics is available. Under the X window system, the "`xterm`" command is used to obtain the text window.

The UNIX/Linux graphical subsystem will accept the following options. It is unlikely that the user will ever need these.

`--name` *appname*
This option specifies the name under which resources for the application should be found. This option is useful in shell aliases to distinguish between invocations of an application, without resorting to creating links to alter the executable file name. This option can also be given as "`-name`".

`--class` *classname*
This option specifies the application class name under which resources for the application should be found.

`--sync`
This option indicates that requests to the X server should be sent synchronously, instead of asynchronously. Since Xlib normally buffers requests to the server, errors do not necessarily get reported immediately after they occur. This option turns off the buffering so that the application can be debugged. It should never be used with a working program. This option can also be given as "`-synchronous`".

`--no-xshm`
If set, the X server will not use shared memory.

3.2 Environment Variables

Environment variables are keyword/value pairs that are made available to an application by the command shell or operating system. The value of an environment variable is a text string, which may be empty. Environment variables can be set by the user to control various defaults in *WRspice*.

3.2.1 Unix/Linux

Environment variables are maintained by the user's command shell. It is often convenient to set environment variables in a shell startup file such as `.cshrc` or `.login` for the C-shell or `.profile` for the

Bourne shell. These files reside in the user's home directory. See the manual page for your shell for more information.

For the C-shell, the command that sets an environment variable is

`setenv` *variable_name* [*value*]

For example,

```
setenv XT_DUMMY "hello world!"
```

Note that if the value contains white space, it should be quoted. Note also that it is not necessary to have a value, in which case the variable acts as a boolean (set or not set).

In the C-shell, one can use `setenv` without arguments, or `printenv`, to list all of the environment variables currently set.

For a modern Bourne-type shell, such as `bash`, the corresponding command is

`export` *variable_name*[=*value*]

In this type of shell one can list the variables currently set by giving the `set` command with no arguments.

3.2.2 Microsoft Windows

Under Windows, environment variables can be set in a DOS box with the "`set`" command before starting the program from the command line, or in the `AUTOEXEC.BAT` file, or from the **System** entry in the **Control Panel**. Only the latter two methods work if the programs are started from an icon. If using a Cygwin bash-box, environment variables can be set in the startup file as under Unix.

3.2.3 *WRspice* Environment Variables

All of the *XicTools* programs respond to the XT_PREFIX environment variable. When the tools are installed in a non-standard location, i.e., other than `/usr/local`, this can be set to the directory prefix which effectively replaces "`/usr/local`", and the programs will be able to access the installation library files without further directives.

There are several environment variables which can be used to alter some of the *WRspice* defaults. On startup, *WRspice* checks for the following variables in the environment, and alters internal defaults accordingly. The defaults can be modified when the program is built, the defaults listed below are those assigned in the distribution.

HOME
: This is used only in the Microsoft Windows version, and can be set to a full directory path which will be taken as the user's home directory.

DISPLAY
: This variable defines the X window system display that *WRspice* will use, but is ignored if the `-d` option is used on the *WRspice* command line. The display must be specified for graphics to be enabled in *WRspice*.

EDITOR
: If defined to the invoking string for a text editor, that editor will be used in the edit command. This is superseded by the SPICE_EDITOR variable, if set.

SPICE_EDITOR
: The text editor called by the edit command can be set with this variable. The variable is defined to the command string one would type to invoke the editor. This will supersede the EDITOR variable, if set, but which would otherwise have the same effect. If no editor is specified in the environment, or with the `editor` shell variable, which supersedes the environment variables, a default internal editor is used. The default internal editor can also be specified by setting SPICE_EDITOR to nothing, "`default`", or "`xeditor`".

TMPDIR
: This specifies a directory to use for temporary files, and is superseded by SPICE_TMP_DIR, if defined. The default location if not specified is `/tmp`.

SPICE_TMP_DIR
: When *WRspice* creates a temporary file, it will look for a directory named by the SPICE_TMP_DIR environment variable, and if not found the directory named in the TMPDIR variable, and if still not found the file will be created in `/tmp`.

SPICE_EXEC_DIR
: This variable can be used to define the directory containing the *XicTools* binaries, used by the aspice command and the `wrspiced` daemon. If not set, the default is "`/usr/local/bin`", or, if `XT_PREFIX` is set, its value replaces "`/usr/local`".

SPICE_PATH
: This can be used to set the full path to the *WRspice* executable, for the aspice command and the `wrspiced` daemon. If not set, the default is "`/usr/local/bin/wrspice`", or, if `XT_PREFIX` is set, its value replaces "`/usr/local`". The SPICE_EXEC_DIR variable can also be used for this purpose, unless the `wrspice` executable has been renamed. The `spicepath` shell variable, if set, will override the path set in the environment.

SPICE_LIB_DIR
: This variable can be used to change the default location where *WRspice* looks for system startup files. If not set, the internal default is "`/usr/local/lib/xictools/wrspice/startup`", or, if `XT_PREFIX` is set, its value replaces "`/usr/local`".

SPICE_INP_PATH
: This can be set to a list of directories to search for input files and scripts. If not set, the internal default is "`( . /usr/local/lib/xictools/wrspice/scripts )`", or, if `XT_PREFIX` is set, its value replaces "`/usr/local`".

SPICE_HLP_PATH
: This can be set to a list of directories to search for help database files. If not set, the internal default is "`( /usr/local/lib/xictools/wrspice/help )`", or, if `XT_PREFIX` is set, its value replaces "`/usr/local`". This is superseded by the `helppath` shell variable, if set.

SPICE_NEWS_FILE
: This variable can be set to the full path to a text file which is printed when *WRspice* starts. If not set, the file `/usr/local/lib/xictools/wrspice/startup/news` will be printed, if it exists (if `XT_PREFIX` is set, its value replaces "`/usr/local`").

SPICE_BUGADDR
: This variable can be set to an Internet mail address to use in the bug reporting command. If not set, the built in default is the Whiteley Research technical support address.

SPICE_OPTCHAR
: This variable can be defined to a character that will be used to flag options on the *WRspice* command line. If not defined, the option character is "−".

SPICE_ASCIIRAWFILE
: If this variable is defined to "0" (zero), *WRspice* will create binary plot-data files (rawfiles). If not set or set to something else, *WRspice* will create the default ascii-format rawfiles. The `filetype` shell variable can also be used to set the mode, which will supersede the environment variable. The rawfiles are normally created with the write command.

SPICE_HOST
: This variable can be used to set the host name to use for remote SPICE runs. The host name can optionally be suffixed by a colon followed by the port number to use for communication with the `wrspiced` daemon. If not given, the port is obtained from the operating system for "spice/tcp", or 3004 if this is not defined. There is no default for this variable. Hosts can also be specified with the rhost command, and given with the `rhost` shell variable.

SPICE_DAEMONLOG
: This variable is used by the `wrspiced` daemon program to set an alternate path for the log file. The default path is `/tmp/wrspiced.log`.

3.3 Initialization Files

WRspice will attempt to source startup files used for initialization when the program is started. First, a file named "`wrspiceinit`" is searched for in the system startup directory, and if found is sourced into *WRspice*. The system startup directory has a default location built into the program (`/usr/local/lib/xictools/wrspice/startup`, or if XT_PREFIX is set in the environment, its value replaces "`/usr/local`"), but this can be changed by setting the SPICE_LIB_DIR environment variable to another location.

A file named "`.wrspiceinit`" is searched for in the current directory, and the user's home directory. The first instance found is sourced. If running on Microsoft Windows which does not support the notion of a home directory, *WRspice* will look in the environment for a variable named "HOME", and its value will be taken as the path to the "home directory". If not set, the search is skipped.

These files have identical format, and contain ordinary script commands, which can be used to set the default behavior of *WRspice*. The first line is ignored, but all remaining lines are taken as script commands. The special directive tbsetup, which can only appear in these files, provides the setup information for the graphical interface. Unlike ordinary input files, it is not necessary to enclose the commands in `.control` or `.exec` blocks in the startup files.

When using the X-window system, the X resource-passing mechanism can be used to set the default colors used in plots. The resource mechanism is otherwise ignored in the current version of *WRspice*. The base names for the color resources are "`color0`" through "`color19`", with the corresponding class names capitalized. Thus, one way to define alternative plotting colors is to create a file named "`Wrspice`" in the user's home directory, which contains lines like

```
*color2:  red
```

for each new color definition. The color name should be known to the X window system, i.e., be listed in the `rgb.txt` file in the X-windows system library.

The same definitions could be placed in a `.wrspiceinit` file with lines like "`set color2 = red`".

3.4 The Tool Control Window

When the DISPLAY variable is found in the environment upon program startup, *WRspice* assumes that a graphical (X-window) server is available, and will enable its graphical components. If this initialization fails, *WRspice* will terminate. If the DISPLAY variable is not set, and the `-d` option is not used on the command line to specify the display, then *WRspice* will run in text-only mode. In this mode, the core functionality is available, but not the graphical niceties such as plotting (other than the infamous crude line-printer plots of yore). Under Microsoft Windows, graphics is (of course) always available.

When a graphical interface is available, *WRspice* by default provides a small "Tool Control" window which provides menus for controlling *WRspice*, and a display containing a tabulation of memory statistics. The menus contain buttons which bring up graphical screens, from which much most of *WRspice* can be controlled in a (perhaps) more user-friendly fashion. The locations of the pop-ups and their active/inactive status at program startup can be preset by the user.

When running *WRspice* through the *Xic* program, by default the Tool Control window will appear when connection to *WRspice* has been established, but only when *WRspice* is running on the local machine. The *Xic* variable NoSpiceTools can be set, before the connection is established, to prevent the Tool Control window from appearing.

The text area lists the following quantities, though some may not be listed if the operating system does not provide this information. The listing is updated every few seconds. The first line of text shows the current running status of *WRspice*: idle, running, or stopped. Also shown on this line is the elapsed wall-clock time from the last status change. The time format is hours:minutes:seconds. The seconds entry has two decimal places (resolution is .01 second). The hours field is not included if zero. The **user** and **system** lines are similar, but display the total user cpu time used by the process, and the total system cpu time, respectively. The user time generally includes input/output processing time, where the system time is pure cpu usage. The **data size** entry displays the total storage in Kb used by the program. This size is limited to the **program limit**, which is given in the following line. This limit can be changed with the `maxdata` variable. The final **system limit** line displays the maximum memory available from the system for the process.

WRspice supports the xdnd and Motif drag and drop protocols. One is able to drag files from many file manager programs into the Tool Control window of *WRspice* or the main window of *Xic*, and that file will be loaded into the program. The File Selection and Files Listing pop-ups participate in the protocols as sources and receivers. The text editor and mail pop-ups are drag receivers.

The file must be a standard file on the same machine. If it is from a tar file, or on a different machine, first drag it to the desktop or to a directory, then into *WRspice*. (Note: The GNOME `gmc` file manager allows one to view the contents of tar files, etc. as a "virtual file system". Window Maker and Enlightenment window managers, at least, are drag/drop aware.)

In the upper left of the Tool Control window is the **WR** button, which contains the Whiteley Research corporate logo. Pressing this button brings up a mail client (see 3.5), pre-loaded with the the address of the Whiteley Research support staff. The text field containing the address, as well as the subject, can be changed. This can be used to send questions and bug reports to Whiteley Research, or to send messages or data files to colleagues.

To the right of the **WR** button are the **Run** (green arrow) and **Stop** (red X) buttons. Pressing the **Run** button is equivalent to giving the run command on the command line, without arguments. Pressing the **Stop** button issues an interrupt signal that will pause a simulation in progress, the same

as if the user typed **Ctrl-C** in the console window.

To the right of the buttons is a menu bar with four entries: **File**, **Edit**, **Tools**, and **Help**. Pressing the mouse button 1 on these entries brings up a drop-down menu containing various commands.

The **File** menu contains commands for manipulating disk files.

Open
Bring up a file manager panel. This panel allows files and directories to be created, deleted, and renamed, or read into *WRspice*. The file manager is described in 3.6.

Source
This button creates a dialog, soliciting a file to input into *WRspice*, as with the source command. When the dialog is active, the Tool Control text window is active as a drag receiver, so that file names can be dragged and dropped from compatible windows (such as the file manager), and the file name will be loaded into the dialog.

Load
This button is similar in operation to the **Source** button, however the file is expected to be a rawfile, which is the native plot-file format. Like the **Source** dialog, the **Load** dialog supports drag and drop protocols through the text area of the Tool Control window. The file is read and data are made available for analysis, similar to the load command.

Update
The **Update** button will save the current tool configuration, and the next time *WRspice* is started, the same tools will be available, in the same locations on-screen. The tools are available in the **Tools** menu to be described.

Initialization of the graphical interface is directed from the system init file `wrspiceinit` found in the startup directory, or more appropriately from the user's startup file `.wrspiceinit` as found in the current or user's home directory . A special command `tbsetup`, which is only recognized in these files, performs the initialization. The `tbsetup` function takes a long, messy command line, however fortunately the user has an easy way to automatically add this line, using the the **Update** command. This action is also available from the command line update command.

Quit
The **Quit** button will terminate the *WRspice* session, after confirmation if there is unsaved work. This can also be accomplished with the quit or exit command line functions.

The **Edit** menu contains commands used to modify input files.

Text Editor
This button brings up a text editor, similar to the edit command. The default text editor is described in 3.7, but the user can specify a different editor with the EDITOR or SPICE_EDITOR environment variables, or with the `editor` variable.

Xic
This button launches the *Xic* program, if it is available.

For schematic capture, the *Xic* program must be installed. When either editor is started, the currently loaded circuit, if any, and if it has graphical information in the case of *Xic*, is loaded into the editor. The simulation process can be initiated and controlled through *Xic*. The internal text editor has a **Source** button, which allows the modified circuit to be passed directly back to *WRspice* for simulation. If another editor is used, changes will have to be saved to disk and sourced from *WRspice*.

The **Tools** menu contains a configurable collection of command buttons which initiate pop-ups which control or display various aspects or *WRspice*. Each of these pop-ups is a graphical short cut to a collection of command line commands. Many users prefer the point-and-click interface to the command line, though some do not. With *WRspice*, the user has a choice. The command functions available in the **Tools** menu are listed below.

Circuits
Bring up a panel (see 3.4.5) listing the circuits that are currently in memory. The panel provides capabilities for choosing the current circuit and deleting circuits from memory.

Colors
Bring up a panel (see 3.4.4) to set the colors used in the plots.

Commands
Bring up a panel (see 3.4.4) to customize the variables associated with the various commands.

Debug
Bring up a panel (see 3.4.4) used to control the debugging modes of *WRspice*.

Files
Pop up a panel (see 3.4.6) which displays a list of the files found along the `sourcepath`. Files can be sourced or edited through this panel.

Fonts
This brings up a panel (see 3.4.7) that allows selection of the font used in the text areas of the various tools, and in plot windows used for displaying simulation results.

Plot Defs
Bring up a panel (see 3.4.4) to control the myriad of variables associated with plotting.

Plots
Bring up a panel (see 3.4.8) which displays a list of the plots in memory. The panel provides capabilities for selecting the current plot, and for deleting plots.

Shell
Bring up a panel (see 3.4.4) used to control the default settings associated with the *WRspice* shell.

Sim Defs
Pop up a panel (see 3.4.4) used to modify the variables which affect circuit simulation.

Trace
Pop up a panel (see 3.4.9) which lists the "debugs" currently in effect. The "debugs" are directives to interactively plot, or trace variables during simulation, or to pause the simulation when a condition is met. The panel provides capability for deactivating or deleting "debugs".

Variables
Pop up a panel (see 3.4.10) displaying the shell variables currently set.

Vectors
Pop up a panel (see 3.4.11) listing the vectors in the current plot. The panel provides an interface for plotting or printing these vectors.

The **Help** menu provides entry into the help system, and provides access to other information. The buttons found in this menu are listed below.

Help
Bring up the help viewer loaded with the top-level page. The help system provides an extensive cross-referenced HTML database, and contains the latest information on *WRspice* features. The help system is described in 3.11.

About Bring up a text window with the current *WRspice* version number, copyright, and legal disclaimer.

Notes

Bring up a text browser loaded with the release notes for the current release of *WRspice*. The release notes provide detailed information about changes in the present release, and serve as a supplement to the manual. Changes and new features should also have been incorporated into the help database.

3.4.1 The tbsetup Command

The **Tools** menu in the Tool Control window is initialized in the user's `.wrspiceinit` file (if any) by a `tbsetup` command. This command can appear only in startup files. The command is placed in the file automatically, and a `.wrspiceinit` file created if necessary, by invoking the `update` command from the prompt line or pressing the **Update** button in the **File** menu of the Tool Control window. The file can be hand-edited if needed.

If there is no initialization found, the configuration is set to an internal default (all tools invisible at startup).

The command string takes the following form:

`tbsetup [old] [vert] [toolbar on|off` *x y*`] [`*name1* `on|off` *x y*`] [`*name2* `...`

The `old` and `vert` keywords are ignored if given. They exist for backward compatibility.

For each block, the first token gives the tool name, with the "`toolbar`" entry indicating the Tool Control window itself. For other than the Tool Control window, which is always shown, the `on` and `off` keywords specify whether the tools are shown at startup. The two numbers that follow give the position of the upper left corner of the tool on the screen (the screen origin is the upper left corner, coordinates are in pixels). For other than the first (Tool Control window) entry, the blocks can be rearranged or deleted. The **Tools** menu will show only the tools listed, in the order given. Thus, the **Tools** menu can be customized.

Here is how to generate an initial configuration. Simply start *WRspice* on a system that supports graphics, and the Tool Control window will appear somewhere on-screen. After you have opened some of the tools from the **Tools** menu and arranged them to your preference and/or moved the Tool Control window, press the **Update** button in the **File** menu. A `.wrspiceinit` file will be created, or an existing file updated, Alternatively, the `update` command can also be entered from the command line. Hand editing of the `.wrspiceinit` file may be used to remove buttons or change the button order in the **Tools** menu. Once edited, only the tools present will be updated.

3.4.2 The update Command

This command will update the user's `.wrspiceinit` file to reflect the current tool setup. The window arrangement should be the same the next time the user starts *WRspice*. This command is also performed when the user presses the **Update** button in the **File** menu of the Tool Control window.

3.4.3 The setfont Command

Syntax: `setfont` *font_num font_specifier*

This command can be used to set the fonts employed in the graphical interface. Although this can be given at a prompt, it is intended to be invoked in a startup script.

The first argument is an integer 1–6 (1–4 on Windows) which designates the font category. The index corresponds to the entries in the drop-down menu of font categories found in the Font Selection panel.

The rest of the line is a font description string. This varies between graphics types.

Unix/Linux
: For gtk1 releases, the name is the X Logical Font Descriptor name for a font available on the user's system, or an alias. For gtk2 releases, the name is a Pango font description name. There is a very modest attempt to interpret a specification of the wrong type.

Windows The name is in one or the following formats:

New standard (*WRspice* release 2.3.58 and later)
face_name pixel_height

Example: `Lucida Console 12`

Old standard (deprecated)
(*pixel_height*)*face_name*

Example: `(12)Lucida Console`

The *face_name* is the name of a font family installed on the system, and the *pixel_height* is the on-screen size.

You will probably never need to use the setfont command directly. All settable fonts are saved in the `.wrspiceinit` startup file when the **Update** menu command in the **File** menu is given, or the `update` command is invoked.

3.4.4 The Variable Setting Tools

The **Plot Defs**, **Colors**, **Shell**, **Sim Defs**, **Commands**, and **Debug** buttons in the **Tools** menu of the Tool Control window bring up panels which control the values of internal variables. These variables can also be set with the set command, though the panels may provide a more convenient interface.

The panels each contain a **Help** button. Pressing the **Help** button pops up a window containing a listing of the variables which can be set with the panel. Clicking on a word in the listing will bring up the help viewer with a description of that variable.

Some of the panels are organized into multiple pages, with each page containing the variables in a particular category. For example, in the **Commands** pop-up, the variables are paged according to the command they modify.

Each variable has its own "box" in the panel. This box contains a **Set** button, and optionally a **Def** button and a text input area. The text input area can take several forms, depending on the type of variable: string, integer, or real. Boolean variables have only the **Set** button. Text can be entered into the area, or in some cases the up/down arrows to the right of the text area can be clicked to adjust the text.

When the **Set** button is active, the variable is set to the value shown in the text area (if any), and the text area is frozen, i.e., can't be edited. The text area can be changed only with the **Set** button inactive, in which case the value in the text area is arbitrary. The value is only known to *WRspice* when the **Set** button is active, in which case the variable should appear in the listing brought up by the **Variables** button in the **Tools** menu in the Tool Control window.

The **Def** button will enter the default value for the variable into the text area. This button becomes active if the text area is modified.

3.4.5 The Circuits Tool

This panel is available from the **Circuits** button in the **Tools** menu of the Tool Control window. It provides a listing of the circuits currently in memory. A circuit description can be read from a file with the source command. If the user clicks on the text of one of the listed circuits, that circuit becomes the current circuit, i.e., the circuit which will simulate with the run command. The panel contains the following buttons:

Delete Current
Delete the current circuit from memory.

Help
Bring up help on this panel.

Dismiss
Remove the circuits panel from the screen.

3.4.6 The Files Tool

This panel is available from the **Files** button in the **Tools** menu of the Tool Control window. It provides a listing of files found in each directory of the "`sourcepath`" search path. The `sourcepath` is a list of directories that are searched for circuit description files, if a file name is given to *WRspice* without a path prefix.

Under Unix/Linux, The main window displays a "notebook" listing of the symbol files found along the search path. Each directory in the search path has a separate page in the notebook listing. Clicking on the notebook tabs will bring the pages into view.

Under Windows, the pop-up contains a drop-down menu which has an entry for each directory in the search path. The main text area lists the files found in the currently selected directory.

A file from the list can be selected by clicking with mouse button 1 on the text. The text can be deselected by clicking in the text window away from any text. The file listing participates in the drag/drop protocol as a drag source and drop receiver.

The following buttons are provided:

Edit
If a file is selected, bring up a text editor with the file loaded as with the edit command. If no file is selected, the button becomes active, and the user can click on a file in the text window to edit it.

Source
If a file is selected, source it as with the source command. If no file is selected, the button becomes active, and the user can click on a file in the text window to source it.

Help
Bring up help on this panel.

Dismiss
Remove the files panel from the screen.

3.4.7 The Fonts Tool

This panel allows selection of the fonts used in the graphical interface. A drop-down menu provides selection of the various font targets. Pressing the **Apply** button will immediately apply the selected font to all visible windows which use the font.

The drop-down font targets list contains the following entries:

Fixed Pitch Text Window Font
This sets the font used in pop-up multi-line text windows, such as the Files Listing and others, where the names are formatted into columns.

Proportional Text Window Font
This sets the font used in pop-up multi-line text windows where text is not formatted, such as the error message pop-up.

Fixed Pitch Drawing Window Font
This is the font used in the plot windows and in the Tool Control window.

Text Editor Font
This is the font used in the Text Editor pop-up.

HTML Viewer Proportional Font (Unix/Linux only)
This is the base font used for proportional text in the HTML viewer (help windows). If set, this will override the font set in the `.mozyrc` file, if any.

HTML Viewer Fixed Pitch Font (Unix/Linux only)
This is the base fixed-pitch font used by the HTML viewer. If set, this will override the font set in the `.mozyrc` file, if any.

The **Font** button in the **Options** menu of the Text Editor brings up a similar panel, as does the **Font** button in ther **Options** menu of the help viewer.

These fonts can be set in the `.wrspiceinit` startup file by giving `setfont` commands. These are inserted automatically when the **Update** button in the **File** menu is pressed, or an `update` command is given.

3.4.8 The Plots Tool

This panel is available from the **Plots** button in the **Tools** menu of the Tool Control window. It provides a listing of the plots currently in memory. A plot is a collection of output data vectors generated during a simulation run. In addition, there is always the `constants` plot, which contains some useful physical constants. A plot from the list can be selected as the current plot by clicking on the text in the list. The current plot is used to resolve vector names in expressions given to commands and elsewhere. The panel contains the following buttons:

New Plot
This creates an empty plot structure, and makes it the current plot.

Delete Current
Delete the current plot from memory.

Help
Bring up help on this panel.

Dismiss
Remove the plots panel from the screen.

3.4.9 The Trace Tool

This panel is available from the **Trace** button in the **Tools** menu of the Tool Control window. It provides a listing of the "debugs" (i.e., traces, breakpoints, interactive plots) currently in effect. These "debugs" are set with the save, stop, trace, and iplot commands. A debug can be made inactive by clicking on the text in the list, in which case an 'I' appears in the first column. Click on the text a second time to restore activation. Inactive debugs are ignored during simulation. The following buttons are available:

Help
Bring up help on this panel.

Delete Inactive
Delete the debugs selected as inactive, i.e., those listed with "I" in the first column.

Dismiss
Remove the trace panel from the screen.

3.4.10 The Variables Tool

This panel is available from the **Variables** button in the **Tools** menu of the Tool Control window. It provides a listing of the variables currently set in the shell, either with the set command or by other means. The format of the listing is the same as that used by the set command without arguments. The following buttons are available:

Help
Bring up help on this panel.

Dismiss
Remove the variables panel from the screen.

3.4.11 The Vectors Tool

This panel is available from the **Vectors** button in the **Tools** menu of the Tool Control window. It provides a listing of the vectors in the current plot. The current plot can be selected with the **Plots** button in the **Tools** menu. A vector can be selected by clicking on the text, and selection is indicated by a '>' symbol in the first column. Any number of vectors can be selected. Click on the vector entry a second time to deselect it. The selected vectors are used by the buttons described below.

Help
Bring up help on this panel.

Print
Print the values of the selected vectors on-screen.

Plot
Plot the values of the selected vectors.

Delete
Delete the selected vectors from the plot.

Dismiss
Remove the vectors panel from the screen.

3.5 The Mail Client

The mail client can be used to send mail to arbitrary mail addresses, though when the panel appears, it is pre-loaded with the address of Whiteley Research technical support. The text field containing the address, as well as the subject, can be changed.

The main text window is a text editor with operations similar to the text editor used elsewhere in *Xic* and *WRspice*. The **File** menu contains commands to read another text file into the editor at the location of the cursor (**Read**), save the text to a file (**Save As**) and send the text to a printer (**Print**). When done, the **Send Mail** command in the **File** menu is invoked to actually send the message. Alternatively, one can quit the mail client without sending mail by pressing **Quit**.

The **Edit** menu contains commands to cut, copy, and paste text.

The **Options** menu contains a **Search** command to find a text string in the text. The **Attach** command is used to add a mime attachment to the message. Pressing this button will cause prompting for the name of a file to attach. While the prompt pop-up is visible, dragging a file into the mail client will load that file name into the pop-up. This is also true of the **Read** command. Attachments are shown as icons arrayed along the tool bar of the mail client. Pressing the mouse button over an attachment icon will allow the attachment to be removed.

In the Windows version, since Windows does not provide a reliable interface for Internet mail, the mail client and crash-dump report may not work. Mail is sent by passing the message to a Windows interface called "MAPI", which in turn relies on another installed program to actually send the mail. The mail system is known to work if Outlook Express is installed and configured as follows. Outlook Express is installed automatically with Internet Explorer.

In order for mail to work from *XicTools*, Outlook Express should be the "Simple MAPI mail client". To configure this, from Outlook Express select **Tools/Options/General** from the menu. Then, check the box next to "Make Outlook Express my default Simple MAPI client".

3.6 The File Manager

The File Selection pop-up allows the user the navigate the host's file systems, and select a file for input to the program.

The panel provides two windows; the left window displays the subdirectories in a tree format, and the right window displays a listing of files in a columnar form. The panel is similar in operation to the Windows Explorer tool provided by Microsoft.

When the panel first appears, the directories listing contains a single entry, which is shown selected, and the files window contains a list of files found in that directory. The tree "root" is selected by the application, and may or may not be the current directory. If the directory contains subdirectories, a

small box containing a '+' symbol will appear next to the directory entry. Clicking on the '+' will cause the subdirectories to be displayed in the directory listing, and the '+' will change to a '−'. Clicking again on the '−' will hide the subdirectory entries. Clicking on a subdirectory name will select that subdirectory, and list its files in the files listing window.

Clicking on the blue triangle in the menu bar will push the current tree root to its parent directory. If the tree root is pushed to the top level directory, the blue triangle is grayed. The label at the bottom of the panel displays the current root of the tree. There is also a **New Root** item in the **File** menu, which allows the user to enter a new root directory for the tree listing. In Windows, this must be used to list files on a drive other than the current drive.

The **Up** menu is similar, but it produces a drop-down list of parent directories. Selecting one of the parents will set the root to that parent, the same as pressing the blue triangle button multiple times to climb the directory tree.

The **New CWD** button in the **File** menu allows the user to enter a new current working directory for the program. This will also reset the root to the new current working directory. The small dialog window which receives the input, and also a similar dialog window associated with the **New Root** button, are sensitive as drop receivers for files. In particular, one can drag a directory from the tree listing and drop it on the dialog, and the text of the dialog will be set to the full path to the directory.

In the Unix/Linux version, the '+' box will appear with subdirectories only after the subdirectory is selected. In the Windows version, the the '+' boxes will be visible without selection.

The files listed in the files listing always correspond to the currently selected directory. File names can be selected in the files listing window, and once selected, the files can be transferred to the calling application. The **Go** button, which has a green octagon icon, accomplishes this, as does the **Open** entry in the **File** menu. These buttons are only active when a file is selected. One can also double-click the file name which will send the file to the application, whether or not the name was selected.

Selected files can be dragged and dropped into the application, as an alternative to the **Go** button. Files and directories can also be dragged/dropped between multiple instances of the File Selection pop-up, or to other file manager programs, or to other directories within the same file manager pop-up. The currently selected directory is the target for files dropped in the files listing window. When dragging in the directory listing, the underlying directory is highlighted. The highlighted directory will be the drop target.

The drag/drop operation is affected by which mouse button is used for dragging, and by pressing the **Shift** and **Ctrl** buttons during the drag. The actions are slightly different for Unix and Windows systems. In Unix/Linux, the normal operation (button 1 with no keys pressed) for drag/drop is copying. The other options are as follows:

Unix Operations	
Button 1	Copy
Shift-Button 1	Move
Control-Button 1	Copy
Shift-Control-Button 1	Link
Button 2/3	Ask

Above, "Ask" means that a dialog will appear asking the user what operation to perform. Options are move, copy, or (symbolically) link. Both the source and destinations are shown in the pop-up, and can be modified.

If a directory is the source for a copy, the directory and all files and subdirectories are copied recursively, as with the "`-R`" option of the Unix `cp` command.

Under Windows, only button 1 can be used for dragging, and there is no "Ask" option. The basic operation (no keys pressed) is move, however this can be reset by the underlying window. In Windows, directories can only be moved, not copied.

Windows Operations	
Button 1	Copy
Shift-Button 1	Move
Control-Button 1	Copy

Only one file or directory can be selected. When the operation is Copy, the cursor icon contains a '+' in all cases, which will change when the **Shift** key is pressed (if a Move operation is permitted) indicating the change to a Move operation.

The **File** menu contains a number of commands which provide additional manipulations. The **New Folder** button will create a subdirectory in the currently selected directory (after prompting for a name). The **Delete** button will delete the currently selected file. If no file is selected, and the currently selected directory has no files or subdirectories, it will be deleted. The **Rename** command allows the name of the currently selected file to be changed. If no file is selected, the name change applies to the currently selected directory.

While the panel is active, a monitor is applied to the listed files and directories which will automatically update the display if the directories change. The listings should respond to external file or directory additions or deletions within half a second.

3.7 The Text Editor

WRspice provides a general-purpose text editor window. It is used for editing text files or blocks, and may be invoked in read-only mode for use as a file viewer. In that mode, commands which modify the text are not available.

The following commands are found in the **File** menu of the editor. Not all of these commands may be available, for example the **Open** button is absent when editing text blocks.

Open
Bring up the **File Selection** panel. This may be used to select a file to load into the editor. This is the same file manager available from the **Open** button in the **File** menu of the **Tool Control Window**.

Load
Bring up a dialog which solicits the name of a file to edit. If the current document is modified and not saved, a warning will be issued, and the file will not be loaded. Pressing **Load** a second time will load the new file, discarding the current document. The editor is sensitive as a drop receiver. If a file is dragged into the editor and dropped, and neither of the **Load** or **Read** dialogs is visible, the **Load** dialog will appear with the name of the dropped file preloaded into the dialog text area. If the drop occurs with the **Load** dialog visible, the dropped file name will be entered into the **Load** dialog.

Read
Bring up a dialog which solicits the name of a file whose text is to be inserted into the document at the cursor position. When this pop-up is visible, and a file is dragged into the editor and dropped, the name of that file is entered into the dialog text area.

Save
Save the document to disk, or back to the application if editing a text block under the control of some command.

Save As
Pop up a dialog which solicits a new file name to save the current document under. If there is selected text, the selected text will be saved, not the entire document.

Print
Bring up a pop-up which enables the document to be printed to a printer, or saved to a file.

Write DOS
This menu item appears only in the Windows versions. It controls the line termination format used in files written by the text editor. The default is to use the archaic DOS two-byte termination. If this button is unset, the more modern and efficient UNIX-style termination is used. Older Windows programs such as Notepad require DOS termination. Most newer objects and programs can use either format, as can the *XicTools* programs.

Quit
Exit the editor. If the document is modified and not saved, a warning is issued, and the editor is not exited. Pressing **Quit** again will exit the editor without saving.

The editor can also be dismissed with the window manager "dismiss window" function, which may be an '**X**' button in the title bar. This has the same effect as the **Quit** button.

The following commands are found in the **Edit** menu of the text editor.

Cut
Delete selected text to an internal buffer (the "clipboard").

Copy
Copy selected text to the clipboard.

Paste
Paste the contents of the clipboard into the document at the cursor location.

The following commands are found in the **Options** menu of the editor.

Search
Pop up a dialog which solicits a regular expression to search for in the document. The up and down arrow buttons will perform the search, in the direction of the arrows. If the **No Case** button is active, case will be ignored in the search. The next matching text in the document will be highlighted. If there is no match, "not found" will be displayed in the message area of the pop-up.

Source
Read the content of the editor into *WRspice* as through the source command. One can also save the file to disk, and use the source command directly.

Font
This brings up a tool for selecting the font to use in the text window. Selecting a font will change the present font, and will set the default font for new text editor class windows. This includes the file browser and mail client pop-ups.

Unless a font is set with the **Font** button, text editor windows will use a font defined in the XEDITOR_FONT environment variable, if set. Under Unix/Linux, this variable can be set to the

name of an X font. Under Windows, the variable is set to a string in the form "(*size*)*name*", i.e., the font size in parentheses followed by the font name, for example the default font is "`(10)Lucida Console`".

Characters are entered into the document as typed, at the current cursor location. The keys with special bindings are listed below.

Unix/Linux Bindings

Ctrl-A	Move cursor to beginning of line
Ctrl-B	Move cursor backward one character
Ctrl-C	Copy selected text to clipboard
Ctrl-D	Delete next character
Ctrl-E	Move cursor to end of line
Ctrl-F	Move cursor forward one character
Ctrl-H	Delete previous character
Ctrl-N	Move cursor down one line
Ctrl-P	Move cursor up one line
Ctrl-U	Delete current line
Ctrl-V	Paste clipboard at cursor
Ctrl-W	Delete backward one word
Ctrl-X	Cut selection to clipboard
Alt-B	Move cursor backward one word
Alt-D	Delete word
Alt-F	Move cursor forward one word
Home	Move cursor to beginning of line
Ctrl-Home	Move cursor to top of document
End	Move cursor to end of line
Ctrl-End	Move cursor to end of document
PageUp	Move up one page
PageDown	Move down one page
Up	Move cursor up one line
Down	Move cursor down one line
Left	Move cursor left one character
Ctrl-Left	Move cursor left one word
Right	Move cursor right one character
Ctrl-Right	Move cursor right one word
Backspace	Delete previous character
Ctrl-Backspace	Delete previous word
Clear	Delete current line
Shift-Insert	Paste clipboard at cursor
Ctrl-Insert	Copy selected text to clipboard
Delete	Delete next character
Shift-Delete	Cut selected text to clipboard
Ctrl-Delete	Delete next word

Clicking with button 1 will move the cursor to that location. Double clicking will select the clicked-on word. Triple clicking will select the clicked-on line. Button 1 is also used to select text by dragging the pointer over the text to select.

Clicking with button 2 will paste the contents of the clipboard (or any selected text) into the document at the click location.

Button 3 will also select text, but the selected text is not copied to the clipboard.

Microsoft Windows Bindings

Ctrl-A	Select all text
Ctrl-C	Copy selected text to the clipboard
Ctrl-V	Paste the clipboard contents at the cursor
Ctrl-X	Cut selected text to clipboard
Backspace	Delete selected text, or the character to the left of the cursor if no text is selected
Ctrl-Backspace	Delete the word to the left of the cursor
Delete	Delete selected text, or the character to the right of the cursor if no text is selected
Shift-Delete	Performs a "cut", i.e., copies to the clipboard before deleting
Ctrl-Delete	Delete the word to the right of the cursor
Insert	Toggle insert/overwrite mode for typed characters
Shift-Insert	Performs a "paste" from the clipboard
Home	Move cursor to start of line
Ctrl-Home	Move cursor to the top of the document
End	Move to end of line
Ctrl-End	Move cursor to the end of the document
Page Up	Scroll up one page
Ctrl-Page Up	Move cursor to start of first visible line
Page Down	Scroll down one page
Ctrl-Page Down	Move cursor to end of last visible line

The arrow keys move the cursor, and with **Ctrl** pressed the left and right arrow keys move the cursor word-by-word. Holding the **Shift** key while moving the cursor with the mouse or with other keys will select the intervening text. The left mouse button is used to move the cursor by clicking, and to select text by dragging.

3.8 The Plot Panel

This panel displays and controls aspects of plots generated from a simulation with the plot command. The plot window contains a row of buttons on the right side. The button presence is determined by the nature of the data plotted, as not all data support all features. The buttons that may be present are listed below.

Dismiss
Remove the plot from the screen.

Help
Bring up the help viewer with pertinent information.

Redraw
Redraw the graph. This would be necessary to see new colors if the colors are changed, with the **Colors** pop-up from the **Tools** menu of the Tool Control window.

Print
Bring up the printer control panel which controls hardcopy generation. The resulting hardcopy data from the plot can be sent to a printer or saved in a file.

Save
Save the plot to a file containing the plot data. The user is prompted for a name for the file. If the name is given an extension that starts with "`raw`", the plot data will be saved in rawfile format. Otherwise, the format will be as generated by the `print` command.

Points
If this button is active, the data points are marked with a glyph. This is mutually exclusive with the **Comb** button.

Comb
If this button, which is mutually exclusive with the **Points** button, is active, the data will be presented as a histogram. If neither of **Points** or **Comb** is selected, a (possibly interpolated) line drawing connecting the points will be presented.

Log X
If the data are consistent with a logarithmic scale of the x-axis, this button will appear. When active, a log scale will be used on the x-axis.

Log Y
If the data are consistent with a logarithmic scale of the y-axis, this button will appear. When active, a log scale will be used on the y-axis. Note that all traces must be consistent with a log scale, and all traces will use a log scale if this button is active.

Marker
When active, a marker will be attached to the cursor, and the scale factors of the plot will be replaced with the current position of the marker relative to the data. This is useful for actually obtaining numerical data from the plot. If button 1 is clicked, a reference mark will be left behind, and the readout will be relative to the values at the reference. Clicking with button 2 will remove the reference.

When using the marker in a polar or Smith plot, the display indicates the real, imaginary, radius, and angle in degrees of the current marker position. The radius and angle are shown in the lower left corner of the plot window. In Smith plots, a family of real and imaginary values are shown, corresponding to a set of values usually displayed along the x axis. For the imaginary contours, the values correspond to the values printed in left to right order. If mouse button three is used to zoom into a section of the Smith plot such that the x axis is invisible, the values corresponding to the displayed real contours are listed along the top of the plot window. These numbers correspond to the displayed real contours in left to right order. They also correspond to the imaginary contours, however depending on the location in the Smith chart, not all imaginary contours, or additional contours, may be visible. Ambiguity can be resolved through use of the marker.

Separate
When active, each trace will be assigned a portion of the overall vertical plot area, and a separate scale. The traces will be scaled so as to not overlap. Otherwise, the entire vertical area is used for each trace. The button will appear if there is more than one trace in the plot.

Single, **Group**

In the most general case, two buttons in a "radio group" control the y-axis scales of the traces. These buttons are labeled **Single** and **Group**. If neither button is active, each trace will have an independent "best fit" y scale. If **Single** is active, all traces will be plotted on the same "best fit" y scale. If **Group** is active, the traces are scaled according to their data type. The types are voltage, current, and other. Each group will have a separate "best fit" scale. If the trace is from a node voltage, then it will have type voltage, however functions of voltages will probably have type other. This is similarly true for currents.

The **Single** and **Group** buttons are prevented from being active at the same time. If there is only one trace, or the traces are all of the same type, the **Group** button will not appear. If there is only one trace, then the **Single** button will also not appear.

If button 3 is pressed and held while pointing at the graph, an outline box is shown, which follows the cursor, anchored at the location pointed to. Releasing button 3 will create a new plot of the area in the box. Pressing **Ctrl**-button 1 is equivalent to button 3 for this operation.

Text that appears in plots will use a font that can be changed most easily from the font selection panel obtained from the **Fonts** button in the **Tools** menu of the Tool Control window. This font can also be changed with the setfont command.

Any text typed while the pointer is in the plot window will appear on the plot (and hardcopies). This is useful for annotation. A mechanism exists for editing this text, and other text, such as the plot title, which appears in the plot window. When the user presses a character key, the character appears in the plot window at the current location of the pointer. Additional characters will be added after the first, even if the pointer moves (but remains in the plot window) until **Enter** is pressed. This terminates the current entry, allowing the user to enter a new string at a different location.

The **Bsp** (backspace) key will remove the last character typed, while entering a string. Previously entered strings, and other text which appears on the plot can be edited by deleting characters with **Bsp** and retyping. First, the desired string must be selected, by clicking with button 1. Alternatively, the left and right arrow keys can perform this function. Pressing one of these keys will move the pointer to the end of the new current string. Each string in the plot can be selected by repeatedly pressing one of these keys. The right arrow key cycles through the internal list in a forward sense, the left arrow cycles through the list backwards (the "oldest" string becomes the current string). Once a string is selected, pressing **Bsp** once "unterminates" the string, so that characters typed will be added to that string. Additional presses of the **Bsp** key will delete characters. If all characters have been deleted, pressing **Bsp** once more will delete the string anchor position, as indicated by the pointer moving to another string. If the termination is deleted, the string can be reterminated by pressing the **Enter** key.

The up and down arrow keys cycle the color of the current string through an internal list of 19 colors. These are the same colors used for plot traces. To change the color of the title, for example, one would click on the title string, then press the up or down arrow keys until the title color is satisfactory.

In plots, most text and the traces can be moved by dragging with mouse button 1, either within a plot or between plot windows. Thus, the order of the traces can be changed. Traces can be "grabbed" by pressing button 1 near the trace legend, but not over the legend text itself. A square wave marker is attached to the pointer when a trace is being dragged. Traces are actually copied, not moved, between plots. If the scale is not compatible between the two plot windows, the copy will fail. This is an easy way to see differences between simulation runs.

If a single multidimensional variable is plotted, the dimensions are shown using different colors, with a legend printed at the left. Clicking on one of the colored numbers will toggle visibility of that trace. If there are too many traces to list, a label "more" will exist at the bottom of the listing. Clicking on

this label will cycle through all of the trace labels.

The plot windows contain icons for changing the scales. These are triangles; the x-axis icons are in the lower left corner, and the y-axis icons are arrayed along the right edge. Clicking on one of these icons has the following effects:

right or **up**	
button 1	move the scale interval to the right or up
button 2 or **Shift**–button 1	extend the right or top scale factor to the right or up
button 3 or **Ctrl**–button 1	contract the right or top scale factor to the left or down
left or **down**	
button 1	move the scale interval to the left or down
button 2 or **Shift**–button 1	extend the left or bottom scale factor to the left or down
button 3 or **Ctrl**–button 1	contract the left or bottom scale factor to the right or up

3.9 The Mplot Panel

This panel appears when plotting results from operating range or Monte Carlo analysis, and is brought up by the mplot command. The display consists of an array of cells, each of which represent the results of a single trial. As the results become available, the cells indicate a pass or fail, In operating range analysis, the cells indicate a particular bias condition according to the axes. In Monte Carlo analysis, the position of the cells has no significance. In this case the display indicates the number of trials completed.

The panel includes a **Help** button which brings up a help message, a **Redraw** button to redraw the plot if, for example, the plotting colors are redefined, and a **Print** button for generating hard copy output of the plot.

Text entered while the pointer is in the mplot window will appear in the plot, and hardcopies. This text, and other text which appears in the plot, can be edited in the manner of text in plot windows.

The cells in an mplot can be selected/deselected by clicking on them. Clicking with button 1 will select/deselect that cell. Using button 2, the row containing the cell will be selected or deselected, and with button 3 the column will be selected or deselected. A selected cell will be shown with a colored background, with an index number printed.

Only one mplot window can have selections. Clicking in a new window will deselect all selections in other mplot windows. See the description of the mplot command in 4.5.5 for information about use of the selections.

3.10 The Print Control Panel

The **Print** button in the plot and mplot panels brings up a panel which provides an interface to the hardcopy drivers. The panel is a highly configurable multi-purpose printer interface used in many parts of *Xic* and *WRspice*. This section describes all of the available features, however many of these features may not be available, depending upon the context when the panel was invoked. For example, a modified version of this panel is used for printing text files. In that case, only the **Dismiss**, **To File**, and **Print** buttons are included.

Under Windows, the **Printer** field contains the name of a connected printer, initially the system default printer. The up/down control to the right can be pressed to cycle through the list of available printers.

Under Unix/Linux, the operating system command used to generate the plot is entered into the **Print Command** text area of the Print Control panel. In this string, the characters "`%s`" will be replaced with the name of the (temporary) file being printed. If there is no "`%s`", the file name will be added to the end of the string. The string is sent to the operating system to generate the plot.

The temporary file used to hold plot data before it is sent to the printer is *not* deleted, it is recommended that the print command include the option to delete the file when plotting is finished. A shell script `wr_lp` is provided for this purpose. This script is described below. Some operating systems provide a "`-r`" option to the `lpr` command which will remove the file when printing is complete. This option can be used in the command string as an alternative to using the `wr_lp` script. Also, large plots may require the use of the "`-s`" option to `lpr`, which will avoid a system limit on file size.

If the **To File** button is active, then this same field contains the name of the file to receive the plot data, and nothing is sent to the printer. The user must enter a name or path to the file, which will be created.

The size and location of the plot on the page can be specified with the **Width**, **Height**, **Left**, and **Top/Bottom** text areas. The dimensions are in inches, unless the **Metric** button is set, in which case the dimensions are in millimeters. The **Width**, **Height**, and offsets are always relative to the page in portrait orientation (even in landscape mode). The vertical offset is relative to either the top of the page, or the bottom of the page, depending on the details of the coordinate system used by the driver. The label is changed from **Top** to **Bottom** in the latter case. Thus, different sized pages are supported, without the driver having to know the exact page size.

The labels for the image height and width in the **Print** pop-up are actually buttons. When pressed, the entry area for height/width is grayed, and the auto-height or auto-width feature is activated. Only one of these modes can be active. In auto-height, the printed height is determined by the given width, and the aspect ratio of the area printed. Similarly, in auto-width, the width is determined by the given height and the aspect ratio of the area to print. In auto-height mode, the height will be the minimum corresponding to the given width. This is particularly useful for printers with roll paper.

The full-page values for many standard paper sizes are selectable in the drop-down **Media** menu below the text areas. Selecting a paper size will load the appropriate values into the text areas to produce a full page image. Under Windows, the "Windows Native" driver requires that the actual paper type be selected. Otherwise, this merely specifies the default size of the image.

Portrait or landscape orientation is selectable by the **Portrait** and **Landscape** buttons. These interlocking buttons switch between portrait and landscape orientation. In portrait mode, the plot is in the same orientation as seen on-screen, and in landscape mode, the image is rotated 90 degrees. However, if the **Best Fit** button is active, the image can have either orientation.

When the **Best Fit** button is active, the driver is allowed to rotate the image 90 degrees if this improves the fit to the aspect ratio of the plotting area. This supersedes the Portrait/Landscape setting for the image.

The available output formats are listed in a drop-down menu and are described below. Printer resolutions are selectable in the adjacent resolution menu. Not all drivers support multiple resolutions. Higher resolutions generate larger files which take more time to process.

Pressing the **Print** button actually generates the plot or creates the output file. This should be pressed once the appropriate parameters have been set. A pop-up message appears indicating success or failure of the operation.

The **Dismiss** button retires the Print Control panel.

3.10.1 Print Drivers

The printing system for *Xic* and *WRspice* provides a number of built-in drivers for producing output in various file formats. In Windows, an additional "Windows Native" driver uses the operating system to provide formatting, thus providing support for any graphical printer known to Windows. The data formats are selected from a drop-down menu available in the **Print** panel. The name of the currently selected format is displayed on the panel.

Except for the "Windows Native" driver all formatting is done in the *Xic*/*WRspice* printer drivers, and the result is sent to the printer as "raw" data. This means that the selected printer *must* understand the format. In practice, this means that the printer selected must be a PostScript printer, and one of the PostScript formats used, or the printer can be an HP Laserjet, and the PCL format used, etc. The available formats are listed below.

PostScript bitmap
: Generates a two color PostScript bitmap of the plotted area.

PostScript bitmap, encoded
: Also produces a two color PostScript bitmap, but uses compression to reduce file size. Some elderly printers may not support the compression feature.

PostScript bitmap color
: Produces a PostScript RGB bitmap of the plotted area. These files can grow quite large, as three bytes per pixel must be stored.

PostScript bitmap color, encoded
: Generates a compressed PostScript RGB bitmap of the plotted area. Due to the file size, this format should be used in preference to the non-compressing format, unless the local printer does not support PostScript run length decoding.

Postscript line draw, mono
: This driver produces a two color PostScript graphics list representing the plotted area.

PostScript line draw, color
: Produces an RGB color PostScript graphics list representing the plotted area.

HP laser PCL
: Produces monochrome output suitable for HP and compatible printers. This typically processes more quickly than PostScript on these printers.

HPGL line draw, color
: This driver produces color output in Hewlett-Packard Graphics Language, suitable for a variety of printers and plotters.

WordPerfect Graphics, mono
: This driver produces two-color output in the WordPerfect 5.1 graphical format, for inclusion in documents.

WordPerfect Graphics, color
: This driver produces full-color output in the WordPerfect 5.1 graphical format, for inclusion in documents.

Windows Native
: This driver is available under Microsoft Windows only. This selection bypasses the drivers in *Xic*

or *WRspice* and uses the driver supplied by Windows. Thus, any graphics printer supported by Windows should work with this driver.

The Windows Native driver should be used when there is no other choice. If the printer has an oddball or proprietary interface, then the Windows Native driver is the one to use. However, for a PostScript printer, better results will probably be obtained with one of the built-in drivers (PostScript Line Draw, etc). The same is true if the printer understands PCL, as do most laser printers. This may vary between printers, so one should experiment and use whatever works best.

In the Unix/Linux versions, selecting a page size from the **Media** menu will load that size into the entry areas that control printed image size. This is the only effect, and there is no communication of actual page size to the printer. This is true as well under Windows, except in the Windows Native driver. Microsoft's driver will clip the image to the page size before sending it to the printer, and will send a message to the printer giving the selected paper size. The printer may not print if the given paper size is not what is in the machine. Thus, when using this driver, it is necessary to select the actual paper size in use.

Xfig line draw, color

Xfig is a free (and very nice) drafting program available over the Internet. Through the transfig program, which should be available from the same place, output can be further converted to a dozen or so different formats.

Imlib: jpeg, tiff, png, etc

This driver converts into a multitude of bitmap file formats. This supports file generation only. The type of file is determined by the extension of the file name provided (the file name should have one!). The driver can convert to several formats internally, and can convert to many more by making use of "helper" programs that may be on your system.

Internal formats:

Extension	Format
`ppm, pnm, pgm`	Portable Bitmap (netpbm)
`ps`	PostScript
`jpg, jpeg`	JPEG
`png`	PNG
`tif, tiff`	TIFF

On Unix/Linux systems, if you have the ImageMagick or netpbm packages installed (available as freeware on the Internet) then many more formats are available, including GIF. The imsave library, which is used to implement this driver and otherwise generate image files, uses a special search path to find helper functions (`convert` from ImageMagick, the netpbm functions, `cjpeg` and `djpeg`). The search path (a colon-delimited list of directories) can be provided in the environment variable IMSAVE_PATH. If not set, the internal path is "`/usr/local/bin:/usr/X11R6/bin`". The helper function capability is not available under Microsoft Windows.

The choice between PostScript line draw and bitmap formats is somewhat arbitrary. Although the data format is radically different, the plots should look substantially the same. A bitmap format typically takes about the same amount of time to process, independent of the data shown, whereas a line draw format takes longer with more objects to render. For very simple layouts and all schematics and *WRspice* plots, the line draw formats are the better choice, but for most layouts the bitmap format will be more efficient.

The necessary preamble for Encapsulated Postscript (EPSF-3.0) is included in all PostScript files, so that they may be included in other documents without modification.

3.10.2 wr_lp

The default print command for many of the hard copy drivers is a call to the script "wr_lp". This script is located in the same directory as the program binaries, and can be customized by the (super) user. The script is a wrapper around the system printer command (usually "lpr") and is designed to solve a problem. The application will not delete temporary files created for printing/plotting, since there is no way generally for the application to know when the job is finished and it is safe to delete the file. The wr_lp script runs the UNIX "at" command, which will delete the file one hour after the print job is submitted. This is true whether or not the application is running, or if the user is no longer logged in. See the manual page for the at command to make sure that it is set up correctly.

Some versions of the lpr command provide a "-r" option, which will remove the file when printing is complete (see "man lpr"). Use of this option is an alternative to using the wr_lp script.

3.11 The *WRspice* Help System

The *WRspice* help system provides a cross-referenced rich-text (HTML) database on the commands and features of the program. The system is entered at the top level by pressing the **Help** button in the **Help** menu of the Tool Control window, or by giving the help command without arguments. If a command name or other known keyword is given as an argument to the help command, the help system will start by displaying the help for that topic.

When graphics is not available, the help text will be presented in a text-only format in the console window. The HTML to ascii text converter only handles the most common HTML tags, so some descriptions may look a little strange. The figures (and all images) are not shown, and clickable links will not be available, other than the "references" and "seealso" topics.

Under Windows, the help viewer and HTML-based info windows use a Microsoft technology called OLE (Object Linking and Embedding) and the Internet Explorer HTML viewer. It actually uses the internals of Internet Explorer to do the HTML rendering. This requires that Internet Explorer 4.0 or newer be installed. Although this technology is admittedly rather nifty, it has rather amazing internal complexity, and when all is said and done, sad and severe limitations. Although the basic functionality is provided, many of the more advanced/obscure features found in the Unix/Linux help system are missing.

Clicking on a colored HTML reference will bring up the text of the selected topic. If button 1 is used to click, the text will appear in the same window. If button 2 is used to click, a new help window containing the selected topic will appear.

The help system operates in one of two modes. The default mode is to use a single window for each invocation of the help command. This is the only mode available in *WRspice*. In the multi-window implementation, which can be selected in *Xic* by setting the boolean variable HelpMultiWin with the !set command, a separate window is brought up for each press of a command button or menu item while in help mode. In either case, clicking on a link may or may not produce a new window, depending upon whether button 1 or button 2 was clicked.

Text shown in the viewer that is not part of an image can be selected by dragging with button 1, and can be pasted into other windows in the usual way.

The viewer can be used to display any text file or URL. The name given to the help command, or to to the **Open** command in the viewer's **File** menu, can be

- A keyword for an entry in the help database.

- A path to a file on the local machine.
- An arbitrary URL accessible through the Internet.

If the given name can be resolved, the resulting page will be displayed in the viewer. Also, the HTML viewer is sensitive as a drop receiver. If a file name or URL is dragged into the viewer and dropped, that file or URL is read into the viewer, after confirmation.

The ability to access general URLs should be convenient for accessing information from the Internet while using *Xic* and *WRspice*. The prefix "`http://`" *must* be provided with the URL. Thus, for example,

```
help http://www.wrcad.com
```

will bring up the Whiteley Research web page. The links can be followed by clicking in the usual way. Of course, the computer must have Internet access for web pages to be accessible.

The URL given in the **Open** command or through the help command is not relative to the current page, however if a '+' is given before the URL, it will be treated as relative. For example, if the viewer is currently displaying `http://www.foo.bar`, if one enters "`/dir/file.html`", the display will be updated to `/dir/file.html` on the local machine. If instead one enters "`+/dir/file.html`", the display will be loaded with `http://www.foo.bar/dir/file.html`.

The HTTP capability imposes some obvious limitations on the string tokens which can be used in the help database. These keywords should not use the '/' character, or begin with a protocol specifier such as "`http:`".

HTML files on a local machine can be loaded by giving the full path name to the file. Relative references will be found. HTML files will also be found if they are located in the help path, however relative references will be found only if the referenced file is also in the help path. If a directory is referenced rather than a file, a formatted list of the files in the directory is shown.

If a filename passed to the viewer has one of the following extensions, the text is shown verbatim. The (case insensitive) extensions for plain-text files are "`.txt`", "`.doc`", "`.log`", "`.scr`", "`.sh`", "`.c`", "`.cc`", and "`.h`".

In the *WRspice* help system, link references to files with a "`.cir`" extension will be sourced into *WRspice* when the link is clicked on. Thus, if one has a circuit file named "`mycircuit.cir`", and the HTML text in the help window contains a reference like

```
<a html="mycircuit.cir">click here</a>
```

then clicking on the "click here" tag will source `mycircuit.cir` into *WRspice*. Similarly, link references to files with a "`.raw`" extension will be loaded into *WRspice* (as a *rawfile*, i.e. a plot data file) when the anchor is clicked.

This feature may solve a big problem. How many *WRspice* users have directories full of old simulation files, the details about which are long forgotton or buried in some notebook somewhere? Now the documentation task may be somewhat simpler. While doing simulations, one can maintain a text file containing notes about the circuit and results, with HTML anchor tags to the actual circuit and data files. Then, one can load the text file into the *WRspice* help system (if the notes are in a file "`notes.html`", one just types "`help notes.html`"), and browse the notes and have one-click access to the original files and plot data. The notes file need not contain any other HTML constructs besides the anchor references.

Holding **Shift** while clicking on an anchor that points to a URL which specifies a file on a remote system will download the file, as in Netscape. Downloading makes use of the `httpget` utility program

available in the Accessories distribution. Installation of the accessories is required for downloading to be available under Unix/Linux. References to files with extensions "`.rpm`", "`.gz`", and other common binary file suffixes will automatically cause downloading rather than viewing. When downloading, the file selection pop-up will appear, pre-loaded with the file name (or "`http_return`" if the name is not known) in the current directory. One can change the saved name and the directory of the file to be downloaded. Pressing the **Download** button will start downloading. A pop-up will appear that monitors the transfer, which can be aborted with the **Cancel** button.

3.11.1 The HTML Viewer

There are three colored buttons in the menu bar of the viewer. The left-facing arrow button (back) will return to the previous topic shown in the window. The right-facing arrow button (forward) will advance to the next topic, if the back button has been used. The **Stop** button will stop HTTP transfers in progress.

There are four drop-down menus in the menu bar: **File**, which contains basic commands for loading and printing, **Options**, which contains commands for setting display attributes, **Bookmarks**, which allows saving frequently used references, and **Help** which provides documentation.

The **Open** button in the **File** menu pops up a dialog into which a new keyword, URL, or file name can be entered. The **Open File** button brings up the file manager. The **Ok** button (green octagon) on the file browser will load the selected file into the viewer (the file should be a viewable file). The file can also be dragged into the viewer from the file manager.

The **Save** button in the **File** menu allows the text of the current window to be saved in a file. This functionality is also provided by the **Print** button. The saved text is pure ASCII.

The **Print** button brings up a pop-up which allows the user to send the help text to a printer, or to a file. The format of the text is set by the drop-down menu, with the current setting indicated on the menu button. The choices are PostScript in four fonts (Times, Helvetica, New Century Schoolbook, and Lucida Bright), HTML, or plain text. If the **To File** button is active, output goes to that file, otherwise the command string is executed to send output to a printer. If the characters "`%s`" appear in the command string, they are replaced with the temporary print file name, otherwise the temporary file name is appended to the string.

The **Reload** button in the **File** menu will re-read the input file and redisplay the contents. This can be useful when writing new help text or HTML files, as it will show changes made to the input file. However, if you edit a "`.hlp`" file, the internally cached offsets for the topics below the editing point will be wrong, and will not display correctly. When developing a help text topic, placing it in a separate file will avoid this problem. If the displayed object is a web page, the page will be redisplayed from the disk cache if it is enabled, rather than being downloaded again.

The **Quit** button in the **File** menu removes the help window.

The **Search** button in the **Options** menu brings up a dialog which solicits a regular expression to use as a search key into the help database. The regular expression syntax follows POSIX 1003.2 extended format (roughly that used by the Unix egrep command). The search is case-insensitive. When the search is complete, a new display appears, with the database entries which contained a match listed in the "References" field. The library functions which implement the regular expression evaluation differ slightly between systems. Further information can be found in the Unix manual pages for "regex".

The **Set Font** button in the **Options** menu will bring up a font selection pop-up. One can choose a typeface from among those listed in the left panel. The base size can be selected in the right panel. There are two separate font families used by the viewer: the normal, proportional-spaced font, and a

fixed-pitch font for preformatted and "typewriter" text. The latter does not actually have to be fixed-pitch, however preformatted columns will not line up unless this is so. Pressing **Apply Font** will set the currently selected font as the viewer's proportional font, and **Apply Fixed Font** will set the currently selected font as the viewers "fixed" font. In either case, the display will be redrawn using the new font. The new font selection will be updated in the `.mozyrc` file if it exists, so that subsequent windows will use the new font.

The **Filter** page in the popup can be used to limit the choices. In particular, if you want a large font, selecting only **Scalable** in **Font Types** eliminates the ragged-looking bitmap fonts from the selection. One can also limit the choices to monospaced fonts when selecting the fixed-pitch font.

A disk cache of downloaded pages and images is maintained. The cache is located in the user's home directory under a subdirectory named "`.wr_cache`". A file named "`directory`" in this directory contains a human-readable listing of the cache files and the original URLs. This directory may also contain a file named "`cookies`" which contains a list of cookies received from web sites. The cache files are named "`wr_cache`*N*"" where *N* is an integer. The listing consists of a line with internal data, followed by data for the cache files. Each such line has three columns. The first column indicates the file number *N*. The second column is 0 if the `wr_cache`*N* file exists and is complete, 1 otherwise. The third column is the source URL for the file. The number of files saved is limited, defaulting to 64. The cache only pertains to files obtained through HTTP transfer.

A page will not be downloaded if it exists in the cache, unless the modification time of the page is newer than the modification time of the cache file.

The **Don't Cache** button in the **Options** menu will disable caching of downloaded pages and images.

The **Clear Cache** button in the **Options** menu will clear the internal references to the cache. The files, however, are not cleared.

The **Reload Cache** button in the **Options** menu will clear and reload the internal cache references from the files that presently exist in the cache directory.

The **Show Cache** button in the **Options** menu brings up a listing of the URLs in the internal cache. Clicking on one of the URLs in the listing will load that page or image into the viewer. This is particularly useful on a system that is not continuously on-line. One can access the pages while on-line, then read them later, from cache, without being on-line.

Support is provided for Netscape-style cookies. Cookies are small fragments if information stored by the browser and transmitted to or received from the web site. There is a **No Cookies** button in the help window **Options** menu to disable sending and receiving cookies. The NoCookies keyword in the `.mozyrc` file (see below) can be set to 1 to disable cookies by default, or 0 for the default (enabled) behavior. With cookies, it is possible to view the New York Times web site, for example, which requires registration. It is also possible to view some commerce sites that require cookies. There is no encryption, so it is not a good idea to send sensitive info like credit card numbers, though.

The **No Cookies** button will disable sending and receiving of cookies, if set.

Image support is provided for gif, jpeg, png, tiff, xbm, and xpm. Animated gifs are supported as well. Images found on the local file system are always displayed immediately (unless debugging options are set in the startup file). The treatment of images that must be downloaded is set by a button group in the **Options** menu. One and only one of these choices is active. If **No Images** is chosen, images that aren't local will not be displayed at all. If **Sync Images** is chosen, images are downloaded as they are encountered. All downloading will be complete before the page is displayed. If **Delayed Images** is chosen, images are downloaded after the page is displayed. The display will be updated as the images are received. If **Progressive Images** is chosen, images are downloaded after the page is displayed, and

images are displayed in sections as downloading progresses.

There are choices as to how anchors (the clickable references) are displayed. If the **Anchor Plain** button in the **Options** menu is selected, anchors will be displayed with standard blue text. If **Anchor Buttons** is selected, a button metaphor will be used to display the anchors. If **Anchor Underline** is selected, the anchor will consist of underlined blue text. The underlining style can be changed in the "`mozyrc`" startup file. One and only one of these three choices is active. In addition, if **Anchor Highlight** is selected, the anchors are highlighted when the pointer passes over them.

If the **Bad HTML Warnings** button in the **Options** menu is active, messages about incorrect HTML format are emitted to standard output.

If the **Freeze Animations** button in the **Options** menu is active, active animations are frozen at the current frame. New animations will stop after the first frame is shown. This is for users who find animations distracting.

If the **Log Transactions** button in the **Options** menu is active, the header text emitted and received during HTTP transactions is printed on the terminal screen. This is for debugging and hacking.

The **Bookmarks** menu contains entries to add and delete entries, plus a list of entries. The entries, previously added by the user, are help keywords, file names, or URLs that can be accessed by selecting the entry. Thus, frequently accessed pages can be saved for convenient access. Pressing the **Add** button will add the page currently displayed in the viewer to the list. The next time the **Bookmarks** menu is displayed, the topic should appear in the menu. To remove a topic, the **Delete** button is pressed. Then, the menu is brought up again, and the item to delete is selected. This will remove the item from the menu. Selecting any of the other items in the menu will display the item in the viewer. The bookmark entries are saved in a file named "`bookmarks`" which is located in the same directory containing the cache files.

3.11.2 The Help Database

The help system uses a fast hashed lookup table containing cached file offsets to the entry text. A modular database provides flexibility and portability. The files are located by default in the directories named "help" under the library tree, which is usually rooted at `/usr/local/lib/xictools`. All of the files with suffix "`.hlp`" in the directories along the help search path are parsed, and reference pointers added to the internal list, the first time the help command is issued in the application. In addition, other types of files, such as GIF image files ("`.gif`" suffix) which are referenced in the HTML help text may be present as well.

The "`.hlp`" files have a simple format allowing users to create and modify them. Each help item is indexed by a keyword which should be unique in the database. The help text may be in HTML or plain text format. The format is described in A.2.

3.11.3 Help System Forms Processing

Support is provided for HTML forms. When the form "Submit" button is pressed, a temporary file is created which contains the form output data. The file consists of key/value pairs in the following formats:

name=*single_token*
name="*whatever*"

There is no white space around '=', and white space should separate the pairs (they are on separate lines).

The action string from the "`<form ...>`" tag determines how this file is used. The file is a temporary file, and is deleted immediately after use. If the action string is in the form "`action_local_`*xxxx*", then the form data are processed internally.

If the full path for the action string begins with "`http://`" or "`ftp://`", then the form data are encoded into a query string and sent to the location (though it is likely an error for ftp). Otherwise, the file will processed locally. This enables the output from the form to be processed by a local shell script or program, which can be very useful. The command given as the action string is given the file contents as standard input. The command standard output will appear in the HTML viewer window. Thus, one can create HTML form front-ends for favorite shell commands and programs.

3.11.4 Help System Initialization File

When a help window pops up, an initialization file is read, if it exists. This file is named "`.mozyrc`" and is sought in the user's home directory. This file is never created, but may be updated by certain operations, such as the **Save** button in the Font Selector, if it exists. A sample `.mozyrc` file is provided in the startup directory of the *Xic* and *WRspice* distributions, and a listing is provided in A.2.2. See the comments in the file for the defaults that can be changed. One should place this text as "`.mozyrc`" in the home directory to change the defaults.

Incidently "mozy" is the name of the stand-alone version of the HTML viewer/web browser available on the Whiteley Research web site.

3.12 The *WRspice* Shell

The command line interpreter in *WRspice* provides many of the features of a UNIX shell. The interpreter, in addition to parsing and responding to command text input, is used as an interpreter for control scripts which control *WRspice* operation. In addition, circuit descriptions have all shell variables expanded during the sourcing process. Thus, shell variables can be used to set circuit parameters.

Various features are available in the *WRspice* shell which are similar to the user interface of the C-Shell. These include IO redirection, history substitution, aliases, global substitution, and command completion.

3.12.1 Command Line Editing

The *WRspice* shell contains a line editor system similar to that found in some UNIX shells. The left and right arrow keys can be used to move the cursor within the line of text, so that text can be entered or modified at any point. Pressing the **Enter** key sends the line of text to *WRspice*, regardless of where the cursor is at the time. The up arrow key will load the line of text with the previously entered line progressively. The down arrow key cycles back through the history text. **Ctrl-E** places the cursor at the end of text, **Ctrl-A** places the cursor at the beginning of the line. **Bsp** (backspace) erases the character to the left of the cursor, **Delete** deletes the character at the cursor, and **Ctrl-K** will delete from the cursor to the end of the line. **Ctrl-U** will delete the entire line.

The following keys perform editing functions:

Ctrl-A	Move cursor to beginning of line
Ctrl-D	List possible completion matches
Tab	Insert completion match, if any
Ctrl-E	Move cursor to end of line
Ctrl-H, Bsp	Erase character to left of cursor
Ctrl-K	Delete to end of line
Ctri-U	Delete line
Ctrl-V	Insert following character verbatim
Delete	Delete character at cursor
Left arrow	Move cursor left
Right arrow	Move cursor right
Up arrow	Back through history list
Down arrow	Forward through history list

By default, command line editing is enabled in interactive mode, which means that *WRspice* takes control of the low level functions of the terminal window. Command line editing can be disabled by setting the `noedit` variable (with the `set` command). If the terminal window doesn't work with the editor, it may be necessary that "`set noedit`" appear in the *WRspice* startup (`.wrspiceinit`) file. When `noedit` is set, the command completion character is **Esc**, rather than **Tab**.

Some terminals may not send the expected character or sequence when one of these keys is pressed, consequently there is a limited key mapping facility available. This mapping is manipulated with the `mapkey` command, which allows most of the keys and combinations listed above to be remapped.

Unless *WRspice* can read the system terminfo/termcap data it needs, it will not allow command line editing, and a warning message will be issued. This may mean that the TERM or TERMINFO environment variables are not set or bogus, or the system terminal info database is incomplete or bad. One can enter alternative terminal names with the `-t` command line option to potentially fix this problem. The non-editing mode is like a standard terminal line, where backspace is available, but the arrow keys and others that move the cursor have no special significance. This is the mode used when "`set noedit`" is given.

3.12.2 Command Completion

Tenex-style command, filename, and keyword completion is available. If **Ctrl-D** (EOF) is typed, a list of the commands or possible arguments is printed. If **Tab** (or instead, **Esc** if command line editing is disabled) is typed, then *WRspice* will try to complete the word being typed based on the choices available, or if there is more than one possibility, it will complete as much as it can. Command completion knows about commands, most keywords, variable and vector names, file names, and several other types of arguments. To get a list of all commands, the user can type **Ctrl-D** at the *WRspice* prompt. Note that for keyboard input, the `EOF` character, **Ctrl-D**, does *not* exit the shell.

Command completion is disabled if the `-q` option is given on the *WRspice* command line, or if the `nocc` variable is set.

3.12.3 History Substitution

History substitutions, similar to C-shell history substitutions, are also available. History substitutions are prefixed by the character `!`, or at the beginning of a line, the character `^`. Briefly, the string `!!` is replaced by the previous command, the string `!`*prefix* is replaced by the last command with that prefix, the string `!?`*pattern* is replaced by the last command containing that pattern, the string `!`*number*

is replaced by the event with that number, and ^*oldpattern*^*newpattern* is replaced by the previous command with *newpattern* substituted for *oldpattern*.

Additionally, a !*string* sequence may be followed by a modifier prefixed with a :. This modifier may select one or more words from the event — :1 selects the first word, :2-5 selects the second through the fifth word, :$ selects the last word, and :$-0 selects all of the words but reverses their order.

Two other : modifiers are supported: :p will cause the command to be printed but not executed, and :s^*old*^*new* will replace the pattern *old* with the pattern *new*. The sequence ^*old*^*new* is synonymous with !!:s^*old*^*new*.

All the commands typed by the user are saved on the history list. This may be examined with the history command, and its maximum length changed by changing the value of the history variable.

3.12.4 Alias Substitution

After history expansion, if the first word on the command line has been defined as an alias (with the alias command), the text for which it is an alias for is substituted. The alias may contain references to the arguments provided on the command line, in which case the appropriate arguments are substituted in. If there are no such references, any arguments given are appended to the end of the alias text.

If a command line starts with a backslash '\' any alias substitution is inhibited.

Aliases are defined with the alias command, and may be removed with the unalias command.

3.12.5 Global Substitution

The characters ~, {, } have the same effects as they do in the C-Shell, i.e., home directory and alternative expansion. In alternative expansion, if a token contains a form like "{foo,bar,baz}", the token is replicated with each replication containing one of the list items from the curly braces replacing the curly brace construct. For example, the string "stuff{string1,string2,...stringN}morestuff" is replaced by the list of words "stuffstring1morestuff stuffstring2morestuff ... stuffstringNmorestuff". Curly braces may be nested. A particularly useful example is

```
plot v({4,5,7})
```

which is equivalent to

```
plot v(4) v(5) V(7)
```

The string ~*user* (tilde at the beginning of a word) is replaced by the given user's home directory, or if the first component of the pathname is simply "~", the current user's home directory is understood.

It is possible to use the wildcard characters *, ?, [, and] to match file names, where * denotes 0 or more characters, ? denotes one character, and [...] denotes one of the specified characters, but these substitutions are performed only if the variable noglob is unset. The pattern [^abc] will match all characters except a, b, and c. The noglob variable is normally set so that the symbols have their usual meanings in algebraic expressions. This can be unset with the unset command if command "globbing" is desired.

3.12.6 Quoting

Words may be quoted with the characters (") (double quote), (') (single quote), and (`) (back quote). A word enclosed by any of these quotes may contain white space. A string enclosed by double quotes may have further special-character substitutions done on it, but it will be considered a single token by the shell. A number so quoted is considered a string. A string enclosed by single quotes also has all its special characters protected. Thus no global expansion (*, ?, etc), variable expansion ($), or history substitution (^, !) will be done. A string enclosed by backquotes is considered a command to the shell and is executed, and the output of the command replaces the text. The backslash character performs the usual single character quoting function. In addition, **Ctrl-V** also provides a single character quoting function from the keyboard.

3.12.7 I/O Redirection

The input to or output from commands may be changed from the terminal to a file by including IO redirection on the command line. The possible redirections are:

> *file*
> Send the output of the command into the file. The file is created if it doesn't exist.

>> *file*
> Append output to the file. The file is created if it doesn't exist.

>& *file*
> Send both the output and the error messages to the file. The file is created if it doesn't exist.

>>& *file*
> Append both the output and the error messages to the file. The file is created if it doesn't exist.

< *file*
> Read input from the file.

Both an input redirection and an output redirection may be present on a command line. No more than one of each may be present, however. IO redirections must be at the end of the command line.

3.12.8 Semicolon Termination

More than one command may be put on one line, separated by semicolons ';'. The semicolons must be isolated by white space, however. Thus a multi-command alias might be written

alias *word 'command1* ; *command2* ; ...'

3.12.9 Variables and Variable Substitution

Shell variables can be set with the set command, or graphically with some of the tools available in the **Tools** menu of the Tool Control window. In particular, the **Shell** button in the **Tools** menu brings up a panel which allows those variables which control shell behavior to be set. Both methods of setting and unsetting the shell variables are equivalent. The **Variables** tool in the **Tools** menu provides a listing of the variables currently set, and is updated dynamically when variables are set and unset. A variable

with any alphanumeric name can be set, though there are quite a number of predefined variable names which have significance to *WRspice*.

Shell variables have boolean type if they are defined without assigning any text to them. Otherwise, the variables take a single text token as their defining value, or a list of text tokens if the assigned value consists of a list of tokens surrounded by space-separated parentheses. See 4.1.8 for details of the syntax of the set command.

The values of variables previously set can be accessed in commands, circuit descriptions, or elsewhere by writing `$`*varname* where the value of the variable is to appear. However, if a backslash (\) precedes `$`, the variable substitution is not performed. The special variable references `$$` and `$<` are replaced by the process ID of the program and a line of input which is read from the terminal when the variable is evaluated, respectively. Also, the notation `$?foo` evaluates to 1 if the variable `foo` is defined, 0 otherwise, and `$#foo` evaluates to the number of elements in `foo` if it is a list, 1 if it is a number or string, and 0 if it is a boolean variable. If `foo` is a valid variable, and is of type list, then the expression `$foo`[*low-high*] represents a range of elements. The values in the range specification `[...]` can also be shell variable references. Either the upper index or the lower may be left out, and the reverse of a list may be obtained with `$foo`[*len*-0].

If a variable reference has the form `$&`*word*, then *word* is assumed to be a vector, and its value is used to satisfy the reference. Vectors consist of one or more real or complex numbers, and are produced, among other ways, during simulation, in which case they represent simulation output. The shell variable substitution mechanism allows reference to all of the vectors in scope. The reference can be followed by range specifiers in square brackets, consistent with the dimensionality and size of the vector. The range specifier can itself contain shell variable references. The complete information on vectors and vector expressions is presented in 3.13.

The sequences `$?&`*vector* and `$#&`*vector* are accepted. The first expands to 1 if *vector* is defined with the let command or otherwise, 0 if not. The second expands to the vector length or 0 if *vector* is undefined. This is analogous to `$?`*variable* and `$#`*variable* for shell variables.

The notation `$&`(*expression*) is replaced by the value of the vector *expression*. A range specification can be added, for example

```
echo $&(a+1)[2]
```

prints the third entry in vector `a+1`, or 0 if out of range.

When a circuit file is sourced into *WRspice*, each line of the circuit description has variable substitution performed by the shell. Thus, shell variables can be used to define circuit parameters, if within the circuit description the parameter is specified in the form of a variable reference. The variable substitution in a SPICE deck allows a concatenation character '`%`'. This is used between a variable and other text, which would otherwise mask the variable. For example

```
set value = 10
v1 1 0 pulse(0 $value%m 5p 10p)
```

expands to

```
v1 1 0 pulse(0 10m 5p 10p).
```

Without the `%`, the pattern match would fail.

3.12.10 Commands and Scripts

Command files are files containing circuit descriptions and/or shell commands. The first line of a command file is ignored, so must be blank or a comment. This is a result of the source command being used for both circuit input and command file execution.

A pure script file, i.e., one which does not include a circuit description, consists of an unread "title" line, followed by a control block. The control block begins with a "`.control`" line, continues with one or more executable statements, and terminates with a "`.endc`" line. In *WRspice*, an "`.exec`" line can be used rather than the `.control` line, though for backward compatibility with SPICE3, it is recommended that the traditional `.control` be used. The executable statements are any statements understandable to the *WRspice* shell. Typically, such statements appear just as they would be entered on the command line if given as text input. Scripts can call other scripts to any depth.

Before a script is read, the variables `argc` and `argv` are set to the number of words on the command line, and a list of those words respectively. Their previous values (if any) are pushed onto a stack, and popped back in place when the script terminates. Thus, within a command script, these predefined variables are available for use in the script. Otherwise, command files may not be reentrant since there are no local variables, however, the procedures may explicitly manipulate a stack.

There may be various command scripts installed in the default scripts directory, and the default `sourcepath` includes this directory, so one can use these command files (almost) like built-in commands. In addition to scripts, there is an executable data structure called a "codeblock". Codeblocks are derived from scripts, but store the command text internally, so are somewhat more efficient. A codeblock has the same name (in general) as the script file from which it was derived. See the description of the codeblock command (in 4.4.1) for more information.

When a line of input is given to *WRspice*, the first word on the line determines how the line is processed. The following logic is used to make this determination.

1. If the word is an alias, the line is replaced with the result after alias substitution, and the line is re-parsed.
2. If the word matches the name of a codeblock in memory, the codeblock is executed.
3. If the word matches the name of an internal command, that command is executed.
4. If the first word is a vector name and is followed by "=", the line is taken to be an implicit let command (an assignment), in which case the line is executed as if it were preceded by the word "let".
5. If the word matches the name of a file found in one of the directories of the current `sourcepath` (search path), an implicit source command is assumed. The line is executed as if it were preceded by the word "source". Thus, typing the name of a circuit or script file will source or run the file.
6. If the variable `unixcom` is set, and the word matches the name of a command known to the operating system, the line will be sent to the operating system for execution.

If the variable `unixcom` is set and the operating system is supportive, commands which are not built-ins are considered shell commands and executed as if the program were a shell. However, using this option increases the start-up time of the program. Probably *WRspice* should not be used as a login shell.

WRspice can be used as the "shell" in UNIX shell scripts. In these scripts, the `wrspice` executable should be called, using the convention applicable to the user's UNIX shell. This generally requires that

the first line of the script begin with the characters "`#!`" and be followed by a space-separated program invocation string. The remainder of the file should consist of standard *WRspice* command file lines, the first line of which (second line of the file) will be ignored.

For example, below is a script that can be saved in a file, which should be made executable (using the UNIX command "`chmod +x` *filename*"). From the UNIX shell, typing the name of the file will run *WRspice* on the example file `mosamp2.cir` and display the plot.

```
#!  wrspice -J
#
.exec
source /usr/local/lib/xictools/wrspice/examples/mosamp2.cir
run
plot v(20) v(66)
set noaskquit
pause
quit
.endc
```

Typing the name of the file is the same as executing "`wrspice -J` *file*". *WRspice* ignores the `#!...` line, so that the next line is the "title" line and is also ignored. The `-J` (JSPICE3 compatibility) means to not bring up the Tool Control window.

3.13 Plots, Vectors and Expressions

WRspice data are in the form of vectors: time, voltage, etc. Each vector has a type, and vectors can be operated on and combined algebraically in ways consistent with their types. During a simulation, each of the circuit variables, plus the scale vector, are filled with data from the simulation. For each simulation, the resulting vectors are associated with a "plot". The plot is given a name (such as "tran2"), and stored in a list with other plots. There is one hard coded plot called "`constants`" which contains various length 1 vectors set to constant values.

An analysis will produce a plot consisting of vectors representing simulated output data. The "current plot" is usually the last plot produced by an analysis run, or the `constants` plot if no analyses have been run or rawfiles loaded. Vectors created with the let command are added to the current plot. The current plot can be changed with the setplot command, or with the **Plots** button in the **Tools** menu. A vector from any plot can be referenced with the notation

plotname.vecname

where *plotname* is the name of the plot, and *vecname* is the vector name.

When a rawfile is read into *WRspice* with the load command, a plot containing vectors is produced, as if the anaylsis had been run. The new plot becomes the current plot.

Vectors can be created with the let and compose commands. Newly-created vectors are added to the current plot, unless a *plotname* field is specified as part of the vector reference name. For example, entering

```
let myvec = constants.e
```

will assign a vector `myvec` in the current plot the values of the vector `e` in the `constants` plot. The let command without arguments will print a listing of vectors in the current plot.

The vectors actually produced depend on the type of analysis, but the most common output is the node voltage. Node voltages are denoted by vectors of the form `v(`*N*`)`, where *N* is a name representing the node. Although the notation looks like a function call, the construct actually refers to a vector, and may be used in expressions whenever a vector is syntactically expected. Another common form is *name*`#branch`, which represents the "branch" current through voltage sources and inductors. The SPICE algorithm adds a term to the matrix for these elements, which represents the current flowing through the device. As there is a specific matrix element for the current for these devices, the value is available as an output variable. The *name* is the name of the voltage source or inductor. For compatibility with SPICE2, the construct `i(`*name*`)` is internally mapped to *name*`#branch`, and the two notations can be used interchangeably.

Additional mappings familiar from SPICE2 are also recognized in *WRspice*. In addition to `v()` and `i()`, the following are recognized:

`vm(), im()`	Magnitude
`vp(), ip()`	Phase
`vr(), ir()`	Real part
`vi(), ii()`	Imaginary part
`vdb(), idb()`	Decibel value ($20log_{10}()$)

These are most useful for complex vectors as are produced in ac analysis. The general form for the argument inside the parentheses of the node voltage constructs is `(`*node1* `[,`*node2*`])`, where if both *node1* and *node2* are given, the vector represents the voltage difference between nodes *node1* and *node2*. If *node2* and the corresponding comma are omitted, then the ground node, at zero potential, is implied. In the branch current construct, the single argument is always the name of a "branch" device, either a voltage source or inductor.

Most simply, vector names can be any alphanumeric word that starts with an alpha character. Vector names may also be of the form *string*(*something*), if the *string* is not the name of a built-in or user-defined function. In *WRspice*, a vector name beginning with the '`@`' symbol is considered a reference to an internal device or model parameter, or a circuit parameter. If the vector name is of the form `@`*name*`[`*param*`]`, this denotes the parameter *param* of the device or model named *name*. Of course, there must be a device or model with that name defined for the current circuit and *param* must be a valid parameter name for that device or model type. See the documentation or use the show command for a listing of the parameters available. If the vector name is of the form `@`*param*, this refers to a parameter of the circuit with the name *param*. The parameters obtained in this manner are those defined on a `.options` line in the current circuit, those defined on a `.param` line in a current circuit, and the keywords of the rusage command, sought in that order. These vectors may be used as arguments to commands and functions which take vector arguments.

Vectors have an indexing that begins with 0, and an index, or range of indices, can be specified in square brackets following the vector name. The notation *vec*[*lower,upper*], where *lower* and *upper* are numbers, denotes the range of elements from *vec* between *lower* and *upper*. The notation *vec*[*num*] denotes the *num*'th element of *vec*. If *upper* is lower than *lower*, the order of the elements in the vector is reversed.

Using the let command, a vector may be assigned the values of a vector already defined, or a floating-point number (a real scalar), or a comma separated pair of numbers (a complex scalar). A number may be written in any format acceptable to SPICE2, such as `14.6MEG` or `-1.231e-4`. Note that one can use either scientific notation or one of the abbreviations like MEG or G (case insensitive), but not both. As

with SPICE2, a number may have trailing alphabetic characters after it, which can indicate the units.

See the description of the let command for more information. The compose command can also be used to assign values to a vector.

Vectors typically have defined units. The units are carried through a computation, and simplified when the result is generated. Presently, the system can not handle fractional powers. The units of a vector can be set with the settype command.

Vectors also posses a dimensionality. A scalar is a vector of the lowest dimensionality. Most vectors are one-dimensional lists of numbers. Certain types of analysis produce multidimensional vectors, which are analogous to arrays. This dimensionality is indicated when the vectors are listed with the display command or the let command without arguments. Plotting a multidimensional vector will produce a family of traces. Elements and sub-dimensional vectors are specified with multiple square brackets, with the bracket on the right having the lowest dimensionality.

For example, one might issue the following command:

```
.ac dec 10 1Hz 1Mhz dc v1 0 2 .1 v2 4.5 5.5 .25
```

which will perform an ac analysis with the dc sources `v1` and `v2` stepped through the ranges 0–2 step .1 for `v1`, 4.5–5.5 step .25 for `v2`. The resulting output vectors will have dimensions [5,21,61], i.e., 5 values for `v2`, 21 for `v1`, and 61 for the ac analysis. Typing "`plot v(1)`" (for example) would plot all 21*5 analyses on the same scale (this would not be too useful). However, one can plot subranges by entering, for example, "`plot v(1)[1]`" which would plot the results for `v2` = 4.75, or "`plot v(1)[1][2]`" for `v2` = 4.75, `v1` = .2, etc. Range specifications also work, for example "`plot v(1)[2][0,2]`" plots the values for `v2` = 5, `v1` = 0, .1, .2. The memory space required to hold the multidimensional plot data can grow quite large, so one should be reasonable.

An expression is an algebraic combination of already defined vectors, scalars (a scalar is a vector of length 1), constants, operators and functions. Some examples of expressions are:

```
cos(time) + db(v(3))
sin(cos(log(10)))
TIME*rnd(v(9)) - 15*cos(vin#branch)^7.9e5
not ((ac3.freq[32] & tran1.time[10]) gt 3)
```

Vectors can be evaluated by the shell parser by adding the prefix `$&` to the vector's name. This is useful, for example, when the value of a vector needs to be passed to the shell's echo command, or in circuit description files where vectors are to be evaluated by the shell as the file is read. Similar to the shell constructs, `$?&`*word* expands to 1 if *word* is a defined vector, 0 otherwise. Also `$#&`*word* expands to the length of *word* if *word* is a defined vector, or 0 if not found. Additionally, the notation `$&(`*vector expression*`)` is replaced by the value of the vector expression. A range specification can be added, for example `echo $&(a+1)[2]` prints the third entry in `a+1`, or 0 if out of range. If white space exists in the `$&(...)` construct, it probably should be quoted. Finally, the shell recognizes the construct `$&v($`*something*`)` as a reference to a SPICE node voltage, so that one can index node voltages as `echo $&v($&i)`, for example. A range specification can be added, which can contain shell variables. This is true for both vectors (`$&` prefix) and variables.

Vector expressions can also contain calls to the built-in "tran" functions ordinarily used in voltage/current source specifications in transient analysis. These are the `pulse()`, `pwl()`, etc. functions described in 2.12.3. If assigned to a vector, the vector will have a length equal to the current scale (e.g., the time values of the last transient analysis plot), and be filled in with values just as if the analysis was run with the given source specification. For example

```
(run transient analysis: tran .1n 10n)
let a = pulse(0 1 1n 1n)
```

Vector a will have length 101 and contain the pulse values.

There are three such functions, sin(), exp(), and gauss(), that have the same names as math functions. The math functions always return data of the same length as the argument(s), and take 1 argument for sin, exp and 2 for gauss. When one of these names is encountered in an expression, *WRspice* counts the arguments. If the number of arguments is 1 for sin/exp or 1 or 2 for gauss, the math function is called, otherwise the tran function is called. It may be necessary to give the gauss function a phony additional argument to force calling the tran function.

3.13.1 The constants Plot

The following values are defined in a plot named "constants". This is the default plot if no rawfile has been loaded and no simulation has been run. These constants are visible no matter what the current plot is, but they are overridden by a vector with the same name in the current plot. The constants plot can not be deleted, and its vectors are read-only. The values are in MKS units.

boltz	Boltzmann's constant (1.38062e-23 joules/degree kelvin)
c	The speed of light (2.997925e8 meters/second)
e	The base of natural logarithms (2.71828182844590452353)
echarge	The charge on an electron (1.60219e-19 coulombs)
false	False value (0)
i	The square root of -1, can be expressed as (0,1)
kelvin	Absolute 0 in Centigrade (-273.15 degrees)
no	False value (0)
pi	π (3.14159265358979323846)
planck	Planck's constant (6.62620e-34 joule-seconds)
true	Truth value (1)
yes	Truth value (1)

3.13.2 Operators in Expressions

The operations available in vector expressions are listed below. They all take two operands, except for unary minus and negation.

addition operator: +
Add the two operands.

subtraction amd negation operator: −
Evaluates to the first argument minus the second, and also may be used as unary minus. In commands that take a list of vectors as arguments, such as plot, one can force a unary minus with '−−', for example:

```
plot v(1) −v(4)
```

would plot v(1)-v(4) (i.e., the difference) whereas

```
plot v(1) −−v(4)
```

would plot `v(1)` and −`v(4)` (two traces).

multiply operator: ∗
Multiply the two operands.

divide operator: /
The first operand divided by the second.

modulo operator: %
This operates in the manner of the C `fmod()` function, returning the remainder. That is, for `x`%`y`, the value of `x`−`i`∗`y` is returned for some integer `i` such that the result has the same sign of `x` and magnitude less than the magnitude of `y`. An error is indicated if `y` is zero. If `x` or `y` is complex, the magnitudes are used in the division.

power operator: ^
Evaluates to the first operand raised to the power of the second.

and operator: & or `and`
Evaluates to 1 if both operands are non-zero, 0 otherwise.

or operator: | or `or`
Evaluates to 1 if either of the two operands is nonzero, 0 otherwise.

not operator: ~ or `not`
Evaluates to 1 if the operand is 0, 0 otherwise.

greater-than operator: > or `gt`
Evaluates to 1 if the first operand is greater than the second, 0 otherwise.

greater-than-or-equal operator: >= or `ge`
Evaluates to 1 if the first operand is greater than or equal to the second, 0 otherwise.

less-than operator: < or `lt`
Evaluates to 1 if the first argument is less than the second, 0 otherwise.

less-than-or-equal operator: <= or `le`
Evaluates to 1 if the first argument is less than or equal to the second, 0 otherwise.

not-equal operator: <> or `ne`
Evaluates to 1 if the two operands are not equal, 0 otherwise.

equal operator: = or `eq`
Evaluates to 1 if both operands are equal, 0 otherwise.

comma operator: `,`
The notation *a*,*b* refers to the complex number with real part *a* and imaginary part *b*. Such a construction may not be used in the argument list to a macro function, however, since commas are used to separate the arguments and parentheses may be ignored. The expression $a + \mathtt{j}(b)$ is equivalent. The comma does *not* behave as an operation (return a value) as it does in C.

The logical operations are & (and), | (or), ~ (not), and their synonyms. A nonzero operand is considered "true". The relational operations are <, >, <=, >=, =, and <> (not equal), and their synonyms. If used in an algebraic expression they work like they would in C, producing values of 0 or 1. The synonyms are useful when < and > might be confused with IO redirection (which is almost always).

Vectors may be indexed by *value*[*index*] or *value*[*low*,*high*].

The first notation refers to the *index*'th element of *value*. The second notation refers to all of the elements of *value* which fall between the *high*'th and the *low*'th element, inclusive. If *high* is less than *low*, the order of the elements in the result is reversed. Note that a complex index will have the same effect as using the real part for the lower value and the imaginary part for the upper, since this is the way the parser reads this expression. Multi-dimensional vectors are referenced as Vec[*ind*N][*ind*N-1]...[*ind*0], where each of the *ind*I can be a range, or single value. The range must be within the vector's spanning space. If fewer than the vector's dimensions are specified, the resulting object is a sub-dimensional vector.

Finally, there is the ran operator: *value1*[[*value2*]] or *value*[[*low,high*]].

The first notation refers to all the elements of *value1* for which the element of the corresponding scale equals *value2*. The second notation refers to all of the elements of value for which the corresponding elements of the scale fall between *high* and *low*, inclusive. If *high* is less than *low*, the order of the elements in the result is reversed.

3.13.3 Math Functions

There are a number of built-in math functions which take and return vectors. Generally, these functions operate on the supplied vector term-by-term, returning a vector of the same length as that given.

The pre-defined functions available are listed below. In general, all operations and functions will work on either real or complex values, providing complex data output when necessary.

It should be noted that the mathematics subsystem used to evaluate expressions in voltage/current sources is completely different. In that subsystem, functions take real valued scalars as input. Although many of the same functions are available in both systems, the correspondence is not absolute.

`abs(`*vector*`)`
: Each point of the returned vector is the absolute value of the corresponding point of *vector*. This is the same as the `mag` function.

`acos(`*vector*`)`
: Each point of the returned vector is the arc-cosine of the corresponding point of *vector*. This and all trig functions operate with radians unless the `units` variable is set to `degrees`.

`acosh(`*vector*`)`
: Each point of the returned vector is the arc-hyperbolic cosine of the corresponding point of *vector*. This and all trig functions operate with radians unless the `units` variable is set to `degrees`.

`asin(`*vector*`)`
: Each point of the returned vector is the arc-sine of the corresponding point of *vector*. This and all trig functions operate with radians unless the `units` variable is set to `degrees`.

`asinh(`*vector*`)`
: Each point of the returned vector is the arc-hyperbolic sine of the corresponding point of *vector*. This and all trig functions operate with radians unless the `units` variable is set to `degrees`.

`atan(`*vector*`)`
: Each point of the returned vector is the arc-tangent of the corresponding point of *vector*. This and all trig functions operate with radians unless the `units` variable is set to `degrees`.

`atanh(`*vector*`)`
: Each point of the returned vector is the arc-hyperbolic tangent of the corresponding point of *vector*. This and all trig functions operate with radians unless the `units` variable is set to `degrees`.

`cbrt(`*vector*`)`
Each point of the returned vector is a cube root of the corresponding point of *vector*.

`ceil(`*vector*`)`
This function return the smallest integer greater than or equal to the argument, in the manner of the C function of the same name. If the argument is complex, the operation is performed on both components, with the result being complex. This operation is performed at each point in the given *vector*.

`cos(`*vector*`)`
Each point of the returned vector is the cosine of the corresponding point of *vector*. This and all trig functions operate with radians unless the `units` variable is set to `degrees`.

`cosh(`*vector*`)`
Each point of the returned vector is the hyperbolic cosine of the corresponding point of *vector*. This and all trig functions operate with radians unless the `units` variable is set to `degrees`.

`db(`*vector*`)`
Each point of the returned vector is the decibel value (`20 * log10(mag())`) of the corresponding point of *vector*.

`deriv(`*vector*`)`
This calculates the derivative of the given *vector*, using numeric differentiation by interpolating a polynomial. However, it may be prone to numerical errors, particularly with iterated differentiation. The implementation only calculates the derivative with respect to the real component of that vector's scale. The polynomial degree used for differentiation can be specified with the `dpolydegree` variable. If `dpolydegree` is unset, the value taken is 2 (quadratic). The valid range is 0-7.

`erf(`*vector*`)`
Each point of the real returned vector is the error function of the corresponding real point of *vector*.

`erfc(`*vector*`)`
Each point of the real returned vector is the complementary error function of the corresponding real point of *vector*.

`exp(`*vector*`)`
Each point of the returned vector is the exponentiation (e^x) of the corresponding point of *vector*.

`fft(`*vector*`)`
The `fft` function returns the Fourier transform of *vector*, using the present scale of *vector*. The scale should be linear and monotonic. The length is zero-padded to the next binary power. Only the real values are considered in the transform, so that the negative frequency terms are the complex conjugates of the positive frequency terms. The negative frequency terms are not included in the (complex) vector returned. A scale for the returned vector is also generated and linked to the returned vector.

`floor(`*vector*`)`
This function return the largest integer less than or equal to the argument, in the manner of the C function of the same name. If the argument is complex, the operation is performed on both components, with the result being complex. The operation is performed at each point of the argument.

`gamma(`*vector*`)`
Each point of the real returned vector is the gamma function of the corresponding real point of *vector*.

`gauss(`*vector*`)`
The result is a real vector of Gaussian random numbers, with a term-by-term standard deviation and mean of the corresponding real and imaginary parts of *vector*. If *vector* is real, the means are 0. The random number sequence can be reset with the `seed` command.

`ifft(`*vector*`)`
The `ifft` function returns the inverse Fourier transform of *vector*, using the present scale of *vector*. The scale should be linear and monotonically increasing, starting at 0. Negative frequency terms are assumed to be complex conjugates of the positive frequency terms. The length is zero-padded to the next binary power. A scale for the returned vector is also generated and linked to the returned vector. The returned vector is always real.

`im(`*vector*`)`
Each point of the real returned vector is the imaginary part of the corresponding point of the given *vector*. This function can also be calld as "`imag`()".

`int(`*vector*`)`
The returned value is the nearest integer to the argument, in the manner of the C `rint` function. If the argument is complex, the operation is performed on each component with the result being complex. The operation is performed at each point in the argument.

`integ(`*vector*`)`
The returned vector is the (trapezoidal) integral of *vector* with respect to the *vector*'s scale (which must exist).

`interpolate(`*vector*`)`
This function takes its data and interpolates it onto a grid which is determined by the default scale of the currently active plot. The degree is determined by the `polydegree` variable. This is useful if the argument belongs to a plot which is not the current one. Some restrictions are that the current scale, the *vector*'s scale, and the argument must be real, and that either both scales must be strictly increasing or strictly decreasing if they are different.

`j(`*vector*`)`
Each point of the returned vector is the corresponding point of *vector* multiplied by the square root of -1.

`j0(`*vector*`)`
Each point of the real returned vector is the Bessel order 0 function of the corresponding real point of *vector*. Unlike most of the functions, this function operates only on the real part of a complex argument, and always returns a real valued result.

`j1(`*vector*`)`
Each point of the real returned vector is the Bessel order 1 function of the corresponding real point of *vector*. Unlike most of the functions, this function operates only on the real part of a complex argument, and always returns a real valued result.

`jn(`*n*, *vector*`)`
Each point of the real returned vector is the Bessel order *n* function of the corresponding real point of *vector*. This function takes two parameters, unlike most other functions. The first is the function order, which is truncated to an integer, and the second is the function argument. If a (single) complex vector is given as an argument, then for each element the real part is taken as the function order and the imaginary part is the function argument. The return is always real-valued.

`length(`*vector*`)`
This function returns the scalar length of *vector*.

`ln(`*vector*`)`
Each point of the returned vector is the natural logarithm of the corresponding point of *vector*.

`log(`*vector*`)`
Each point of the returned vector is the base-10 logarithm of the corresponding point of *vector*.

`mag(`*vector*`)`
Each point of the real returned vector is the magnitude of the corresponding point of *vector*.

`mean(`*vector*`)`
This function returns is a scalar (a length 1 vector) that is the mean of the elements of *vector*.

`norm(`*vector*`)`
Each point of the returned vector is the corresponding point of the given vector multiplied by the magnitude of the inverse of the largest value in the given vector. The returned vector is therefor normalized to 1 (i. e, the largest magnitude of any component will be 1).

`ph(`*vector*`)`
Each point of the real returned vector is the phase in radians of the corresponding point of *vector*.

`pos(`*vector*`)`
This function returns a real vector which is 1 if the corresponding element of the argument has a non-zero real part, and 0 otherwise.

`re(`*vector*`)`
Each point of the real returned vector is the real part of the corresponding point of *vector*. The function can also be called as "`real`()".

`rms(`*vector*`)`
This function integrates the magnitude-squared of *vector* over the *vector*'s scale (using trapezoidal integration), divides by the scale range, and returns the square root of this result. If the *vector* has no scale, the square root of the sum of the squares of the elements is returned.

`rnd(`*vector*`)`
This function returns a vector which contains random values between 0 and the corresponding element of *vector*. If *vector* is complex then the random value is also complex. The random number sequence can be reset with the `seed` command.

`sin(`*vector*`)`
Each point of the returned vector is the sine of the corresponding point of *vector*. This and all trig functions operate with radians unless the `units` variable is set to `degrees`.

`sinh(`*vector*`)`
Each point of the returned vector is the hyperbolic sine of the corresponding point of *vector*. This and all trig functions operate with radians unless the `units` variable is set to `degrees`.

`sqrt(`*vector*`)`
Each point of the returned vector is the square root of the corresponding point of *vector*.

`sum(`*vector*`)`
The return value is a scalar which is the sum of the elements of the vector *vector*.

`tan(`*vector*`)`
Each point of the returned vector is the tangent of the corresponding point of *vector*. This and all trig functions operate with radians unless the `units` variable is set to `degrees`.

`tanh(`*vector*`)`
Each point of the returned vector is the hyperbolic tangent of the corresponding point of *vector*. This and all trig functions operate with radians unless the `units` variable is set to `degrees`.

`unitvec(`*vector*`)`
This function returns a real vector consisting of all 1's, with length equal to the magnitude of the first element of the argument.

`vector(`*vector*`)`
This function returns a vector consisting of the integers from 0 up to the magnitude of the first element of its argument.

`y0(`*vector*`)`
Each point of the real returned vector is the Neumann order 0 function of the corresponding real point of *vector*. Unlike most of the functions, this function operates only on the real part of a complex argument, and always returns a real valued result.

`y1(`*vector*`)`
Each point of the real returned vector is the Neumann order 1 function of the corresponding real point of *vector*. Unlike most of the functions, this function operates only on the real part of a complex argument, and always returns a real valued result.

`yn(`*n*, *vector*`)`
Each point of the real returned vector is the Neumann order *n* function of the corresponding real point of *vector*. This function takes two arguments, unlike most other functions. The first is the function order, which is truncated to an integer, and the second is the function argument. If a (single) complex vector is given as an argument, then for each element the real part is taken as the function order and the imaginary part is the function argument. The return is always real-valued.

3.13.4 Expression Lists

Some commands, such as print and plot, take expression lists as arguments. In the simplest form, an expression list is a space-separated list of vectors. In the general form, an expression list is a sequence of expressions involving vectors. The parsing is context dependent, i.e., white space does not necessarily terminate an expression. This leads to ambiguities. For example, the command

```
plot v(2) -v(3)
```

can be interpreted as two vectors, or as a single vector representing the difference. *WRspice* will assume the latter.

There are two ways to ensure that the former interpretation prevails. First, two minus signs "−−" can be used in the second vector. This will enforce interpretation as a unary minus, with which the parser will recognize two separate vectors. Second, and more generally, double quotes around an expression will enforce interpretation as a separate vector, but only if white space precedes the opening double quote. Thus, the correct form is

```
plot v(2) "-v(3)"
```

There are situations where the name of a vector is so strange that it can't be accessed in the usual way. For example, if a list-type special variable is saved with the save command, the plot may contain a vector with a name like "b1[ic,0]". To access this vector, one can't simply type the name, since the name is an expression which will actually lead to an evaluation error. One has to fool the expression parser into taking the name as a string. This will happen if the name is not the lead in a token and the name is double quoted. If the name is the leading part of a token, it should be backslash-double-double quoted.

To use the double quotes to enforce string interpretation, one should have, for example,

```
plot v(2) \""b1[ic,0]"\"
```

The extra set of quotes is needed only if the string is at the start of a token, thus

```
plot 2*"b1[ic,0]"
```

is ok. This may be a bit confusing, but this feature is seldom used, and a bit of experimentation will illustrate the behavior.

These commands can accept the *plotname.vecname* notation, where either field may be the wildcard "all". If the plotname is all, matching vectors from all plots are specified, and if the vector name is all, all vectors in the specified plots are referenced. The constants plot is never matched by a plot wildcard. Note that you may not use binary operations on expressions involving wildcards - it is not obvious what "all + all" should denote, for instance.

3.13.5 Set and Let

Novice *WRspice* users may be confused by the different interpretations of shell variables and vectors. Any variable can be defined with the set command, and undefined with unset. If defined, the value of the variable is the string, if given. For example, if

```
set a = 10*2
```

is entered, the value of a (obtained as $a) is the string "10*2" and *not* the integer 20.

Some internally used variables have boolean values, such as

```
set unixcom
```

which if set causes certain modes or functions to be active.

Vectors, however, always have numeric values, and can be created with let and compose, and deleted with unlet. If one enters

```
let a = 10*2, or more simply
a = 10*2
```

the value of the vector a is 20.

Vectors can be set to shell variables, in which case they take on the interpreted numerical values. For example,

```
set a=10*2
b = $a
```

would assign the string `10*2` to the shell variable `a`, but the vector `b` would contain the value `20`.

The inputs to most commands are vectors, however some commands, such as `echo`, substitute for shell variables. For example,

```
set a = "foo"
set b = "bar"
echo $a$b
```

would print "`foobar`".

Shell variables are expanded by `echo`, and in *WRspice* input when sourced. If the value of a vector is needed in shell expansion, then the special prefix `$&` should be added. This tells the shell interpreter that the following symbol is a vector, to be replaced by its value. For example,

```
let a = 2.0e-2
echo $&a
```

will print `2.00000e-2`. However

```
let a = 2.0e-2
echo $a
```

would give an error message (unless `a` is also a shell variable), and

```
let a = 2.0e-2
echo a
```

would print "`a`".

Double quotes will cause multiple tokens to be taken as one, for example

```
set a = "a string"
```

will set `a` accordingly, whereas

```
set a = a string
```

will set shell variable `a` to "`a`" and shell variable `string` to boolean true.

Single quotes do about the same thing, but suppress shell variable expansion. For example:

```
set a = foo
set b = bar
echo $a $b
```

and

```
set a = foo
set b = bar
echo "$a $b"
```

would print "`foo bar`", whereas

```
set a = foo
set b = bar
echo '$a $b'
```

would print "`$a $b`".

In the present version, `$` can not be nested. For example,

```
set a = foo
set b = bar
set c = '$a$b'
echo $c
```

prints "`$a$b`", not "`foobar`". However,

```
set a = foo
set b = bar
set c = $a$b
echo $c
```

does print "`foobar`" (the value of `c`).

Shell variables that are lists are referenced with zero-based index, for example

```
set a = ( aa bb cc )
echo $a[1]
```

prints "`bb`".

Actually, what can be in the brackets is `[`*lo*`-`*hi*`]`, where *lo* defaults to 0 and *hi* defaults to the length - 1. If $lo > hi$, the list is reversed.

If the reference is to a vector, as in

```
compose a values .1 .2 .3
echo $&a[1]
```

the index is also zero-based, so "`2.0000e-1`" is printed.

The `[]` subscripting is interpreted a little differently by the shell and by the vector parser. If a variable starts with `$`, as in `$&value[]`, the `[]` is interpreted by the shell parser. In this case, the terms inside `[]` must be interpreted as shell variables, with the (optional) *low-high* notation. In a vector expression, i. e. , one using `value[]`, the terms inside `[]` will be interpreted as vector expressions, with the optional *low*,*high* notation. Thus,

```
if (value[index] = 0)
```

is perfectly legal for vectors `value` and `index`. Also, equivalently,

```
if ($&value[$&index] = 0)
```

is also ok, though not as efficient. However

```
if ($&value1[index] = 0)
```

is an error, as the shell parser does not know that `index` is a vector.

Shell variables can be used freely in vector expressions, however one must keep in mind how the variables are interpreted. During parsing, the shell variables are evaluated, and their values put back into the expression as constants. Then the expression is evaluated as a vector expression.

3.14 Control Structures

Control structures operate on expressions involving vectors, constants, and ($-substituted) shell variables. A non-zero result (of any element, if the length is greater than 1) indicates "true". The following control structures are available:

repeat [*number*]
statement
...
`end`

Execute the statements in the block defined by the repeat line and the corresponding end statement *number* times, or indefinitely if no *number* is given. The *number* must be a constant, or a shell variable reference that evaluates to a constant, which may be a vector reference in the `$&` form. A vector name is not valid.

`while` *condition*
statement
...
`end`

The while line, together with a matching end statement, defines a block of commands that are executed while the *condition* remains true. The *condition* is an expression which is considered true if it evaluates to a nonzero value, or if a vector, any component is nonzero. The test is performed at the top of the loop, so that if the *condition* is initially false, the statements are not executed.

`dowhile` *condition*
statement
...
`end`

The dowhile line, together with a matching end statement, defines a block of commands that are executed while the *condition* remains true. The *condition* is an expression which is considered true if it evaluates to a nonzero value, or if a vector, any component is nonzero. Unlike the while statement, the test is performed at the bottom of the loop – so that the loop executes at least once.

```
foreach var value ...
statement
...
end
```

The foreach statement opens a block which will be executed once for each *value* given. Each time through, the *var* will be set to successive *value*s. After the loop is exited it will have the last value that was assigned to it. The *var* can be accessed in the loop with the `$var` notation, i.e., it should be treated as a shell variable, not a vector. This is set to each *value* as a text item.

```
if condition
statement
...
else
statement
...
end
```

If the *condition* is non-zero then the first set of statements is executed, otherwise the second set. The `else` and the second set of statements may be omitted.

```
label labelname
```

This defines a label which can be used as an argument to a goto statememt.

```
goto label
```

If there is a label statement defining the *label* in the block or an enclosing block, control is transfered there. If the goto is used outside of a block, the label must appear ahead of the goto (i.e., a forward `goto` may occur only within a block). There is a begin macro pre-defined as "`if 1`" which may be used if forward label references are required outside of a block construct.

```
continue [n]
```

If there is a `while`, `dowhile`, `foreach` or repeat block enclosing this statement, the next iteration begins immediately and control passes to the top of the block. Otherwise an error results. If a *number* is given, that many surrounding blocks are continued. If there are not that many blocks, an error results.

```
break [n]
```

If there is a `while`, `dowhile`, `foreach`, or repeat block enclosing this statement, control passes out of the block. Otherwise an error results. If a *number* is given, that many surrounding blocks are exited. If there are not that many blocks, an error results.

```
end
```

This statement terminates a block. It is an error for an end to appear without a matching if, while, dowhile, foreach, or repeat statement. The keywords `endif`, `endwhile`, `enddowhile`, `endforeach`, and `endrepeat` are internally aliased to `end`.

Control structures may be nested. When a block is entered and the input is from the keyboard, the prompt becomes a number of >'s equalling the depth of blocks the user has entered. The current control structures may be examined with the debugging command cdump.

3.15 Scripts and Batch Mode

Scripts can be written to automate a large number of runs on a circuit, saving the output in a sequence of rawfiles. Typically this is done in the background, using *WRspice* in batch or server modes. This section addresses some of the subtleties of using these modes.

A pure script file, i.e., one which does not include a circuit description, consists of an unread "title" line, followed by a control block. The control block begins with a "`.control`" line, continues with one or more executable statements, and terminates with a "`.endc`" line. In *WRspice*, an "`.exec`" line can be used rather than the `.control` line, though for backward compatibility with SPICE3, it is recommended that the traditional `.control` be used. The executable statements are any statements understandable to the *WRspice* shell. Typically, such statements appear just as they would be entered on the command line if given as text input. Scripts can call other scripts to any depth.

Before a script is read, the variables `argc` and `argv` are set to the number of words on the command line, and a list of those words respectively. Their previous values (if any) are pushed onto a stack, and popped back in place when the script terminates. Thus, within a command script, these predefined variables are available for use in the script.

If *WRspice* is run with the `-b` flag, it will process the circuit in batch mode, similar to SPICE2. The files are input on the command line, and if no files are listed, the standard input is read. Most of the control lines recognized by SPICE2 will be handled, including `.plot`, `.print`, and `.four`. The output will be a margin analysis file if a margin run is input, otherwise if the `-r` option is used (`-r` *rawfilename*), a rawfile is produced. Otherwise, output is sent to the standard output, in the form of ascii plots and print output as directed by `.plot/.print` lines. The format of this output is somewhat different from SPICE2, however, and much less information is available from an operating point analysis. Some SPICE2 options are not supported.

Server mode is similar to batch mode, however input is taken only from the standard input, and output is exclusively to the standard output. The output is either in rawfile or margin analysis format, and inappropriate command line options such as `-r`, `-b` are ignored.

In server mode, lines in a control file such as

```
.control
cktfile
tran 1p 100p
...
```

will generate raw output (on stdout). The same lines in batch mode will produce nothing, unless followed by a `write` *rawfilename* line. Similarly, a `run` *rawfilename* will perform the analysis in the circuit (*not* the control) file, and create *rawfilename* in batch mode. In server mode, the output is to stdout, and *rawfilename* is ignored. In a control file, `run` will generate raw data on stdout in server mode, but will do nothing in batch mode unless followed by a write command. This is true whether or not the `-r` option is

used in invoking *WRspice*. However, if the input file is a circuit file, and there are `.plot/.print` lines, if `-r` is not given, the ascii plots and prints will be generated on stdout in batch mode.

Examples:

circuit file `(cktfile)`:

```
title
.tran 1p 100p
.plot tran v(1)
...
.end
```

control file `(ctrlfile)`:

```
blank line
.control
cktfile
run
.endc
```

`wrspice -b cktfile`
generates an ascii plot on stdout.

`wrspice -b -r raw cktfile`
generates a rawfile named "raw".

`wrspice -s <cktfile`
generates a rawfile on stdout.

`wrspice -s <ctrlfile`
generates a rawfile on stdout.

`wrspice -b ctrlfile`
generates nothing.

`wrspice -b -r raw ctrlfile`
generates nothing.

control file `(ctrlfile)`:

```
blank line
.control
cktfile
run f1
.endc
```

`wrspice -s <ctrlfile`
generates a rawfile on stdout (tran 2p 50p), ignores file f1.

`wrspice -b -r xxx ctrlfile`
puts tran 1p 100p in rawfile f1, ignores xxx.

control file (ctrlfile):
blank line

.control
cktfile
tran 2p 50p
.endc

`wrspice -s <ctrlfile`
generates a rawfile on stdout (tran 2p 50p).

`wrspice -b -r xxx ctrlfile`
generates nothing.

control file (ctrlfile):
blank line
.control
cktfile
tran 2p 50p
write f1
.endc

`wrspice -s <ctrlfile`
generates a rawfile on stdout (tran 2p 50p), puts constants in file f1.

`wrspice -b [-r xxx] ctrlfile`
puts tran 2p 50p in rawfile f1. If -r xxx was specified and f1 not, then file xxx would be used. If neither file was specified, "rawspice.raw" would be used.

3.16 The *WRspice* Daemon and Remote SPICE Runs

WRspice can be accessed and run from a remote system for asynchronous simulation runs, for assistance in computationally intensive tasks such as Monte Carlo analysis, and as a simulator for the *Xic* graphical editor. This is made possible through a daemon (background) process which controls *WRspice* on the remote machine. The daemon has the executable name "wrspiced", and should be run as a root process on the remote machine. Typically, this can be initiated in the system startup script, or manually. Of course, the remote machine must have a valid *WRspice* executable present.

The `wrspiced` program is described in B.4.

Chapter 4

WRspice Commands

When a line is entered, it is interpreted as one of several things. First, it may be an alias, in which case the line is replaced with the result after alias substitution, and the line is re-parsed. Second, it may be the name of a codeblock, which is a user-defined command obtained from a script file, in which case the codeblock is executed. Third, it may be a pre-defined command, in which case it is executed. Fourth, it may be an assignment statement, which consists of a vector name, an '=' symbol, and an expression, in which case it is executed as if it were preceded by the word "`let`". Fifth, it may be the name of a circuit file, in which case it is loaded as if with a source command, or it may be the name of a command script – *WRspice* searches the current `sourcepath` (search path) for the file and executes it when it is found. The effect of this is identical to the effect of "`source` *file*", except that the variables `argc` and `argv` are set. Finally, it may be a UNIX command, in which case if the variable `unixcom` is set, it is executed as though it were typed to the operating system shell.

The following subsections discuss the built-in commands and variables.

4.1 Shell Commands

The commands listed below are built into the *WRspice* shell, or control shell operation.

Shell Commands	
alias	Create alias
cd	Change directory
echo	Print string
history	Print command history
mapkey	Create keyboard mapping
pause	Pause script execution
rehash	Update command database
set	Set a variable
shell	Execute operating system commands
shift	Shift argument list
unalias	Destroy alias
unset	Unset a variable
usrset	Print list of internally used variables

4.1.1 alias

The alias command is used to create aliases, as in the C-shell.

```
alias [word] [text]
```

The alias command causes *word* to be aliased to *text*. Whenever a command line beginning with *word* is typed, *text* is substituted. Arguments are either appended to the end, or substituted in if history characters are present in the text. With no argument, a list of the current aliases is displayed.

In the body of the alias text, any strings of the form `!:`*number* are replaced with the *number*'th argument of the actual command line. Note that when the alias is defined with the alias command, these strings must be quoted to prevent history substitution from replacing the `!`'s before the alias command can get to them. Thus the command

```
alias foo echo '!:2' '!:1'
```

causes "`foo bar baz`"" to be replaced with "`echo baz bar`". Other `!` modifiers as described in the section on history substitution may also be used, always referring to the actual command line arguments given. If a command line starts with a backslash '\' any alias substitution is inhibited.

4.1.2 cd

The cd command is used to change the current working directory.

```
cd [directory]
```

The command will change the current working directory to *directory*, or to the user's home directory if none is given.

4.1.3 echo

The echo command will print its arguments on the standard output.

```
echo [-n][stuff ...]
```

If the `-n` option is given, then the arguments are echoed without a trailing newline.

4.1.4 history

The history command prints the last commands executed.

```
history [-r] [number]
```

The command will print out the last *number* commands typed by the user, or all the commands saved if *number* is not given. The number of commands saved is determined by the value of the `history` variable. If the `-r` flag is given, the list is printed in reverse order.

4.1.5 mapkey

The mapkey command provides limited keyboard mapping support.

`mapkey [ -r [`*filename*`] | -w [`*filename*`] | `*keyname data*` ]`

Only the keys that are used for command line editing are mappable. This is to account for "strange" terminals that may not send the expected data when a key is pressed.

The following keys can be mapped:

Ctrl-A
Ctrl-D
Ctrl-E
Ctrl-K
Ctrl-U
Ctrl-V
Tab
Backspace
Delete
LeftArrow
RightArrow
UpArrow
DownArrow

Of these, the arrow keys and **Delete** are most likely to need remapping.

If no argument is given, the user is prompted to press each of these keys, and the internal map is updated. After doing this, the keys should have their expected effect when pressed while entering a *WRspice* command.

If "`-w` [*filename*]" is given, the present internal map will be saved in the named file, or "`wrs_keymap`" in the current directory if no *filename* is given.

If "`-r` [*filename*]" is given, the file will be read as a key mapping file, and the internal map will be updated. The *filename*, if not given, defaults to "`wrs_keymap`". If no path is given, it will be found in the current directory or the startup directory.

If "*keyname data...*" is given, a single key in the internal map can be updated. The format is the same as the entries in the mapping file, i.e., one of the names above, followed by one or more hex bytes of data. The bytes represent the stream sent when the named key is pressed, and will henceforth be interpreted as the pressing of that key. The bytes should be in hex format, and the first byte of a multi-byte sequence must be the **Escape** character (1b).

Example (from real life):

After installing the latest X-window system, suppose one finds that, when running *WRspice* in an `xterm` window, the **Delete** key no longer deletes the character under the cursor in *WRspice*, but instead injects some gibberish. There are three ways to fix this. The first two are specific to the `xterm` program, and instruct the `xterm` to send the ascii Del character when **Delete** is pressed, rather than use the new default which is to send the VT-100 "delete character" string. The third method is to map this string into the delete function in *WRspice*.

1. From the main `xterm` menu, find and click on the "**Delete is DEL**" entry. Usually, holding the **Ctrl** key and clicking in the `xterm` with button 1 displays this menu.

2. Create a file named "XTerm" in your home directory, containing the line

```
*deleteIsDEL: true
```

3. In *WRspice*, type "mapkey" and follow the prompts. You can save the new map, and add a line to your .wrspiceinit file to read it when *WRspice* starts.

4.1.6 pause

The pause command is used in scripts to cause the executing script to wait for a keypress. The function takes no arguments, and the keypress is discarded.

4.1.7 rehash

The rehash command rebuilds the command list from the files found along the user's executable file search path. The command will recalculate the internal hash tables used when looking up operating system commands, and make all operating system commands in the user's PATH available for command completion. This command takes no arguments, and has effect only when the unixcom variable is set.

4.1.8 set

The set command allows the user to examine and set shell variables.

```
set [varname [= value] ...]
```

In addition, shell variables are set which correspond to definitions supplied on the .options line of the current circuit, and there are additional shell variables which are set automatically in accord with the current plot. The shell variables that are currently active can be listed with the set command given without arguments, and are also listed within the **Variables** window brought up from the **Tools** menu of the Tool Control window. In these listings, a '+' symbol is prepended to variables defined from a .options line in the current circuit, and a '*' symbol is prepended to those variables defined for the current plot. These variable definitions will change as the current circuit and current plot change. Some variables are read-only and may not be changed by the user, though this is not indicated in the listing.

Before a simulation starts, the options from the .options line of the current circuit are merged with any of the same name that have been set using the shell. The result of the merge is that options that are booleans will be set if set in either case, and those that take values will assume the value set through the shell if conflicting definitions are given. The merge will be suppressed if the shell variable noshellopts is set *from the shell*, in which case the only options used will be those from the .options line, and those that are redefined using the set command will be ignored.

Above, the *varname* is the name of the shell variable to set, and *value*, if present, is a single token to be assigned. Multiple variables can be assigned with a single set command. If *value* is missing (along with the '='), then *varname* is of boolean type and always taken as "true" when set. If *value* is a pure number not double quoted, then *varname* will reference that number. Otherwise, *varname* will reference *value* as a character string, unless *value* is a list. A list is a space-separated list of tokens in space-separated parentheses, as in

```
set mylist = ( abc def 1.2 xxdone )
```

which sets the variable `mylist` to the list of four tokens. The unset command can be used to delete a variable.

The value of a variable *word* may be inserted into a command by writing `$`*word*. If a variable is set to a list of values that are enclosed in parentheses (which must be separated from their values by white space), the value of the variable is the list. The **Variables** button in the **Tools** menu also displays the variables which are currently set. There are quite a number of variables with internal meaning to *WRspice*, and in fact this is the mechanism by which most *WRspice* defaults are specified. Several of the other buttons in the **Tools** menu, including **Commands**, **Debug**, **Plot Defs**, **Shell**, and **Sim Defs** bring up panels from which these special variables can be modified.

The predefined variables which have meaning to *WRspice* (see 4.7) can be listed with the usrset command. In general, variables set in the `.options` line are available for expansion in `$`*varname* references, but do not otherwise affect the workings of the shell.

4.1.9 shell

The shell command will pass its arguments to the operating system shell.

`shell [`*command*`]`

The command will fork a shell if no *command* is given, or execute the arguments as a command to the operating system.

4.1.10 shift

The shift command facilitates handling of list variables in shell scripts.

`shift [`*varname*`] [`*number*`]`

If *varname* is the name of a list variable, it is shifted to the left by *number* elements, i.e., the *number* leftmost elements are removed. The default *varname* is `argv`, and the default *number* is 1.

4.1.11 unalias

The unalias command is used to remove aliases previously set with the alias command.

`unalias [`*word* `...]`

The command removes any aliases associated with each of the *word*s. The argument may be "`*`", in which case all aliases are deleted.

4.1.12 unset

The unset command will remove the definitions of shell variables, previously defined with the set command, passed as arguments.

`unset [`*varname* `...]`

All of the named variables are unset (undefined). The argument may be "*", in which case all variables are unset (although this is usually not something that one would want to do).

4.1.13 usrset

The usrset command prints a (long) list of all of the variables used internally by *WRspice* which can be set with the set command.

`usrset [-c][-d][-p][-sh ][-si] [`*keyword* `...]`

WRspice provides a substantial number of internal switches and variables which can be configured with the set command. The usrset command prints a listing and brief description of each of the variables with internal significance to *WRspice*. If no arguments are given, all of the variables which control *WRspice* will be printed. The options print sets of keywords associated with certain functions, which are in turn associated with a particular panel accessible from the Tool Control window.

Option	Toolbar Button	Description
`-c`	**Commands**	Variables which control *WRspice* commands
`-d`	**Debug**	Debugging variables
`-p`	**Plot Defs**	Variables which control plotting
`-sh`	**Shell**	Variables which control the shell
`-si`	**Sim Defs**	Simulation control and SPICE options

Other arguments are taken as variable names, which will result in a description of that variable being printed.

4.2 Simulation Control Commands

The commands described in this section control the execution of *WRspice* simulations. By default, there is no activity on screen during a simulation. One can monitor the progress of a run in two ways. First, the iplot command can be used to plot a variable as the simulation is progressing. To plot `v(1)`, for example, one would type, before the run is started, "`iplot v(1)`". During the run, `v(1)` will be plotted on screen, with the plot rescaled as necessary. One can also print variables. For example, the trace command can be used, by typing "`trace time`" before the run starts, to cause the time value to be printed at each output point during transient analysis. These two commands are examples of what are called "debugs". Other debugs include `stop when` and `stop after`. A debug remains in effect until deleted with the delete command, and the debugs in effect can be listed with the status command. The debugs can also be listed, deleted, or made inactive with the **Trace** tool from the **Tools** menu. The run can be paused at any time by typing **Ctrl-C** in the controlling text window.

It is possible to transparently execute simulations on a remote machine while in *WRspice*, if the remote machine has a SPICE daemon running. It is also possible to run simulations asynchronously on the present machine. These jobs are not available for use with the iplot command, however. The jobs command can be used to monitor their status.

Many of these commands operate on the "current circuit" which by default is the last circuit entered into *WRspice* explicitly with the source command, or implicitly by typing the file name. The setcirc command can be used to change the current circuit. The **Circuits** button in the **Tools** menu also allows setting of the current circuit.

When a circuit file is read, any references to shell variables are expanded to their definitions. Shell variables are referenced as `$`*name*, where *name* has been set with the set command or in the `.options` line. This expansion occurs only when the file is sourced, or the reset command is given, so that if the variable is changed, the circuit must be sourced or reset to make the change evident in the circuit. If a variable is set in the shell and also in the `.options` line, the value from the shell is used.

Simulation Commands	
ac	Perform ac analysis
alter	Change circuit parameter
aspice	Initiate asynchronous run
check	Initiate range analysis
dc	Initiate dc analysis
delete	Delete watchpoint
destroy	Delete plot
disto	Initiate distortion analysis
dump	Print circuit matrix
free	Delete circuits and/or plots
listing	List current circuit
loop	Perform analysis over range
noise	Initiate noise analysis
op	Compute operating point
pz	Initiate pole-zero analysis
reset	Reset simulator
resume	Resume run in progress
rhost	Identify remote SPICE host
rspice	Initiate remote SPICE run
run	Initiate simulation
save	List vectors to save during run
sens	Initiate sensitivity analysis
setcirc	Set current circuit
show	List parameters
state	Print circuit state
status	Print trace status
step	Advance simulator
stop	Specify stop condition
tf	Initiate transfer function analysis
trace	Set trace
tran	Initiate transient analysis
version	Print program version

4.2.1 ac

The ac command initiates an ac analysis of the current circuit.

```
ac ac_args [dc dc_args]
```

The *ac_args* are the same as appear in a `.ac` line (see 2.7.1). If a dc sweep specification follows, the ac analysis is performed at each point of the dc analysis (see 2.7.2).

4.2.2 alter

The alter command is used to change circuit parameter values.

`alter` *device_list* `,` *parameter_name value* `...`

This command in its present form can be issued before a simulation is started, after the circuit has been loaded. It can be used to alter device and model parameters. The syntax is similar to that of the show command, however the parameter list contains (name value) pairs. The parameter names are expected to be recognized for each device in the list. A comma separates the device list from the parameters.

Example:

```
alter r* , resistance 50
```

This changes all resistors in the circuit to 50 ohms.

Currently, this command has no (or undefined) effect if a run is currently in progress.

4.2.3 aspice

The aspice command allows simulation jobs to be run in the background on the present machine.

`aspice` *infile* [*outfile*]

This command will run a simulation asynchronously with *infile* as an input circuit. If *outfile* is given, the output is saved in this file, otherwise a temporary file is used. After this command is issued, the job is started in the background, and one may continue using *WRspice* interactively. When the job is finished, the rawfile is loaded and becomes the current plot, and any output generated is printed. Specifically, *WRspice* forks off a new process with the standard input set to *infile*, and which writes the standard output to *outfile*. The forked program is expected to create a rawfile with name given by a `-r` command line option. The forked command is effectively "*program* `-S -r` *rawfile* `<`*infile* `>`*outfile*, where *program* is the `spicepath` variable (which defaults to calling *WRspice*), *rawfile* is a temporary file name, and *outfile* is the file given, or a temporary file name. Although the aspice command is designed for use with *WRspice*, it may be used with other simulators capable of emulating the *WRspice* server mode protocol. One may specify the pathname of the program to be run with the `spicepath` variable, or by setting an environment variable.

4.2.4 check

The check command is used to initiate margin analysis. Margin analysis can consist of either a swept operating range analysis, or a Monte Carlo analysis.

`check [-v][-a][-m][-r][-c][-f][-s][-k][-h][`*analysis*`]`

See Chapter 5 for a full description of operating range and Monte Carlo analysis. The current circuit is evaluated, and must have an associated block of control statements which contain the pass/fail script. A second associated block of executable statements contains initialization commands. These blocks can be provided in the circuit file, or be previously defined codeblocks bound to the circuit. Codeblocks

are executable data structures described in 4.4.1. Setting up the file in one of the formats described described in Chapter 5 will ensure that these blocks are created and bound transparently, however it is possible to do this by hand.

The option characters can be grouped following a single "-", or entered separately. The *analysis* to be performed is given, otherwise it is found in the circuit deck. In interactive mode, if no analysis is specified, the user will be prompted for an analysis string. The current circuit must be from a margin analysis file, or have appropriate bound codeblocks. If `-v` (verbose) is given, results and other messages are printed on-screen as the analysis is performed, otherwise the analysis is silent, except for any printing statements executed in the associated command scripts. The mplot command can be used to follow progress graphically.

If the `-a` flag is given, operating range analysis is performed at every point (all-points mode). Otherwise, the analysis attempts to limit computation by identifying the contour containing the points of operation. This algorithm can be confused by operating ranges with strange shapes, or which possess islands of fail points. If the input file contains a `.checkall` line, then the `-a` flag to the check command is redundant, all points will be checked in this case.

If the `-m` option is given, Monte Carlo analysis is performed, rather than operating range analysis. This is the default if a `.monte` line appeared in the file; the `-m` option is only required if there is no `.monte` line. The `-a` option is ignored if `-m` is given, as is `.checkall`. Monte Carlo analysis files differ from operating range files only in the header (or header codeblock). During Monte Carlo analysis, the header block is executed before every simulation so that variables can be updated. In operating range analysis variables are initialized by the header block only once, at the start of analysis.

If the `-r` (remote) option is given, remote servers will be assigned simulation runs, allowing parallelism to increase analysis speed. The remote servers must have been specified through the rhost command, and each must have a `wrspiced` server running. More information on remote asynchronous runs can ge found in 4.2.18 and 4.2.19.

The `-c` (clear) option will clear any margin analysis in progress if the analysis has been paused, for example by pressing **Ctrl-C**. If the command line contains only `-c`, no new analysis is started. If something else appears on the command line, a new analysis is started after clearing the old analysis. A paused margin analysis is resumed if the check command is given which does not have the `-c` option set, and any arguments given in this case are ignored. The resume command will also restart a paused margin analysis.

Ordinarily, during operating range and Monte Carlo analysis, only the current data point is retained. The amount of data retained can be altered with the `-f`, `-s`, and `-k` options. However, if a `.measure` line appears in the circuit deck, or the iplot debug is active, data will be retained internally so that the `.measure` or iplot is operational.

The `-f` option will cause the data for the current trial to be retained. This is implied if any `.measure` lines are present, or if an iplot is active. The data are overwritten for each new trial. The data for the last trial are available after the analysis is complete, or can be accessed for intermediate trials if the analysis is paused.

The `-s` option also causes retention of the data for the current trial, but in addition will dump the data to a family of rawfiles, similar to the `segment` keyword of the `.tran` line (though this works with other than transient analysis). The default file name is the name of the range analysis output file, suffixed with "`.s`*NN*", where *NN* is 00, 01, etc. Each trial generates a new suffix in sequence.

With the `-k` option, all data are retained, in a multi-dimensional plot. Note that this can be huge, so use of the `maxdata` variable and `.save` lines may be necessary. One can see the variations by plotting some or all of the dimensions of the output. Recall forms like `v(1)[`*N*`]` refer to the *N+1*'th trial, and

`v(1)[N,M]` includes the data for the *N+1*'th to the *M+1*'th trials. The mplot command has a facility for displaying trial data in a simplified manner.

Finally, the `-h` (help) option will simply print a brief summary of the options to the check command.

If an iplot is active, `-f` (current trial data retention) is implied. The data will be plotted for each trial in the same iplot, erasing after each trial is complete. If `-k` is given, all data will be plotted, without erasure. Note that an iplot doubles internal memory requirements.

During operating range analysis, a file is usually created and placed in the current directory for output. This file is named with the base name of the input file, with an extension `.d`*NN*, where *NN* is replaced with `00`, `01`, etc. — the first case where the filename is unique. If for some reason the input file name is unknown, the basename "check" will be used. Similarly, in Monte Carlo analysis, a file named *basename*.`m`*NN* is generated. In either case, the shell variable `mplot_cur` is set to the current output file name. These files can be plotted on-screen with the "`mplot [`*filename*`]`" command.

The results from operating range/Monte Carlo analysis are hidden away in the resulting plot structure. The plot can be displayed by entering "`mplot` *vec*" where *vec* is any vector in the plot.

When a `.measure` is included in an iterative analysis, data are saved as follows. Before each iteration, the previous result vector and its scale are saved to the end of a "history" vector and scale, and are then deleted. The result vector and scale are recreated when the measurement is completed during the iteration. Thus, at the end of the analysis, for a measurement named "example", one would have the following vectors:

`example`	the result from the final trial
`example_scale`	the measurement interval or point in the last trial
`example_hist`	results from the prior trials
`example_hist_scale`	intervals from the prior trials

Thus, during each trial, the result vector will have the same properties as in a standard run. It can be used in the `.control` block of a Monte Carlo or operating range file (recall that `$?`*vector* can be used to query existence, and that if there is no `checkPNTS` vector defined, the `.control` block is called once at the end of each trial).

In the current circuit, the parameters to be varied are usually included as shell variables `$value1` and `$value2`. These are special hard-coded shell variables which contain the parameter values during simulation. Before the file is sourced (recall that variable substitution occurs during the read-in), these variables can be set with the set command, and the file simulated just as any other circuit. Initially, the variables `$value1` and `$value2` are set to zero. The `value1` and `value2` names can be changed to other names, and other mechanisms can be used to supply trial values, as described in Chapter 5.

Briefly, operating range analysis works as follows. The analysis range and other parameters are specified by setting certain vectors in the header script, or by hand. The range is evaluated by rows (varying `value1`) for each column (`value2`) point. Columns are then reevaluated if the terminating pass point was not previously found. For a row, starting at the left, points are evaluated until a pass point is found. The algorithm skips to the right, and evaluates toward the left until a pass point is found. This minimizes simulation time, however strange operating ranges, such as those that are reentrant or have islands, will not be reproduced correctly. The only fool-proof method is to evaluate every point, which will occur if the `-a` option is given, or the `.checkall` line was given in the input file.

The range of evaluation is set with *center*, *step*, and *number* variables. The *number* is the number of steps to take above and below the *center*. Thus, if *number* is 1, the range is over the three points *center-step*, *center*, and *center+step*. One can set ranges for `value1` and `value2`, or alternatively one can set `value2`, and the algorithm can determine the operating range for `value1` at each `value2` point. These values represent the parameter variation range in operating range analysis, but serve only to

determine the number of trials in Monte Carlo analysis.

There are a number of vectors with defined names which control operating range and Monte Carlo analysis. In addition, there are relevant shell variables. The check command creates a plot structure, which contains all of the special control vectors, plus vectors for each circuit node and branch. This plot becomes the current plot after the analysis starts. The special vectors which have relevance to the operating range analysis are listed below.

`checkPNTS` (real, length >= 1)
: These are the points of the scale variable (e.g., time in transient analysis) at which the pass/fail test is applied. If a fail is encountered, the simulation is stopped and the next trial started. This vector is usually specified as an array, with the compose command, and is used in operating range and Monte Carlo analysis. If not specified, the evaluation is performed after the trial completes.

`checkVAL1` (real, length 1)
: This is the initial central value of the first parameter to be varied during operating range analysis. It is not used in Monte Carlo analysis.

`checkDEL1` (real, length 1)
: The first central value will be incremented or decremented by this value between trials in operating range analysis. It is not used in Monte carlo analysis.

`checkSTP1` (integer, length 1)
: This is the number of trials above and below the central value. In Monte Carlo analysis, it partially specifies the number of simulation runs to perform, and specifies one coordinate of the visual array used to monitor progress (with the mplot command). In operating range analysis, the default is zero. In Monte Carlo analysis, the default is 3.

`checkVAL2, checkDEL2, checkSTP2`
: These are as above, but relate to the second parameter to be varied in the circuit in operating range analysis. In Monte Carlo analysis, only `checkSTP2` is used, in a manner analogous to `checkSTP1`. The total number of simulations in Monte Carlo analysis is (2*`checkSTP1` + 1)*(2*`checkSTP2` + 1), the same as would be checked in operating range analysis.

`checkFAIL` (integer, length 1, 0 or 1)
: This is the global pass/fail flag, which is set after each trial, 1 indicates failure. This variable is used in both operating range and Monte Carlo analysis.

`opmin1, opmax1, opmin2, opmax2` (real, length >= 1)
: The operating range analysis can be directed to find the operating range extrema of the one parameter for each value of the other parameter. These vectors contain the values found. They are not used in Monte Carlo analysis.

`value` (real, length variable)
: This vector can be used to pass trial values to the circuit, otherwise shell variables are used. This pertains to operating range and Monte Carlo analysis.

`checkN1, checkN2` (integer, length 1)
: These are the indices into the value array of the two parameters being varied in operating range analysis. The other entries are fixed. These vectors are not used if shell variables pass the trial values to the circuit, and are not used in Monte Carlo analysis.

The shell variables are:

`checkiterate` (0-10)
: This is the binary search depth used in finding operating range extrema. This is not used in Monte Carlo analysis.

`value1, value2`
: These variables are set to the current trial values to be used in the circuit (parameters 1 and 2). The *WRspice* deck should reference these variables (as `$value1` and `$value2`) as the parameters to vary. Alternatively, the value array can be used for this purpose. These variables can be used in Monte Carlo analysis. Additionally, these variables, and a variable named "`value`" can be set to a string. When done, the variable or vector named by the string will take on the functionality of the assigned-to variable. For example, if `set value1 = L1` is given, the variable `L1` is used to pass trial parameter 1 values to the circuit (references are `$L1`).

The `checkVAL1`, `checkDEL1`, etc. vectors used must be defined and properly initialized, either in the deck or directly from the shell.

The shell variables `value1` and `value2` are set to the current variable 1 and variable 2 values. In addition, vector variables can be set. This is needed for scripts such as optimization where the parameter to be varied is required to be under program control. If a vector named `value` exists, as does a vector named `checkN1`, then the vector entry `value[`*checkN1*`]` is set to `$value1` if `checkN1` is in the range of `value`. Similarly, if a vector `checkN2` exists, then the vector entry `value[`*checkN2*`]` is set to `$value2`, if `checkN2` is in the range of `value`. Thus, instead of invoking `$value1` and `$value2` in the *WRspice* text, one can instead invoke `$&value[$&checkN1]`, `$&value[$&checkN2]`, where we have previously defined the vectors `value`, `checkN1`, `checkN2`. The file could have a number of parameters set to `$&value[0]`, `$&value[1]`, ... If `checkN1` is set to 2, for example, `$&value[2]` would be varied, and the other values would be fixed at predefined entries. The name "value" can be redefined by setting a shell variable named "`value`" to the name of another vector.

If any of the shell variables `value1`, `value2`, or a *shell* variable `value` are set to a string, then the shell variable or vector named in the string will have the same function as the assigned-to variable. For example, if in the header one has "`set value1 = L1`", then the variable reference `$L1` would be used in the file to introduce variations, rather than `$value1`. Similarly, if we have issued "`set value = myvec`", the vector `myvec` would contain values to vary (using the pointer vectors `checkN1` and `checkN2`), and a reference would have the form `$&myvec[$&checkN1]`. Note that the alternate variables are not automatically defined before the circuit is parsed, so that they should be set to some value in the header. The default `$value1` and `$value2` are predefined to zero.

In Monte Carlo analysis, the header block is executed before each simulation. In the header block, shell variables and vectors may be set for each new trial. These variables and vectors can be used in the SPICE text to modify circuit parameters. The names of the variables used, and whether to use vectors or variables, is up to the user (variables are a little more efficient). Monte Carlo analysis does not use predefined names for parameter data. Typically, the `gauss()` function is used to specify a random value for the variables in the header block.

One can keep track of the progress of the analysis in two ways. *WRspice* will print the analysis point on the screen, plus indicate whether the circuit failed or passed at the point, if the `-v` option was given to the check command. Echo commands can be used in the input file to provide more information on-screen, which is printed whether or not the `-v` option was given. The second method uses the mplot command, which graphically records the pass/fail points. In this mode, the relevant arguments to mplot are as follows.

`mplot -on`
: This will cause subsequent operating range analysis results to be plotted while the analysis is

running.

`mplot -off`
This will return to the default (no graphical output while simulating).

The analysis can search for the actual edge of the operating region for each row and column. These data are stored in vectors named `opmin1`, `opmax1`, `opmin2`, and `opmax2` with length equal to the number of points of the fixed variable. For example, `opmin1[0]` will contain the minimum parameter 1 value when parameter 2 is equal to *central2* - *delta2***steps2*, and `opmin1[2*`*steps2*`]` will contain the minimum parameter 1 value when parameter 2 is *central2* + *delta2***steps2*.

The binary search depth is controlled by a shell variable `checkiterate`, with possible values of 0–10. If set to 1–10, the search is performed (setting to 0 skips the range finding). Higher values provide more accuracy but take more time. If the search is performed, a vector called `range` and its scale `r_scale` are also produced. These contain the Y and X coordinates of the operating range contour, which can be plotted with the command "`plot range`".

A typical approach is to first unset `checkiterate`, `checkSTP1`, and `checkSTP2`. The `check` command is used to run a single-point analysis, while changing the values of `value1` and `value2` until a pass point is found. After the pass point is found, `checkiterate` can be set to a positive value, which will yield the ranges for the two variables. Then, the `checkSTP1` and other variables can be set to cover this range with desired granularity, and the analysis performed again.

When only one point is checked (`checkSTP1 = checkSTP2 = 0`), no output file is generated. If `checkiterate` is nonzero and the `-a` option is given, and a vector is used to supply trial values, the range of each entry in the vector is determined, and stored in the `opmin1` and `opmax1` vectors. A mask vector can be defined, with the same length as the value vector and same name with the suffix "`_mask`". Value entries corresponding to nonzero entries of this vector do not have the range computed. If the `-a` flag is not given, the range is found in the usual way. The central value must pass, or the range will not be computed.

See Chapter 5 for more information on performing operating range and Monte Carlo analysis, and the suggested file formats.

4.2.5 dc

The `dc` command performs a swept dc analysis of the current circuit.

`dc` *.dc dc_args*

The *dc_args* are the same as used in the `.dc` line (see 2.7.2).

4.2.6 delete

The `delete` command is used to remove "debugs" (traces or breakpoints) from the debug list.

`delete [[in]active] [all | stop | trace | iplot | save |` *number*`] ...]`

With no arguments, a list of existing debugs is printed, and the user is prompted for one to delete. The `status` command also prints a list of debugs. Debugs can also be controlled with the panel brought up with the **Trace** button in the **Tools** menu.

If the `inactive/active` keyword is given, breakpoints listed to the right but before another `(in)active` keyword are deleted only if they are inactive/active. Otherwise, they are deleted unconditionally. If one of `stop`, `trace`, `iplot`, or `save` is given, debugs of that type only are deleted. These keywords can appear in combination.

Each debug is assigned a unique number, which is available through the status command. This number can also be entered on the command line causing that debug to be deleted (if the activity matches the `inactive` keyword, if given). A range of numbers can be given, for example "2-6". There must be no white space in the range token.

Examples:

Delete all traces and iplots:
```
delete trace iplot
```

Delete all inactive debugs:
```
delete inactive all
```

Delete all traces and inactive iplots:
```
delete traces inactive iplots
```

4.2.7 destroy

The destroy command will delete plot structures.

`destroy [all] | [`*plotname* `...]`

Giving this command will throw away the data in the named plots and reclaim the storage space. This can be necessary if a lot of large simulations are being done. *WRspice* should warn the user if the size of the program is approaching the maximum allowable size (within about 90%), but it is advisable to run the rusage command occasionally if running out of space is a possibility. If the argument to destroy is `all`, all plots except the constants plot will be thrown away. It is not possible to destroy the constants plot. If no argument is given the current plot is destroyed.

4.2.8 disto

The disto command will initiate distortion analysis of the current circuit.

`disto` *disto_args* `[dc` *dc_args*`]`

The *disto_args* are the same as appear in a `.disto` line (see 2.7.3). If a dc sweep specification follows, the distortion analysis is performed at each point of the dc analysis (see 2.7.2).

4.2.9 dump

The dump command may be useful for analyzing convergence problems. This command sends a print of the internal matrix data structure last used by the simulator for the current circuit to the standard output. It is used for program debugging, but may be useful if convergence problems are encountered. The command takes no arguments.

4.2.10 free

The free command is used to free memory used by circuit and plot structures.

```
free [c[ircuit]] [p[lot]] [a[ll]] [y[es]]
```

This command releases the memory used to store plot and circuit structures for reuse by *WRspice*. The virtual memory space used by plots in particular can grow quite large. If `free` is given without an argument, the user is queried as to whether to delete the current plot and circuit structures (independently). If the argument `all` is given, the user is queried as to whether to delete all plot and circuit structures. If the argument `circuit` is given, only circuits will be acted on. Similarly, if the argument `plot` is given, only plots will be acted on. If neither `circuit` or `plot` is given, both circuits and plots will be acted on. If the argument `yes` is given, the user prompting is skipped, and the action performed. Only the first letter of the keywords is needed. Plots can also be freed from the panel brought up by the **Plots** button in the **Tools** menu, and circuits can be freed from the panel brought up with the **Circuits** button. The destroy command can also be used to free plots.

4.2.11 listing

The listing command is used to generate a listing of the current circuit.

```
listing [l[ogical]] [p[hysical]] [d[eck]] [e[xpand]]
```

The command will print a listing of the current circuit to the standard output. The arguments control the format of the listing. A `logical` listing is one in which comments are removed and continuation lines are appended to the end of the continued line. A `physical` listing is one in which comments and continuation lines are preserved. A `deck` listing is a `physical` listing without line numbers, so as to be acceptable to the circuit parser — it recreates the input file verbatim. The last option, `expand`, is orthogonal to the previous three — it requests that the circuit be printed after subcircuit expansion. Note that only in an expanded listing are error messages associated with particular lines visible. If no argument is given, `logical` is understood.

4.2.12 loop

The loop command is used to perform a simulation analysis over a range of conditions.

```
loop [-c] [min1 [max1 [step1]] [min2 [max2 [step2]]] [analysis]]
```

The command works like a dc sweep, however the shell variables `value1` and `value2`, which are accessible in the circuit as `$value1` and `$value2`, are incremented, as in operating range analysis (see Chapter 5). The specified *analysis* is performed at each point, yielding multidimensional output vectors. If *analysis* is omitted, an analysis specification is expected to be found in the deck. Only one analysis can be specified (no dc chaining). The deck should contain `$value1` and `$value2` (if used) for substitution, and be the current circuit. If an analysis loop is paused, for example by pressing **Ctrl-C**, it can be resumed by entering the loop command again. No arguments are required in this case, however if the `-c` option is given the old analysis is cleared, and a new analysis started if further parameters are supplied. The `-c` is ignored if there was no loop analysis in progress. The resume command will also resume a paused loop analysis. The reset command given with the `-c` option will also clear any paused loop analysis.

4.2.13 noise

The noise command initiates a small-signal noise analysis of the current circuit.

```
noise noise_args [dc dc_args]
```

The *noise_args* are the same as appear in a `.noise` line (see 2.7.4). If a dc sweep specification follows, the noise analysis is performed at each point of the dc analysis (see 2.7.2).

4.2.14 op

The op command will initiate dc operating point analysis of the current circuit (see 2.7.5). The command takes no arguments.

4.2.15 pz

The pz command will initiate pole-zero analysis on the current circuit.

```
pz pz_args
```

The *pz_args* are the same as appear in a `.pz` line (see 2.7.6).

4.2.16 reset

The reset command will reinitialize the current circuit.

```
reset [-c]
```

The command will throw out any intermediate data in the circuit (e.g, after a breakpoint or user pause with **Ctrl-C**) and re-parse the deck. Any standard analysis in progress will be cleared, however the state of any margin analysis (started with the check command), or loop analysis (started with the loop command) is retained by default. However, if the `-c` option is given, these too are cleared. Thus, the reset command can be used to update the shell variables in a deck with or without affecting the status of a margin or loop analysis in progress.

4.2.17 resume

The resume command will resume an analysis in progress. The simulation can be stopped by typing an interrupt (**Ctrl-C**) or with the `stop` command. If no analysis is currently in progress, the effect is the same as the run command. Each circuit can have one each of a standard analysis, a loop analysis (started with the loop command), and a margin analysis (from the check command) in progress. The resume command will resume standard analysis, margin analysis, and loop analysis in that precedence. Paused margin and loop analysis can also be restarted with the check and loop commands. These commands, and the reset command, can be used to clear stopped analyses. The resume command takes no arguments.

4.2.18 rhost

The rhost command allows addition of host names which are available for remote *WRspice* runs.

`rhost [-a][-d] [`*hostname*`]`

This command allows manipulation of a list of host names which are available for remote *WRspice* runs with the rspice command. If no arguments are given, the list of hosts is printed. The `-a` and `-d` options allow a host name to be added to or deleted from the list, respectively. The default is `-a`. Hosts are added to the list if they have been specified in the environment or with the `rhost` variable, and a job has been submitted. The *hostname* can optionally be suffixed with a colon followed by the port number to use to communicate with the `wrspiced` daemon. If not given, the port number is obtained from the operating system for "spice/tcp", or 3004 if this is not defined.

4.2.19 rspice

The rspice command is used to initiate simulation runs on a remote machine.

`rspice` *inputfile*
or
`rspice [-h` *host*`][-p` *program*`][-f` *inputfile*`] [`*analysis*`]`

This command initiates a remove *WRspice* job, using the *inputfile* as input, or the current circuit if no *inputfile* is given. If the `-h` option is not used, the default host can be specified in the environment before *WRspice* is started with the SPICE_HOST environment variable, or with the `rhost` variable. In addition, a list of hosts which are nominally available for remote runs can be generated with the rhost command. The default host used is the host known to *WRspice* that has the fewest active submissions, or which appears last on the list (hosts are added to the front of the list). If the `-p` option is not used, *WRspice* will use the program found in the `rprogram` variable, and if not set will use the same *program* as the aspice command. If the `-f` option is not used, the current circuit is submitted, otherwise *inputfile* is submitted. If there is no *analysis* specification, there must be an analysis specified in *inputfile*. If the current circuit is being submitted, there must be an *analysis* specification given on the command line.

Once the job is submitted, *WRspice* returns to interactive mode. When the job is complete, the standard output of the job, if any, is printed, and the rawfile generated becomes the current plot.

Remote runs can only be performed on machines which have the `wrspiced` daemon operating, and have permission to execute the target program.

4.2.20 run

The run command initiates the analysis found in the deck associated with the current circuit.

`run [`*rawfile*`]`

The command will run the simulation as specified in the input file. If there were any of the analysis specification lines (`.dc`, `.tran`, etc.) they are executed. The output is put in *rawfile* if it was given, in addition to being available interactively. If a standard analysis run is in progress and halted with the stop command or by pressing **Ctrl-C**, the run command will resume that run. This applies only to standard analyses, and not margin analysis or loop analysis. Standard analyses started with the analysis commands tran, dc, etc. , will always start a new analysis, after clearing any paused standard analysis.

4.2.21 save

The save command can be used to save a particular set of outputs from a simulation run.

```
save [all] [nodename ...]
```

The command will save a set of outputs, the rest will be discarded. If a node has been mentioned in a save command, it will appear in the working plot after a run has completed, or in the rawfile if *WRspice* is run in batch mode (in this case, the command can be given in the input file as `.save` ...). If a node is traced or used in an iplot it will also be saved. If no save commands are given, all nodes will be saved. The save can be deleted with the delete command, or from the panel brought up by the **Trace** button in the **Tools** menu.

If a save command is given at the prompt in interactive mode, it is placed in a global list, and activity will persist until deleted (with the delete command). If the command is given in a file, the command will be added to a list for the current circuit, and will apply only to that circuit. Thus, for example, a *WRspice* file can contain lines like

```
*# save v(1) ...
```

and the action will be performed as that circuit is run, but the "`save v(1)` ..." directive will not apply to other circuits.

One can save "special" variables, i.e., device/circuit parameters that begin with '`@`'. If a device parameter is a list type, only a single component can be saved. The single component can be specified with an integer, or with a vector name that evaluates to an integer. For example, the initial condition values for a Josephson junction can be accessed as a list, say for a junction named "b1", one can specify

`@b1[ic,0]` or `@b1[ic][0]`

which are equivalent, and each the same as `@b1[vj]`, the initial voltage. Similarly,

`@b1[ic,1]` or `@b1[ic][1]`

are equivalent, each being the same as `@b1[phi]`, the initial phase.

One can also have

```
let val = 1          (this vector is defined somewhere)
@b1[ic,val] or @b1[ic][val]
```

Thus, commands like

```
save @b1[ic,0]
```

or equivalently

```
save @b1[ic][0]
```

are accepted. Note that "`save @b1[ic]`" is the same as "`save @b1[ic,0]`". The "0" can be an integer, or a vector name that evaluates to an integer.

4.2.22 sens

The sens command initiates sensitivity analysis on the current circuit.

`sens` *sens_args* `[dc` *dc_args*`]`

The *sens_args* are the same as appear in a `.sens` line (see 2.7.7). If a dc sweep specification follows, the sensitivity analysis is performed at each point of the dc analysis (see 2.7.2).

4.2.23 setcirc

The setcirc command will set the "current circuit" assumed by *WRspice*.

`setcirc [`*circuit_name*`]`

The current circuit is the one that is used by the analysis commands. When a circuit is loaded with the source command it becomes the current circuit. If no arguments are given, a list of circuits is printed, and the user is requested to choose one. The current circuit can also be selected from the panel brought up by the **Circuits** button in the **Tools** menu.

4.2.24 show

The show command is used to display information about devices, models, and internal statistics.

`show [-r|-o|-d|-n` *nodename*`|-m|-D[M]|-M] [`*args*`] [,` *parmlist*`]`

If `-r` is given, system resource values are printed. The *args* are resource keywords as in the rusage command, and there is no *parmlist*. If there are no *args*, `all` is assumed.

If `-o` is given, *WRspice* option values are printed. These values are obtained from the `.options` line of the current circuit, or have been set with the set command. If no *args* are given, the default is `all`. There is no *parmlist*.

If `-d` is given, or if no option is given, device parameters are printed. The *args* are device names, and the *parmlist*, which is separated from the device list by a comma, consists of device parameter keywords. The parameters are expected to apply to each device in the list. Both lists can contain "globbing" (wild-card) characters with similar behavior to globbing (global substitution) in the *WRspice* shell. Briefly, '`?`' matches any character, '`*`' matches any set of characters or no characters, "`[abc]`" matches the characters '`a`', '`b`', and '`c`', and "`a{bc,de}`" matches "`abc`" and "`ade`". Either the device *args* or the *parmlist* can be "`all`", and the default is "`all, all`" ("`all`" is equivalent to '`*`'). Either the device *args* or the *parmlist* can be "`all`", and the default is "`all, all`". If the *parmlist* is the keyword "`none`", then no parameters are listed, only the devices with their resolved model names. This can be useful for determining which model is actually used for a MOS device, if L/W model selection is being used. The command "`show -d m*,none`" will display the name of the model used for each MOS device.

If `-n` is given, followed by the name of a circuit node, the output is in the same form as for `-d` however only devices connected to the named node are displayed.

If `-m` is given, model parameters are printed. The *args* are model names, and the *parmlist* is the list of model parameters to print. Wild-carding is accepted in both lists. The default is `all, all`. The

parameters are expected to apply to each model in the list. See the entries for the various devices and models for the parameter names, or type the show command without a parameter list to see the current values for all available parameters for the devices or models mentioned.

Spaces around the "," are optional, as is the "," itself if no parameters are given. If no argument is given to the show command, all parameters of all devices in the current circuit will be displayed.

The `-D` and `-M` options are similar, but keywords and descriptions from the internal models are listed, and no values are shown. It is not necessary to have a circuit loaded, as it is with `-d` and `-m`. The *args* are single characters which key the devices in *WRspice*, such as '`c`' for capacitors, '`q`' for bipolar transistors, etc. . For devices with a `level` model parameter such as MOSFETs, a digit indicating the model level can follow the key argument, without any space.

If these options are given with no argument, the device or model info is printed for each device or model (both for "`show -DM`") found in the device library. If an argument is given, only the matching device or model will be be shown, but all of the parameters will be listed in addition. The `-D` option lists the instance parameters, and `-M` the model parameters, and `-DM` will list both. In the listing, the letters 'RO' indicate a read-only parameter, which is a computed quantity not set in the instance or model lines. The letters 'NR' indicate a parameter that can't be read, i.e., it is input-only. Recall that device parameters can be accessed as vectors with the `@`*devname*[*param*] construct. There is no *parmlist* for the `-D` and `-M` options.

For example, to print the resistance of all resistors in the current circuit, enter

```
show -d r*, resistance
```

The `-d` above is optional, being the default. To print the `cbs` and `cbd` parameters of mosfets `m1`–`m4`

```
show m[1-4], c{bd,bs}
```

To print the current value of the relative tolerance option, enter

```
show -o reltol
```

Entering

```
show -DM q m5
```

will list the instance and model parameters of bipolar transistors and level 5 (BSIM2) MOSFETs.

4.2.25 state

The state command will print the name and a summary of the state of the current circuit. The command takes no arguments.

4.2.26 status

The status command is used to print a list of the "debugs" currently in force. The command will print out a summary of all the trace, stop, save, and iplot commands that are active. Each debug is assigned a unique number, which can be supplied to the delete command to remove the debug. The debug list can also be manipulated from the panel brought up with the **Trace** button in the **Tools** menu. The command takes no arguments.

4.2.27 step

The step command allows single-stepping through a transient simulation.

```
step [number]
```

The command will simulate through the number of user output points given, or one, if no number is given.

4.2.28 stop

The stop command will add a breakpoint to the debug list.

```
stop [before | at | after num] [when expr1 op expr2] ...
```

When the condition is true, simulation will stop, and can be resumed, after clearing the breakpoint, with the resume command. The breakpoints can be cleared with the delete command, and listed with the status command. The panel brought up by the **Trace** button in the **Tools** menu can also be used to manipulate breakpoints.

The first clause specifies the user output points at which the break is in effect. If an `after` clause is given, the simulation will stop after *num* points. The `at` clause stops only at *num* points, and the `before` clause stops only before *num* points. These are useful in conjunction with a `when` clause or can be omitted. If a `when` clause is included, at each point, the *expr*1,2 expressions will be evaluated and *expr1 op expr2* will be checked, and if it is true, the simulation will stop. If more than one `when` or `after` clause is put on one line, the conjunction of the conditions is checked. The *ops* are relational operators from the list below. Note that for this command, < and > do not denote IO redirection.

`eq` or =	equal to
`ne` or <>	not equal to
`gt` or >	greater than
`lt` or <	less than
`ge` or >=	greater than or equal to
`le` or <=	less than or equal to

If a stop command is given at the prompt in interactive mode, it is placed in a global list, and activity will persist until deleted (with the delete command). If the command is given in a file, the command will be added to a list for the current circuit, and will apply only to that circuit. Thus, for example, a *WRspice* file can contain lines like

```
*# stop when ...
```

and the action will be performed as that circuit is run, but the "`stop when` ..." directive will not apply to other circuits.

4.2.29 tf

The tf command initiates transfer function analysis of the current circuit.

```
tf tf_args [dc dc_args]
```

The arguments appear as they would in a `.tf` line (see 2.7.8). If a dc sweep specification follows, the transfer function analysis will be performed at each dc sweep point (see 2.7.2).

4.2.30 trace

The trace command will add a "debug" which prints the value of the expression(s) at each user analysis point.

`trace` *expr* `[...]`

At each time point, the expressions on the command line will be evaluated, and their values printed on the standard output.

If a trace command is given at the prompt in interactive mode, it is placed in a global list, and activity will persist until deleted (with the delete command). If the command is given in a file, the command will be added to a list for the current circuit, and will apply only to that circuit. Thus, for example, a *WRspice* file can contain lines like

```
*# trace v(1)
```

and the trace will be performed as that circuit is run, but the "`trace v(1)`" directive will not apply to other circuits.

The traces in effect can be listed with the status command, deleted with the delete command, and otherwise manipulated from the panel brought up with the **Trace** button in the **Tools** menu. At every user analysis point, the expressions given are evaluated and the results printed on the standard output.

4.2.31 tran

The tran command initiates transient analysis of the current circuit.

`tran` *tran_args* `[dc` *dc_args*`]`

The arguments are the same as those of a `.tran` line (see 2.7.9). Output is retained at *tstart*, *tstop*, and multiples of *tstep* in between, unless the variable `steptype` is set to `nousertp`. In this case, output is retained at each internally generated time point in the range. Accuracy can be slightly improved if the steptype variables set to `hitusertp`. If a dc sweep specification follows, the transient analysis is performed at each sweep point.

4.2.32 version

The version command is used to determine the version of *WRspice* running.

`version [`*version_name*`]`

With no arguments, this command prints out the current version of *WRspice*. If there are arguments, it compares the current version with the given version and prints a warning if they differ. A version command is usually included in the rawfile.

4.3 Data Manipulation Commands

The following commands perform various operations on vectors, which are the basic data structures of *WRspice*. Vectors from the current plot can be referenced by name. A listing of the vectors for the current plot is obtained by typing the let or display commands without arguments, or pressing the **Vectors** button in the **Tools** menu. Vectors for other than the current plot are referenced by *plotname.vecname*, for example, `tran2.v(1)`. The current plot can be changed with the setplot command, or from the panel brought up by the **Plots** button in the **Tools** menu.

Vectors can be created and manipulated in many ways. For example, typing

```
let diff = v(1) - v(2)
```

creates a new vector `diff`. All vectors can be printed, plotted, or used in expressions. They can be deleted with the unlet command.

Data Manipulation Commands	
compose	Create vector
cross	Vector cross operation
define	Define a macro function
deftype	Define a data type
diff	Compare plots and vectors
display	Print vector list
fourier	Perform spectral analysis
let	Create a vector
linearize	Linearize vector data
pick	Create vector from reduced data
print	Print vectors
seed	Seed random number generator
setplot	Set current plot
setscale	Assign scale to vector
settype	Assign type to vector
spec	Perform spectral analysis
undefine	Undefine macro function
unlet	Undefine vector

4.3.1 compose

The compose command is used to create vectors. It has two forms:

`compose` *vecname param* = *value* `[...]`
or
`compose` *vecname* `values` *value* `[...]`

Both forms of this command create a new vector called *vecname*. In the first form, the values in the vector are determined by the parameters given. In the second form, the given values are used to form the vector.

The possible parameters are:

`start`	The value at which the vector should start
`stop`	The value at which the vector should end
`step`	The difference between successive elements
`lin`	The number of points, linearly spaced
`log`	The number of points, logarithmically spaced
`dec`	The number of points per decade, log spaced
`center`	Where to center the range of points
`span`	The size of the range of points
`gauss`	The number of points in the gaussian distribution
`mean`	The mean value for the gaussian distribution
`sd`	The standard deviation for the gaussian distribution
`random`	The number of randomly selected points

In the first form of the command, at least one of `lin`, `log`, `dec`, `gauss`, or `random` should be specified, or each of `start`, `stop`, and `step`. For `random`, `span` defaults to 2.0, and `center` defaults to 0. For `gauss`, `sd` defaults to 1.0, and `mean` defaults to 0. The random number sequences can be reset by calling the seed command. The parameter `start` defaults to zero, unless implicitly set by other parameters. If `start > stop`, the order is reversed. Parameters `start`, `stop`, and `step` apply to `lin` (the default), `log`, and `dec`. Parameters `center` and `span` apply to `random`, and `sd` and `mean` apply to `gauss`.

4.3.2 cross

The cross command creates a new vector.

`cross` *vecname number source* `[...]`

The vector is constructed, with name *vecname* and values consisting of the *number*'th element of each of the source vectors.

4.3.3 define

The define command is used to specify user-defined vector functions.

`define [`*function*`(`*arg1*`, `*arg2*`, ...)]  [`*expression*`]`

This will define the user-definable function with the name *function* and arguments *arg1*, *arg2*, ... to be *expression*, which may involve the arguments. When the function is later used, the arguments that are is given are substituted for the formal arguments. If *expression* is not present, any definitions for *function* are printed, and if there are no arguments then all currently active definitions are printed. Note that one may have different functions defined with the same name but different arities. Some useful definitions (which are part of the default environment) are:

```
define max(x,y) (x > y) * x + (x <= y) * y
define min(x,y) (x < y) * x + (x >= y) * y
```

4.3.4 deftype

The deftype command defines a new data type.

`deftype v` *typename* [*abbrev*]
`deftype p` *plottype* [*pattern* ...]

The first form defines a new type for vectors. The *typename* may then be used as a vector type specification in a rawfile, and will be used by the display command to indicate the type. If an *abbrev* is given, values of that type can be named *abbrev*(*something*) where *something* is the name given in the rawfile (and *something* doesn't contain parentheses). The second form defines a plot type. The (one word) name for a plot with any of the patterns present in its plot type name as given in the rawfile will be *plottype*N, where N is a positive integer incremented every time a rawfile is read or a new plot is defined. This is an obscure command that might be useful for exporting rawfile data to other programs. If a vector's value indicates furlongs per fortnight, its type can be so defined.

4.3.5 diff

The diff command compares two vectors or entire plots.

`diff` *plot1 plot2* [*vecname* ...]

The command will compare all the vectors in the specified plots, or only the named vectors if any are given. If there are different vectors in the two plots, or any values in the vectors differ significantly, the difference is reported. The variables `diff_abstol`, `diff_reltol`, and `diff_vntol` are used to determine if two values are "significantly" different.

4.3.6 display

The display command prints a summary of all of the currently defined vectors, or the names supplied.

`display` [*vecname* ...]

This command will list the names, types and lengths of the vectors defined in the current plot, and whether the vector is real or complex. If one or more *vecame*s are given the list is restricted to those vectors. Additional information is also given: if there is a minimum or maximum value for the vector defined, this is listed (see A.1 for the manner in which this and the rest of the per-vector parameters are defined), if there is a default grid type or a default plot type, they are mentioned, and if there is a default color or a default scale for the vector it is noted. Additionally, one vector in the plot will have the notation `[default scale]` appended — this vector will be used as the x-scale for the plot command if none is given or if the vectors named have no default scales of their own. See the plot command (4.5.6) for more information on scales.

The vectors are sorted by name unless the variable `nosort` is set. The let command without arguments is equivalent to the display command without arguments.

4.3.7 fourier

The fourier command performs Fourier analysis.

`fourier` *fundamental_frequency* [*value* ...]

The command initiates a fourier analysis of each of the given values, using the first 10 multiples of the fundamental frequency (or the first `nfreqs`, if that variable is set). The values may be any valid expression. The values are interpolated onto a fixed-space grid with the number of points given by the `fourgridsize` variable, or 200 if it is not set. The interpolation will be of degree `polydegree` if that variable is set, or 1. If `polydegree` is 0, then no interpolation will be done. This is likely to give erroneous results if the time scale is not monotonic. This command is executed when a `.four` line is present in the input file and *WRspice* is being run in batch mode.

4.3.8 let

The let command is used to assign vectors.

`let [`*vecname* `[=` *expr*`]]`

With no arguments, the list of vectors from the current plot is printed, similar to the display command. If an argument appears without an assignment, the value of the named vector is printed. Otherwise, a vector is created with name *vecname* and value given by the expression *expr*. None of the vector options such as default scale, color, etc. that are read from the rawfile are preserved when a vector is created in this manner.

The syntax `let a[3] =` *vec* is valid. If *vec* is a vector, then `a[3] =` *vec*`[0]`, `a[4] =` *vec*`[1]`, etc., If undefined, `a` is defined, and new entries that are not explicitly set are zeroed. The length of `a` is set or modified to accommodate *vec*. The syntax `a[0] =` *vec* is also valid, and is equivalent to `a =` *vec*. If *vec* is a vector, then `a` is a copy of *vec*. If *vec* is a scalar (unit length vector), then `a` is also a scalar with the value of *vec*.

The units suffix of a constant value is used to assign the units of any vector to which the constant is assigned. This means, for example, in

```
let a = v(1)/15o
```

`a` has units of current (A). Use the settype command without arguments to see a list of recognized types.

If alpha characters appear after a number, the initial characters are checked as a scale factor. Recognized sequences are “t, g, k, u, n, p, f, m, meg, mil”. Remaining characters are parsed as a units string. This is all case insensitive.

The “let” is actually optional; the let command will be applied to a line with the second token being “=”. This is somewhat less efficient, however.

4.3.9 linearize

This linearize command is used to create linearized vectors from vectors whose scales are not evenly spaced.

`linearize [`*vecname* `...]`

The command will force data from a transient analysis to conform to a linear scale, if the plot has been created using raw timepoints. This is the case only when the `steptype` variable is set to “`nousertp`”.

The linearize command will create a new plot with all of the vectors in the current plot, or only those mentioned if arguments are given. The new vectors will be interpolated onto a linear time scale, which is determined by the values of `tstep`, `tstart`, and `tstop` in the currently active transient analysis. The currently loaded deck must include a transient analysis, or a tran command may be run interactively, and the current plot must be from this transient analysis. The variable `polydegree` determines the degree of interpolation.

4.3.10 pick

The pick command creates a new vector from elements of other vectors.

`pick` *vecname offset period vector* [*vector ...*]

The command creates a vector *vecname* and fills it with every *period*'th value starting with *offset* from the vectors. The *offset* and *period* are integers. For example, for

```
pick xx 1 2 v1 v2
```

we obtain

```
xx[0] = v1[1]
xx[1] = v2[1]
xx[2] = v1[3]
xx[3] = v2[3]
```

and so on.

4.3.11 print

The print command is used to print vector data on-screen or to a file using output redirection.

`print [/`*format*`] [col | line]` *expr* `[...]`

The command prints the values of the given expressions to the standard output. The default is to use exponential format for all values, with the number of digits given by the `numdgt` variable. However, a fixed-point format can be specified. The *format* must be the first token after `print` and has the form of a forward slash, followed by an optional integer, followed by "`f`". The optional integer supplies the number of decimal places printed, and the "`f`" indicates fixed-point notation. Any other character will revert to the default exponential notation. If the number isn't given, the default is the value of the `numdgt` variable if set, or 5.

All vectors listed will be printed in the same format, except for the scale vector, which is printed by default in the `col` mode, which is printed with the default notation.

If `col` is specified, the values are printed in columns, with the scale in the leftmost column on each page. This is the default for multi-valued vectors. The scale vector (`time`, `frequency`) will always be in the first column unless the variable `noprintscale` is true.

If `line` is specified, the value of each expression is printed on one line (or more if needed). If all expressions have a length of 1, the default style is `line`, otherwise `col` is the default.

The options `width`, `height`, and `nobreak` are effective for this command, however if printing on-screen, the actual screen or window size will be used. If the expression is "`all`", all of the vectors available are printed. Thus "`print col all >` *file*" will print everything to the *file* in SPICE2 format.

Examples:

```
print /3f v(5)
```

This prints v(5) to three decimal places in fixed-point notation.

```
print /4f v(2) v(3) v(4) > myfile
```

This prints the vectors to four decimal places in the file "myfile".

```
print 2*v(2)+v(3) v(4)-v(1)
```

This prints the computed quantities using the default format.

4.3.12 seed

The seed command will reset the internal random number generator.

```
seed [seed_integer]
```

The *seed_integer*, if given, will be used to seed the new random number sequence. This command enables sequences of "random" values obtained from functions like `rnd()` and `gauss()` to be repeatable (the default seed is random).

4.3.13 setplot

The setplot command can be used to set the current plot, or to create a new, empty plot and make it the current plot.

```
setplot [plotname]
```

The same functionality is available from the **Plots** button in the **Tools** menu. The setplot command will set the current plot to the plot with the given *plotname*, or if no name is given, prompt the user with a menu. The plots are named as they are loaded, with names like `tran1` or `ac2`. These names are shown by the setplot and display commands and are used by other commands. If the "New plot" item is selected, the current plot will become a new one with no vectors defined. Note that here the word "plot" refers to a group of vectors that are the result of one *WRspice* simulation run. When more than one file is loaded in, or more than one plot is present in one file, *WRspice* keeps them separate and only shows the vectors in the current plot. One generally accesses a given plot by first making it the current plot.

The current plot can also be changed by resetting the `curplot` variable. There are three read-only variables which are reset internally whenever the current plot changes. Each contains a string describing a feature of the current plot. These are `curplotdate`, `curplotname`, and `curplottitle`.

4.3.14 setscale

The setscale command is used to set the vector used as a scale when plotting other vectors.

```
setscale [plot or vector] [vectors ...]
```

Each plot has a default scale, which can be set with this command. Each vector has a scale variable, which if set will override the default scale of the plot. These also can be set with this command. This command takes as input the names of a plot and a new scale vector in that plot, or the names of vectors from the current plot. The wildcard forms using "`all`" and the plot prefix form *plot.vector* are not allowed in this command. If only one argument is given, i.e.

```
setscale vector
```

then *vector* is assigned as the default scale of the current plot. The vector must already exist in the current plot.

If two arguments are given, the first argument is initially interpreted as the name of a plot, and the second argument is the name of a vector in that plot to use as the scale. The plot has names like "`tran1`" or "`ac2`" and the *vector* must exist in that plot.

If the first argument is not a plot name, or there are more than two arguments, the arguments are expected to be vectors in the current plot, and the last vector will be assigned as the scale for the other listed vectors.

The scales assigned to vectors can be removed by assigning the vector that is the current default scale for the plot, or the scale vector name given can have the special names "`none`" or "`default`". The scale for plots can't be removed, since a plot must always have a default scale (if any vectors are defined).

The let command without arguments lists the vectors and will show the scales, if any.

4.3.15 settype

The settype command is used to change the data types of the vectors in a plot.

```
settype [type] [vector ...]
```

The command will change the type of the named vectors to *type*. With no arguments, the list of recognized types and abbreviations is printed. The *type* field can consist of a single name, or a single token containing a list of abbreviations. The token list can contain a digit power after an abbreviation, and a single '/' for denominator units. Examples are "F/M2", "Wb2/Hz". Units of vectors generated during analysis are set automatically.

Type names can also be found in the description of the rawfile format in A.1, or they may be defined with the deftype command.

The primitive built-in types and abbreviations are:

time	S
frequency	Hz
voltage	V
current	A
charge	Cl
flux	Wb
capacitance	F
inductance	H
resistance	O
conductance	Si
length	M
area	M2
temperature	C
power	W

The codes from the rawfile are:

Name	Description	SPICE2 Numeric Code
notype	Dimensionless value	0
time	Time	1
frequency	Frequency	2
voltage	Voltage	3
current	Current	4
output-noise	SPICE2 .noise result	5
input-noise	SPICE2 .noise result	6
HD2	SPICE2 .disto result	7
HD3	SPICE2 .disto result	8
DIM2	SPICE2 .disto result	9
SIM2	SPICE2 .disto result	10
DIM3	SPICE2 .disto result	11
pole	SPICE3 pz result	12
zero	SPICE3 pz result	13

4.3.16 spec

The spec command will create a new plot consisting of the Fourier transforms of the vectors given on the command line.

spec *start_freq stop_freq step_freq vector* [...]

This is based on a SPICE3 spec command by Anthony Parker of Macquarie University in Sydney Australia, which is available as part of the patch set from
http://www.elec.mq.edu.au/cnerf/spice/spice.html.

The command will create a new plot consisting of the Fourier transforms of the vectors given on the command line. Each vector given should be a transient analysis result, i.e., have time as a scale, and each should have the same time scale. The Fourier transform will be computed using the frequency parameters given, and will use a window function as given with the specwindow variable.

The following variables control operation of the spec command. Each can be set with the set command, or equivalently from the **Fourier** tab of the **Commands** tool.

`spectrace`
: This enables messages to be printed during Fourier analysis with the spec command, for debugging purposes.

`specwindow`
: This variable is set to one of the following strings, which will determine the type of windowing used for the Fourier transform in the spec command. If not set, the default is `hanning`.

`bartlet`	Bartlet (triangle) window
`blackman`	Blackman order 2 window
`cosine`	Hanning (cosine) window
`gaussian`	Gaussian window
`hamming`	Hamming window
`hanning`	Hanning (cosine) window
`none`	No windowing
`rectangular`	Rectangular window
`triangle`	Bartlet (triangle) window

`specwindoworder`
: This can be set to an integer in the range 2–8. This sets the order when the gaussian window is used in the spec command. If not set, order 2 is used.

4.3.17 undefine

The undefine command is used to undefine user-defined functions that have previously been defined with the define command.

`undefine` *word* `[...]`

The command deletes the definitions of the user-defined functions passed as arguments. If the argument is "*", then all macro functions are deleted. Note that all functions with the given names are removed, so there is no way to delete a function with a particular arity without deleting all functions with that name.

4.3.18 unlet

The unlet command will delete the vectors listed as arguments.

`unlet` *vecname* `[...]`

The current plot is assumed, though the *plot.vector* notation is accepted. When the default scale vector is deleted, another random vector will become the default scale. The names can be "`all`", indicating that all matching vectors should be removed. If the vector name is "`all`", all vectors in the the plot are removed, but the plot itself is not deleted. Giving "`all.all`" will clear the vectors in all plots (not very useful). To delete a plot, use the destroy or free commands.

4.4 Input and Output Commands

The following functions manage input to *WRspice*, or allow *WRspice* output to be saved in files.

Input and Output Commands	
codeblock	Manipulate codeblocks
edit	Edit text file
load	Read plot data from rawfile
sced	Bring up schematic editor
source	Read circuit input file
write	Write data to rawfile
xeditor	Edit text file

4.4.1 codeblock

The codeblock command manipulates codeblocks.

```
codeblock [-options] [filename]
```

A codeblock is a stored executable structure derived from a script file. Being internal representations, codeblocks execute more efficiently than script files. A codeblock generally has the same name as the script file from which it was derived.

Option characters, which may be lumped or given separate tokens, are listed below.

p	print the text of a block (synonym t)
f	remove the block (synonym d)
a	add a block
b	bind the block to the "controls" of the current circuit
be	bind the block to the "execs" of the current circuit

If no *filename* is given, all the blocks in the internal list are listed by name, commands are printed if p is given, and the blocks are deleted if f is given (a and b are ignored), and the current circuit codeblock pointer is undefined.

Otherwise, the named file/block is acted on. Added blocks overwrite existing blocks of the same name.

When a command is entered in response to a prompt or in a script (or another codeblock), the blocks are checked first, then the *WRspice* internal commands, then scripts, then vectors (for the implicit let in *vector = something*) and finally operating system commands if unixcom is set.

A codeblock can be "bound" to the current circuit with the b and be options. If be, the block is bound as an "exec" codeblock, and if b is given, the block is bound as a "control" codeblock. Each circuit has one of each type, which are by default derived from the .exec and .control statements from the circuit file. Binding an external codeblock overrides the blocks obtained from the file. If no *filename* was given, the existing binding is deleted from the current circuit, according to whether the b or be was given. Separate calls are required to unbind both blocks.

Operating range and Monte Carlo analysis can make use of "bound" codeblocks. In both types of analysis, the "controls" codeblock execution sets a variable indicating whether the circuit simulated properly according to user specified criteria. When a margin analysis file is input, the lines between .control and .endc become the default controls codeblock. Similarly, the lines between .exec and .endc become the default exec codeblock. A bound codeblock will always supersede the default codeblock.

Once a codeblock has been added, it can be executed by simply entering its name, as if it were a shell command. If a name conflicts with an internal command or script, the codeblock has precedence.

4.4.2 edit

The edit command allows the text of an input file to be edited.

`edit [-n][-r] [`*filename*`]`

The command will bring up a text editor loaded with the named file. If no file name is given, the file associated with the current circuit will be edited. If no file is associated with the current circuit, the current circuit will be printed into a temporary file which is opened for editing. If no circuits are present, an empty file is opened for editing. Pressing the **Text Editor** button in the **Edit** menu of the Tool Control window is equivalent to giving the edit command without arguments.

The editor used is named by the `editor` variable, the SPICE_EDITOR environment variable, or the EDITOR environment variable, in that order. If none of these is set, or the first one found is set to "`xeditor`", the internal editor is used, if graphics is available. If graphics is not available and no editor is specified, *WRspice* will attempt to use the "`vi`" editor. The internal editor has the advantage of asynchronous deck sources with the edit window displayed at all times, through the **Source** button in the editor's **Options** menu. The xeditor command is similar to the edit command, but will always call the internal editor. See 4.4.7 for a description of the internal editor.

If an external editor is used, if graphics is available the default action is to start the editor in a new `xterm` window. This can be suppressed if the `noeditwin` variable is set. This variable should be set if the external editor creates its own window to avoid the unneeded `xterm`. It can also be set for an editor such as `vi`, in which case the editing will take place in the same window used to interact with *WRspice*.

The `-r` and `-n` options are available only when the internal editor is *not* being used, and the editor is a text-mode editor such as `vi` and `noeditwin` is set so that editing takes place in the console controlling *WRspice*. If this is the case, after quitting the editor, the file will be sourced automatically if the text was saved. The `-n` (no source) option prevents this, and should be given if the editor is used to browse files that are not SPICE input files. The `-r` (reuse) option will reuse the existing circuit for the automatic source, rather than creating a new one. This saves memory, but prevents revisiting earlier revisions of the circuit. If the internal editor, or any editor that creates its own window is used, *WRspice* will pop up the editor and resume command prompting. There is no automatic source in this case.

4.4.3 load

The load command can be used to load a rawfile.

`load` *filename* `[...]`

The load command loads data from the rawfiles (in either binary or ascii format) named. Rawfiles are the native *WRspice* data output file, produced by the write and (possibly) the run commands. The default *filename* is "`rawspice.raw`", or the argument to the `-r` command line option if one was given. The file data will be converted into internal plot structures containing vectors available for printing, plotting, and other manipulation just as if the analysis had been run. The last plot read becomes the current plot. Raw data files can also be loaded from the **Load** button in the **Files** menu of the Tool Control window. A name given without a path prefix is searched for in the source path (`sourcepath` variable).

The load command is Internet aware, i.e., if a given filename has an "`http://`" or "`ftp://`" prefix, the file will be downloaded from the internet and loaded. The file is transferred as a temporary file, so if a permanent local copy is desired, the write command should be used to save file to disk.

4.4.4 sced

The sced command brings up the *Xic* graphical editor (if available) in electrical mode.

```
sced [filename ...]
```

This allows schematic capture, with most of the *WRspice* functionality directly available through the *Xic* interface. If the *Xic* graphical editor is not available for execution, this command will exit with a message indicating that *Xic* is not available. Otherwise, the sced command will bring up the schematic capture front-end with file *filename*, which must be an *Xic* input file (*not* a standard *WRspice* circuit file!). If the current circuit originated from *Xic*, that file will be loaded into *Xic* if no *filename* is given.

When *Xic* saves a native-mode top-level cell containing a schematic, the circuit SPICE listing is appended to the file. *WRspice* is smart enough to ignore the geometric information in these files and read only the circuit listing.

Xic can also be started from the **Xic** button in the **Edit** menu of the Tool Control window.

4.4.5 source

The source command is used to load circuit files and command scripts.

```
source file
```

This command will read and processes circuit descriptions and command text from the *file*. This is done implicitly if the name of the file is simply given as a command. The source operation will be performed, provided that the file name does not clash with a *WRspice* command. If the file does not have a path prefix, it is searched for in the search path specified by the `sourcepath` variable. If not in the search path, a full path name should be given. Shell variables found in the circuit deck (but not in the commands text) are evaluated during the source. The reset command can be used to update these variables if they are later changed by the shell after sourcing.

The source command is Internet aware, i.e., if a given filename has an "`http://`" or "`ftp://`" prefix, the file will be downloaded from the internet and sourced. The file is transferred as a temporary file, so if a permanent local copy is desired, the edit or listing commands should be used to save the circuit description to disk.

Command lines must be surrounded by the lines `.exec` or `.control` and `.endc` in the file, or prefixed by "`*@`" or "`*#`" in order to be recognized as commands. Commands found in `.exec` blocks or `*@` lines are executed before the circuit is parsed, thus can set variables used in the circuit. Commands found in `.control` blocks or `*#` lines are executed after the circuit is parsed, so a control line of "`ac ...`" will work the same as the corresponding `.ac` line, for example. Use of the "comment" control prefixes `*@` and `*#` makes it possible to embed commands in *WRspice* input files that will be ignored by earlier versions of SPICE.

The first line in any input file is considered a title line and is not parsed but kept as the name of the circuit. The exceptions to this rule are old format margin analysis input files and *Xic* files.

4.4.6 write

The write command is used to save simulation data in rawfile format.

`write [`*file* `[`*expr* `...]]`

The command writes out the *exprs* to *file*. First vectors are grouped together by plots, and written out as such. For example, if the expression list contained three vectors from one plot and two from another, then two plots will be written, one with three vectors and one with two. Additionally, if the scale for a vector isn't present, it is automatically written out as well. The default format is ascii, but this can be changed with the `filetype` variable, or the SPICE_ASCIIRAWFILE environment variable. The default filename is `rawspice.raw`, or the argument to the `-r` flag on the *WRspice* command line, if there was one, and the default expression list is "`all`". If the `appendwrite` variable is set, the data will be appended to an existing file.

4.4.7 xeditor

The xeditor command invokes a text editing window for editing circuit and other text files. It is available only when graphics is enabled.

`xeditor [`*file*`]`

This is similar to the edit command, however the internal editor is always used. The `editor` variable and the environment variables used by the edit command are ignored by the xeditor command.

The xeditor command brings up a general-purpose text editor window. The same pop-up editor is invoked in read-only mode by the **Notes** button of the **Help** menu in the Tool Control window for use as a file viewer. In that mode, commands which modify the text are not available.

See 3.7 for more information about the text editor.

4.5 Graphical Output Commands

The following commands display the output of simulations graphically, either on-screen or on a printing device. Many take as input a list of vectors or expressions to plot, and in some cases ambiguities may arise. These can often be resolved through additional parentheses `(...)` surrounding troublesome expressions. In addition, a double minus sign (`--`) forces a unary minus. For example, the command

```
plot v(1) -v(4)
```

would plot the difference of `v(1)` and `v(4)` (one trace), whereas the command

```
plot v(1) --v(4)
```

would plot `v(1)` and `-v(4)` (two traces).

In the expression list, a "." token is replaced with the vector list found in a `.plot` line from the file with the same analysis type as the current plot. For example, if the file contains

```
.tran .1u 10u
.plot tran v(1) v(2)
```

then one can type "`run`", then "`plot .`" to plot `v(1)` and `v(2)`.

Graphical Output Commands	
asciiplot	Generate line printer plot
combine	Combine plots
hardcopy	Send plot to printer
iplot	Plot during simulation
mplot	Plot range analysis output
plot	Plot simulation results
xgraph	Plot simulation results using `xgraph`

4.5.1 asciiplot

The asciiplot command generates a crude plot on a character mode device. It is not often used in modern environments, but is retained for compatibility with SPICE2.

`asciiplot` *plotargs*

The *plotargs* are vectors or expressions to be plotted, as with the plot command. The plot is sent to the standard output, so one can put it into a file with "`asciiplot` *plotargs* `>` *file*". The variables `width`, `height`, and `nobreak` determine the width and height of the plot, and whether there are page breaks, respectively, though if the asciiplot is printed on-screen or in a window, the plot width and height are determined by the window size.

There are problems if one tries to asciiplot something with an X-scale that is not monotonic (i.e., something like `sin(time)`), because `asciiplot` uses a simple-minded sort of linear interpolation. Also, most of the variables that the plot command recognizes aren't used by asciiplot. The scaling and other variables are set with the set command as for the plot command. These variables can also be set with the **Plot Defs** tool from the **Tools** menu of the Tool Control window.

The `nointerp` variable is used only by the asciiplot command. Normally asciiplot interpolates data onto a linear scale before plotting it. If this option is given this won't be done — each line will correspond to one data point as generated by the simulation. Since data are already linearized unless from a transient analysis with `steptype` set to `nousertp`, setting this variable will avoid a redundant linearization.

Ordinarily, the first vector plotted has its values also printed in the first column. This can be suppressed by setting the variable `noasciiplotvalue`. When printing, the number of significant digits used can be set with `numdgt` variable.

This command is completely obsolete, but is retained for nostalgia for those who fondly remember punched cards and line printers.

4.5.2 combine

The combine command takes no arguments. The command will combine the two most recent plots, if similar, into a single plot, and expands the dimensionality of the resulting plot. The two plots must have identical vector names and compatible lengths. The purpose of this command is to create a single multi-dimensional plot from sequences of runs. The most recent plot is added to the end of the previous plot, and is deleted.

Example:

```
while i < 5
    (set parameters for run)
    run
    if i > 0
        combine
    end
    i = i + 1
end
```

This will combine all the data from the five runs into a single plot.

4.5.3 hardcopy

The hardcopy command is used to generate hardcopy plots of simulation data on a printer or plotter. This capability is similar to the **Print** button which appears on each of the on-screen plots from the plot command.

```
hardcopy [setupargs] plotargs
setupargs:  -d driver -c command -f filename -r resolution -w width -h height
-x left_marg -y top_marg -l
```

The hardcopy command uses the internal hardcopy drivers to generate a hard copy of the vectors and expressions given in *plotargs*. The *plotargs* are vectors or expressions to be plotted, as with the plot command. If no *plotargs* are provided, the arguments are taken to be the same as those given to the last plotting command given (these include plot, asciiplot, hardcopy, and xgraph). The *setupargs* override the current values established using the set command or the **Plot Defs** tool in the **Tools** menu, and default to the driver defaults if not specified either way.

The `-d` *driver* specifies the name of a hardcopy driver using one of the keywords known to the `hcopydriver` variable. If the `-d` option is not specified, the `hcopydriver` variable will be used if set. If no driver is set, or set to an unrecognized driver name, the hardcopy is aborted.

The `-c` *command* option specifies the operating system command used to send the job to the printer, and overrides the value of the `hcopycommand` variable, which is otherwise used if set. The value is a string (which must be quoted it it contains space), where the characters `%s` are replaced by the name of the (possibly temporary) file used to store the plot data. If no `%s` appears, the file name is appended to the end of the command string. In BSD Unix, the command string might be "`lpr -h -Pmyprinter`", for example. See the man page for the print command on your machine for more information. If there is no command string given using the `-c` option and `hcopycommand` is undefined, the data will be saved in a file, but not printed.

The `-f` *filename* option gives a file name to store the plot data. There is no analogous "set" variable. If given, the plot will be saved in the file, and *not* sent to the printer.

The `-r` *resolution* command will set the printer to use the specified resolution, if that resolution is supported by the driver and the printer. If not given, the value of `hcopyresol` is used, if set, otherwise the driver default is used. The default is almost always the best choice.

The `-w` *width* and `-h` *height* options set the size of the image as it would appear on a portrait-oriented page. The numbers given represent inches, unless followed by "cm" which indicates centimeters. If these options are not given, the `hcopywidth` and `hcopyheight` variables are used if set, otherwise the driver defaults are used.

The `-x` *xoffset* and `-y` *yoffset* options control the position of the image on the page, as defined in portrait orientation. The *yoffset* may be measured from the top or bottom of the page, depending upon the driver. These values default to those in the variables `hcopyxoff` and `hcopyyoff` if set, otherwise driver defaults are used. The numbers represent inches, unless followed by "cm" indicating centimeters.

If the `-l` option is given, or the `hcopylandscape` variable is set, the image will be rotated and printed in landscape orientation.

The variables which control plot presentation also control the presentation of the hardcopy (see 4.5.6). The hardcopy command is suited for use in scripts. For general plotting, the **Print** button in the plot windows brings up a panel which provides a superior user interface.

4.5.4 iplot

The iplot command adds an incremental plot to the "debug" list. While a simulation is running, the plots will be generated, allowing immediate feedback as to whether the simulation is producing the "right" results.

`iplot` *plotargs*

The iplots can be deleted with the delete command, and can also be specified and deleted using the panel brought up by the **Trace** button in the **Tools** menu. The status command will list the "debugs", including iplots. The *plotargs* are vectors or expressions to be plotted, as with the plot command. The variables which control plotting also apply to iplots. These are set with the set command, or with the **Plot Defs** tool in the **Tools** menu of the Tool Control window.

If an iplot command is given at the prompt in interactive mode, it is placed in a global list, and activity will persist until deleted (with the delete command or with the **Trace** tool). If the command is given in a file, the command will be added to a list for the current circuit, and will apply only to that circuit. Thus, for example, a *WRspice* file can contain lines like

```
*# iplot v(1)
```

and the iplot will be performed as that circuit is run, but the "`iplot v(1)`" directive will not apply to other circuits.

4.5.5 mplot

The mplot command is used to plot the results from margin analysis, which includes operating range and Monte Carlo analyses. It is also used to set and clear interactive margin analysis plotting.

`mplot [[-on|-off] | [-c][`*filename* `...]  |` *vector*`]`

The *filenames* are names of files produced by the margin analysis. If no file is specified, the file produced by the last margin analysis run in the current session is assumed. If no margin analysis files have been produced in the current session, the file named "`check.dat`" is assumed. It is also assumed that these files exist in the current directory. The name of the most recent margin analysis output file produced in the current session is saved in the `mplot_cur` variable.

The results from operating range/Monte Carlo analysis are also hidden away in the resulting plot structure. The mplot can be displayed by entering "`mplot` *vector*" where *vector* is any vector in the plot.

The `-c` switch combines the operating range data from the files on the command line into a single display, if possible. Thus, if two or more successive operating range analysis runs are required to obtain the total operating range, then it is possible to plot all of the results on a single graph with the `-c` option. The data must have identical coordinate spacing and projected origins to be combinable.

There are two switches, `-on` and `-off`, which control whether or not operating range analysis results are plotted on the screen during analysis, similar to the iplot command. Entering `mplot -on` will cause margin analysis results to be plotted while simulating, and `mplot -off` will turn this feature off.

The display consists of an array of cells, each of which represent the results of a single trial. As the results become available, the cells indicate a pass or fail, In operating range analysis, the cells indicate a particular bias condition according to the axes. In Monte Carlo analysis, the position of the cells has no significance. In this case the display indicates the number of trials completed.

The panel includes a **Help** button which brings up the appropriate topic in the help system, a **Redraw** button to redraw the plot if, for example, the plotting colors are redefined, and a **Print** button for generating hard copy output of the plot.

Text entered while the pointer is in the mplot window will appear in the plot, and hardcopies. This text, and other text which appears in the plot, can be edited in the manner of text in plot windows.

The cells in an mplot can be selected/deselected by clicking on them. Clicking with button 1 will select/deselect that cell. Using button 2, the row containing the cell will be selected or deselected, and with button 3 the column will be selected or deselected. A selected cell will be shown with a colored background, with an index number printed.

Only one mplot window can have selections. Clicking in a new window will deselect all selections in other mplot windows.

At present, the selections are used to facilitate plotting of multidimensional plots, such as those obtained from the `-k` option of the check command. If selections exist, only the data from the selected cells will be plotted from the associated multidimensional vectors in the plot command.

For example, after running "`check -k`", suppose one has a resulting vector v(1) (which will contain data from all of the trials). If not using "`mplot -on`" during the run, one can type "`mplot`" after the run to display the pass/fail results. In the mplot window, select one of the cells. Then, "`plot v(1)`" will plot the v(1) from that trial only. If no cells are selected, or all cells are selected, "`plot v(1)`" would show the superimposed v(1) traces from all trials. The index number that appears in the cell is the vector index, so for example if a single box is selected with index 4, "`plot v(1)`" would be equivalent to "`plot v(1)[4]`". Note that the selection mechanism allows combinations of traces to be plotted which can't easily be obtained from indexing.

This capability is carried a step further for general multidimensional plots. If one enters "`mplot` *vector*" where *vector* is the name of a multidimensional vector from whatever source, an mplot will appear. If the vector originated from operating range or Monte Carlo analysis, the resulting mplot will appear (the pass/fail results are saved in the plot structure, as well as in the output file). Otherwise, the mplot has nothing to do with range analysis, and all cells are marked "fail". Either case allows the selection mechanism to be used for displaying the plots. Suppose for example one has a multidimensional plot from a loop-transient analysis. Entering "`mplot time`" will bring up a dummy mplot whose cells represent the loop iterations (time is the scale vector for the plot, but any data vector in the plot would suffice). Then, by selecting the cells, one can choose which iterations will be visible when vectors from the plot are plotted with the plot command.

4.5.6 plot

The plot command is used to plot simulation output on-screen. Each execution of a plot command will bring up a window which displays the plot, along with several command buttons. Each plot will remain on-screen until dismissed with the **Dismiss** button.

```
plot [expr ... ]  [vs x-expr]
```

The set of expressions can be followed with a `vs` *x-expr* clause, which will produce an x-y plot using the values of *x-expr* as the x scale.

If no arguments are given, the arguments to the last given plot command are used. If the argument list contains a token consisting of a single period ("."), this is replaced with the vector list found in a `.plot` line from the input file with the same analysis type as the current plot. For example, if the input file contains

```
.tran .1u 10u
.plot tran v(1) v(2)
```

then one can type "`run`" followed by "`plot .`" to plot `v(1)` and `v(2)`.

The plot style can be controlled by a number of variables (listed below), which can be set with the set command, which are listed in 4.7.3. These define default behavior, as the plot window contains buttons which also determine presentation. The **Plot Defs** tool from the **Tools** menu of the Tool Control window can also be used to set these variables. The **Colors** tool from the **Tools** menu can be used to change the colors used for plotting.

Variable	_temp_*name*?
color	
combplot	*
gridsize	
gridstyle	
group	*
lingrid	*
linplot	*
loglog	*
multi	*
nogrid	*
nointerp	*
plotposn	
plotstyle	
pointchars	
pointplot	*
polar	*
polydegree	
polysteps	
scaletype	
single	*
smith	*
smithgrid	*

ticmarks	
title	*
xcompress	*
xdelta	*
xindices	*
xlabel	*
xlimit	*
xlog	*
ydelta	*
ylabel	*
ylimit	*
ylog	*
ysep	*

For each of the variables above with an asterisk in the right column, if a variable named _temp_*varname* is set, its value will be used rather than that of *varname*. This allows temporary overriding of the nominal settings of the variables. In addition, there are certain variables such as `gridstyle` which can be set to one of several keywords. If the keyword itself is set as a boolean variable, it will override the string variable. For example, one could issue "`set gridstyle = lingrid`" to set a nominally linear grid. This can be changed by, for example, "`set loglog`" (or "`set _temp_loglog`"), but it is an error to have two or more such keywords set as booleans at a time.

When a plot is read from a rawfile, defaults for the presentation attributes are set as specified in the rawfile. These can be overridden by reseting the attributes in *WRspice*, with the exception of the color specification in the rawfile. If given, that color will be used for a particular trace independent of the current setting within *WRspice*. *WRspice* never sets the color specification, when writing a rawfile, unless that color was indicated from a previous rawfile. If a certain unalterable color is desired for a trace, the rawfile can be edited with a text editor to specify that color.

Any text typed while the pointer is in the plot window will appear on the plot (and hardcopies). This is useful for annotation. Entered and existing text can be edited and moved. In addition, traces in the plot can be moved to change the order, or moved to other (x-scale compatible) plot windows. The description of the plot window (3.8) contains more information.

4.5.7 xgraph

The xgraph command will produce plots using the UNIX xgraph utility.

`xgraph` *file plotargs*

This command is similar to the plot command, however the xgraph program (a widely-used plotting package) actually generates the plots. If the given *file* is either "`tmp`" or "`temp`", then a temporary file is used to hold the data while being plotted. Polar and Smith plots are not supported, otherwise the variables associated with the plot command apply.

The `xglinewidth` variable specifies the line width in pixels to use in the plots. If not set, a minimum line width is used.

If `xgmarkers` is set, point plots will use cross marks, otherwise big pixels are used.

4.6 Miscellaneous Commands

These commands perform miscellaneous functions.

Miscellaneous Commands	
bug	Submit bug report
jobs	Check asynchronous jobs
help	Enter help system
qhelp	Print command summaries
quit	Exit program
rusage	Print resource usage
setrdb	Set X resources
strcmp	Compare strings
where	Print nonconvergence information

4.6.1 bug

The bug command facilitates sending bug reports and other messages to the *WRspice* administrator. Issuing the bug command will pop up a mail editing window if graphics is available, or will allow a message to be entered on the command line if not. The environment variable SPICE_BUGADDR is used to set the Internet address to which bug reports are sent (this can be changed in the pop-up mail editor window). If not set, the report is sent to the Whiteley Research technical support staff. This command takes no arguments.

The mail editor window can also be displayed by pressing the WR button in the Tool Control window.

4.6.2 jobs

The jobs command prints the status of asynchronous *WRspice* runs initiated. The command produces a report on the asynchronous *WRspice* jobs currently running. Asynchronous jobs can be started with the aspice command locally, or on a remote system with the rspice command. *WRspice* checks to see if the jobs are finished every time a command is executed. If a job is finished, then the data are loaded and become available. This command takes no arguments.

4.6.3 help

The help command brings up a help window describing the topic keyword passed as an argument to the command, or the top-level entry if no argument is given.

```
help [-c | topic]
```

When graphics is not available, the help text is presented in a text-only format on the terminal. The HTML to ascii text converter only handles the most common HTML tags, so some descriptions may look a little strange. The figures (and all images) are not shown, and links are not available, except for the "subtopics" and "references" lists.

The help data files are found in directories specified in the `helppath` variable, or from the SPICE_HLP_PATH environment variable. If for some reason the help directory is not found, a very

minimal internal text-mode help system will be provided. The single character '?' is internally aliased to "help".

If the single argument "-c" is given, the internal topic hash tables are cleared. Since the topics are hashed as offsets into the files, if a topic text changes, the offsets will be incorrect. After changes are made to a help file, or new help files are added, if in *WRspice* and the help database has already been cached by viewing any help topic, giving "help -c" will ensure that new topics are found and present topics display correctly.

The helpinitxpos variable specifies the distance in pixels from the left edge of the screen to the left edge of the help window, when it first appears. If not set, the value taken is 100 pixels. The helpinitypos variable specifies the distance in pixels from the top edge of the screen to the top edge of the help window, when it first appears. If not set, the value taken is 100 pixels.

See 3.11 for more information about the *WRspice* help system.

4.6.4 qhelp

The qhelp command prints a brief description of each command listed as an argument. If no arguments are given, all commands are listed. This is not part of the main help system.

4.6.5 quit

The quit command terminates the *WRspice* session. If there are circuits that are in the middle of a simulation, or plots that have not been saved in a file, the user is reminded of this and asked to confirm. The variable noaskquit disables this. *WRspice* can also be terminated from the **Quit** button in the **File** menu of the Tool Control window. The command takes no arguments.

4.6.6 rusage

The rusage command is used to obtain information about the consumption of system resources and other statistics during the *WRspice* session.

```
rusage [all] [resource ...]
```

If any resource keywords are given, only those resources are printed. All resources are printed if the keyword all is given. With no arguments, only total time and space usage are printed. The show command can also be used to obtain resource statistics. The recognized keywords are listed below.

accept
: This keyword prints the number of accepted time points from the last transient analysis.

elapsed
: This keyword prints the total amount of time that has elapsed since the last call of the rusage command, or to the program start time.

faults
: This keyword prints the number of page faults caused by the program thus far.

`loadtime`
If given, print the time spent loading the matrix in the last simulation run. This includes the time spent in computation of device characteristics.

`lutime`
The `lutime` keyword will print the time spent in LU factorization of the matrix during the last simulation run.

`rejected`
This keyword prints the number of rejected time points in the last transient analysis.

`solvetime`
This will print the time spent solving the matrix equations, after LU decomposition, in the last simulation run.

`space`
This keyword will print the memory presently in use by *WRspice*.

`time`
This keyword will print the CPU time used by the last simulation run.

`totaltime`
If this keyword is given, the total CPU time used in the present session will be printed.

`totiter`
This keyword prints the total number of Newton iterations used in the last analysis.

`traniter`
The `traniter` keyword will print the number of iterations used in the last transient analysis.

`tranpoints`
This keyword prints the number of internal time steps used in the last transient analysis.

`transolvetime`
This keyword prints the matrix solution time required by the last transient analysis.

`trantime`
This keyword will print the total time spent in transient analysis in the last transient analysis.

4.6.7 setrdb

The setrdb command adds resources to the X resource database.

`setrdb` *resource*`:` *value*

The user interface toolset currently used to implement the *WRspice* user interface is the GTK+ toolkit (`www.gtk.org`) which does not use the X resource mechanism.

WRspice presently only recognizes resource strings which set the plotting colors for the plot command. The names of these resources are "`color0`" through "`color19`", which correspond directly to the shell variables of the same name, and to the colors listed in the **Colors** tool of the **Tools** menu of the Tool Control window. To set a color using the *setrdb* command, one can use forms like

"`setrdb *color2: pink`"

4.6.8 strcmp

The strcmp command is used for string comparison in scripts.

`strcmp` *varname string1 string2*

The value of *varname* is set to a number that is greater than, equal to, or less than 0 according to whether *string1* is lexically before, equal to, or after *string2*.

4.6.9 where

The where command, which takes no arguments, prints information about the last nonconvergence, for debugging purposes.

4.7 Variables

Shell variables can be set from the shell with the set command. In addition, shell variables are set which correspond to definitions supplied on the .options line of the current circuit, and there are additional shell variables which are set automatically in accord with the current plot. The shell variables that are currently active can be listed with the set command given without arguments, and are also listed within the **Variables** window brought up from the **Tools** menu of the Tool Control window. In these listings, a '+' symbol is prepended to variables defined from a .options line in the current circuit, and a '∗' symbol is prepended to those variables defined for the current plot. These variable definitions will change as the current circuit and current plot change. Some variables are read-only and may not be changed by the user, though this is not indicated in the listing.

Before a simulation starts, the options from the .options line of the current circuit are merged with any of the same name that have been set using the shell. The result of the merge is that options that are booleans will be set if set in either case, and those that take values will assume the value set through the shell if conflicting definitions are given. The merge will be suppressed if the shell variable noshellopts is set *from the shell*, in which case the only options used will be those from the .options line, and those that are redefined using the set command will be ignored.

There are many shell variables that have special meaning to *WRspice*. Note the difference between a variable and a vector — a variable is manipulated with the commands set and unset, and may be substituted in a command line with the $*varname* notation. A vector is a datum which can be plotted, manipulated algebraically, and so forth.

While any variable may be set, only the following ones will have any significance to *WRspice*. In general, variables set in the .options line are available for expansion in $*varname* references, but do not otherwise affect the workings of the shell.

4.7.1 Shell Variables

These variables control behavior of the *WRspice* shell. Most of these variables can be set indirectly from the **Shell** tool in the **Tools** menu of the Tool Control window.

argc
: This read-only variable is set to the number of arguments used to invoke the currently executing script, including the script name. This can be referenced from within scripts only.

argv
: This is a read-only list of tokens from the invoking line of the currently executing script, including the script name. This can be referred to within scripts only.

cktvars
: When this boolean variable is set with the set command or the **Shell** tool (*not* in a SPICE .options line), variables set in the .options line of the current circuit will be treated the same as variables set with the set command.

  With this variable unset, the legacy behavior is maintained, i.e., variables set in .options will work in variable substitution, but will be ignored in most commands.

  In releases prior to 2.2.61, when a variable is set in a .options line, it becomes visible almost like it was set with the set command, when the circuit containing the .options line is the current circuit. In the variables listing (set command without arguments or the **Variables** tool), these have a '+' in the first column. However, they are not part of the normal variable database, and

they only "work" in special cases. For example, they will work in variable substitution, but won't affect the defaults in most commands, such as the plot command. If the same variable is also set with set, the set definition will have precedence. The variables set with .options can't be unset, except by changing the current circuit.

This was confusing to the user. If a .options line contains an assignment for a plot-specific variable (for example), the variable will appear to be active when listed, but it will have no effect on the plot command. It can be argued that making the circuit variables behave the same as those set with the set command would be an improvement. In this case, variables listed in the set or **Variables** tool listing will always have effect, and one can set any variable in the .options line, and have it always "work".

On the other hand, circuit variables can't be unset, so a variable in the current circuit would always have effect, desired or not. Also, changing present behavior would possibly adversely affect existing users who expect the current behavior, and this change might break existing scripts.

The cktvars variable gives the user control over how to handle the circuit variables.

height
: This variable sets the number of lines assumed in a page to use when printing output. If not set, the height is determined from the output device, if possible, and set to 66 otherwise.

history
: The history variable sets the number of commands saved in the history list. The default is 1000.

ignoreeof
: If this boolean variable is set, the EOF character (**Ctrl-D**) is ignored in file input. If not set, an EOF character will terminate the input. When typed as keyboard input, **Ctrl-D** prints a list of completion matches, if command completion is in use.

noaskquit
: If this variable is set, *WRspice* will skip the exit confirmation prompting it there are simulations in progress or unsaved data when a quit command has been given.

nocc
: If this boolean variable is set, command completion will be disabled.

noclobber
: If this boolean variable is set, files will not be overwritten with input/output redirection.

noedit
: By default, command line editing is enabled in interactive mode, which means that *WRspice* takes control of the low level functions of the terminal window. This can be defeated if noedit is set. If the terminal window doesn't work properly with the editor, it is recommended that "set noedit" appear in the .wrspiceinit file. Note that the command completion character is **Tab** when editing is on, and **Esc** otherwise.

  This variable is ignored under Microsoft Windows. The editing is always enabled in that case.

noerrwin
: In interactive mode, error messages are generally printed in a separate pop-up window. When this variable is set, error messages will appear in the console window instead. This variable is automatically set when *WRspice* is started in JSPICE3 emulation mode (-j command line option given).

noglob
: If this boolean variable is set, global pattern matching using the characters '*', '?', '[', and ']' is disabled. This variable is set by default, since '*' is often used in algebraic expressions.

`nomoremode`
If `nomoremode` is not set, whenever a large amount of text is being printed to the screen (e.g., from the print or asciiplot commands), the output will be stopped every screenful and will continue when a character is typed. The following characters have special meaning:

- `q` Discard the rest of the output
- `c` Print the rest of the output without pausing
- `?` Print a help message

If `nomoremode` is set, all output will be printed without pauses.

`nonomatch`
If set, and `noglob` is unset and a global expression cannot be matched, the global characters will be used literally. If not set, lack of a match produces an error.

`nosort`
If this boolean is set, lists of output are not sorted alphabetically.

`prompt`
This variable contains a string to use as the command prompt. In this string, the '!' character is replaced by the event number, and "`-p`" is replaced by the current directory. If the program is reading lines which form a part of a control block, the prompt becomes a set of '>' characters, one for each level of control structure. The default prompt is "`$program ! -> `".

`sourcepath`
This list variable contains directories to search for command scripts or input files.

`unixcom`
When this boolean is set, *WRspice* will attempt to execute unrecognized commands as operating system commands.

`width`
This variable sets the number of columns assumed in printed output. If not set, the width is determined from the output device, if possible, and set to 80 otherwise.

4.7.2 Command Specific Variables

These variables control the operation of specific *WRspice* commands and functions. Most of these variables can be set indirectly from the **Commands** tool in the **Tools** menu of the Tool Control window.

`appendwrite`
When set, data written with the write command will be appended to the rawfile, if the rawfile already exists. If not set, the rawfile will be overwritten.

`checkiterate`
This sets the binary search depth used in finding operating range extrema in operating range analysis initiated with the check command. It can be set to an integer value 0–10. If not set or set to zero, the search is skipped.

`diff_abstol`
This variable sets the absolute error tolerance used by the diff command. The default is 1e-12.

`diff_reltol`
This variable sets the relative error tolerance used by the diff command. The default is 1e-3.

`diff_vntol`
This variable sets the absolute voltage tolerance used by the diff command. The default is 1e-6.

`dpolydegree`
This variable sets the polynomial degree used by the `deriv()` function for differentiation. If not set, the value is 2 (quadratic). The valid range is 0–7.

`editor`
This variable is set to the name or path of the text editor to be used within *WRspice*. This overrides the SPICE_EDITOR and EDITOR environment variables. If no editor is set, the internal editor xeditor is used if graphics is available, otherwise the `vi` editor is used.

`filetype`
This variable can be set to either of the keywords `ascii` or `binary`. It determines whether ascii or binary format is used in the generated rawfiles. The default is ascii, but this can be overridden with the environment variable SPICE_ASCIIRAWFILE, which is set to "1" (for ascii), or "0" (zero, for binary).

`fourgridsize`
When a fourier command is given, the data are first interpolated onto a linear grid. The size of the grid is given by this variable. If unspecified, a size of 200 is used.

`helpinitxpos`
This variable specifies the distance in pixels from the left edge of the screen to the left edge of the help window, when it first appears. If not set, the value taken is 100 pixels.

`helpinitypos`
This variable specifies the distance in pixels from the top edge of the screen to the top edge of the help window, when it first appears. If not set, the value taken is 100 pixels.

`helppath`
This variable specifies the search path used to locate directories containing help database files. This variable takes its initial value from the SPICE_HLP_PATH environment variable, if set, otherwise it assumes a built-in default "`( /usr/local/lib/xictools/wrspice/help )`", or, if XT_PREFIX is defined in the environment, its value replaces "`/usr/local`".

`mplot_cur`
This variable contains the name of the last margin analysis output file generated. This variable can be set, but has no effect, as the file names are generated internally.

`nfreqs`
This variable specifies how many multiples of the fundamental frequency to print in the fourier command. If not set, 10 values are printed.

`noeditwin`
If this boolean variable is set, no window is created for the text editor. This is desirable for editors that create their own windows.

`nopadding`
If set, binary rawfiles with vectors of less than maximum length are not zero padded.

`nopage`
If set, page breaks are suppressed in the print and asciiplot commands. The `nobreak` variable is similar, but suppresses page breaks only in the asciiplot command.

When given in the `.options` line, page ejects are suppressed in printed output, in batch mode.

`noprintscale`
When doing a `print col`, the value of the scale variable is usually printed in the first column of each page. Setting this boolean variable suppresses this.

`numdgt`
This variable specifies the number of significant digits to print in print, asciiplot, fourier, and some other commands. The default precision is six digits.

This variable sets the number of significant digits printed in output from batch mode, when used in the `.options` line.

`rawfile`
This variable sets the default name for the rawfile to be produced. The default is as entered with the `-r` command line option, or "`rawspice.raw`".

`rawfileprec`
This variable sets the number of digits used to print data in an ascii rawfile. The default is 15.

`rhost`
This variable specifies the name of the default machine to submit remote simulations to. This machine must have a `wrspiced` daemon running. The default machine can also be specified in the SPICE_HOST environment variable, which will be overridden if `rhost` is set. Additional machines can be added to an internal list with the rhost command. The host name can be optionally suffixed with a colon followed by the port number to use to communicate with the `wrspiced` daemon. The port must match that set up by the daemon. If not given, the port number is obtained from the operating system for "spice/tcp" or 3004 if this is not defined.

`rprogram`
The name of the program to run when an rspice command is given. If not set, the program path used will be determined as in the aspice command.

`spectrace`
This enables messages to be printed during Fourier analysis with the spec command, for debugging purposes.

`specwindow`
This variable is set to one of the following strings, which will determine the type of windowing used for the Fourier transform in the spec command. If not set, the default is `hanning`.

`bartlet`	Bartlet (triangle) window
`blackman`	Blackman order 2 window
`cosine`	Hanning (cosine) window
`gaussian`	Gaussian window
`hamming`	Hamming window
`hanning`	Hanning (cosine) window
`none`	No windowing
`rectangular`	Rectangular window
`triangle`	Bartlet (triangle) window

`specwindoworder`
This can be set to an integer in the range 2–8. This sets the order when the gaussian window is used in the spec command. If not set, order 2 is used.

`spicepath`
This variable can be set to a path to a simulator executable which will be executed when asynchronous jobs are submitted with the aspice command. If not set, the path used will default to the

value of the environment variable SPICE_PATH. If this environment variable is not set, the path is constructed from the value of the environment variable SPICE_EXEC_DIR prepended to the name of the presently running program. If the SPICE_EXEC_DIR variable is not set, the path used is that of the presently running *WRspice*.

`units`
If this variable is set to "`degrees`", all trig functions will use degrees instead of radians for the units of their arguments. The default is "`radians`".

4.7.3 Plot Variables

These variables control the numerous plotting modes and capabilities of the plot, hardcopy, xgraph, and asciiplot commands. Most of these variables can be set indirectly from the **Plot Defs** panel and the **Colors** panel in the **Tools** menu of the Tool Control window.

`color`*N*
If a variable with the name "`color`*N*" (*N* 1–19) is set to the name of a color the *N*'th value used in a window will have this color. The value of `color0` denotes the background color and `color1` denotes the grid and text color. The color names recognized are those found in the `rgb.txt` file in the X-window system library. These mappings are built into *WRspice* and apply whether or not X is being run. The colors can also be set using the panel brought up by the **Colors** button in the **Tools** menu, and can be set through the X-resource mechanism (see 3.3) and the setrdb command.

The "name" for a color can be given in HTML-style notation: #*rrggbb*, where *rr*, *gg*, *bb* are the hex values for the red, green and blue component of the color.

`combplot`
This is a keyword of the `plotstyle` variable, or can be set as a boolean. It directs the use of a comb plot (histogram) instead of connected points. Each point is connected to the bottom of the plot area by a line.

`curplot`
This variable holds the name of the current plot. It can be set to another plot name (as listed in the `plots` variable), which will become the current plot. This variable can also be set to "`new`", in which case a new, empty plot is created and becomes the current plot.

`curplotdate`
This read-only variable contains the creation date of the current plot.

`curplotname`
This read-only variable contains a description of the type of simulation which produced the current plot. Note that this is not the name used by the setplot command, but rather a description of the type of simulation done.

`curplottitle`
This read only variable contains the title of the circuit associated with the current plot.

`gridsize`
If this variable is set to an integer greater than zero and less than or equal to 10000, this number will be used as the number of equally spaced points to use for the X-axis when plotting in the plot command. The plot data will be interpolated to these linearly spaced points, and the use of this variable makes sense only when the raw data are not equally spaced, as from transient analysis with the `steptype` variable set to `nousertp`. Otherwise the current scale will be used (which may

not have equally spaced points). If the current scale isn't strictly monotonic, then this option will have no effect. The degree of the interpolation is given by the variable `polydegree`.

`gridstyle`
This variable is used to determine the style of grid used by the commands plot, hardcopy, and asciiplot. It can be set to one of the following values:

`lingrid`	Use a linear grid
`loglog`	Use a log scales for both axes
`xlog`	Use a log scale for the X axis
`ylog`	Use a log scale for the Y axis
`polar`	Use a polar grid
`smith`	Transform data and use a Smith grid
`smithgrid`	Use a Smith grid

`group`
This is a keyword of the `scaletype` variable, or can be set as a boolean. It indicates the use of common scales for three categories of data: voltages, currents, and anything else. Each group will have its own Y-scale.

`hcopycommand`
This variable specifies the operating system command which the hardcopy command will use to send a job to the printer. If the string contains the characters `%s`, those characters will be replaced by the name of the file being used to store the plot data, otherwise the file name will be appended to the end of the string, separated by a space character. This allows reference to the file in the middle of the string. For example, suppose that your site has some strange printer, but that there is a filter which converts postscript to a format recognized by that printer. The command string might be "`myfilt <%s |lpr -Pstrange_printer`". Note that double quotes must be used in the set command since the string contains space:

```
set hcopycommand = "myfilt <%s |lpr -Pstrange_printer"
```

`hcopydriver`
This variable specifies the default driver to use in the hardcopy command. The variable should be set to one of the following keywords:

Keyword	Description
`hp_laser_pcl`	mono HP laser
`hpgl_line_draw_color`	color HPGL
`postscript_bitmap`	mono postscript
`postscript_bitmap_encoded`	mono postscript, compressed
`postscript_bitmap_color`	color postscript
`postscript_bitmap_color_encoded`	color postscript, compressed
`postscript_line_draw`	mono postscript, vector draw
`postscript_line_draw_color`	color postscript
`windows_native`	Microsoft Windows native
`wpm`	mono WordPerfect graphics
`wp`	color WordPerfect graphics
`imlib`	tiff, gif, jpeg, png graphics
`xfig_line_draw_color`	format for the `xfig` program

These drivers correspond to the drivers available in the format menu of the **Print** panel from the plot windows.

For PostScript, the line draw drivers are most appropriate for SPICE plots. The bitmap formats will work, but are less efficient for simple line drawings. More information on these drivers can be found in 3.10.1.

`hcopyheight`
This variable sets the default height of the image on the page, as measured in portrait orientation, used by the hardcopy command. It is specified as a floating point number representing inches, unless followed by "cm" (without space) which indicates centimeters. The default is typically 10.5 inches, but this is driver dependent.

`hcopylandscape`
This boolean variable, used by the hardcopy command, will cause plots to be printed in landscape orientation when set.

`hcopyresol`
This variable sets the default resolution used by the driver to generate hardcopy data in the hardcopy command. In almost all cases, the default resolution which is achieved by not setting this variable is the best choice. One case where this may not be true is with the `hp_laser_pcl` driver, where the choices are 75, 100, 150, and 300 (default 150).

`hcopywidth`
This variable sets the default width of the image on the page, as measured in portrait orientation, used by the hardcopy command. It is specified as a floating point number representing inches, unless followed by "cm" (without space) which indicates centimeters. The default is typically 8.0 inches, but this is driver dependent.

`hcopyxoff`
This variable sets the distance of the image from the left edge of the page, viewed in portrait orientation, used by the hardcopy command. It is specified as a floating point number representing inches, unless followed by "cm" (without space) which indicates centimeters. The default is typically 0.25 inches, but this is driver dependent.

`hcopyyoff`
This variable sets the vertical position of the image on the page, viewed in portrait orientation, used by the hardcopy command. Some drivers measure this distance from the top of the page, others from the bottom. This is a consequence of the internal coordinate systems used by the drivers, and the lack of assumption of a particular page size. The offset is specified as a floating point number representing inches, unless followed by "cm" (without space) which indicates centimeters. The default is typically 0.25 inches, but this is driver dependent.

`lingrid`
This is a keyword of the `gridstyle` variable, or can be set as a boolean. It specifies use of a linear grid. This is the default grid type.

`linplot`
This is a keyword of the `plotstyle` variable, or can be set as a boolean. It specifies the display of plot data as points connected by lines. This is the default.

`loglog`
This is a keyword of the `gridstyle` variable, or can be set as a boolean. It specifies use of a log-log grid.

`multi`
This is a keyword of the `scaletype` variable, or can be set as a boolean. It indicates the use of separate Y-scales for each trace of the plot (the default).

`nobreak`
If set, suppress page breaks when doing an asciiplot. The `nopage` variable is similar, but suppresses page breaks in both the asciiplot and print commands.

`noasciiplotvalue`
If set, suppress printing the value of the first variable plotted with asciiplot on the left side of the graph.

`nogrid`
Setting this boolean variable specifies plotting without use of a grid. The data will be plotted with only the border lines at the bottom and left sides of the plotting area.

`nointerp`
This variable is used only by the asciiplot command. Normally asciiplot interpolates data onto a linear scale before plotting it. If this option is given this won't be done — each line will correspond to one data point as generated by the simulation. Since data are already linearized unless from a transient analysis with `steptype` set to `nousertp`, setting this variable will avoid a redundant linearization.

`plotgeom`
This variable sets the size of subsequently created plot windows. It can be set as a string "*wid hei*" or as a list `(` *wid hei* `)`. The *wid* and *hei* are the width and height in pixels.

For Microsoft Windows, the default (when `plotgeom` is unset) width and height are 500, 300 and these apply to the whole window. Due to Microsoft's silly and unnecessary conversion to "dialog units", the actual pixel size may be slightly different.

For others, the default width and height are 400, 300 and these apply to the plotting area only.

The acceptable numbers for the width and height are 100—2000. In the string form, a non-numeric character can separate the two numbers, e.g., `"300x400"` is ok.

`plotposn`*N*
This variable can be used to set the screen position of the *N*'th plot window. It can be specified as a list, as

```
set plotposn0 = ( 100 200 )
```

or as a string, as in

```
set plotposn2 = "150 250".
```

The *N* can range from 0–15. If not set, the plots are positioned by an internal algorithm.

`plots`
This list variable is read-only, and contains the names of the plots available. The `curplot` variable can be set to any of these, or to the word "new", in which case a new, empty plot is created.

`plotstyle`
This variable is used to determine the plot style in the commands plot, hardcopy, and asciiplot. Its value may be one of:

`linplot`	Connect points with line segments
`combplot`	Connect each point to the X-axis
`pointplot`	Plot each point as a discrete glyph

`pointchars`
If this variable is set as a boolean, alpha characters will be used as glyphs for point plots (i.e., the `pointplot` mode is active) in a plot command. If set to a string, the characters in this string are used to plot successive data values. The default is "`oxabcdefhgijklmnpqrstuvwyz`".

`pointplot`
This is a keyword of the `plotstyle` variable, or can be set as a boolean. This will cause data to be plotted as unconnected points. Each successive expression is plotted with a different glyph to mark the points. The glyphs default to an internally generated set, however alpha characters can be used if the variable `pointchars` is set.

`polar`
This is a keyword of the `gridstyle` variable, or can be set as a boolean. It specifies use of a polar grid instead of a rectangular grid.

`polydegree`
This variable determines the degree of the polynomial that is fit to points when a plot is done. If it is not set or set to 1, then the points are connected by lines. If it is greater than 1, then a polynomial curve is fit to the points. If the value of `polydegree` is n, then for each $n + 1$ adjacent points, an nth degree curve is fit. If this is not possible (due to the fact that the points aren't monotonic), the curve is rotated 90 degrees and another attempt is made. If is is still unsuccessful, n is decreased by 1 and the process is repeated. Thus four points in the shape of a diamond may be fit with quadratics to approximate a circle (although it's not clear that this situation comes up often in circuit simulation). The variable `gridsize` determines the size of the grid on which the curve is fit if the data are monotonic. If the `gridsize` variable is zero or not set, or the scale is non-monotonic, no polynomial fitting is done.

`polysteps`
This variable sets the number of intermediate points to plot between each actual point used for interpolation. If not set, 10 points are used.

`scaletype`
This variable is used to determine the treatment of the Y-axis scaling used in displaying the curves in the plot command. Its value may be one of:

`multi`	Use separate Y-scales for each trace (the default)
`single`	Use common Y-scale for all traces
`group`	Use same scale for voltages, currents, and others

`single`
This is a keyword of the `scaletype` variable, or can be set as a boolean. It indicates the use of a common Y-scale for all traces in the plot.

`smith`
This is a keyword of the `gridstyle` variable, or can be set as a boolean. It specifies use of a smith grid instead of a rectangular or polar grid, and an implicit transformation of the data into the "reflection coefficient" space through the relation $S = (z-1)/(z+1)$, where z is the complex input data.

`smithgrid`
This is a keyword of the `gridstyle` variable, or can be set as a boolean. It specifies use of a Smith grid instead of a rectangular or polar grid, and plots the data directly, without transformation. The data must fall within the unit circle in the complex plane to be visible.

`ticmarks`
If this variable is set as a boolean, than an "x" will be printed every 10 points for each curve plotted. This variable may also be set as a number, which will be the number of points between each tic mark. If interpolation is used for plotting, the ticmarks feature is disabled.

`title`
This variable provides a string to use as the title printed in the plot. If not specified, the title is taken as the name of the current plot.

`xcompress`
This variable can be set to an integer *value*. It specifies that we plot only one out of every *value* points in each of the vectors.

`xdelta`
This value is used as the spacing between grid lines on the x-axis, if set.

`xglinewidth`
This variable specifies the line width in pixels to use in xgraph plots. If not set, a minimum line width is used.

`xgmarkers`
If set, xgraph point plots will use cross marks, otherwise big pixels are used.

`xindices`
This variable can be set as a list `(` *lower upper* `)` or as a string "*lower upper*", where *lower* and *upper* are integers. Only data points with indices between *lower* and *upper* are plotted. The value of *upper* must be greater or equal to *lower*.

`xlabel`
This variable provides a string to be used as the label for the x-axis. If not set, the name of the scale vector is used.

`xlimit`
This variable can be set as a list `(` *lower upper* `)` or as a string "*lower upper*", where *lower* and *upper* are reals. The plot area in the x-direction is restricted to lie between `lower` and *upper*. The area actually used may be somewhat larger to provide nicely spaced grid lines, however.

`xlog`
This is a keyword of the `gridstyle` variable, or can be set as a boolean. It specifies use of a log scale for the x-axis and a linear scale for the y-axis.

`ydelta`
This value is used as the spacing between grid lines on the y-axis, if set.

`ylabel`
This variable provides a string to be used as the label for the y-axis. If not set, no label is printed.

`ylimit`
This variable can be set as a list `(` *lower upper* `)` or as a string "*lower upper*", where *lower* and *upper* are reals. Setting this variable will limit the plot area in the y-direction to lie between *lower* and *upper*. It may be expanded slightly to allow for nicely spaced grid lines.

`ylog`
This is a keyword of the `gridstyle` variable, or can be set as a boolean. It specifies use of a log scale for the y-axis and a linear scale for the x-axis.

`ysep`
If this boolean is set, the traces will be provided with their own portion of the vertical axis, so as to not overlap. Otherwise, each trace may occupy the entire vertical range on the plot.

4.7.4 Simulation Option Variables

These variables control defaults related to simulation. Most of these variables can be set indirectly from the **Sim Defs** tool in the **Tools** menu of the Tool Control window, which is equivalent to using the `set` command. They (and any variable) can also be included in an `.options` line in the input file. Before a simulation starts, the options from the `.options` line are merged with any that have been set using the shell. The result of the merge is that options that are booleans well be set if set in either case, and those that take values will assume the value set through the shell if conflicting definitions are given. The merge will be suppressed if the shell variable `noshellopts` is set from the shell, in which case the only options used will be those from the `.options` line, and those set using the `set` command will be ignored.

In the listing of variables provided by the `set` command without arguments, the variables set in the `.options` line of the current circuit will be listed with a "`+`" in the first column. The variables set in the `.options` line are available for substitution (into `$`*variable* references) when the circuit is the current circuit, but otherwise do not affect the shell. For example, setting the variable `noglob` from a `.options` line will *not* affect the global expansion of the shell, but references to `$noglob` would behave as if the boolean `noglob` was set, while the circuit is the current circuit.

Shell variables set in the `.options` line are set before the rest of the SPICE text is expanded, so that shell variable references in the text can be defined from the `.options` line, as in the `.exec` block. The `.exec` lines are executed before the `.options` lines are expanded.

`abstol`
This variable sets the absolute error tolerance used in convergence testing. The default value is 1e-12.

`bypass`
When this bool variable is set, *WRspice* will bypass computation of unchanging elements, which may improve execution speed.

`chgtol`
This variable sets the charge tolerance used in convergence testing. The default value is 1e-14.

`defad`
This variable sets the value for MOS drain diffusion area. The default is 0.0.

`defas`
This sets the value for MOS source diffusion area. The default is 0.0.

`defl`
This sets the value for MOS channel length. The default is 100.0 micrometers.

`defw`
This variable sets the value for MOS channel width. The default is 100.0 micrometers.

`dphimax`
This variable sets the maximum allowable phase change of sinusoidal and exponential sources between internal time points in transient analysis.

Consider a circuit consisting of a sinusoidal voltage source driving a resistor network. The internal transient time steps are normally determined from a truncation error estimation from the numerical integration of reactive elements. Since there are no such elements in this case, a large, fixed time step is used. This may not be sufficient to reasonably define the sinusoidal source waveform, so the timestep is cut. This variable sets the time scale for the cut. The default value of $\pi/5$ provides about 10 points per cycle. All of the built-in source functions that are exponential or sinusoidal reference this variable in the timestep cutting algorithm.

This variable also limits the transient time step when Josephson junction devices are present, i.e., it is equivalent to the `jjdphimax` variable in JSPICE3.

The variable can be set in the range $\pi/1000$ – pi, and is taken as $\pi/5$ if unset.

`gmin`
This sets the value of `gmin`, the minimum conductance allowed by the program. The default value is 1e-12.

`gminsteps`
This variable sets the number of gmin steps used in finding the operating point, if convergence initially fails. The default is 10. In gmin stepping, conductances are added to the matrix diagonal, and reduced by a factor of 10 for each gmin step so that on the final step the conductances are equal to the `gmin` variable. This iterative process most often leads to convergence.

`interplev`
In transient analysis, in the default `steptype` mode, internal timepoint data are interpolated onto the external (user supplied) time points. Only the interpolated data are saved. This variable sets the polynomial degree of interpolation, in the range 1–3. The default is 1 (linear interpolation).

`itl1`
The `itl1` variable sets the dc iteration limit before convergence failure is indicated. The default is 400.

`itl2`
The `itl2` variable sets the dc transfer curve iteration limit before convergence failure is indicated. The default is 100.

`itl4`
This variable sets the number of timepoint iterations in transient analysis above which convergence failure is indicated. The default is 10.

`maxdata`
This variable sets the maximum allowable memory stored as plot data during an analysis, in kilobytes. The default is 32000. For all analyses except transient with the `steptype` variable set to "`nousertp`", the run will abort at the beginning if the memory would exceed the limit. Otherwise, the run will end when the limit is reached.

`maxord`
This variable sets the maximum order of the integration method used. Setting this to 1 will always use rectangular integration. If unset, the value taken is 2, which is the maximum order for the default trapezoidal integration. If Gear integration is used, the maximum order is 7.

`method`
This string can be set to either of the keywords "`trap`", which is the default and sets trapezoidal integration, or "`gear`", for Gear integration. The `maxord` variable sets the maximum order of the integration.

`minbreak`
This sets the minimum interval between breakpoints in transient analysis, which is otherwise set to 1e-20.

`modelcard`
This variable allows the keyword that specifies a model to be reset. If unset, the keyword is "`.model`".

`nobjthack`
If this boolean is set, bipolar transistors are assumed to have four nodes. Otherwise, three nodes are acceptable. This only affects subcircuit expansion.

`noiter`
Not currently implemented.

During transient analysis, at each new time step, Newton iterations are used to solve the nonlinear circuit equations. The first iteration, the prediction step, uses extrapolation from past values to obtain a best guess at the solution for use as input. Additional iterations use the previous output values as input.

In cases where the nonlinearity is weak, or where the internal time step is forced to be small (as when Josephson junctions are present) iterations beyond the predictor sometimes lead to unneeded accuracy. Setting the variable `noiter` causes skipping of iterations beyond the prediction step, and also skipping of certain other code. This maximizes the simulation rate, but can lead to errors if not used carefully. Much the same effect can be obtained by setting `reltol` to a large value, however `noiter` is more efficient as convergence testing and matrix loading are skipped, as there is a-priori knowledge that no iterations are to take place. The iteration count and total internal timepoint count are available from the rusage command.

`nojjtp`
During transient analysis with Josephson junctions present, the default time step is given by $T = \phi/vmax$, where $\phi = \Phi_0/2\pi$ (ϕ = 3.291086546e-16, Φ_0 is the magnetic flux quantum) and $vmax = max(Vj, sqrt(\phi Jc/C))$. If the variable `nojjtp` is set, the timestep is determined from a truncation error calculation, as is the case when Josephson junctions are not present in the circuit. The user should experiment to determine which timestep leads to faster execution.

`noopiter`
When computing the operating point, the iterator is called directly, and if this fails gmin stepping and source stepping are used to try and obtain convergence. If this variable is set, the direct convergence attempt is bypassed, and the flow begins with gmin stepping.

`noshellopts`
If set, do not use *WRspice* options that have been set interactively through the shell. Use only options that appear in a `.options` line in the circuit file when running a simulation of the circuit.

`oldlimit`
When set, the SPICE2 limiting algorithm for MOS devices is used. Otherwise, an improved limiting procedure is used.

`pivrel`
This variable sets the relative ratio between the largest column entry and an acceptable pivot value. The default value is 1e-3. In the numerical pivoting algorithm the allowed minimum pivot value is determined by `epsrel=max(pivrel*maxval,pivtol)` where `maxval` is the maximum element in the column where a pivot is sought (partial pivoting).

`pivtol`
This variable sets the absolute minimum value for a matrix entry to be accepted as a pivot. The default value is 1e-13.

`reltol`
This sets the relative error tolerance used in convergence testing. The default value is 0.001 (0.1 percent).

`renumber`
When set, the source lines are renumbered sequentially after subcircuit expansion.

`savecurrent`
If this variable is set, then all device current special vectors are saved in the plot by default during analysis. This enables plotting of device currents using the `@`*device*`[`*param*`]` construct.

`srcsteps`
If gmin stepping fails, convergence is attempted by raising the sources step-wise to their specified values, iteratively. This variable sets the number of steps used in this technique. The default is 10.

`steptype`
This string can be set to one of three keywords which determine the data output mode in transient analysis. It can be set to "`interpolate`", which is the default, "`hitusertp`", or "`nousertp`".

If not set, or set to "`interpolate`", output points are interpolated from internal time points to the user time increments, with degree 1 (the default) to 3, set by the `interplev` variable.

If set to "`hitusertp`", then during transient analysis the time step will be cut so as to land on the user time points. This requires more simulation time, but provides the greatest accuracy.

Setting to "`nousertp`" will cause internal timepoint data to be saved, either in internal data structures in interactive mode or in the rawfile in batch mode. The amount of data can grow quite large.

`subend`
This variable allows the keyword which ends a subcircuit definition to be changed. If unset, the keyword is "`.ends`".

`subinvoke`
This variable allows the prefix which invokes a subcircuit to be changed. If unset, the prefix is "`x`".

`substart`
This variable allows the keyword which begins a subcircuit definition to be changed. If unset, the keyword is "`.subckt`".

`temp`
This variable specifies the assumed operating temperature of the circuit under simulation. The default value is 27C (300 K).

`tnom`
The `tnom` variable sets the nominal temperature. The default value is 27C (300 K). This is the temperature at which device model parameters are assumed to have been measured.

`trtol`
This sets the transient error tolerance. The default value is 7.0. This parameter is an estimate of the factor by which *WRspice* overestimates the actual truncation error.

`trytocompact`
This boolean variable is applicable only to the LTRA model. When specified, the simulator tries to condense LTRA transmission line past history of input voltages and currents.

`vntol`
This variable sets the absolute voltage error tolerance used in convergence testing. The default value is 1 microvolt.

In addition, the following variables have no significance to *WRspice* when in interactive mode, but control operations in batch mode. Also, there are SPICE2 options which are recognized but do nothing. These are listed below.

`acct`
When *WRspice* is run in batch mode, print out accounting information at the end of the run. This boolean variable has meaning only when set in the input file in a `.options` line.

`cptime`
The SPICE2 option to set the maximum allowable cpu time for the job. This has no effect in *WRspice*.

`itl3`
The SPICE2 option to set the lower transient iteration limit for timestep control. This is not used in *WRspice*.

`itl5`
The SPICE2 option to set the maximum number of iterations for the job. This is not used in *WRspice*.

`limpts`
The SPICE2 variable which sets the maximum number of points per analysis. This is not used in *WRspice*.

`limtim`
The SPICE2 option to reserve time for output generation. This is not used in *WRspice*.

`list`
When *WRspice* is run in batch mode, list the circuit before running the simulation. This boolean variable has meaning only when set in the `.options` line of the input file.

`lvlcod`
The SPICE2 option to generate machine code. This is not used in *WRspice*.

`lvltim`
The SPICE2 variable to set the type of timestep control. This is not used in *WRspice*.

`node`
The SPICE2 variable to print a node summary. This boolean variable has meaning only when set in the `.options` line of the input file.

`nomod`
The SPICE2 variable to suppress printing of a summary of models. This is not used in *WRspice*.

`opts`
When *WRspice* is run in batch mode, print out all the variables set and their values. This boolean variable has meaning only when set in the `.options` line of the input file.

4.7.5 Debugging Variables

These variables turn on debugging modes and otherwise provide debugging utility. Most of these variables can be set indirectly from the **Debug** tool in the **Tools** menu of the Tool Control window.

`debug`
: This variable may be a boolean (i.e., set to nothing), in which case all debugging is turned on, a string token from the list below, in which case the string specifies which part of the program to enable debugging for, or a list of these strings, which enables any combination. The possible values are:

`async`	The aspice and rspice code
`control`	The control structure code
`cshpar`	The C-shell pre-processor and parser
`eval`	The expression evaluation routines
`ginterface`	Graphics package interface routines
`helpsys`	The help system
`plot`	The plotting routines
`parser`	The parser for expressions
`siminterface`	The interface to the simulator
`vecdb`	The vector database

`display`
: This variable contains the display name for X used by the graphics system, generally of the form *host*:*number*. This variable is read-only.

`dontplot`
: This variable disables the plotting system for debugging purposes. When this variable is set, and a plot command is given, no graphical operations are performed.

`noparse`
: This variable turns off the parsing of circuit descriptions when a file is sourced with the source command, for debugging purposes. The circuit cannot be simulated if it isn't parsed.

`nosubckt`
: This variable disables the expansion of subcircuits when set, for debugging purposes. A circuit with subcircuits cannot be parsed if this is set.

`program`
: This variable contains the full path name of the program.

`strictnumparse`
: When this variable is set, *WRspice* will not allow trailing characters after a number, unless they are separated from the number with an underscore ('_'). This may prevent errors, for example writing "`1meter`" and expecting it to have a value of 1.

Chapter 5

Margin Analysis

WRspice has provision for automated operating range and Monte Carlo analysis. Both types of analysis perform repeated simulation runs with varying parameters, and record whether or not the circuit "worked" with that parameter set. Writing the code that tests whether the circuit is functioning properly or not is probably the major challenge in applying these analyses. It is usually helpful to have a thorough understanding of how the circuit behaves before performing margin analysis. The margin analysis is one of the later steps in circuit design.

Both types of margin analysis can use a file format which contains the SPICE deck plus executable statements. There are actually two formats recognized, one for compatibility with the JSPICE3 program, and a new format particular to *WRspice*. Use of one of these formats is the most straightforward method of initiating margin analysis, however there are short-cuts and hooks for more advanced users. The scripting capability is a powerful tool, and in general allows much tedium to be automated.

5.1 Operating Range Analysis

In operating range analysis, a suitably configured source file containing a circuit description is evaluated over a two dimensional area of parameter space, producing an output file describing a true/false result at each evaluated point. The algorithm and implementation are designed to be as efficient as possible to speed execution. Results can be viewed graphically during or after simulation.

As with conventional circuit and command files, operating range analysis files can be sourced by simply typing in the file name. If the file name happens to conflict with a *WRspice* command, then the file can be input with the source command by typing

`source` *filename*

In batch mode, the operating range analysis is performed immediately. Otherwise, actual operating range analysis is performed with the check command (see 4.2.4). In batch mode, the check command is run automatically, if the file has certain properties to be described.

In order to initiate margin analysis with the check command, the current circuit must be from a margin analysis file, or have appropriate bound codeblocks. Every circuit suitable for margin analysis must have a control block which contains a shell routine which will evaluate the circuit variables and establish whether or not the operation is correct. If operation is incorrect, a vector named "`checkFAIL`"

must be set to a non-zero value. These control statements can be supplied in the circuit file in a block initiated with a `.control` line and ending with a `.endc` line, or through another file added as a codeblock and bound to the "controls" of the circuit, through use of the codeblock command.

A second block of statements, the "header" or "exec" block, is typically required, though it is not an error if none is provided. This block provides initializing statements, and is executed at the start of operating range analysis, or at the start of each trial in Monte Carlo analysis. This block can be provided in the circuit file within an `.exec` and an `.endc` line, or can be a bound codeblock, bound to the "execs" of the circuit.

Monte Carlo analysis files differ from operating range files only in the header or `.exec` lines (or header codeblock). During Monte Carlo analysis, the header block is executed before every simulation so that variables can be updated. In operating range analysis variables are initialized by the header block only once, at the start of analysis.

If the circuit has a line with the characters `.monte`, then Monte Carlo analysis is assumed, and the `-m` option to the check command is unnecessary. Similarly, a `.checkall` line will imply the checking of all points in operating range analysis, making the `-a` option to the check command unnecessary. A line containing the characters `.check` will indicate (the default) operating range analysis. One of these lines must appear if the file is to be analyzed in batch mode. These lines also suppress the automatic execution of the `.exec` lines and the `.control` lines as the file is sourced (the `.exec` lines are actually executed, but no vectors are saved, to enable correct shell variable expansion). A line containing the string `.noexec` appearing in the circuit file will have the same effect.

There are a number of vectors with defined names which control operating range and Monte Carlo analysis. In addition, there are relevant shell variables. The vectors created for use in an analysis run are assigned to a plot structure created for the analysis. This plot becomes the current plot after the analysis starts. These vectors are usually set in the header (`.exec`) block, unless the defaults are used. They can also be set by hand, or under the control of another script, if the current plot is the `constants` plot, before starting the analysis. The pre-named vectors are as follows:

`checkPNTS` (real, length >= 1)
: These are the points of the scale variable (e.g., `time` in transient analysis) at which the pass/fail test is applied. If a fail is encountered, the simulation is stopped and the next trial started. If not specified, the pass/fail test is applied after the trial is finished. The `checkPNTS` variable is usually set in the header to a list of values with the compose command.

`checkVAL1` (real, length 1)
: This is the initial central value of the first parameter to be varied during operating range analysis. It is not used in Monte Carlo analysis.

`checkDEL1` (real, length 1)
: The first central value will be incremented or decremented by this value between trials in operating range analysis. It is not used in Monte Carlo analysis.

`checkSTP1` (integer, length 1)
: This is the number of trials above and below the central value. In Monte Carlo analysis, it partially specifies the number of simulation runs to perform, and specifies the X-axis of the visual array used to monitor progress (with the mplot command). In operating range analysis, the default is zero. In Monte Carlo analysis, the default is 3.

`checkVAL2 checkDEL2 checkSTP2`
: These are as above, but relate to the second parameter to be varied in the circuit in operating range analysis. In Monte Carlo analysis, only `checkSTP2` is used, in a manner analogous to `checkSTP1`.

The total number of simulations in Monte Carlo analysis is `(2*checkSTP1 + 1)*(2*checkSTP2 + 1)`, the same as would be checked in operating range analysis. The `checkSTP2` variable sets the number of cells in the Y-axis of the plot produced by mplot.

`checkFAIL` (integer, length 1, 0 or nonzero)
This is the global pass/fail flag, which is set after each trial, nonzero indicates failure. This variable is used in both operating range and Monte Carlo analysis. This variable is set by the code which evaluates the pass/fail criteria.

`opmin1, opmax1` (real, length >= 1)
The operating range analysis can be directed to find the operating range extrema of the first parameter for each value of the second parameter. These vectors contain the values found, and are automatically generated if the range finding feature is enabled. They are not generated in Monte Carlo analysis.

`opmin2, opmax2` (real, length >= 1)
The operating range analysis can be directed to find the operating range extrema of the second parameter for each value of the first parameter. These vectors contain the values found, and are automatically generated if the range finding feature is enabled. They are not generated in Monte Carlo analysis.

`range, r_scale` (real, length >= 1)
If the range finder was active, these vectors are automatically created and added to the plot. The `range` vector and its scale `r_scale` contain all of the extrema data, formatted in such a way that the path is the contour of the boundary of the pass region. The plot command can be used to display this contour by entering "`plot range`".

`value` (real, length variable)
This vector can be used to pass trial values to the circuit, otherwise shell variables are used. This pertains to operating range and Monte Carlo analysis. The name of this vector can be redefined by setting a shell variable named "value" to a new name.

`checkN1, checkN2` (integer, length 1)
These are the indices into the `value` array of the two parameters being varied in operating range analysis. The other entries are fixed. These vectors are not used if shell variables pass the trial values to the circuit, and are not used in Monte Carlo analysis.

The shell variables are:

`checkiterate` (integer 0-10)
This sets the binary search depth used in finding operating range extrema. If not set or set to zero, the search is skipped. The binary search is used to find the exact values of the operating region boundary, and has no relevance to the usual set of pass/fail outputs generated with the check command. If nonzero, during operating range analysis and *not* in all-points mode, the extrema for each row and column are found, and saved in the `opmin1`, `opmax1`, `opmin2`, and `opmax2` vectors, which are then used to generate the `range` and `r_scale` vectors described above.

If both the the input vectors `checkSTP1` and `checkSTP2` are unset or set to zero, the range finder behaves somewhat differently. In this case, if the all-points mode is active, and the file is using an input "value" vector rather than shell variables for alterable parameters, then the range of each of these parameters is determined. A masking facility allows some of these inputs to be skipped. If the all-points mode is not set, the range for the two variables is found. The range finder is described in more detail below. The range finder is not used in Monte Carlo analysis, and the `checkiterate` variable is ignored in that case.

`value1, value2`
: The `value1` and `value2` variables are set to the current trial values to be used in the circuit (parameters 1 and 2). The SPICE deck should reference these variables (as `$value1` and `$value2`) as the parameters to vary. Alternatively, the *vector* `value` array can be used for this purpose. These variables can be used in Monte Carlo analysis, but are not set implicitly.

If any of the shell variables `value1`, `value2`, or a *shell* variable named "`value`" are set to a string, then the shell variable or vector named in the string will have the same function as the assigned-to variable. For example, if in the header one has `set value1 = C1`, then the variable reference `$C1` would be used in the file to introduce variations, rather than `$value1`. Similarly, if we have issued `set value = myvec`, the vector `myvec` would contain values to vary (using the pointer vectors `checkN1` and `checkN2`), and a reference would have the form `$&myvec[$&checkN1]`. Note that the alternate variables are not automatically defined before the circuit is parsed, so that they should be set to some value in the header. The default `$value1` and `$value2` are predefined to zero.

The `checkVAL1`, `checkDEL1`, etc. vectors to be used must be defined and properly initialized, either in the deck or directly from the shell, before analysis.

The operating range analysis sets the shell variables `value1` and `value2` to the variables being varied. In addition, vector variables can be set. This is needed for scripts such as optimization where the parameter to be varied is required to be under program control. If a vector called `value` is defined, and a vector called `checkN1` is defined, and `checkN1 >= 0` and `checkN1 <` the length of `value`, then `value[checkN1]` is set to `$value1`. Similarly, if a vector called `value` is defined, and a vector called `checkN2` is defined, and `checkN2 >= 0` and `checkN2 <` the length of `value`, then `value[checkN2]` is set to `$value2`. Thus, instead of invoking `$value1` and `$value2` in the SPICE text, one can instead invoke `$&value[$&checkN1], $&value[$&checkN2]`, where we have previously defined the vectors `value`, `checkN1`, `checkN2`. Thus, the file could have a number of parameters set to `$&value[0], $&value[1], ...` . If `checkN1` is set to 2, for example, `$&value[2]` would be varied as parameter 1. The unreferenced values would be fixed at predefined entries. As mentioned above, the `value1`, `value2`, and `value` names can be redefined by assigning the name of a new variable to the shell variable name being reassigned, using the `set` command.

The range is constructed by row, where columns represent different values for `value1`. A second pass fills in concave contours in column order, thus the same pattern should be obtained independently of the parameter ordering. Patterns with islands or reentrancy may not be displayed correctly. The only way to make the algorithm completely foolproof is to check every point, which is achieved by giving the `-a` option to the `check` command, or by using `.checkall`.

During the analysis, a binary search can be employed to determine the actual values of the edges of the operating region. This feature is enabled by setting the shell variable `checkiterate` to some value between 1 and 10. This is the depth of the binary search used to find the endpoint. A binary search will be performed during conventional operating range analysis only, and is skipped (other than in the exception noted below) if in all-points mode (`-a` flag or `.checkall` line given). The search is skipped if there are no pass points in the row or column. The computed values are stored in the `opmin1`, etc. vectors, where the zeroth element corresponds to the lowest value of the fixed parameter. For example, `opmin1[0]` is the minimum value of parameter 1 when parameter 2 is value2 - steps2*delta2. Entries of these vectors corresponding to points that were not found are zero.

The value to set for the `checkiterate` variable is a trade-off between accuracy and execution time. If the boundary is found within the parameter range defined by the input vectors (and as plotted with the `mplot` command), the error is bounded by $delta/2^n$, where *delta* is the appropriate `checkDEL1` or `checkDEL2` value, and n is the `checkiterate` value. If the extremum is found outside of the given parameter space, the error may be $val/2^n$, where *val* is the value at the edge of the parameter space

nearest the solution.

After an operating range analysis with range finding is complete, two new vectors, `range` and `r_scale`, are created from the `opmin1`, etc. vectors and added to the current plot. These vectors incorporate all of the nonzero entries in such a way that they form a path describing the boundary of the operating region, with `range` containing Y-data and `r_scale` containing X-data. This contour can be displayed by plotting the `range` vector with the `plot` command.

The algorithm used the evaluate a row is shown below. This is the normal algorithm; if the `-a` flag is given to the `check` command, or a `.checkall` line was found in the file, the points are simply stepped through, and no binary searching is done.

```
for each value2 value {
    start at left
    value1 = central1 - delta1 * nsteps1
    loop {
        analyze
        record point
        if (pass) break
        value1 = value1 + delta1
        if (value1 > central1 + delta1 * nsteps1) break
    }
    if (pass)
        do binary search for lower extremum

    start at right
    value1 = central1 + delta1 * nsteps1
    loop {
        analyze
        record point
        if (pass) break
        value1 = value1 - delta1
        if (value1 < central1 - delta1 * nsteps1) break
    }
    if (pass)
        do binary search for upper extremum
}
```

If both `checkSTP1` and `checkSTP2` are zero or not defined, the range finder can have an additional operating mode. This mode is made active if the all-points mode is active (`-a` option or `.checkall` given), and a vector is being used to supply trial values, rather than shell variables. If a vector named "value" is defined, or a vector defined whose name is assigned to the shell variable named "value", the range of each of the components can be computed. Note that the vector can have arbitrarily many entries, and each of these ranges can be found. The range finding can be skipped for certain entries by defining a mask vector. This is a vector with the same length as the `value` vector, and the same name as the `value` vector but suffixed with "`_mask`" as in `value_mask`. Each non-zero entry in the mask signifies that the corresponding variable in the `value` array will *not* be tested for range. Additionally, any entry in the `value` vector which is zero will not be tested. If no mask vector is defined, the range will be computed for all nonzero entries. The results are placed, somewhat arbitrarily, in the `opmin1` and `opmax1` vectors, which will have lengths equal to that of the `value` vector. Skipped entries will be zero. No `range` vector will be produced, since it is not relevant in this mode.

If not in all-points mode, the range will be computed for the shell variables. The `opmin1`, etc. will

contain the maximum and minimum values (length 1). The `range` vector will contain the four points found. Note that the central value must be a pass point in either of these modes, or the range finding is skipped. There is no output file produced when both `checkSTP1` and `checkSTP2` are zero or undefined.

One can keep track of the progress of the analysis in two ways. *WRspice* will print the analysis point on the screen, plus indicate whether the circuit failed or passed at the point, if the `-v` option is given to the check command. Shell echo commands can be used in the executable blocks to provide more information on screen, and echoed output is printed whether or not `-v` is given. The second method uses the mplot command, which graphically records the pass/fail points. If "`mplot -on`" is given before the analysis, the results are plotted as simulation proceeds.

During operating range analysis, a file named *basename*.d*xx* is created in the current directory, where *basename* is the base name of the input file, and *xx* is 00–99, set automatically to avoid clobbering existing files. The output file name is stored in the `mplot_cur` shell variable.

There is a special echof command that allows text to be printed in the output file. The `echof` command is used exactly as the echo command. If there is no output file open, the command returns with no action. The echof command can be used in either `.control` or `.exec` blocks in the input file.

5.2 Operating Range Analysis File Format

There are two recognized file formats which can be used as input for operating range analysis. One, the "old format", is retained for compatibility with an older version of SPICE. *WRspice* recognizes a second "new format" which is more consistent with standard *WRspice* input file organization. In both cases, the input file which specifies operating range analysis consists of three sections:

1. an initializing header
2. a body of control statements
3. the circuit description

5.2.1 Initializing Header

In the old format, the file must begin with a line containing only the string

```
.check
```

which is followed by shell commands. The header block in the old format is terminated with a line containing only the string

```
.control
```

which also begins the control statement block.

In the new format, the first line of the file is taken to be a title line and is otherwise ignored, consistent with other types of input files for *WRspice*. The header statements are found within a block which starts with a line containing only the string

```
.exec
```

and ends with a line containing only the string

```
.endc
```

in other words, a standard `.exec` block. The comment prefix `*@` can also be used to enter header block text, as in described in 2.10.1. The new format file for margin analysis should also contain a line with only the string

```
.check
```

somewhere in the text. Unlike the old format, the ordering of the `.exec block` and the `.check` line is unimportant.

The lines in the header block initialize internally defined variables. The variables are those listed above as user-set, including the `checkiterate` shell variable. Variables which are not used (such as those for variable 2 in a one dimensional case) can be ignored.

An example header is given below:

Old format:

```
.check
compose checkPNTS values 50p 100p 150p 200p
checkVAL1 = 12
checkDEL1 = .5
checkSTP1 = 5
checkVAL2 = .5
checkDEL2 = .1
checkSTP2 = 2
```

New format:

```
* Title for this file
.check
.exec
compose checkPNTS values 50p 100p 150p 200p
checkVAL1 = 12
checkDEL1 = .5
checkSTP1 = 5
checkVAL2 = .5
checkDEL2 = .1
checkSTP2 = 2
.endc
```

The variables `checkFAIL`, `checkSTP1`, and `checkSTP2` are integers. The other variables are real, except for `checkPNTS` which is a real vector.

The header block can also be supplied as a bound codeblock. This is accomplished, for example, with the command

`codeblock -abe` *filename*

where *filename* is the name of a file which contains the statements to be used in the header block. If an `exec` codeblock is bound to the circuit, the bound block is executed rather than any locally specified header block.

5.2.2 Control Statements

The control statement block is almost identical in the old and new formats. In the old format, the control block immediately follows the header block, though in the new format this is not necessary. The control statements are evaluated at each of the `checkPNTS`, and set the `checkFAIL` flag if the logic determines that the circuit run has failed.

This control block begins with a line containing only the string

```
.control
```

and ends with a line containing only

```
.endc
```

i.e., the standard form for a *WRspice* control block (see 2.10.1).

The enclosed lines are *WRspice* script statements that perform a logical comparison of circuit variables and set the `checkFAIL` variable accordingly.

The control block can also be supplied as a bound codeblock. This is accomplished, for example, with the command

```
codeblock -ab filename
```

where *filename* is the name of a file which contains the statements to be used in the control block. If a `control` codeblock is bound to the circuit, the bound block is executed rather than any locally specified control block.

5.2.3 Circuit Description

In the old format, the circuit description starts immediately after the end of the control block, with the title line. In the new format, the title line is the first line of the file, and the circuit description is by definition what is left after removing the `.exec` and `.control` blocks.

This circuit description section of the file consists of conventional *WRspice* format circuit description lines. The parameters to be varied are replaced with `$value1` and `$value2`. Alternatively, one can define a vector called `value`, and unit length vectors `checkN1` and `checkN2`. Then, the parameters to be varied can be replaced with `$&value[$&checkN1]` and `$&value[$&checkN2]`. During analysis, the `$value1` and `$value2` (and the value vector entries, if used) are replaced with the current values of the variables.

Note that in the circuit description, it is often useful to use the concatenation character `%` to add a suffix. For examples, the file line might be

```
v1 0 1 pulse (0 5m 10p ...)
```

where we want to vary the "5m". If the value of $value1 is 5, one could replace this line with

```
v1 0 1 pulse (0 $value1%m 10p ...)
```

Without the %, the variable substitution would fail. Alternatively, one could set $value1 to 5e-3, and not use the "m" suffix in the file.

5.3 Example Operating Range Analysis Control File

The listing that follows is an operating range analysis control file for a Josephson binary counter circuit.

```
3 stage Josephson counter, operating range analysis
.check
.exec
# Margins of a Josephson binary counter
# This is an example of an operating range analysis input file
#
# After sourcing the file, optionally enter "mplot -on" to see results
# graphically, then "check" to initiate run.  The results will be left
# in a file.
#
compose checkPNTS values 50p 135p 185p 235p 285p 335p 385p 435p 485p
checkFAIL = 0
# above two lines are required in header, the rest are optional
#
# central value of first variable, number of evaluation steps above and
# below, step delta:
checkVAL1 = 13
checkSTP1 = 5
checkDEL1 = .5
#
# same thing for second variable
checkVAL2 = 38
checkSTP2 = 5
checkDEL2 = 1
#
# one can define other initialized constants here as well
failthres = 1
#
# end of header
.endc
.control
#
# The following code is evaluated just after the time variable exceeds
# each one of the checkPNTS
#
if time > checkPNTS[0]
        if time < checkPNTS[1]
# time is 50p, set quiescent phase differences.  Uninitialized variables
```

```
# do not require declaration in header
                p0 = v(200) - v(201)
                p1 = v(300) - v(301)
                p2 = v(400) - v(401)
                checkFAIL = 0
# echo "tp1" to screen
                echo tp1
        end
end
if time > checkPNTS[1]
        if time < checkPNTS[2]
# time = 135p, state should be '001'. if not set checkFAIL to 1
# pi and the other variables in the 'constants' plot are known
                if abs(v(200) - v(201) + p0 - 2*pi) > failthres
                        checkFAIL = 1;
                end
                if abs(v(300) - v(301) - p1) > failthres
                        checkFAIL = 1;
                end
                if abs(v(400) - v(401) - p2) > failthres
                        checkFAIL = 1;
                end
                echo tp2
        end
end
if time > checkPNTS[2]
        if time < checkPNTS[3]
# time = 185p, state should be '010'. if not set checkFAIL to 1
                if abs(v(200) - v(201) - p0) > failthres
                        checkFAIL = 1;
                end
                if abs(v(300) - v(301) + p1 - 2*pi) > failthres
                        checkFAIL = 1;
                end
                if abs(v(400) - v(401) - p2) > failthres
                        checkFAIL = 1;
                end
                echo tp3
        end
end
if time > checkPNTS[3]
        if time < checkPNTS[4]
# time = 235p, state should be '011'. if not set checkFAIL to 1
                if abs(v(200) - v(201) + p0 - 2*pi) > failthres
                        checkFAIL = 1;
                end
                if abs(v(300) - v(301) + p1 - 2*pi) > failthres
                        checkFAIL = 1;
                end
                if abs(v(400) - v(401) - p2) > failthres
                        checkFAIL = 1;
```

```
                end
                echo tp4
        end
end
if time > checkPNTS[4]
        if time < checkPNTS[5]
# time = 285p, state should be '100'. if not set checkFAIL to 1
                if abs(v(200) - v(201) - p0) > failthres
                        checkFAIL = 1;
                end
                if abs(v(300) - v(301) - p1) > failthres
                        checkFAIL = 1;
                end
                if abs(v(400) - v(401) + p2 - 2*pi) > failthres
                        checkFAIL = 1;
                end
                echo tp5
        end
end
if time > checkPNTS[5]
        if time < checkPNTS[6]
# time = 335p, state should be '101'. if not set checkFAIL to 1
                if abs(v(200) - v(201) + p0 - 2*pi) > failthres
                        checkFAIL = 1;
                end
                if abs(v(300) - v(301) - p1) > failthres
                        checkFAIL = 1;
                end
                if abs(v(400) - v(401) + p2 - 2*pi) > failthres
                        checkFAIL = 1;
                end
                echo tp6
        end
end
if time > checkPNTS[6]
        if time < checkPNTS[7]
# time = 385p, state should be '110'. if not set checkFAIL to 1
                if abs(v(200) - v(201) - p0) > failthres
                        checkFAIL = 1;
                end
                if abs(v(300) - v(301) + p1 - 2*pi) > failthres
                        checkFAIL = 1;
                end
                if abs(v(400) - v(401) + p2 - 2*pi) > failthres
                        checkFAIL = 1;
                end
                echo tp7
        end
end
if time > checkPNTS[7]
        if time < checkPNTS[8]
```

```
# time = 435p, state should be '111'. if not set checkFAIL to 1
                if abs(v(200) - v(201) + p0 - 2*pi) > failthres
                        checkFAIL = 1;
                end
                if abs(v(300) - v(301) + p1 - 2*pi) > failthres
                        checkFAIL = 1;
                end
                if abs(v(400) - v(401) + p2 - 2*pi) > failthres
                        checkFAIL = 1;
                end
        end
end
#
# end of pass/fail logic
.endc
.tran 1p 500p uic
.subckt count 1 4 5 6 7
c1 4 0 3.2p
r1 3 8 .4
r2 4 9 1.1
b1 3 0 6 jj1
b2 5 0 7 jj1
l1 3 4 2.0p
l2 4 5 2.0p
l3 1 2 2.0p
l4 2 0 2.0p
l5 8 0 1.4p
l6 9 0 .1p
k1 l1 l3 .99
k2 l2 l4 .99
.ends count
r1 17 2 50
r2 1 6 50
r3 1 10 50
r4 1 14 50
r5 3 18 50
r6 7 13 50
r7 11 13 50
r8 15 13 50
r9 3 20 50
r10 4 5 .43
r11 8 9 .43
r12 12 19 .43
r13 16 30 .5
l1 5 6 2.1p
l2 9 10 2.1p
l3 19 14 2.1p
l4 30 0 2p
x1 3 2 4 100 101 count
x2 7 6 8 200 201 count
x3 11 10 12 300 301 count
```

```
x4 15 14 16 400 401 count
*
* These are the sources which vary
* In general, the $value1 or $value2 symbols can replace any numerical
* parameter in the circuit description.  No checking is done as to whether
* the substitution makes sense.
*
*flux bias
v1 13 0 pulse(0 $value1%m 10p 10p)
*gate bias
v2 1 0 pulse(0 $value2%m 10p 10p)
*
*
v3 20 0 pwl(0 0 70p 0
+ 75p  15m 90p  15m 100p -15m 115p -15m
+ 125p 15m 140p 15m 150p -15m 165p -15m
+ 175p 15m 190p 15m 200p -15m 215p -15m
+ 225p 15m 240p 15m 250p -15m 265p -15m
+ 275p 15m 290p 15m 300p -15m 315p -15m
+ 325p 15m 340p 15m 350p -15m 365p -15m
+ 375p 15m 390p 15m 400p -15m 415p -15m
+ 425p 15m 440p 15m 450p -15m 465p -15m 500p -15m)
*
* flux bias first stage
v4 18 0 pulse(0 13m 10p 10p)
*gate bias first stage
v5 17 0 pulse(0 39m 8p 10p)
*
*Nb 3000 A/cm2   area = 20 square microns
.model jj1 jj(rtype=1,cct=1,icon=10m,vg=2.8m,delv=0.08m,
+ icrit=0.6m,r0=49.999998,rn=2.745098,cap=0.777093p)
.end
```

5.4 Monte Carlo Analysis

WRspice has a built-in facility for performing Monte Carlo analysis, where one or more circuit variables are set according to a random distribution, and the circuit analyzed for functionality. The file formats and operation are very similar to operating range analysis.

As in operating range analysis, a complete input file consists of three sections: a header, an executable script analyzing operation, and the circuit deck. Unlike operating range analysis, however, the header block is executed before every simulation run, so that circuit variables may be changed (not just initialized) in the header. As in operating range analysis, an "old format" and a "new format" are recognized. These formats are identical in Monte Carlo analysis, except that instead of a line containing the string `.check`, Monte Carlo files contain the keyword `.monte`. This must be the first line of the file in the old format, but can appear anywhere in a new format file. If both keywords appear in the file (not a good idea), then Monte Carlo analysis is assumed.

As with conventional circuit and command files, Monte Carlo analysis files can be sourced by simply typing in the file name. If the file name happens to conflict with a command, then the file can be input

with the source command. If not in batch mode, the analysis is initiated with the check command, otherwise the analysis is performed immediately.

Monte Carlo analysis is enforced by supplying the `-m` option to the check command, which initiates analysis. The `-m` option is only necessary if the input file does not contain a `.monte` line. If the `-r` option is given, the simulations will be parceled out to remote servers, allowing parallelism in computation.

Output from a Monte Carlo run is saved in a file with base name that of the circuit, with a suffix "`.m`*xx*", where *xx* is a sequentially assigned number so as to make the file name unique. The output file name is stored in the `mplot_cur` shell variable.

The number of runs performed in Monte Carlo analysis is set by the `checkSTP1` and `checkSTP2` variables, as in operating range analysis. The number of points will be `(2*checkSTP1 + 1)*(2*checkSTP2 + 1)`. If the values are not given, they default to 3 (49 points).

In Monte Carlo analysis, the header block is executed before each simulation. In the header block, shell variables and vectors may be set for each new trial. These variables and vectors can be used in the SPICE text to modify circuit parameters. The names of the variables used, and whether to use vectors or variables, is up to the user (variables are a little more efficient). Monte Carlo analysis does not use predefined names for parameter data. Typically, the `gauss()` function is used to specify a random value for the variables in the header block.

There is a special echof command that allows text to be printed in the output file. This is the means by which the trial values are recorded, as there is no default recording mechanism. The file by default records only the success or failure of each run. The echof command is used exactly as the echo command. If there is no output file open, the command returns with no action. The echof command can be used in either `.control` or `.exec` blocks in the input file.

Monte Carlo results can be viewed during analysis ar afterward with the mplot command. Giving "`mplot -on`" will display results while simulating, as in operating range analysis. The display consists of `(2*checkSTP1 + 1) * (2*checkSTP2 + 1)` squares, as in operating range analysis, with each square indicating pass or fail. In Monte Carlo analysis, the squares are simply filled in in sequence, and their placement has nothing to do with the actual circuit values.

5.5 Example Monte Carlo Analysis Control File

The following is an example new format Monte Carlo input file:

```
3 stage counter
.exec
# Monte Carlo analysis of a Josephson binary counter
# This is an example of a Monte Carlo analysis input file
#
compose checkPNTS values 50p 135p 185p 235p 285p 335p 385p 435p 485p
#
set value1 = $&(13*gauss(.2,1))
set value2 = $&(38*gauss(.2,1))
# put the values in the output file
echof $value1 $value2
#
# one can define other initialized constants here as well
failthres = 1
```

```
#
# end of header
.endc
.control
#
# The following code is evaluated just after the time variable exceeds
# each one of the checkPNTS
#
echo $&time
if time > checkPNTS[0]
    if time < checkPNTS[1]
# time is 50p, set quiescent phase differences.  Uninitialized variables
# do not require declaration in header
        p0 = v(200) - v(201)
        p1 = v(300) - v(301)
        p2 = v(400) - v(401)
        checkFAIL = 0
        echo Test values: $value1 $value2
    else
        echo -n "  Checking at time $&time ... "
    end
end
if time > checkPNTS[1]
    if time < checkPNTS[2]
# time = 135p, state should be '001'. if not set checkFAIL to 1
# pi and the other variables in the 'constants' plot are known
        if abs(v(200) - v(201) + p0 - 2*pi) > failthres
            checkFAIL = 1;
        end
        if abs(v(300) - v(301) - p1) > failthres
            checkFAIL = 1;
        end
        if abs(v(400) - v(401) - p2) > failthres
            checkFAIL = 1;
        end
    end
end
if time > checkPNTS[2]
    if time < checkPNTS[3]
# time = 185p, state should be '010'. if not set checkFAIL to 1
        if abs(v(200) - v(201) - p0) > failthres
            checkFAIL = 1;
        end
        if abs(v(300) - v(301) + p1 - 2*pi) > failthres
            checkFAIL = 1;
        end
        if abs(v(400) - v(401) - p2) > failthres
            checkFAIL = 1;
        end
    end
end
```

```
if time > checkPNTS[3]
    if time < checkPNTS[4]
# time = 235p, state should be '011'. if not set checkFAIL to 1
        if abs(v(200) - v(201) + p0 - 2*pi) > failthres
            checkFAIL = 1;
        end
        if abs(v(300) - v(301) + p1 - 2*pi) > failthres
            checkFAIL = 1;
        end
        if abs(v(400) - v(401) - p2) > failthres
            checkFAIL = 1;
        end
    end
end
if time > checkPNTS[4]
    if time < checkPNTS[5]
# time = 285p, state should be '100'. if not set checkFAIL to 1
        if abs(v(200) - v(201) - p0) > failthres
            checkFAIL = 1;
        end
        if abs(v(300) - v(301) - p1) > failthres
            checkFAIL = 1;
        end
        if abs(v(400) - v(401) + p2 - 2*pi) > failthres
            checkFAIL = 1;
        end
    end
end
if time > checkPNTS[5]
    if time < checkPNTS[6]
# time = 335p, state should be '101'. if not set checkFAIL to 1
        if abs(v(200) - v(201) + p0 - 2*pi) > failthres
            checkFAIL = 1;
        end
        if abs(v(300) - v(301) - p1) > failthres
            checkFAIL = 1;
        end
        if abs(v(400) - v(401) + p2 - 2*pi) > failthres
            checkFAIL = 1;
        end
    end
end
if time > checkPNTS[6]
    if time < checkPNTS[7]
# time = 385p, state should be '110'. if not set checkFAIL to 1
        if abs(v(200) - v(201) - p0) > failthres
            checkFAIL = 1;
        end
        if abs(v(300) - v(301) + p1 - 2*pi) > failthres
            checkFAIL = 1;
        end
```

```
        if abs(v(400) - v(401) + p2 - 2*pi) > failthres
            checkFAIL = 1;
        end
    end
end
if time > checkPNTS[7]
    if time < checkPNTS[8]
# time = 435p, state should be '111'. if not set checkFAIL to 1
        if abs(v(200) - v(201) + p0 - 2*pi) > failthres
            checkFAIL = 1;
        end
        if abs(v(300) - v(301) + p1 - 2*pi) > failthres
            checkFAIL = 1;
        end
        if abs(v(400) - v(401) + p2 - 2*pi) > failthres
            checkFAIL = 1;
        end
end
if time > checkPNTS[1]
    if checkFAIL <> 0
        echo FAILED
    else
        echo OK
    end
end
#
# end of pass/fail logic
.endc
.tran 1p 500p uic
.subckt count 1 4 5 6 7
c1 4 0 3.2p
r1 3 8 .4
r2 4 9 1.1
b1 3 0 6 jj1
b2 5 0 7 jj1
l1 3 4 2.0p
l2 4 5 2.0p
l3 1 2 2.0p
l4 2 0 2.0p
l5 8 0 1.4p
l6 9 0 .1p
k1 l1 l3 .99
k2 l2 l4 .99
.ends count
r1 17 2 50
r2 1 6 50
r3 1 10 50
r4 1 14 50
r5 3 18 50
r6 7 13 50
r7 11 13 50
```

```
r8 15 13 50
r9 3 20 50
r10 4 5 .43
r11 8 9 .43
r12 12 19 .43
r13 16 30 .5
l1 5 6 2.1p
l2 9 10 2.1p
l3 19 14 2.1p
l4 30 0 2p
x1 3 2 4 100 101 count
x2 7 6 8 200 201 count
x3 11 10 12 300 301 count
x4 15 14 16 400 401 count
*
* These are the sources which vary
* In general, the $value1 or $value2 symbols can replace any
* numerical parameter in the circuit description.  No checking
* is done as to whether the substitution makes sense.
*
*flux bias
v1 13 0 pulse(0 $value1%m 10p 10p)
*gate bias
v2 1 0 pulse(0 $value2%m 10p 10p)
*
*
v3 20 0 pwl(0 0 70p 0
+ 75p  15m 90p  15m 100p -15m 115p -15m
+ 125p 15m 140p 15m 150p -15m 165p -15m
+ 175p 15m 190p 15m 200p -15m 215p -15m
+ 225p 15m 240p 15m 250p -15m 265p -15m
+ 275p 15m 290p 15m 300p -15m 315p -15m
+ 325p 15m 340p 15m 350p -15m 365p -15m
+ 375p 15m 390p 15m 400p -15m 415p -15m
+ 425p 15m 440p 15m 450p -15m 465p -15m 500p -15m)
*
* flux bias first stage
v4 18 0 pulse(0 13m 10p 10p)
*gate bias first stage
v5 17 0 pulse(0 39m 8p 10p)
*
*Nb 3000 A/cm2   area = 20 square microns
.model jj1 jj(rtype=1,cct=1,icon=10m,vg=2.8m,delv=0.08m,
+ icrit=0.6m,r0=49.999998,rn=2.745098,cap=0.777093p)
.end
```

5.6 Circuit Margin Optimization

There are three scripts which implement a margin optimization algorithm used by Clark Hamilton at NIST. These files (kept in the scripts directory) are `optimize`, `margins`, and `merit`. The main script is `optimize`, which is invoked with the name of the file to be optimized as an argument.

This facility is for advanced users. The present status of the scripts is unknown, and it is possible that they may require modification before use. They are provided as an example of how the *WRspice* scripting facility can be employed for optimization.

An example input file, which defines and initializes various variables and vectors as well as providing a circuit to optimize, is shown below. To perform optimization, one gives "`optimize` *filename*".

```
.check
set checkiterate = 3
let checkN1 = 0
compose checkPNTS values 1n 2n
let value[19] = 0
let flags[19] = 0
let flags[0] = 1
let value[0] = .8
.control
if (TIME >= checkPNTS[0])
&#32   checkFAIL = 0
&#32   if ((abs(v(1)) > 1.5) or (abs(v(1)) < .5))
&#32       checkFAIL = 1
&#32   endif
endif
.endc
optimization test
i1 0 1 pulse(0 1 0 1n)
r1 1 0 $&value[0]
.tran .01n 1.1n
.end
```

This is the simplest way to input the file, alternatively one could set the shell variables and vectors externally and/or use a bound codeblock for pass/fail evaluation.

The `margins` script, called by `optimize` calls the `check` command. The variable `checkiterate` must be set to a nonzero value up to 10. This is the binary search depth for finding the operating range.

The vectors `checkN1` and `value` must be defined, `checkN1` is the index into the value array of the variable being adjusted. It is altered by the scripts, but it and `value` must be defined before the script is input or in the header as shown.

The vector `checkPNTS` is the array of points where analysis is performed. Note that due to some strangeness, at least two entries must exist.

The `value` array is initialized to the starting values. The `flags` vector contains 1 for each entry in the array which is to be varied, the others are treated as constants.

The lengths of the vectors `value` and `flags` is 20, which is assumed in the optimization script.

After the analysis is complete, the `value` array will contain the optimized values. Two other arrays,

`lower` and `upper`, are created, and contain the lower and upper limit for each value index.

The scripts provided can be customized by the user for more specific applications, or used as templates for different types of analysis. It is recommended that such scripts be defined as codeblocks to speed execution.

Appendix A

File Formats

A.1 Rawfile Format

Rawfiles produced and read by *WRspice* have either an ascii or a binary format. Ascii format is the preferred format for general use, as it is hardware independent and easy to modify, though the binary format is the most economical in terms of space and speed of access.

The ascii format consists of lines or sets of lines introduced by a keyword. The `Title` and `Date` lines should be the first in the file and should occur only once. There may be any number of plots in the file, each one beginning with the `Plotname`, `Flags`, `No. Variables`, `No. Points`, `Variables`, and `Values` lines. The `Command` and `Option` lines are optional and may occur anywhere between the `Plotname` and `Values` lines. Note that after the `Variables` keyword there must be *numvars* "declarations" of outputs, and after the `Values` keyword, there must be *numpoints* lines, each consisting of *numvars* values. To clarify this discussion, one should create an ascii rawfile with *WRspice* and examine it.

Line Name	Description
`Title`	An arbitrary string describing the circuit
`Date`	A free-format date string
`Plotname`	A string describing the analysis type
`Flags`	Either "`complex`" or "`real`"
`No. Variables`	The number of variables (numvars)
`No. Points`	The number of points (numpoints)
`Command`	An arbitrary *WRspice* command
`Option`	SPICE variables
`Variables`	A number of variable lines (see below)
`Values`	A number of data lines (see below)

Any text on a `Command` line is executed when the file is loaded as if it were typed as a command. By default, *WRspice* puts a version command into every rawfile it creates.

Text on an `Option` line is parsed as if it were the arguments to a *WRspice* set command. The variables set are then available normally, except that they are read-only and are associated with the plot.

A `Variable` line looks like

number name typename `[` *parm=value* `]` `....`

The numbers are irrelevant, the *name* is the name by which this quantity will be referred to by *WRspice*, the *typename* may be either a pre-defined type or one defined with the deftype command, and the *parm*s are listed below.

Below are the predefined data types:

Name	Description	SPICE2 Numeric Code
`notype`	Dimensionless value	0
`time`	Time	1
`frequency`	Frequency	2
`voltage`	Voltage	3
`current`	Current	4
`output-noise`	SPICE2 .noise result	5
`input-noise`	SPICE2 .noise result	6
`HD2`	SPICE2 .disto result	7
`HD3`	SPICE2 .disto result	8
`DIM2`	SPICE2 .disto result	9
`SIM2`	SPICE2 .disto result	10
`DIM3`	SPICE2 .disto result	11
`pole`	SPICE3 pz result	12
`zero`	SPICE3 pz result	13

Below are the available parameters:

`min`	Minimum significant value for this output
`max`	Maximum significant value for this output
`color`	The name of a color to use for this value
`scale`	The name of another output to use as the scale
`grid`	The type of grid to use - numeric codes are:
`0`	Linear grid
`1`	Log-log grid
`2`	X-log/Y-linear grid
`3`	X-linear/Y-log grid
`4`	Polar grid
`5`	Smith grid
`plot`	The plotting style to use - numeric codes are:
`0`	Connected points
`1`	"Comb" style
`2`	Unconnected points
`dims`	The dimensions of this vector

If one of the flags is `complex`, the points look like *r,i* where *r* and *i* are floating point (in `%e` format). Otherwise they are real values in `%e` format. Only one of `real` and `complex` should appear.

The lines are guaranteed to be less than 80 columns wide (unless the plot title or variable names are very long, or a large number of variable options are given), so this format is safe to mail between systems that enforce 80 character lines.

The binary format is similar to the ascii format in organization, except that it is not text-mode. Strings are NULL terminated instead of newline terminated, and the values are in the machine's double precision floating point format instead of in ascii. This makes it much easier to read and write and reduces file size, but the binary format is not portable between machines with different floating point formats.

The circuit title, date, and analysis type name in that order are at the start of the plot, each terminated by a NULL byte. Then the flags field (a short, which is 1 for real data and 2 for complex data), the number of outputs, and the number of points (both integers) are present. Following this is a list of NULL-terminated strings which are command lines. This list is terminated by an extra NULL byte. Then come the options, which consist of the name, followed by the type and the value in binary. The output "declarations" consist of the name, type code, flags, color, grid type, plot type, and dimension information in that order. Next come the values, which are either doubles or pairs of doubles in the case of complex data.

The "old" binary format, which is used by SPICE2, is not accepted by *WRspice*, however the format is given below should it be necessary to write a translator.

SPICE2 Binary Rawfile Format	
Field	**Size in Bytes**
title	80
date	8
time	8
numoutputs	2
the integer 4	2
output names	8 for each output
types of output	2 for each output
node index	2 for each output
plot title	24
data	`numpoints * numoutputs * 8`

The data is in the form of double precision numbers, or pairs of single precision numbers if the data are complex.

The values recognized for the "types of output" fields are described in the Predefined Values table above.

A.2 Help Database Files

The help information is obtained from database files suffixed with `.hlp` found along the help search path. These directories may also contain other files referenced in the help text, such as image files. The help search path can be set in the environment with the variable SPICE_HLP_DIR, and/or may be set with the `helppath` variable, which will override the environment. These files have a simple format, allowing users to create and modify them. Each help entry is associated with one or more keywords, which should be unique in the database. The help system has a debugging mode, which can usually be switched on by the application, which will issue a warning message on `stderr` if a name clash is detected. The files are ascii text, either in DOS or Unix format. Fields are separated by keywords which begin with "`!!`". Although the help system provides rich-text presentation from HTML formatting, entries can be in plain text. A sample plain-text entry has the form:

```
!!KEYWORD
excmd
!!TITLE
Example Command
!!TEXT
    This command exists only in this example.  Note that the
    !!keywords only have effect if they start in the first
    column.  The blank line below is optional.

!!SUBTOPICS
akeyword
anotherkeyword
!!SEEALSO
yetanotherkeyword
```

In this example, the keyword "`excmd`" is used to access the topic, and should be unique among the database entries accessed by the application. The text which appears in the topic (following `!!TEXT`) is shown indented, which is recommended for clarity, but is not required.

In "`.hlp`" files, lines anywhere with '`*`' or '`#`' in the first column are ignored, as they are assumed to be comments. Blank lines outside of the `!!TEXT` field are ignored. Leading white space is stripped, which can be a problem for maintaining indentation in formatted plain text. To add a space which will not be stripped, use the HTML escape "` `".

The following '`!!`' keywords can appear in "`.hlp`" files. These are recognized only in upper case, and must start in the first text column.

`!!`(space) *anything*
: A line beginning with two exclamation points followed by a space character is ignored.

`!!KEYWORD` *keyword-list*
: This keyword signals the start of a new topic. The *keyword-list* consists of one or more tokens, each of which must be unique among all topics in the database. The words are used to identify the topic, and if more than one is listed, the additional words are equivalent aliases. The *keyword-list* may follow `!!KEYWORD` on the same line, or may be listed in the following line, in which case `!!KEYWORD` should appear alone on the line.

  Punctuation is allowed in keywords, only white space characters can not be used. The '`#`' character has special meaning and should not be part of a keyword name. Also, character sequences that could be confused with a URL or directory path should be avoided. The latter basically prohibits the '`/`' character (and also '\' under Windows) from being included in keywords. There are special names starting with '$' which are expanded to application-specific internal variables, as described below. To avoid any possibility of a clash, it is probably best to avoid '$' in general keywords.

  It is often useful to include a meaningful prefix in keywords to ensure uniqueness, for example in *Xic*, all commands have keywords prefixed with "`xic:`".

`!!TITLE` *string*
: The `!!TITLE` specifies the title of the topic, and should follow the `!!KEYWORD` specification. The title text can appear on the same line following `!!TITLE`, or on the next line, in which case `!!TITLE` should appear alone in the line. The title is printed at the top of the topic display, and is used in menus of topics.

`!!TEXT`
: This line signals the beginning of the topic text, which is expected to be plain text. The keyword is

mutually exclusive with the !!HTML keyword. The lines following !!TEXT up to the next !!KEYWORD, !!SEEALSO, or !!SUBTOPICS line or end of file are read into the display window. The plain text is converted to HTML before being sent to the display in the following manner:

1. The title text is enclosed in <H1>...</H1>.
2. Each line of text has a <BR> appended.
3. The subtopics and see-alsos are preceded with added <H3>Subtopics</H3> and <H3>References</H3> lines.
4. The subtopics and see-alsos are converted to links of the form <A HREF="*keyword*">*title*</A> where the *keyword* is the database keyword, and the *title* is the title text for the entry.

Note that the text area can contain HTML tags for various things, such as images. Also note that text formatting is taken from the help file (the <BR> breaks lines), and not reformatted at display time. The !!HTML line should be used rather than !!TEXT if the text requires full HTML formatting.

!!HTML

This line signals the beginning of the topic text, which is expected to be HTML-formatted. The keyword is mutually exclusive with the !!TEXT keyword. The parser understands all of the standard HTML 3.2 syntax, and a few 4.0 extensions. References are to keywords found in the database and general URLs. Image (.gif, etc.) files can be referenced, and are expected to be found along with the .hlp files.

!!IFDEF *word*

This line can appear in the block of text following !!TEXT or !!HTML. In conjunction with the !!ELSE and !!ENDIF directives, it allows for the conditional inclusion of blocks of text in the topic. The *word* is one of the special words defined by the application. Presently, the following words are defined:

in *Xic*	Xic
in *WRspice*	WRspice
in either, under Windows	Windows

If *word* is defined, the text up to the next !!ELSE or !!ENDIF will be included in the topic, and any text following an !!ELSE up to !!ENDIF is discarded. If *word* is not defined, the text up to the next !!ELSE or !!ENDIF is discarded, and any text following an !!ELSE is included. The constructs can be nested. A word that is not recognized or absent is "not defined". Every !!IFDEF should have a corresponding !!ENDIF. The !!ELSE is optional. The !!SEEALSO and !!SUBTOPICS lines can appear within the blocks.

Example:

```
!!HTML
   Here is some text.
!!IFDEF Xic
   You are reading this in Xic.
!!ELSE
!!IFDEF WRspice
   You are reading this in WRspice.
!!ELSE
   You are not reading this in Xic or WRspice.
!!ENDIF
!!ENDIF
```

`!!IFNDEF` *word*
: This keyword can appear in the block of text following `!!TEXT` or `!!HTML`. It is similar to `!!IFDEF` but has the reverse logic.

`!!ELSE`
: This keyword can follow `!!IFDEF` or `!!IFNDEF` and defines the start of a block of text to include in the topic if the condition is not satisfied.

`!!ENDIF`
: This keyword terminates the text blocks to be conditionally included in the topic, using `!!IFDEF` or `!!IFNDEF`.

`!!INCLUDE` *filename*
: The keyword may appear in the text following `!!TEXT` or `!!HTML`. When encountered in the text to be included in the topic, the text of *filename*, which is searched for in the help search path if not an absolute pathname, is added to the displayed text of the current topic. There is no modification of the text from *filename*.

  If the filename is a relative path to a subdirectory of one of the directories of a directory in the help search path, the subdirectory is added to the search list. Thus, an HTML document and associated gif files can be placed in a separate subdirectory in the help tree. The HTML document can be referenced from the main help files with a `!!INCLUDE` directive, and there is no need to explicitly change the help search path.

`!!REDIRECT` *keyword target*
: This will define *keyword* as an alias for *target*. The *target* can be any input token recognizable by the help system, including URLs, named anchors, and local files. For example:

```
!!REDIRECT nyt http://www.nytimes.com
```

  Giving "`!help nyt`" in *Xic* or "`help nyt`" in *WRspice* will bring up a help window containing the New York Times web page.

`!!HEADER`
: The text that follows, up until the next `!!KEYWORD` or `!!FOOTER`, is saved for inclusion in each page composed from the `!!HTML` lines for database keywords. The header is inserted at the top of the page. There can be only one header defined, and if more than one are found in the help files, the first one read will be used.

  In the header text, the literal token `%TITLE%` is replaced with the `!!TITLE` text of the current topic when displayed.

`!!FOOTER`
: The text that follows, up until the next `!!KEYWORD` or `!!HEADER`, is saved for inclusion in each page composed from the `!!HTML` lines for database keywords. The footer is inserted at the bottom of the page. There can be only one footer defined, and if more than one are found in the help files, the first one read will be used.

`!!SEEALSO keyword-list`
: The *keyword-list* consists of a list of keywords that are expected to be defined by `!!KEYWORD` lines elsewhere in the database. A menu of these items is displayed at the bottom of the topic text, under the heading "References". The keywords specified after `!!SEEALSO` can appear on the same line separated with space, or on multiple lines. If a keyword in these lists is not found in the database, the normal action is to ignore the error. The application may provide a debugging mode, whereby unresolved references will produce a warning message.

`!!SUBTOPICS` *keyword-list*
: This produces a menu of the topics found in the *keyword-list* very similar to `!!SEEALSO`, however under the heading "Subtopics". This can be used in addition to `!!SEEALSO`.

A.2.1 Anchor Text

Clickable references in the HTML text have the usual form:

`<a href="`*something*`">`*highlighted text*`</a>`

Here, "*something*" can be a help database keyword or an ordinary URL.

One can use named anchors in help keywords. This means that the '`#`' symbol is holy, and should not be used in help keywords. The named anchors can appear in the `!!HTML` part of the help database entries in the usual HTML way, e.g.

```
!!KEYWORD
somekeyword
...
!!HTML
    ...
    <a name="refname">some text</a>
```

Then, referencing forms like "`!help somekeyword#refname`" and `<a href="somekeyword#refname">blather</a>` will bring up the "somekeyword" topic, but with "some text" at the top of the help window, rather than the start of the document.

There is an additional capability: '$' expansion. If the first character of an anchor URL is '$', it is processed specially. The leading word is replaced with other text, either an internal path, or an environment variable. The internally recognized tokens are:

`$HELP`	replaced by first component of the help path
`$EXAMPLES`	replaced by path to examples
`$DOCS`	replaced by path to docs
`$SCRIPTS`	replaced by path to scripts

Otherwise, if it matches a variable in the environment, it is replaced by the environment variable value. If no match, it is left alone.

The paths are derived from the first component of the Help path. If the HelpPath variable is "`/usr/local/lib/xictools/xic/help /blather/help`", then the substitution paths are `/usr/local/lib/xictools/xic/examples`", etc.

If the first character of an anchor URL is '~', the path is tilde expanded. This is done after '$' substitution. Tildes denote a user's home directory: "`~/mydir`" might expand to "`/users/you/mydir`", and "`~joe/joesdir`" might expand to "`/users/joe/joesdir`", etc.

In *WRspice*, one can source files from anchor text in the HTML viewer, if the anchor text consists of a file name with a "`.cir`" extension. Thus, if one has a circuit file named "`mycircuit.cir`", and the HTML text in the help window contains a reference like

```
<a html="mycircuit.cir">click here</a>
```

then clicking on the "click here" tag will source `mycircuit.cir` into *WRspice*. Similarly, anchor references to files with a "`.raw`" extension will be loaded into *WRspice* (as a *rawfile*, i.e., a plot data file) when the anchor is clicked.

A.2.2 `.mozyrc` File

The help system looks for a file named "`.mozyrc`" in the user's home directory, which contains keywords which define the default behavior of many of the commands and features of the help window. This is used only in UNIX/Linux releases. It is necessary to install this file if one wants alternate selections from the help window, for example different fonts, to be persistent.

A sample `.mozyrc` file listing is provided below. The file can be found in the `startup` directory in the installation tree, under the name "`mozyrc`". To install, edit the file if necessary, then move it to your home directory under the name "`.mozyrc`".

```
# This is the startup file which sets defaults for the mozy web browser
# and the Xic/WRspice HTML viewer.  It should be installed as ".mozyrc"
# in the user's home directory, should the user wish to change the
# defaults.

# --- DISPLAY ATTRIBUTES -------------------------------------------------

# Background color used for pages that don't have a <body> tag,
# such as help text (default #e8e8f0)
DefaultBgColor #e8e8f0


# Background image URL to use for pages that don't have a <body> tag
# (no default)
#DefaultBgImage /some/dir/pretty_picture.jpg

# Text color to use for pages that don't have a <body> tag
# (default black)
DefaultFgText black

# Color to use for links in pages without a <body> tag
# (default blue)
DefaultFgLink blue

# How to handle images:
#  0 Don't display images that require downloading
#  1 Download images when encountered in document
#  2 Download images after document is displayed
#  3 Display images progressively after document is displayed (the default)
ImageLoadMode 3

# How to underline anchors when underlining is enabled
#  0 No underline
```

```
#  1 Single solid underline (default)
#  2 Double solid underline
#  3 Single dashed underline
#  4 Double dashed underline
AnchorUnderline 1

# If this is set to one (the default) anchors are shown as buttons.  If set
# to zero, anchors use the underlining style
AnchorButtons 0

# If set to one (the default) anchors will be highlighted when the pointer
# passes over them.  If zero, there will be no highlighting
AnchorHighlight 1

# The default font families.  This is the XLFD family name with "-size"
# appended.  Defaults:  adobe-times-normal-p-14   misc-fixed-normal-c-14
FontFamily adobe-times-normal-p-14
FixedFontFamily misc-fixed-normal-*-14

# If set to one, animations are frozen.  If zero (the default) animations
# will be shown normally
FreezeAnimations 0

# --- COMMUNICATIONS -----------------------------------------------------

# Time in seconds allowed for a response from a message (0 for no timeout,
# to 600, default 15)
Timeout 15

# Number of times to retry a message after a timeout (0 to 10, default 4)
Retries 4

# The port number used for HTTP communications (1 to 65536, default 80)
HTTP_Port 80

# The port number used for FTP communications (1 to 65536, default 21)
FTP_Port 21

# --- GENERAL ------------------------------------------------------------

# Number of cache files to save (2 to 4096, default 64)
CacheSize 64

# Set to one to disable disk cache, 0 (the default) enables cache
NoCache 0

# Set to one to disable sending and receiving cookies
NoCookies 0

# --- DEBUGGING ----------------------------------------------------------
```

```
# Set this to one to print extended status messages on terminal screen
# (default 0)
DebugMode 0

# Set this to one to print transaction headers to terminal screen
# (default 0)
PrintTransact 0

# Debugging mode for images
#  0 Disable debugging mode (the default)
#  1 Load local images after document is displayed
#  2 Display local images progressively after document is displayed
LocalImageTestMode 0

# Issue warnings about bad HTML syntax to terminal (1) or not (0, the default)
BadHTMLwarnings 0
```

A.3 Example Data Files

A.3.1 Circuit 1: Simple Differential Pair

The following file determines the dc operating point of a simple differential pair. In addition, the ac small-signal response is computed over the frequency range 1Hz to 100MHz.

```
Simple differential pair.
vcc 7 0 12
vee 8 0 -12
vin 1 0 ac 1
rs1 1 2 1k
rs2 6 0 1k
q1 3 2 4 mod1
q2 5 6 4 mod1
rc1 7 3 10k
rc2 7 5 10k
re 4 8 10k
model mod1 npn bf=50 vaf=50 is=1.E-12 rb=100 cjc=.5pf tf=.6ns
.ac dec 10 1 100meg
.end
```

A.3.2 Circuit 2: MOS Output Characteristics

The following file computes the output characteristics of a MOSFET device over the range 0-10V for VDS and 0-5V for VGS.

```
MOS output characteristics
.options node nopage
vds 3 0
vgs 2 0
m1 1 2 0 0 mod1 l=4u w=6u ad=10p as=10p
.model mod1 nmos vto=-2 nsub=1.0e15 uo=550
* vids measures Id, we could have used Vds, but Id would be negative
vids 3 1
.dc vds 0 10 .5 vgs 0 5 1
.end
```

A.3.3 Circuit 3: Simple RTL Inverter

The following file determines the dc transfer curve and the transient pulse response of a simple RTL inverter. RTL was an early logic family which died out in the early 1970's. We could not think of anything more archaic, as *WRspice* does not contain a vacuum tube model. The input is a pulse from 0 to 5 Volts with delay, rise, and fall times of 2ns and a pulse width of 30ns. The transient interval is 0 to 100ns, with printing to be done every nanosecond.

```
Simple Resistor-Transistor Logic (RTL) inverter
vcc 4 0 5
```

```
vin 1 0 pulse 0 5 2ns 2ns 2ns 30ns
rb 1 2 10k
q1 3 2 0 q1
rc 3 4 1k
.model q1 npn bf 20 rb 100 tf .1ns cjc 2pf
.dc vin 0 5 0.1
.tran 1ns 100ns
.end
```

A.3.4 Circuit 4: Four-Bit Adder

The following file simulates a four-bit binary adder, using several subcircuits (now were getting really advanced) to describe various pieces of the overall circuit. Upper case is used throughout, just to prove that it works.

```
ADDER - 4 BIT ALL-NAND-GATE BINARY ADDER
*
*** SUBCIRCUIT DEFINITIONS
.SUBCKT NAND 1 2 3 4
*NODES: INPUT(2), OUTPUT, VCC
Q1 9 5 1 QMOD
D1CLAMP 0 1 DMOD
Q2 9 5 2 QMOD
D2CLAMP 0 2 DMOD
RB 4 5 4K
R1 4 6 1.6K
Q3 6 9 8 QMOD
R2 8 0 1K
RC 4 7 130
Q4 7 6 10 QMOD
DVBEDROP 10 3 DMOD
Q5 3 8 0 QMOD
.ENDS NAND
.SUBCKT ONEBIT 1 2 3 4 5 6
*NODES: INPUT(2), CARRY-IN, OUTPUT, CARRY-OUT, VCC
X1 1 2 7 6 NAND
X2 1 7 8 6 NAND
X3 2 7 9 6 NAND
X4 8 9 10 6 NAND
X5 3 10 11 6 NAND
X6 3 11 12 6 NAND
X7 10 11 13 6 NAND
X8 12 13 4 6 NAND
X9 11 7 5 6 NAND
.ENDS ONEBIT
.SUBCKT TWOBIT 1 2 3 4 5 6 7 8 9
*NODES: INPUT - BIT0(2) / BIT1(2), OUTPUT - BIT0 / BIT1,
*       CARRY-IN, CARRY-OUT, VCC
X1 1 2 7 5 10 9 ONEBIT
X2 3 4 10 6 8 9 ONEBIT
```

```
.ENDS TWOBIT
*
.SUBCKT FOURBIT 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
*NODES: INPUT - BIT0(2) / BIT1(2) / BIT2(2) / BIT3(2),
*       OUTPUT - BIT0 / BIT1 / BIT2 / BIT3, CARRY-IN, CARRY-OUT, VCC
X1 1 2 3 4 9 10 13 16 15 TWOBIT
X2 5 6 7 8 11 12 16 14 15 TWOBIT
.ENDS FOURBIT
*
*** DEFINE NOMINAL CIRCUIT
*
.MODEL DMOD D
.MODEL QMOD NPN(BF=75 RB=100 CJE=1PF CJC=3PF)
VCC 99 0 DC 5V
VIN1A 1 0 PULSE(0 3 0 10NS 10NS   10NS   50NS)
VIN1B 2 0 PULSE(0 3 0 10NS 10NS   20NS  100NS)
VIN2A 3 0 PULSE(0 3 0 10NS 10NS   40NS  200NS)
VIN2B 4 0 PULSE(0 3 0 10NS 10NS   80NS  400NS)
VIN3A 5 0 PULSE(0 3 0 10NS 10NS  160NS  800NS)
VIN3B 6 0 PULSE(0 3 0 10NS 10NS  320NS 1600NS)
VIN4A 7 0 PULSE(0 3 0 10NS 10NS  640NS 3200NS)
VIN4B 8 0 PULSE(0 3 0 10NS 10NS 1280NS 6400NS)
X1 1 2 3 4 5 6 7 8 9 10 11 12 0 13 99 FOURBIT
RBIT0 9 0 1K
RBIT1 10 0 1K
RBIT2 11 0 1K
RBIT3 12 0 1K
RCOUT 13 0 1K
*
*** (FOR THOSE WITH MONEY (AND MEMORY) TO BURN)
* Ah, the good olde days...
.TRAN 1NS 6400NS
.END
```

A.3.5 Circuit 5: Transmission Line Inverter

The following file simulates a transmission-line inverter. Two transmission-line elements are required since two propagation modes are excited. In the case of a coaxial line, the first line (t1) models the inner conductor with respect to the shield, and the second line (t2) models the shield with respect to the outside world.

```
transmission-line inverter
v1 1 0 pulse(0 1 0 0.1N)
r1 1 2 50
x1 2 0 0 4 tline
r2 4 0 50
.subckt tline 1 2 3 4
t1 1 2 3 4 z0=50 td=1.5ns
t2 2 0 4 0 z0=100 td=1ns
.ends tline
```

```
.tran 0.1ns 20ns
.end
```

A.3.6 Circuit 6: Function and Table Demo

Below is a file which illustrates some features exclusive to *WRspice* for specifying the output of sources.

```
WRspice function and table demo
*
* WRspice allows arbitrary functional dependence in sources.  This
* file demonstrates some of the capability.
*
* v1 is numerically equal to the exponentiation of
* 2 times the sine.  "x" is replaced by the time variable.
v1 1 0 exp(2*sin(6.28e9*x))
r1 1 0 1
*
* v2 obtains values from table t1
v2 2 0 table(t1, time)
r2 2 0 1
.table t1 0 0 100p .1 500p 0 750p .2 1000p 0
*
* v3 is a 0-1 ramp
v3 3 0 pwl(0 0 1n 1)
r3 3 0 1
*
* e1 illustrates use of sub-tables.  x is the voltage from v3
e1 4 0 3 0 table(t2, x)
* below is an alternative equivalent form for e1
*e1 4 0 t2(v(3))
r4 4 0 1
.table t2 0 table t3 .5 table t4 .75 .75 1 0
.table t3 0 0 .25 1 .5 0
.table t4 0 0 .5 0 .625 1 .75 .75, x)
*
* e2 produces the same output as e1, but uses a PWL statement.
* when the controlling nodes are given, pwl uses the control source,
* and not time when used in e,f,g,h sources
e2 5 0 3 0 pwl(0 0 .25 1 .5 0 .625 1 .75 .75 1 0)
r5 5 0 1
*
.tran 1p 1n
* type "run", then "plot v(1) v(2) v(3) v(4)"
.end
```

A.3.7 Circuit 7: MOS Convergence Test

Below is an example circuit that illustrates some of the new features, and some older features perhaps not widely appreciated. This also served as a test for improving MOS convergence. To run a convergence

test:

1. Source this circuit.
2. Press the **siminterface** button in the Debug tool.
3. Type "`set value1 = mult`".
4. Type "`loop .5 2.5 .1 op`".

This runs operating point analysis for M values from .5 to 2.5, and displays a little plot of the convergence process. In the display, '+' means an increasing gmin step, '.' is a source step, '-' is a decreasing gmin step.

Note that 100 nS is too coarse to get a decent looking plot with BSIM devices. Use "`tran 1n 10u`" in that case, since the circuit seems to be oscillating at very high frequency.

```
mosamp2 - mos amplifier - transient
.options abstol=10n vntol=10n noopiter mult=2 nqs=1 reltol=1e-4
*.op
.tran 0.1us 10us
.plot  tran v(20) v(66)
* set below to 0 for old MOS model
.param bsim = 1
.if bsim = 1
m1  15 15  1 32 n1 w=88.9u  l=25.4u  m=$mult nqsmod=$nqs
m2   1  1  2 32 n1 w=12.7u  l=266.7u m=$mult nqsmod=$nqs
m3   2  2 30 32 n1 w=88.9u  l=25.4u  m=$mult nqsmod=$nqs
m4  15  5  4 32 n1 w=12.7u  l=106.7u m=$mult nqsmod=$nqs
m5   4  4 30 32 n1 w=88.9u  l=12.7u  m=$mult nqsmod=$nqs
m6  15 15  5 32 n1 w=44.5u  l=25.4u  m=$mult nqsmod=$nqs
m7   5 20  8 32 n1 w=482.6u l=12.7u  m=$mult nqsmod=$nqs
m8   8  2 30 32 n1 w=88.9u  l=25.4u  m=$mult nqsmod=$nqs
m9  15 15  6 32 n1 w=44.5u  l=25.4u  m=$mult nqsmod=$nqs
m10  6 21  8 32 n1 w=482.6u l=12.7u  m=$mult nqsmod=$nqs
m11 15  6  7 32 n1 w=12.7u  l=106.7u m=$mult nqsmod=$nqs
m12  7  4 30 32 n1 w=88.9u  l=12.7u  m=$mult nqsmod=$nqs
m13 15 10  9 32 n1 w=139.7u l=12.7u  m=$mult nqsmod=$nqs
m14  9 11 30 32 n1 w=139.7u l=12.7u  m=$mult nqsmod=$nqs
m15 15 15 12 32 n1 w=12.7u  l=207.8u m=$mult nqsmod=$nqs
m16 12 12 11 32 n1 w=54.1u  l=12.7u  m=$mult nqsmod=$nqs
m17 11 11 30 32 n1 w=54.1u  l=12.7u  m=$mult nqsmod=$nqs
m18 15 15 10 32 n1 w=12.7u  l=45.2u  m=$mult nqsmod=$nqs
m19 10 12 13 32 n1 w=270.5u l=12.7u  m=$mult nqsmod=$nqs
m20 13  7 30 32 n1 w=270.5u l=12.7u  m=$mult nqsmod=$nqs
m21 15 10 14 32 n1 w=254u   l=12.7u  m=$mult nqsmod=$nqs
m22 14 11 30 32 n1 w=241.3u l=12.7u  m=$mult nqsmod=$nqs
m23 15 20 16 32 n1 w=19u    l=38.1u  m=$mult nqsmod=$nqs
m24 16 14 30 32 n1 w=406.4u l=12.7u  m=$mult nqsmod=$nqs
m25 15 15 20 32 n1 w=38.1u  l=42.7u  m=$mult nqsmod=$nqs
m26 20 16 30 32 n1 w=381u   l=25.4u  m=$mult nqsmod=$nqs
m27 20 15 66 32 n1 w=22.9u  l=7.6u   m=$mult nqsmod=$nqs
```

```
.else
m1  15 15  1 32 n1 w=88.9u  l=25.4u  m=$mult
m2   1  1  2 32 n1 w=12.7u  l=266.7u m=$mult
m3   2  2 30 32 n1 w=88.9u  l=25.4u  m=$mult
m4  15  5  4 32 n1 w=12.7u  l=106.7u m=$mult
m5   4  4 30 32 n1 w=88.9u  l=12.7u  m=$mult
m6  15 15  5 32 n1 w=44.5u  l=25.4u  m=$mult
m7   5 20  8 32 n1 w=482.6u l=12.7u  m=$mult
m8   8  2 30 32 n1 w=88.9u  l=25.4u  m=$mult
m9  15 15  6 32 n1 w=44.5u  l=25.4u  m=$mult
m10  6 21  8 32 n1 w=482.6u l=12.7u  m=$mult
m11 15  6  7 32 n1 w=12.7u  l=106.7u m=$mult
m12  7  4 30 32 n1 w=88.9u  l=12.7u  m=$mult
m13 15 10  9 32 n1 w=139.7u l=12.7u  m=$mult
m14  9 11 30 32 n1 w=139.7u l=12.7u  m=$mult
m15 15 15 12 32 n1 w=12.7u  l=207.8u m=$mult
m16 12 12 11 32 n1 w=54.1u  l=12.7u  m=$mult
m17 11 11 30 32 n1 w=54.1u  l=12.7u  m=$mult
m18 15 15 10 32 n1 w=12.7u  l=45.2u  m=$mult
m19 10 12 13 32 n1 w=270.5u l=12.7u  m=$mult
m20 13  7 30 32 n1 w=270.5u l=12.7u  m=$mult
m21 15 10 14 32 n1 w=254u   l=12.7u  m=$mult
m22 14 11 30 32 n1 w=241.3u l=12.7u  m=$mult
m23 15 20 16 32 n1 w=19u    l=38.1u  m=$mult
m24 16 14 30 32 n1 w=406.4u l=12.7u  m=$mult
m25 15 15 20 32 n1 w=38.1u  l=42.7u  m=$mult
m26 20 16 30 32 n1 w=381u   l=25.4u  m=$mult
m27 20 15 66 32 n1 w=22.9u  l=7.6u   m=$mult
.endif
cc 7 9 40pf
cl 66 0 70pf
vin 21 0 AC pulse(0 5 1ns 1ns 1ns 5us 10us)
vccp 15 0 dc +15
vddn 30 0 dc -15
vb 32 0 dc -20
.if bsim
.model n1 nmos(level=8 capmod=3)
.else
.model n1 nmos(nsub=2.2e15 uo=575 ucrit=49k uexp=0.1 tox=0.11u xj=2.95u
+   level=2 cgso=1.5n cgdo=1.5n cbd=4.5f cbs=4.5f ld=2.4485u nss=3.2e10
+   kp=2e-5 phi=0.6 )
.endif
```

A.3.8 Circuit 8: Verilog Pseudo-Random Sequence

This example illustrates use of a `.verilog` block to generate a digital signal, that is then interfaced to and processed by a conventional SPICE circuit. The digital signal is a 511 step pseudo-random sequence, which is converted to a voltage and filtered.

```
* WRspice pseudo-random bit sequence demo
```

```
v1 1 0 a/255-1
r1  1 2 100
c1 2 0 10p
.tran 1p 10n
.plot tran v(1) v(2)

.verilog
module  prbs;
reg [8:0] a, b;
reg clk;
integer cnt;

initial
    begin
    a = 9'hff;
    clk = 0;
    cnt = 0;
    $monitor("%d", cnt, "%b", a, a[0]);
    end

always
    #5 clk = ~clk;

always
    @(posedge clk)
    begin
    a = { a[4]^a[0], a[8:1] };
    if (a ==  9'hff)
        $stop;
    cnt = cnt + 1;
    end

endmodule
.endv
```

A.3.9 Circuit 9: Josephson Junction I-V Curve

```
WRspice jj I-V curve demo
*
* One can plot a pretty decent iv curve using transient analysis.
* This will show the differences between the various model options.
*
b1 1 0 jj1 control=v2
v1 2 0 pwl(0 0 2n 70m 4n 0 5n 0)
r1 2 1 100
*
* for rtype=4, vary v2 between 0 and 1 for no gap to full gap
v2 3 0 .5
*
r2 3 0 1
```

```
*
* It is interesting to set rtype and delv to different values, and note
* the changes.
*
*Nb 1000 A/cm2   area = 30 square microns
.model jj1 jj(rtype=4,cct=1,icon=10m,vg=2.8m,delv=.1m,
+ icrit=0.3m,r0=100,rn=5.4902,cap=1.14195p)
.tran 5p 5n
* type "run", then "plot -b v(1) (-v1#branch)"
.end
```

A.3.10 Circuit 10: Josephson Gap Potential Modulation

```
WRspice jj qp modulation demo
*
* The rtype=4 option of the Josephson model causes the gap potential
* to scale with the external "control current" absolute value.  For
* unit control current (1 Amp) or larger, the full potential is used,
* otherwise it scales linearly to zero.  The transfer function is defined
* externally with controlled sources, as below.  The approximation
* Vg = Vg0*(1-t**4) is pretty good, except near t = 1 (T = Tc, t = T/Tc).
* The actual transfer function is left to the user - in the example below,
* the ambient temperature is 7K, Tc=9.2K, and 1mv of "input" causes 1K
* temperature shift.
*
* For amusement, change cct=1 to cct=0 below.  This runs much more quickly
* as critical current is set to zero.
*
b1 1 0 jj1 control=v2
v1 2 0 pulse(0 35m 10p 10p)
r1 2 1 100
*
v2 3 0
g1 3 0 4 0 function 1 - (1000*x+7)/9.2)^4
v4 4 0 pulse(-1m 1m 10p 10p 10p 20p 60p)
*
*Nb 1000 A/cm2   area = 30 square microns
.model jj1 jj(rtype=4,cct=1,icon=10m,vg=2.8m,delv=.1m,
+ icrit=0.3m,r0=100,rn=5.4902,cap=1.14195p)
.tran 1p 500p uic
* type "run", then plot v(1) and v(4) to see the gap shift and input
.end
```

Appendix B

Utility Programs

The *WRspice* distribution provides a few supplemental utility and accessory programs.

B.1 The `multidec` Utility: Coupled Lossy Transmission Lines

The standalone program `multidec` produces a subcircuit for multiconductor lossy transmission lines in terms of uncoupled (single) simple lossy lines. This decomposition is valid only if the following hold:

1. The electrical parameters (R, G, Cs, Cm, Ls, Lm) of all wires are identical and constant (i.e., independent of frequency).
2. Each line is coupled only to its (maximum 2) nearest neighbors.

The subcircuit is sent to the standard output and is intended to be included in an input file.

The command-line options for `multidec` are as follows:

`-l`<self-inductance Ls>
`-c`<self-capacitance Cs>
`-r`<series-resistance R>
`-g`<parallel-conductance G>
`-k`<coeff-of-inductive-coupling K>
`-x`<mutual-capacitance Cm>
`-o`<subckt-name>
`-n`<number-of-conductors>
`-L`<length>

The inductive coupling coefficient K is the ratio of Lm to Ls. Values for -l, -c, -o, -n and -L must be specified.

Example:

```
multidec -n4 -l9e-9 -c20e-12 -r5.3 -x5e-12 -k0.7 -otest -L5.4
```

This utility was written by J.S. Roychowdhury for use with the lossy transmission line model [13].

B.2 The printtoraw Utility: Print to Rawfile Conversion

The printtoraw program is a stand-alone utility provided with the *WRspice* distribution. This converts the data in files produced by the print command using output redirection into the rawfile format, which can be plotted. This works only for print files in the standard columnar form.

Usage: printtoraw [*printfile*]

The argument, if given, is assumed to be a path to a file that was produced by the *WRspice* print command through redirection. If no argument is given, the standard input is read. The data are converted to rawfile format and dumped to the standard output.

Example:

```
wrspice> run
wrspice> print v(1) v(2) v(3) > myfile
wrspice> quit

bash> printtoraw myfile > myfile.raw

wrspice> load myfile.raw
wrspice> plot all
```

B.3 The proc2mod Utility: BSIM1 Model Generation

This utility, provided with SPICE3, produces a set of BSIM1 models from process-dependent data provided in a "process" file. An example process (.pro) file is provided with the *WRspice* examples. This utility was written by J. Pierret [3], and the reference presumably provides more information.

B.4 The wrspiced Daemon: Remote SPICE Controller

WRspice can be accessed and run from a remote system for asynchronous simulation runs, for assistance in computationally intensive tasks such as Monte Carlo analysis, and as a simulator for the *Xic* graphical editor. This is made possible through a daemon (background) process which controls *WRspice* on the remote machine. The daemon has the executable name "wrspiced", and should be run as a root process on the remote machine. Typically, this can be initiated in the system startup script, or manually. Of course, the remote machine must have a valid *WRspice* executable present.

This section describes how to configure the installation to support remote *WRspice* runs. In this configuration, *WRspice* is installed on a network-reachable remote computer (the "SPICE server"), and we wish to submit jobs to this *WRspice* from within *Xic*, or from within *WRspice* running on local computers (the "clients").

B.4.1 SPICE Server Configuration

The SPICE server must have *WRspice* installed, and be served by a license server. As a prerequisite, *WRspice* should operate on the SPICE server host in the normal way.

In order for the "SPICE server" to actually become a server, the `wrspiced` daemon program must be running. This program is part of the *WRspice* distribution, and is installed in the same directory as the `wrspice` executable. The daemon manages the queue of submitted jobs and responses, and maintains the communications port.

The `wrspiced` daemon will establish itself on port number 3004 by default. If this port is already in use, or can't be used for some other reason, a different port can be specified on the `wrspiced` command line, or in the system services database as "spice/tcp". It is quite rare that operation on other than the default port is necessary.

The system services database is represented by the contents of the file `/etc/services` in simple installations. If using NIS, then the system will get its services information from elsewhere, in which case adding a service to the database will require more than editing the file. Your system administrator should be consulted in this case. To add the spice/tcp service, assuming the simple case, one must edit the `/etc/services` file (you must be root to do this) and add a line like the following:

```
spice       3004/tcp      #Whiteley Research Inc
```

This file probably also has an entry for "xictools 3010/tcp", which is the license server. The "3004" is the default port number to be used for SPICE communications, and should be replaced by the actual port used. This can be any valid port, but must be the same for all machines wishing to use the service.

Even if the default 3004 port is being used, it might be wise to edit the `/etc/services` file as above anyway to prevent someone else from using the port for another purpose. In no case is it *required* that the services database be modified; it is mainly for convenience. An alternate port can be specified when the daemon is started (see below).

The final step in configuring the SPICE server is to start the `wrspiced` daemon. The `wrspiced` program command line has the following form

`wrspiced` [`-l` *logfile*] [`-p` *program*] [`-m` *maxjobs*] [`-t` *port*]

There are four optional arguments.

`-l` *logfile*
: The *logfile* is a path to a file that will receive messages from the daemon. The default is the value of the SPICE_DAEMONLOG environment variable if set, or `/tmp/wrspiced.log`.

`-p` *program*
: This specifies the SPICE program to run, in case for some reason the `wrspice` binary has been renamed, or `wrspice` is not in the expected location. This overrides the values of the SPICE_PATH and SPICE_EXEC_DIR environment variables, which can also be used to set the path to the binary. The default is "`/usr/local/bin/wrspice`".

`-m` *maxjobs*
: This sets the maximum number of jobs that the server will allow to be running at the same time. The default is 2.

`-t` *port*
: This sets the port to be used by the daemon, and overrides any port set in the services database. Clients must use the same port number to connect to the SPICE server.

The daemon is started by simply typing the command. If a machine is to operate continuously as a SPICE server, it is recommended that the `wrspiced` daemon be started in the system initialization scripts. The daemon will run until explicitly killed by a signal, or the machine is halted.

It may be necessary to become root before starting `wrspiced`, as on some systems connection to the port will otherwise be refused due to permission requirements.

B.4.2 Client Configuration

If the SPICE server is using a port number other than the default 3004, it is necessary to inform the clients of the actual port used. This can be done in two ways. One way is to modify the services database, as discussed above, setting the port for "spice/tcp" to the number used by the server. It might be wise to add this entry to the `/etc/services` files of the clients in any case, to let others know that the port is in use.

The second method is to specify the port number when specifying the SPICE host name to *Xic* and *WRspice*. The host name can have the following form:

hostname[:*port*]

i.e., the host name can be suffixed by a colon-delimited number which gives the port.

A *WRspice* server can receive jobs from *Xic*, and from *WRspice* (rspice command). Both programs have means by which the SPICE server can be specified from within the program. One means common to both programs is through use of the SPICE_HOST environment variable. The variable should be set to the host name of the SPICE server, as resolvable by the client, followed by the optional colon and port number. When set, *Xic* will by default use this server for SPICE jobs initiated with the **Run** button in the side menu, and *WRspice* will use this host in the rspice command. In a situation where the SPICE server provides the only SPICE available, the SPICE_HOST variable should be set in the user's shell startup script. In *WRspice* the `rhost` shell variable and the rhost command can also be used to specify the remote host, and these override any value set in the environment.

Note: In *Xic*, when *WRspice* connects, a message is printed in the terminal window similar to

```
Stream established to wrspice://chaucer, port 4573.
```

The "port" in this case is *not* the `wrspiced` port discussed above.

Bibliography

[1] A. Vladimirescu and S. Liu, *The Simulation of MOS Integrated Circuits Using SPICE2*, ERL Memo No. ERL M80/7, Electronics Research Laboratory, University of California, Berkeley, Oct. 1980.

[2] B. J. Sheu, D. L. Scharfetter, and P. K. Ko, *SPICE2 Implementation of BSIM*, ERL Memo No. ERL M85/42, Electronics Research Laboratory, University of California, Berkeley, May 1985.

[3] J. R. Pierret, *A MOS Parameter Extraction Program for the BSIM Model*, ERL Memo Nos. ERL M84/99 and M84/100, Electronics Research Laboratory, University of California, Berkeley, Nov. 1984.

[4] Min-Chie Jeng, *Design and Modeling of Deep-Submicrometer MOSFETs*, ERL Memo Nos. ERL M84/99 and ERL M90/90, Electronics Research Laboratory, University of California, Berkeley, October 1990.

[5] Soyeon Park, *Analysis and SPICE implementation of High Temperature Effects on MOSFET*, Master's Thesis, University of California, Berkeley, December 1986.

[6] Clement Szeto, *Simulator of Temperature effects in MOSFETs (STEIM)*, Master's Thesis, University of California, Berkeley, May 1988.

[7] A. E. Parker and D. J. Skellern, *An Improved FET Model for Computer Simulators*, IEEE Trans. CAD, vol. 9, no.5, pp. 551-553, May 1990.

[8] Y. Cheng, M. Chan, K. Hui, M-C Jeng, Z. Liu, J. Huang, K. Chen, J. Chen, R. Tu, P. Ko and C. Hu, *BSIM3v3 Manual*, Department of Electrical Engineering and Computer Sciences, University of California, Berkeley, 1996.

[9] R. Saleh and A. Yang, Editors, *Simulation and Modeling, IEEE Circuits and Devices*, vol. 8, no. 3, pp. 7-8 and 49, May 1992.

[10] H. Statz et al., *GaAs FET Device and Circuit Simulation in SPICE*, IEEE Transactions on Electron Devices, Vol. 34, Number 2, February 1987 pp. 160-169.

[11] R. E. Jewett, *Josephson Junctions in SPICE2G5*, ERL Memo, Electronics Research Laboratory, University of California, Berkeley, 1982.

[12] S. R. Whiteley, *Josephson Junctions in SPICE3*, IEEE Trans. Magn., vol. 27, no. 2, pp. 2902-2905, March 1991.

[13] J. S. Roychowdhury and D. O. Pederson, *Efficient Transient Simulation of Lossy Interconnect*, Proc. DAC 91.

[14] Shen Lin and Ernest S. Kuh, *Transient Simulation of Lossy Interconnect*, Proc. DAC, pp 81-86, 1992.

This page intentionally left blank.

Index

.ac line, 32
.adc line, 48
.check line, 46, 214
.checkall line, 46
.control line, 45
.dc line, 32
.disto line, 33
.elif line, 28
.else line, 28
.end line, 22
.endc line, 45, 214
.endif line, 28
.endl line, 23
.ends line, 30
.endv line, 47
.exec line, 45
.four line, 40
.global line, 23
.ic line, 24
.if line, 28
.include line, 22
.lib line, 23
.measure line, 41
.model, 75
.monte line, 46, 214, 226
.mozyrc file, 240
.nodeset line, 24
.noexec line, 46
.noise line, 35
.op line, 36
.options line, 25
.param line, 27
.plot line, 40
.print line, 40
.pz line, 36
.save line, 39
.sens line, 37
.spinclude line, 22
.splib line, 23
.subckt line, 29
.table line, 26
.tf line, 37
.title line, 21
.tran line, 38
.verilog line, 47
.wrspiceinit file, 100
/width line, 41
@delta vector, 39

abs function, 138
abstol variable, 207
ac analysis, 8, 32
ac command, 157
acct variable, 211
acos function, 138
acosh function, 138
alias command, 152
alias substitution, 129
alter command, 158
analysis, 6
    ac, 8, 32
    dc, 8, 32
    distortion, 9, 33
    looping, 10
    Monte Carlo, 9
    noise, 9, 35
    operating range, 9
    pole-zero, 8, 36
    sensitivity, 9, 37
    transfer function, 8, 37
    transient, 8, 38
appendwrite variable, 198
area parameter, 72
argc variable, 148, 196
argv variable, 148, 196
asciiplot command, 186
asin function, 138
asinh function, 138
aspice command, 158
atan function, 138
atanh function, 138

batch mode, 95, 148
bipolar transistor instance, 73

bipolar transistor model, 81
break statement, 147
bug command, 192
bypass variable, 207

capacitor, 65
capacitor model, 78
cbrt function, 139
CCCS, 62
CCVS, 63
cd command, 152
ceil function, 139
check command, 158, 213
    variables, 161
    vectors, 161
checkiterate variable, 198, 215
chgtol variable, 207
Circuits tool, 12, 104, 107
cktvars variable, 196
codeblock command, 182
colorN variable, 201
Colors tool, 104, 106
colors, setting, 100
combine command, 186
combplot variable, 201
command completion, 128
command editing, 127
command interpretation, 132
command line options, 95–97
    –class, 97
    –name, 97
    –no-xshm, 97
    –sync, 97
    -b, 95
    -d, 96
    -dnone, 96
    -i, 96
    -j, 96
    -n, 96
    -o, 96
    -p, 96
    -q, 96
    -r, 96
    -s, 96
    -t, 97
    -x, 97
command scripts, 132
Commands tool, 104, 106
comment line, 22
compose command, 173
concatenation character, 220
constants plot, 136
continue statement, 147
control blocks, 45, 148
control structures, 146
convergence, 10, 24
cos function, 139
cosh function, 139
cptime variable, 211
cross command, 174
Ctrl-D, 15
curplot variable, 201
curplotdate variable, 201
curplotname variable, 201
curplottitle variable, 201
current circuit, 12
current flow convention, 20
current plot, 133
current source, 49

daemon, wrspiced, 150
db function, 139
dc analysis, 8, 32
dc analysis, chained, 31
dc command, 163
Debug tool, 104, 106
debug variable, 212
defad variable, 207
defas variable, 207
define command, 174
defl variable, 207
deftype command, 174
defw variable, 207
delete command, 163
dependent source, 60
deriv function, 139
destroy command, 164
device library, 6, 63
device models, 75
diff command, 175
diff_abstol variable, 198
diff_reltol variable, 198
diff_vntol variable, 198
dimensions, vectors, 135
diode instance, 72
diode model, 80
display command, 175
DISPLAY environment variable, 95, 98, 101
display variable, 212
disto command, 164
distortion analysis, 9, 33
dontplot variable, 212

dowhile block, 146
dphimax variable, 207
dpolydegree variable, 139, 199
drag and drop, 101, 111
dump command, 164

echo command, 152
echof command, 226
edit command, 15, 183
EDITOR environment variable, 99
editor variable, 199
end statement, 147
environment variables, 97–100
    DISPLAY, 98
    EDITOR, 99
    HOME, 98
    setting, 97
    SPICE_ASCIIRAWFILE, 100
    SPICE_BUGADDR, 99
    SPICE_DAEMONLOG, 100
    SPICE_EDITOR, 99
    SPICE_EXEC_DIR, 99
    SPICE_HLP_PATH, 99
    SPICE_HOST, 100
    SPICE_INP_PATH, 99
    SPICE_LIB_DIR, 99
    SPICE_NEWS_FILE, 99
    SPICE_OPTCHAR, 100
    SPICE_PATH, 99
    SPICE_TMP_DIR, 99
    TMPDIR, 99
erf function, 139
erfc function, 139
example run, 15
executable comments, 46
exp function, 139
exp tran function, 55
exponential specification, 55
expression list, 142
expression substitution, 131

fft function, 139
file formats, 233
    help files, 235
    rawfile, 233
file manager, 110
File Selection window, 110
Files tool, 104, 107
filetype variable, 199
floor function, 139
Fonts tool, 104, 108
foreach block, 146
fourgridsize variable, 199
fourier command, 175
free command, 165
FreeBSD, 6

gamma function, 140
gauss function, 140
gauss tran function, 55
Gaussian random specification, 55
global nodes, 23
global substitution, 129
gmin variable, 208
gminsteps variable, 208
goto statement, 147
gridsize variable, 201
gridstyle keyword
    lingrid, 203
    loglog, 203
    polar, 205
    smith, 205
    smithgrid, 205
    xlog, 206
    ylog, 206
gridstyle variable, 202
group variable, 202

hardcopy command, 187
hardcopy drivers, 202
hcopycommand variable, 202
hcopydriver variable, 202
hcopyheight variable, 203
hcopylandscape variable, 203
hcopyresol variable, 203
hcopywidth variable, 203
hcopyxoff variable, 203
hcopyyoff variable, 203
height variable, 197
help command, 192
help database, 126
help files, 235
Help menu, 104
help system, 11, 122
help viewer
    Anchor Buttons, 126
    Anchor Highlight, 126
    Anchor Plain, 126
    anchor styles, 126
    Anchor Underline, 126
    back, 124
    Bad HTML Warnings, 126

Clear Cache, 125
cookies, 125
Delayed Images, 125
disk cache, 125
Don't Cache, 125
forward, 124
Freeze Animations, 126
image formats, 125
Log Transactions, 126
No Cookies, 125
No Images, 125
Open, 124
Open File, 124
Print, 124
Progressive Images, 125
Quit, 124
Reload, 124
Reload Cache, 125
Save, 124
Search, 124
Set Font, 124
Show Cache, 125
stop, 124
Sync Images, 125
helpinitxpos variable, 199
helpinitypos variable, 199
helppath variable, 199
history command, 152
history substitution, 128
history variable, 197
HOME environment variable, 98

if block, 147
ifft function, 140
ignoreeof variable, 197
im function, 140
independent source, 49
inductor, 65
initialization files, 100
input format, 19
int function, 140
integ function, 140
interactive simulation, 12
interp tran function, 56
interplev variable, 208
interpolate function, 140
interpolation specification, 56
io redirection, 130
iplot command, 188
itl1 variable, 208
itl2 variable, 208
itl3 variable, 211
itl4 variable, 208
itl5 variable, 211

j function, 140
j0 function, 140
j1 function, 140
JFET instance, 73
JFET model, 83
jn function, 140
jobs command, 192
Josephson junction instance, 75
Josephson junction model, 93

label statement, 147
length function, 141
let command, 14, 134, 176
limpts variable, 211
limtim variable, 211
linearize command, 176
lingrid variable, 203
linplot variable, 203
list variable, 211
listing command, 165
ln function, 141
load command, 183
log function, 141
loglog variable, 203
loop command, 165
LTRA model, 79
lvlcod variable, 211
lvltim variable, 211

mag function, 141
mail window, 110
MAPI, 110
mapkey command, 153
margin analysis, 46
math functions, 138
maxdata variable, 31, 208
maxord variable, 208
mean function, 141
measurement, 12
measurement interval, 41
measurement types, 42
measurements, 41
measurements, chained, 44
measurements, source reference, 44
memory statistics, 101
memory use, 10
MESFET instance, 73
MESFET model, 84

method variable, 208
minbreak variable, 209
modelcard variable, 209
Monte Carlo analysis, 9, 214, 225
    example, 226
MOS L/W selection, 88
MOSFET instance, 74
MOSFET model, 85
mplot command, 188, 226
mplot window, 118
mplot_cur variable, 199
multi variable, 203
multidec program, 251
multidimensional vectors, 135
mutual inductor, 65

name field, 19
nfreqs variable, 199
noasciiplotvalue variable, 204
noaskquit variable, 197
nobjthack variable, 209
nobreak variable, 203
nocc variable, 197
noclobber variable, 197
node names, 20
node references, 134
node variable, 211
noedit variable, 197
noeditwin variable, 199
noerrwin variable, 197
noglob variable, 197
nogrid variable, 204
nointerp variable, 204
noise analysis, 9, 35
noise command, 166
noiter variable, 209
nojjtp variable, 209
nomod variable, 211
nomoremode variable, 197
nonomatch variable, 198
noopiter variable, 209
nopadding variable, 199
nopage variable, 199
noparse variable, 212
noprintscale variable, 199
norm function, 141
noshellopts variable, 209
nosort variable, 198
nosubckt variable, 212
number field, 19
numdgt variable, 200

off parameter, 10, 72
oldlimit variable, 209
op command, 166
operating range analysis, 9, 213
    checkDEL1, 214
    checkDEL2, 214
    checkFAIL, 215
    checkiterate, 215
    checkN1, 215
    checkN2, 215
    checkPNTS, 214
    checkSTP1, 214
    checkSTP2, 214
    checkVAL1, 214
    checkVAL2, 214
    example, 221
    file format, 218
    finding endpoints, 216
    opmax1, 215
    opmax2, 215
    opmin1, 215
    opmin2, 215
    r_scale, 215
    range, 215
    value, 215
    value1, 216
    value2, 216
operating range files, 214
operators, 136
option character, 95
options, 25
opts variable, 211

pause command, 154
pedigree, 6
ph function, 141
pick command, 177
pivrel variable, 209
pivtol variable, 210
platforms, 6
plot command, 13, 190
Plot Defs tool, 104, 106
plot description, 133
plot panel, 115
plot to file, 119
plot window, 115
plotgeom variable, 204
plotposn variable, 204
plots, 15
Plots tool, 104, 108
plots variable, 204

plotstyle keyword
combplot, 201
lineplot, 203
pointplot, 205
plotstyle variable, 204
plotting, 11
pointchars variable, 204
pointplot variable, 205
polar variable, 205
pole-zero analysis, 8, 36
poly specification, 52
polydegree variable, 205
polysteps variable, 205
pos function, 141
print command, 14, 177
print help text, 124
print panel, 118
printtoraw program, 252
proc2mod program, 252
program variable, 212
prompt variable, 198
pulse specification, 56
pulse tran function, 56
PWL specification, 57
pwl tran function, 57
pz command, 166

qhelp command, 193
quit command, 193
quoting, 130

rawfile, 15
rawfile ascii format, 233
rawfile binary format, 234
rawfile options, 233
rawfile variable, 200
rawfile variables, 234
rawfileprec variable, 200
re function, 141
rehash command, 154
reltol variable, 210
renumber variable, 210
repeat block, 146
reset command, 166
resistor, 66
resistor model, 78
resume command, 166
rhost command, 167
rhost variable, 200
rms function, 141
rnd function, 141
rprogram variable, 200
rspice command, 167
run command, 12, 167
rusage command, 193
accept, 193
elapsed, 193
faults, 193
loadtime, 194
lutime, 194
rejected, 194
solvetime, 194
space, 194
time, 194
totaltime, 194
totiter, 194
traniter, 194
tranpoints, 194
transolvetime, 194
trantime, 194

save command, 168
save help text, 124
savecurrent variable, 210
scale factors, 20
scaletype keyword
group, 202
multi, 203
single, 205
scaletype variable, 205
sced command, 184
scripts, 12, 132, 148
search help database, 124
seed command, 178
semicolon termination, 130
sens command, 169
sensitivity analysis, 9, 37
server mode, 148
set and let, 143
set command, 14, 154
setcirc command, 12, 169
setfont command, 105
setplot command, 178
setrdb command, 194
setscale command, 179
settype command, 179
sffm tran function, 58
shell, 12, 127
shell comand, 155
shell commands, 151
shell scripts, 132
Shell tool, 104, 106

shell variables, 130
shift command, 155
show command, 169
Sim Defs tool, 104, 106
sin function, 141
sin tran function, 58
sine pulse specification, 59
sine specification, 58
single frequency FM specification, 58
single quoted expressions, 21, 27
single variable, 205
sinh function, 141
smith variable, 205
smithgrid variable, 205
source, 49
source command, 12, 184
source expressions, 50
sourcepath variable, 107, 198
spec command, 180
spectrace variable, 180, 200
specwindow variable, 181, 200
specwindoworder variable, 181, 200
spice options, 25
SPICE_ASCIIRAWFILE environment variable, 100
SPICE_BUGADDR environment variable, 99
SPICE_DAEMONLOG environment variable, 100
SPICE_EDITOR environment variable, 99
SPICE_EXEC_DIR environment variable, 99
SPICE_HLP_PATH environment variable, 99
SPICE_HOST environment variable, 100
SPICE_INP_PATH environment variable, 99
SPICE_LIB_DIR environment variable, 99
SPICE_NEWS_FILE environment variable, 99
SPICE_OPTCHAR environment variable, 95, 100
SPICE_PATH environment variable, 99
SPICE_TMP_DIR environment variable, 99
spicepath variable, 200
spulse tran function, 59
sqrt function, 141
srcsteps variable, 210
startup file, 100, 105
state command, 170
status command, 170
step command, 171
steptype variable, 210
stop command, 171
strcmp command, 195
strictnumparse variable, 212
subcircuit call, 30
subcircuit declaration, 29
subcircuits, 29
subend variable, 210
subinvoke variable, 210
substart variable, 210
sum function, 141
switch, 66
switch model, 79

table reference specification, 59
table tran function, 59
tan function, 142
tanh function, 142
tbsetup command, 105
temp variable, 76, 210
temperature, 76
    coefficient, 78
text editor, 112
text-mode interface, 11
tf command, 171
ticmarks variable, 205
title line, 21
title variable, 206
TMPDIR environment variable, 99
tnom variable, 76, 210
tool control window, 11, 101
Tools menu, 103
TRA model, 79
trace command, 172
Trace tool, 104, 109
tracing, 11
tran command, 172
tran functions, 53
tran functions, in expressions, 54
transfer function analysis, 8, 37
transient analysis, 8, 38
transmission line, 67
transmission line, lumped, 71
trtol variable, 210
trytocompact variable, 211

uic keyword, 20
unalias command, 155
undefine command, 181
units variable, 201
unitvec function, 142
unixcom variable, 198
unlet command, 181
unset command, 155
Update button, 105
update command, 105
urc, 71

URC model, 80
user-defined functions, 54
usrset command, 156

variable expansion, 131
variable setting tools, 106
variable substitution, 21, 45, 131
variable types, 131
Variables tool, 104, 109, 130
variables, shell, 130
VCCS, 60
VCVS, 61
vector description, 133
vector function, 142
vector indexing, 134
vector substitution, 131, 135
vectors, 13
Vectors tool, 14, 104, 109
vectors, dimension, 31
vectors, dimensions, 135
Verilog, 11
Verilog blocks, 47
Verilog interface, 48
version command, 172
virtual memory use, 10
vntol variable, 211
voltage source, 49

where command, 195
while block, 146
WR button, 101
wr_lp script, 122
write command, 184
wrspiced daemon, 150
wrspiced program, 252

X resources, 100
xcompress variable, 206
xdelta variable, 206
xeditor command, 185
xglinewidth variable, 206
xgmarkers variable, 206
xgraph command, 191
Xic, 6, 12
xindices variable, 206
xlabel variable, 206
xlimit variable, 206
xlog variable, 206

y0 function, 142
y1 function, 142
ydelta variable, 206
ylabel variable, 206
ylimit variable, 206
ylog variable, 206
yn function, 142
ysep variable, 206

This page intentionally left blank.