Gnucap
The Gnu Circuit Analysis Package
Users manual

Albert Davis

February 21, 2025

Contents

Chapter 1

Introduction

1.1 What is it?

Gnucap is a general purpose mixed analog and digital circuit simulator. It performs nonlinear dc and transient analyses, fourier analysis, and ac analysis linearized at an operating point. It is fully interactive and command driven. It can also be run in batch mode. The output is produced as it simulates. Spice compatible models for the MOSFET (levels 1-7) and diode are included in this release.

Since it is fully interactive, it is possible to make changes and re-simulate quickly. This makes Gnucap ideal for experimenting with circuits as you might do in an iterative design or testing design principles as you might do in a course on circuits.

In batch mode it is mostly Spice compatible, so it is often possible to use the same file for both Gnucap and Spice.

The analog simulation is based on traditional nodal analysis with iteration by Newton's method and LU decomposition. An event queue and incremental matrix update speed up the solution considerably for large circuits and provide some of the benefits of relaxation methods but without the drawbacks.

It also has digital devices for true mixed mode simulation. The digital devices may be implemented as either analog subcircuits or as true digital models. The simulator will automatically determine which to use. Networks of digital devices are simulated as digital, with no conversions to analog between gates. This results in digital circuits being simulated faster than on a typical analog simulator, even with behavioral models.

Gnucap also has a simple behavioral modeling language that allows simple behavioral descriptions of most components including capacitors and inductors.

Gnucap is an ongoing research project. It is being released in a preliminary phase in hopes that it will be useful and that others will use it as a thrust or base for their research.

1.2 Starting

To run this program, type and enter the command: `gnucap`, from the command shell.

The prompt `-->` shows that the program is in the command mode. You should enter a command. Normally, the first command will be to `build` a circuit, or to `get` one from the disk. First time users should turn to the tutorial section for further assistance.

To run in batch mode, use `gnucap -b` *`file`*. It will run that file then exit.

To load a file on starting, use `gnucap` *`file`*. This is equivalent to starting with no arguments, then using the `get` command to load a file.

1.3 How to use this manual

The best approach is to read this chapter, then read the command summary at the beginning of chapter 2, then run the examples in the tutorial section. Later, when you want to use the advanced features, go back for more detail.

This manual is designed as a reference for users who are familiar with circuit design, and therefore does not present information on circuit design but only on the use of this program to analyze such a design. Likewise, it is not a text in modeling, although the models section does touch on it.

Throughout this manual, the following notation conventions are used:

- `Typewriter` font represents exactly what you type, or computer output.
- `Underlined typewriter` font is what you type, in a dialogue with the computer.
- Command words are shown in mixed UPPER and lower case. The upper case part must be entered exactly. The lower case part is optional, but if included must be spelled correctly.
- *Italics* indicate that you should substitute the appropriate name or value.
- Braces { } indicate optional parameters.
- Ellipses (...) indicate that an entry may be repeated as many times as needed or desired.

1.4 Command structure

Commands are whole words, but usually you only have to type enough of the word to make it unique. The first three letters will almost always work. In some cases less will do. The whole word is significant, if used, and must be spelled correctly.

In files, commands must be prefixed with a dot (.). This is done for compatibility with other simulation programs, such as SPICE.

Command options should be separated by commas or spaces. In some cases, the commas or spaces are not necessary, but it is good practice to use them.

Upper and lower case are usually the same.

Usually options can be entered in any order. The exceptions to this are numeric parameters, where the order determines their meaning, and command-like parameters, where they are executed in order. If parameters conflict, the last takes precedence.

In general, standard numeric parameters, such as sweep limits, must be entered first, before any options.

Any line starting with * is considered a comment line, and is ignored. Anything on any line following a quote is ignored. This is mainly intended for files.

This program supports abbreviated notation for floating point numeric entries. 'K' means kilo, or 'e3', etc. 'M' and 'm' mean milli, not mega (for Spice compatibility). 'Meg' means mega. Of course, it will also take the standard scientific notation. Letters following values, without spaces, are ignored.

T = Tera = e12
G = Giga = e9
Meg = Mega = e6

K = Kilo = e3
m = milli = e-3
u = micro = e-6
n = nano = e-9
p = pico = e-12
f = femto = e-15

1.5 Standard options

There are several options that are used in many commands that have a consistent meaning.

`Quiet` Suppress all unnecessary output, such as intermediate results, disk reads, activity indicators.

`Echo` Echo all disk reads to the console, as read from the disk.

`Basic` Format the output for compatibility with other software with primitive input parsers, such as C's *scanf* and Basic's *input* statements. It forces exponential notation, instead of our standard abbreviated notation. Any numbers that would ordinarily be printed without an exponent are not changed.

`Pack` Remove extra spaces from the output to save space at the expense of readability.

`<` Take the input from a file. The file name follows in the same line.

`>` Direct the output to a file. The file name follows. If the file already exists, it will ask permission to delete the old one and replace it with a new one with the same name.

`>>` Direct the output to a file. If the file already exists, the new data is appended to it.

1.6 Getting help, and the Gnucap user community

This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

Probably the best source of current information is the web site: `http://www.gnu.org/software/gnucap`. Here, you will find information on the latest developments, including other work related to gnucap, but not strictly part of it.

There are four mailing lists of interest to Gnucap users.

bug-gnucap This list is for bug reports and discussion about bugs in gnucap.

help-gnucap This is a general user discussion list for gnucap. Discussions about the use of gnucap, and sharing of ideas, models, and libraries, are all welcome here. Technical discussions should be light weight and user oriented.

info-gnucap This list is for announcements about gnucap. It is a moderated list. All postings come from the administrator.

gnucap-devel This list is for technical discussions relating to the development of gnucap. Technical discussions about simulator algorithms, modeling, and interfacing are all welcome here.

The web site contains the archives of these lists, and allows you to sign up for them.

1.7 How to contribute

There are a number of ways that you can contribute to help make Gnucap a better system. Perhaps the most important way to contribute is to write high-quality code for solving new problems, and to make your code freely available for others to use.

You can add significant value by developing models, even macro models, that can be distributed. Converting Spice models, publicizing which ones already work, or documenting any features that Gnucap needs to make it work, are all valuable contributions.

If you find Gnucap useful, consider providing additional funding to continue its development. Even a modest amount of additional funding could make a significant difference in the amount of time that is available for development and support.

If you cannot provide funding or contribute code, you can still help make Gnucap better and more reliable by reporting any bugs you find and by offering suggestions for ways to improve Gnucap.

If you are a teacher, you are making a significant contribution simply by using free software in your courses, and showing the students that they really do have a choice in software. You can further the contribution by encouraging student software projects that can be released as free software. You can also further the contribution by writing texts that use free software in the coursework, providing an alternative to those texts that promote closed source commercial software.

If you are an academic researcher, you can contribute by releasing your own software under GPL, and collaborating with others who do. You can help by using only open standards and avoiding proprietary languages such as the modeling languages of some proprietary simulators.

If you are a commercial user, you can help by giving financial support or equipment to the developers. Often, (as is the case with Gnucap), the principal developers are in the academic community, so by supporting free software, you are also supporting academic research and providing financial support for students.

1.8 Licensing

We protect your rights with two steps: (1) copyright the software, and (2) offer you this license which gives you legal permission to copy, distribute and/or modify the software.

Also, for each author's protection and ours, we want to make certain that everyone understands that there is no warranty for this free software. If the software is modified by someone else and passed on, we want its recipients to know that what they have is not the original, so that any problems introduced by others will not reflect on the original authors' reputations.

Finally, any free program is threatened constantly by software patents. We wish to avoid the danger that redistributors of a free program will individually obtain patent licenses, in effect making the program proprietary. To prevent this, we have made it clear that any patent must be licensed for everyone's free use or not licensed at all.

The precise terms and conditions for copying, distribution and modification follow.

GNU GENERAL PUBLIC LICENSE TERMS AND CONDITIONS FOR COPYING, DISTRIBUTION AND MODIFICATION

0. This License applies to any program or other work which contains a notice placed by the copyright holder saying it may be distributed under the terms of this General Public License. The "Program", below, refers to any such program or work, and a "work based on the Program" means either the Program or any derivative work under copyright law: that is to say, a work containing the Program or a portion of it, either verbatim or with modifications and/or translated into another language. (Hereinafter, translation is included without limitation in the term "modification".) Each licensee is addressed as "you".

Activities other than copying, distribution and modification are not covered by this License; they are outside its scope. The act of running the Program is not restricted, and the output from the Program is covered only if its contents constitute a work based on the Program (independent of having been made by running the Program). Whether that is true depends on what the Program does.

1. You may copy and distribute verbatim copies of the Program's source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice and disclaimer of warranty; keep intact all the notices that refer to this License and to the absence of any warranty; and give any other recipients of the Program a copy of this License along with the Program.

You may charge a fee for the physical act of transferring a copy, and you may at your option offer warranty protection in exchange for a fee.

2. You may modify your copy or copies of the Program or any portion of it, thus forming a work based on the Program, and copy and distribute such modifications or work under the terms of Section 1 above, provided that you also meet all of these conditions:

a) You must cause the modified files to carry prominent notices stating that you changed the files and the date of any change.

b) You must cause any work that you distribute or publish, that in whole or in part contains or is derived from the Program or any part thereof, to be licensed as a whole at no charge to all third parties under the terms of this License.

c) If the modified program normally reads commands interactively when run, you must cause it, when started running for such interactive use in the most ordinary way, to print or display an announcement including an appropriate copyright notice and a notice that there is no warranty (or else, saying that you provide a warranty) and that users may redistribute the program under these conditions, and telling the user how to view a copy of this License. (Exception: if the Program itself is interactive but does not normally print such an announcement, your work based on the Program is not required to print an announcement.)

These requirements apply to the modified work as a whole. If identifiable sections of that work are not derived from the Program, and can be reasonably considered independent and separate works in themselves, then this License, and its terms, do not apply to those sections when you distribute them as separate works. But when you distribute the same sections as part of a whole which is a work based on the Program, the distribution of the whole must be on the terms of this License, whose permissions for other licensees extend

to the entire whole, and thus to each and every part regardless of who wrote it.

Thus, it is not the intent of this section to claim rights or contest your rights to work written entirely by you; rather, the intent is to exercise the right to control the distribution of derivative or collective works based on the Program.

In addition, mere aggregation of another work not based on the Program with the Program (or with a work based on the Program) on a volume of a storage or distribution medium does not bring the other work under the scope of this License.

3. You may copy and distribute the Program (or a work based on it, under Section 2) in object code or executable form under the terms of Sections 1 and 2 above provided that you also do one of the following:

a) Accompany it with the complete corresponding machine-readable source code, which must be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,

b) Accompany it with a written offer, valid for at least three years, to give any third party, for a charge no more than your cost of physically performing source distribution, a complete machine-readable copy of the corresponding source code, to be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,

c) Accompany it with the information you received as to the offer to distribute corresponding source code. (This alternative is allowed only for noncommercial distribution and only if you received the program in object code or executable form with such an offer, in accord with Subsection b above.)

The source code for a work means the preferred form of the work for making modifications to it. For an executable work, complete source code means all the source code for all modules it contains, plus any associated interface definition files, plus the scripts used to control compilation and installation of the executable. However, as a special exception, the source code distributed need not include anything that is normally distributed (in either source or binary form) with the major components (compiler, kernel, and so on) of the operating system on which the executable runs, unless that component itself accompanies the executable.

If distribution of executable or object code is made by offering access to copy from a designated place, then offering equivalent access to copy the source code from the same place counts as distribution of the source code, even though third parties are not compelled to copy the source along with the object code.

4. You may not copy, modify, sublicense, or distribute the Program except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense or distribute the Program is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

5. You are not required to accept this License, since you have not signed it. However, nothing else grants you permission to modify or distribute the Program or its derivative works. These actions are prohibited by law if you do not accept this License. Therefore, by modifying or distributing the Program (or any work based on the Program), you indicate your acceptance of this License to do so, and all its terms and conditions for copying, distributing or modifying the Program or works based on it.

6. Each time you redistribute the Program (or any work based on the Program), the recipient automatically receives a license from the original licensor to copy, distribute or modify the Program subject to these terms and conditions. You may not impose any further restrictions on the recipients' exercise of the rights granted herein. You are not responsible for enforcing compliance by third parties to this License.

7. If, as a consequence of a court judgment or allegation of patent infringement or for any other reason (not limited to patent issues), conditions are imposed on you (whether by court order, agreement or otherwise) that contradict the conditions of this License, they do not excuse you from the conditions of this License. If you cannot distribute so as to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as a consequence you may not distribute the Program at all. For example, if a

patent license would not permit royalty-free redistribution of the Program by all those who receive copies directly or indirectly through you, then the only way you could satisfy both it and this License would be to refrain entirely from distribution of the Program.

If any portion of this section is held invalid or unenforceable under any particular circumstance, the balance of the section is intended to apply and the section as a whole is intended to apply in other circumstances.

It is not the purpose of this section to induce you to infringe any patents or other property right claims or to contest validity of any such claims; this section has the sole purpose of protecting the integrity of the free software distribution system, which is implemented by public license practices. Many people have made generous contributions to the wide range of software distributed through that system in reliance on consistent application of that system; it is up to the author/donor to decide if he or she is willing to distribute software through any other system and a licensee cannot impose that choice.

This section is intended to make thoroughly clear what is believed to be a consequence of the rest of this License.

8. If the distribution and/or use of the Program is restricted in certain countries either by patents or by copyrighted interfaces, the original copyright holder who places the Program under this License may add an explicit geographical distribution limitation excluding those countries, so that distribution is permitted only in or among countries not thus excluded. In such case, this License incorporates the limitation as if written in the body of this License.

9. The Free Software Foundation may publish revised and/or new versions of the General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

Each version is given a distinguishing version number. If the Program specifies a version number of this License which applies to it and "any later version", you have the option of following the terms and conditions either of that version or of any later version published by the Free Software Foundation. If the Program does not specify a version number of this License, you may choose any version ever published by the Free Software Foundation.

10. If you wish to incorporate parts of the Program into other free programs whose distribution conditions are different, write to the author to ask for permission. For software which is copyrighted by the Free Software Foundation, write to the Free Software Foundation; we sometimes make exceptions for this. Our decision will be guided by the two goals of preserving the free status of all derivatives of our free software and of promoting the sharing and reuse of software generally.

NO WARRANTY

11. BECAUSE THE PROGRAM IS LICENSED FREE OF CHARGE, THERE IS NO WARRANTY FOR THE PROGRAM, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE PROGRAM "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE PROGRAM IS WITH YOU. SHOULD THE PROGRAM PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.

12. IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MAY MODIFY AND/OR REDISTRIBUTE THE PROGRAM AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE PROGRAM (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR A FAILURE OF THE PROGRAM TO OPERATE WITH ANY OTHER PRO-

GRAMS), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

END OF TERMS AND CONDITIONS

Appendix: How to Apply These Terms to Your New Programs

If you develop a new program, and you want it to be of the greatest possible use to the public, the best way to achieve this is to make it free software which everyone can redistribute and change under these terms.

To do so, attach the following notices to the program. It is safest to attach them to the start of each source file to most effectively convey the exclusion of warranty; and each file should have at least the "copyright" line and a pointer to where the full notice is found.

¡one line to give the program's name and a brief idea of what it does.¿ Copyright (C) 19yy ¡name of author¿

This program is free software; you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation; either version 2 of the License, or (at your option) any later version.

This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

You should have received a copy of the GNU General Public License along with this program; if not, write to the Free Software Foundation, Inc., 675 Mass Ave, Cambridge, MA 02139, USA.

Also add information on how to contact you by electronic and paper mail.

If the program is interactive, make it output a short notice like this when it starts in an interactive mode:

Gnomovision version 69, Copyright (C) 19yy name of author Gnomovision comes with ABSOLUTELY NO WARRANTY; for details type 'show w'. This is free software, and you are welcome to redistribute it under certain conditions; type 'show c' for details.

The hypothetical commands 'show w' and 'show c' should show the appropriate parts of the General Public License. Of course, the commands you use may be called something other than 'show w' and 'show c'; they could even be mouse-clicks or menu items–whatever suits your program.

You should also get your employer (if you work as a programmer) or your school, if any, to sign a "copyright disclaimer" for the program, if necessary. Here is a sample; alter the names:

Yoyodyne, Inc., hereby disclaims all copyright interest in the program 'Gnomovision' (which makes passes at compilers) written by James Hacker.

¡signature of Ty Coon¿, 1 April 1989 Ty Coon, President of Vice

This General Public License does not permit incorporating your program into proprietary programs. If your program is a subroutine library, you may consider it more useful to permit linking proprietary applications with the library. If this is what you want to do, use the GNU Library General Public License instead of this License.

Chapter 2

Command descriptions

2.1 Command Summary

`*` Comment line.

`!` Pass a command to the operating system.

`<` Batch mode.

`>` Direct the "standard output" to a file.

`AC` Performs a small signal AC (frequency domain) analysis. Sweeps frequency.

`ALARM` Select points in circuit check against limits.

`ALTER` Perform analyses in queue. Changes follow. (Not implemented.)

`BUILD` Build a new circuit or change an existing one.

`CHDIR` Change current directory.

`CLEAR` Delete the entire circuit, titles, etc.

`DC` Performs a nonlinear DC analysis, for determining transfer characteristics. Sweeps DC input or component values.

`DELETE` Delete a part, or group of parts.

`DISTO` SPICE command not implemented.

`EDIT` Edit the circuit description using your editor.

`END` Perform analyses in queue. New circuit follows. (Implemented incorrectly.)

`EXIT` Exits the program. (Same as quit.)

`FANOUT` List by node number, the branches that connect to each node.

`FAULT` Temporarily change a component.

`FOURIER` Transient analysis, with results in frequency domain. (Different from SPICE.)

`GENERATOR` View and set the transient analysis function generator.

`GET` Get a circuit from a disk file. Deletes old one first.

`IC` Set transient analysis initial conditions. (Not implemented.)

`INCLUDE` Include a file from disk. Add it the what is already in memory.

`INSERT` Insert a node number. (Make a gap.)

`LIST` List the circuit on the console.

`LOG` Save a record of commands.

`MARK` Mark this time point, so transient analysis will restart here.

`MERGE` Get a file from disk. Add it the what is already in memory.

`MODIFY` Change a value, node, etc. For very simple changes.

`NODESET` Preset node voltages, to assist convergence. (Not implemented.)

`NOISE` SPICE command not implemented.

`OP` Performs a nonlinear DC analysis, for determining quiescent operating conditions. Sweeps temperature.

`OPTIONS` View and set system options. (Same as set.)

`PAUSE` Wait for key hit, while in batch mode.

`PLOT` Select points in circuit (and their range) to plot.

`PRINT` Select which points in the circuit to print as table.

`QUIT` Exits the program. (Same as exit.)

`SAVE` Save the circuit in a file.

`SENS` SPICE command not implemented.

`STATUS` Display resource usage, etc.

`SWEEP` Sweep a component. (Loop function.)

`TEMP` SPICE command not implemented.

`TF` SPICE command not implemented.

`TITLE` View and create the heading line for printouts and files.

`TRANSIENT` Performs a nonlinear transient (time domain) analysis. Sweeps time.

`UNFAULT` Undo faults.

`UNMARK` Undo mark. Release transient start point.

`WIDTH` Set output width.

2.2 ! command

2.2.1 Syntax

`!` *command*

2.2.2 Purpose

Run a program, or escape to a shell.

2.2.3 Comments

Any *command* typed here will be passed to the system for it to execute.

The bare command `!` will spawn an interactive shell. Exiting the shell will return.

2.2.4 Examples

`! ls *.ckt` Run the command `ls *.ckt` as if it were a shell command.

`!` No arguments mean to spawn an interactive shell.

2.3 < command

2.3.1 Syntax

`<` *filename*
`<<` *filename*

2.3.2 Purpose

Run a simulation in batch mode. Gets the commands and circuit from a disk file. `<<` clears the old circuit, first.

2.3.3 Comments

You can invoke the batch mode directly from the command that starts the program. The first command line argument is considered to be an argument for this command.

The file format is almost as you would type it on the keyboard. Commands must be prefixed with a dot, and circuit elements can be entered directly, as if in *build* mode. This is compatible with Spice.

The `log` command makes a file as you work the program, but the format is not correct for this command. To fix it, prefix commands with a dot, and remove the `build` commands.

Any line that starts with `*` a comment line.

Any line that starts with `.` (dot) is a command.

Any line that starts with a letter is a component to be added or changed.

A `<` command in the file transfers control to a new file. Files can be nested.

A bare `<` in the file (or the end of the file) gives it back to the console.

Unlike SPICE, commands are executed in order. This can result in some surprises when using some SPICE files. SPICE queues up commands, then executes them in a predetermined order.

2.3.4 Examples

`< thisone.ckt` Activates batch mode, from the file `thisone.ckt`, in the current directory.

`< runit.bat` Use the file `runit.bat`.

From the shell: on start up:

`gnucap afile` Start up the program. Start using the file `afile.ckt` in batch mode, as if you entered `< afile` as the first command.

`gnucap <afile` Start up the program. Start using the file `afile.ckt` with commands as if you typed them from the keyboard.

2.4 > command

2.4.1 Syntax

`>` *file*
`>>` *file*
`>`

2.4.2 Purpose

Saves a copy of all program output (except help) in a file.

2.4.3 Comments

`>` creates a new file for this output. If the file already exists, the old one is lost, and replaced by the new one.

`>>` appends to an existing file, if it exists, otherwise it creates one.

A bare `>` closes the file.

2.4.4 Examples

`> run1` Save everything in a file `run1` in the current directory. If `run1` already exists, the old one is gone.

`>> allof` Save everything in a file `allof`. If `allof` already exists, it is kept, and the new data is added to the end.

`>` Close the file. Stop saving.

2.5 AC command

2.5.1 Syntax

`AC` {*options ...*} *start stop stepsize* {*options ...*}

2.5.2 Purpose

Performs a small signal, steady state, AC analysis. Sweeps frequency.

2.5.3 Comments

The AC command does a linear analysis about an operating point. *It is absolutely necessary to do an* `OP` *analysis first on any nonlinear circuit. Not doing this is the equivalent of testing it with the power off.*

Three parameters are normally needed for an AC analysis: start frequency, stop frequency and step size, in this order. If all of these are omitted, the values from the most recent AC analysis are used.

If only one frequency is specified, a single point analysis will be done.

If only a new step size is specified, the old start and stop are kept and only the step size is changed. This is indicated by a keyword: `by`, `times`, `decade` or `octave`, or a symbol: `+` or `*`.

If the start frequency is zero, the program will still do an AC analysis. The actual frequency can be considered to be the limit as the frequency approaches zero. It is, therefore, still possible to have a non-zero phase angle, but delays are not shown because they may be infinite.

The nodes to look at must have been previously selected by the `print` or `plot` command. This is different fron Spice.

2.5.4 Options

`+` *stepsize* Linear sweep. Add *stepsize* to get the next step. Same as `By`.

`*` *multiplier* Log sweep. Multiply by *multiplier* to get the next step.

`>` *file* Send results of analysis to *file*.

`>>` *file* Append results to *file*.

`By` *stepsize* Linear sweep. Add *stepsize* to get the next step. Same as `+`.

`Decade` *steps* Log sweep. Use *steps* steps per decade.

`NOPlot` Suppress plotting.

`Octave` *steps* Log sweep. Use *steps* steps per octave.

`PLot` Graphic output, when plotting is normally off.

`Print` Send results to printer.

`Quiet` Suppress console output.

`TEmperature` *degrees* Temperature, degrees C.

`TImes` *multiplier* Log sweep. Multiply by *multiplier* to get the next step.

2.5.5 Examples

`ac 10m` A single point AC analysis at 10 mHz.

`ac 1000 3000 100` Sweep from 1000 Hz to 3000 Hz in 100 Hz steps.

`ac 1000 3000 Octave` Sweep from 1000 Hz to 3000 Hz in octave steps. Since the sweep cannot end at 3000 Hz, in this case, the last step will really be 4000 Hz.

`ac by 250` Keep the same limits as before, but use 250 Hz steps. In this case, it means to sweep from 1000 to 3000 Hz, because that it what it was the last time.

`ac 5000 1000 -250` You can sweep downward, if you want. Remember that the increment would be negative.

`ac 20 20k *2` Double the frequency to get the next step.

`ac 20k 20 *.5` You can do a log sweep downward, too. A multiplier of less than one moves it down.

`ac` Do the same AC sweep again.

`ac >afile` Save the results in the file `afile`. The file will look just like the screen. It will have all probe points. It will be a plot, if plotting is enabled. It will have the numbers in abbreviated notation. (10 nanovolts is `10.n`.)

2.6 `ALARM` command

2.6.1 Syntax

`ALArm`
`ALArm` *mode points ... ...*
`ALArm` *mode + points ... ...*
`ALArm` *mode - points ... ...*
`ALArm` *mode* CLEAR

2.6.2 Purpose

Select points in the circuit to check against user defined limits.

2.6.3 Comments

The '`alarm`' command selects points in the circuit to check against limits. There is no output unless the limits are exceeded. If the limits are exceeded a the value is printed.

There are separate lists of probe points for each type of analysis.

To list the points, use the bare command '`alarm`'.

Syntax for each point is *parameter(node)(limits)*, *parameter(componentlabel)(limits)*, or *parameter(index)(limits)*. Some require a dummy index.

For more information on the data available see the `print` command.

You can add to or delete from an existing list by prefixing with `+` or `-`. `alarm ac + v(3)` adds v(3) to the existing set of AC probes. `alarm ac - q(c5)` removes q(c5) from the list. You can use the wildcard characters `*` and `?` when deleting.

2.6.4 Examples

`alarm ac vm(12)(0,5) vm(13)(-5,5)` Check magnitude of the voltage at node 12 against a range of 0 to 5, and node 13 against a range of -5 to 5 for AC analysis. Print a warning when the limits are exceeded.

`alarm op id(m*)(-100n,100n)` Check current in all mosfets. In op analysis, print a warning for any that are outside the range of -100 to +100 nanoamps. The range goes both positive and negative so it is valid for both N and P channel fets.

`alarm tran v(r83)(0,5) p(r83)(0,1u)` Check the voltage and power of `R83` in the next transient analysis. The voltage range is 0 to 5. The power range is 0 to 1 microwatt. Print a warning when the range is exceeded.

`alarm` List all the probes for all modes.

`alarm dc` Display the DC alarm list.

`alarm ac CLear` Clear the AC list.

2.7 ALTER command

The Spice `Alter` command is not implemented. Similar functionality is available from the `sweep` command.

2.8 BUILD command

2.8.1 Syntax

`Build` {*line*}

2.8.2 Purpose

Builds a new circuit, or replaces lines in an existing one.

2.8.3 Comments

`Build` Lets you enter the circuit from the keyboard. The prompt changes to `>` to show that the program is in the build mode.

At this point, type in the circuit components in standard (Spice type) netlist format.

Component labels must be unique. If not, the old one is modified according to the new data, keeping old values where no new ones were specified.

Ordinarily, components are added to the end of the list. To insert at a particular place, specify the label to insert in front of. Example: `Build R77` will cause new items to be added before `R77`, instead of at the end.

In either case, components being changed or replaced do not change their location in the list.

If it is necessary to start over, `delete all` or `clear` will erase the entire circuit in memory.

To exit this mode, enter a blank line.

2.8.4 Examples

`build` Build a circuit. Add to the end of the list. This will add to the circuit without erasing anything. It will continue until you exit or memory fills up.

`b` This is the same as the previous example. Only the first letter of the 'Build' is necessary.

`build R33` Insert new items in front of R33.

2.9 CHDIR command

2.9.1 Syntax

`ChDir` {*path*}
`CD` {*path*}

2.9.2 Purpose

Changes or displays the current directory name.

2.9.3 Comments

Change the current directory to that specified by *path*. See your system manual for complete syntax.
If no argument is given the current directory is displayed.

2.9.4 Examples

`cd ../ckt` Change the current working directory to `../ckt`.

`cd` Show the current working directory name.

2.10 CLEAR command

2.10.1 Syntax

`CLEAR`

2.10.2 Purpose

Deletes the entire circuit, and blanks the title.

2.10.3 Comments

The entire word `clear` is required.
`Clear` is similar to, but a little more drastic than `delete all`.
After deleting anything, there is no way to get it back.
See also: `delete` command.

2.10.4 Examples

`clear` Delete the entire circuit.

2.11 DC command

2.11.1 Syntax

`DC` *start stop stepsize* {*options ...*}
`DC` *label start stop stepsize* {*options ...*}

2.11.2 Purpose

Performs a nonlinear DC steady state analysis, and sweeps the signal input, or a component value.

2.11.3 Status

Nesting of sweeps is not supported. (SPICE supports two levels of nesting.)

2.11.4 Comments

The nodes to look at must have been previously selected by the `print` or `plot` command.

If there are numeric arguments, without a part label, they represent a ramp from the `generator` function. They are the start value, stop value and step size, in order. They are saved between commands, so no arguments will repeat the previous sweep.

A single parameter represents a single input voltage. Two parameters instruct the computer to analyze for those two points only.

In some cases, you will get one more step outside the specified range of inputs due to internal rounding errors. The last input may be beyond the end point.

This command also sets up a movable operating point for subsequent `AC` analysis, which can be helpful in distortion analysis.

The program will sweep any simple component, including resistors, capacitors, and controlled sources. SPICE sweeps only fixed sources (types `V` and `I`).

2.11.5 Options

`*` *multiplier* Log sweep. Multiply the input by *multiplier* to get the next step. Do not pass zero volts!!

`>` *file* Send results of analysis to *file*.

`>>` *file* Append results to *file*.

`BY` *stepsize* Linear sweep. Add *stepsize* to get the next step.

`Continue` Use the last step of a `OP`, `DC` or `Transient` analysis as the first guess.

`Decade` *steps* Log sweep. Use *steps* steps per decade.

`LOop` Repeat the sweep, backwards.

`NOPlot` Suppress plotting.

`PLot` Graphic output, when plotting is normally off.

`Print` Send results to printer.

`Quiet` Suppress console output.

`REverse` Sweep in the opposite direction.

`TEmperature` *degrees* Temperature, degrees C.

`Times` *multiplier* Log sweep. Multiply the input by *multiplier* to get the next step. Do not pass zero volts!!

`TRace` *n* Show extended information during solution. Must be followed by one of the following:

`Off` No extended trace information (default, override .opt)

`Warnings` Show extended warnings

`Iterations` Show every iteration.

`Verbose` Show extended diagnostics.

2.11.6 Examples

`dc 1` Do a single point DC signal simulation, with '1 volt' input.

`dc -10 15 1` Sweep the circuit input from -10 to +15 in steps of 1. (usually volts.) Do a DC transfer simulation at each step.

`dc` With no parameters, it uses the same ones as the last time. In this case, from -10 to 15 in 1 volt steps.

`dc 20 0 -2` You can sweep downward, by asking for a negative increment. Sometimes, this will result in better convergence, or even different results! (For example, in the case of a bi-stable circuit.)

`dc` Since the last time used the `input` option, go back one more to find what the sweep parameters were. In this case, downward from 20 to 0 in steps of 2. (Because we did it 2 commands ago.)

`dc -2 2 .1 loop` After the sweep, do it again in the opposite direction. In this case, the sweep is -2 to +2 in steps of .1. After it gets to +2, it will go back, and sweep from +2 to -2 in steps of -.1. The plot will be superimposed on the up sweep. This way, you can see hysteresis in the circuit.

`dc temperature 75` Simulate at 75 degrees, this time. Since we didn't specify new sweep parameters, do the same as last time. (Without the loop.)

2.12 DELETE command

2.12.1 Syntax

`DELete` *label ...*
`DELete ALL`

2.12.2 Purpose

Remove a line, or a group of lines, from the circuit description.

2.12.3 Comments

To delete a part, by label, enter the label. (Example 'DEL R15'.) Wildcards '`*`' and '`?`' are allowed, in which case, all that match are deleted.

To delete the entire circuit, the entire word `ALL` must be entered. (Example 'DEL ALL'.)

After deleting anything, there is usually no way to get it back, but if a fault had been applied (see `fault` command) `restore` may have surprising results.

2.12.4 Examples

`delete all` Delete the entire circuit, but save the title.

`del R12` Delete `R12`.

`del R12 C3` Delete `R12` and `C3`.

`del R*` Delete all resistors. (Also, any models and subcircuits starting with `R`.)

2.13 DISTO command

The Spice `disto` command is not implemented. Similar functionality is not available.

2.14 EDIT command

2.14.1 Syntax

`Edit`
`Edit` *file*

2.14.2 Purpose

Use your editor to change the circuit.

2.14.3 Comments

The `edit` command runs your editor on a copy of the circuit in memory, then reloads it.

`Edit` *file* runs your editor on the specified *file*.

If you are only changing a component value, the `modify` command may be easier to use.

The program uses the `EDITOR` environment variable to find the editor to use. The command fails if there is no `EDITOR` defined.

2.14.4 Examples

`edit` Brings up your editor on the circuit.

`edit foo` Edits the file `foo` in your current directory.

2.15 END command

When run in batch mode from the shell, the `END` command cleans up and exits the program.

In script mode (< command) it ends the script and returns to the program prompt.

In interactive mode it exits the program.

2.16 EXIT command

2.16.1 Syntax

`EXIt`

2.16.2 Purpose

Terminates the program.

2.16.3 Comments

'`Quit`' also works.

Be sure you have saved everything you want to!

2.17 FANOUT command

2.17.1 Syntax

`FANout` {*nodes*}

2.17.2 Purpose

Lists connections to each node.

2.17.3 Comments

`Fanout` lists the line number and label of each part connected to each node. If both ends of a part are connected the same place, it is listed twice.

For a partial list, just specify the numbers. A number alone (`17`) says this branch alone. A trailing dash (`23-`) says from here to the end. A leading dash (`-33`) says from the start to here. Two numbers (`9 13`) specify a range.

2.17.4 Examples

`fanout` Lists all the nodes in the circuit, with their connections.

`fanout 99` List parts connecting to node 99.

`fanout 0` List the connections to node 0. (There must be at least one, unless you are editing a model.)

`fanout 78-` List connections to nodes 78 and up.

`fanout 124 127` List connections to nodes 124, 125, 126, 127.

2.18 FAULT command

2.18.1 Syntax

`FAult` *partlabel*=*value* ...

2.18.2 Purpose

Temporarily change a component value.

2.18.3 Comments

This command quickly changes the value of a component, usually with the intention that you will not want to save it.

If you apply this command to a nonlinear or otherwise strange part, it becomes ordinary and linear until the fault is removed.

It is an error to `fault` a model call.

If several components have the same label, the fault value applies to all of them. (They will all have the same value.)

The `unfault` command restores the old values.

2.18.4 Example

`fault R66=1k` R66 now has a value of 1k, regardless of what it was before.

`fault C12=220p L1=1u` C12 is 220 pf and L1 is 1 uH, for now.

`unfault` Clears all faults. It is back to what it was before.

2.19 FOURIER command

2.19.1 Syntax

`Fourier` *start stop stepsize* {*options ...*}

2.19.2 Purpose

Performs a nonlinear time domain (transient) analysis, but displays the results in the frequency domain.

Start, *stop*, and *stepsize* are frequencies.

2.19.3 Comments

This command is slightly different and more flexible than the SPICE counterpart. SPICE always gives you the fundamental and 9 harmonics. Gnucap will do the same if you only specify one frequency. SPICE has the probes on the same line. Gnucap requires you to specify the probes with the `print` command.

SPICE uses the last piece of a transient that was already done. Gnucap does its own transient analysis, continuing from where the most recent one left off, and choosing the step size to match the Fourier Transform to be done. Because of this the Gnucap Fourier analysis is much more accurate than SPICE.

The nodes to look at must have been previously selected by the `print` or `plot` command.

Three parameters are normally needed for a Fourier analysis: start frequency, stop frequency and step size, in this order.

If the start frequency is omitted it is assumed to be 0. The two remaining parameters are stop and step, such that stop > step.

If only one frequency is specified, it is assumed to be step size, which is equivalent to the fundamental frequency. The start frequency is zero and the stop frequency is set according the `harmonics` option (from the `options` command. The default is 9 harmonics.

If two frequencies are specified, they are stop and step. The order doesn't matter since stop is always larger than step.

Actually, this command does a nonlinear time domain analysis, then performs a Fourier transform on the data to get the frequency data. The transient analysis parameters (start, stop, step) are determined by the program as necessary to produce the desired spectral results. The internal time steps are selected to match the Fourier points, so there is no interpolation done.

The underlying transient analysis begins where the previous one left off. If you specify the "cold" option, it begins at time = 0. Often repeating a run will improve the accuracy by giving more time for initial transients to settle out.

See also: `Transient` command.

2.19.4 Options

`>` *file* Send results of analysis to *file*.

`>>` *file* Append results to *file*.

`Cold` Zero initial conditions. Cold start from power-up.

`Quiet` Suppress console output.

`SKip` *count* Force at least *count* internal transient time steps for each one used.

`STiff` Use a different integration method, that will suppress overshoot when the step size is too small.

`TEmperature` *degrees* Temperature, degrees C.

`TRACe` *n* Show extended information during solution. Must be followed by one of the following:

- `Off` No extended trace information (default, override .opt)
- `Warnings` Show extended warnings
- `Alltime` Show all accepted internal time steps.
- `Rejected` Show all internal time steps including rejected steps.
- `Iterations` Show every iteration.
- `Verbose` Show extended diagnostics.

2.19.5 Examples

`fourier 1Meg` Analyze the spectrum assuming a fundamental frequency of 1 mHz. Use the `harmonics` option to determine how many harmonics (usually 9) to display.

`fourier 40 20k 20` Analyze the spectrum from 40 Hz to 20 kHz in 20 Hz steps. This will result in a transient analysis with 25 micro-second steps. (1 / 40k). It will run for .05 second. (1 / 20).

`fourier 0 20k 20` Similar to the previous example, but show the DC and 20 Hz terms, also.

`fourier` No parameters mean use the same ones as the last time. In this case: from 0 to 20 kHz, in 20 Hz steps.

`fourier Skip 10` Do 10 transient steps internally for every step that is used. In this case it means to internally step at 2.5 micro-second, or 10 steps for every one actually used.

`fourier Cold` Restart at time = 0. This will show the spectrum of the power-on transient.

2.20 GENERATOR command

2.20.1 Syntax

`Generator` {*option-name=value ...*}

2.20.2 Purpose

Sets up an input waveform for `transient` and `Fourier` analysis. Emulates a laboratory type function generator.

2.20.3 Comments

This command sets up a singal source that is conceptually separate from the circuit. To use it, make the value of a component "generator(1)", or substitute a scale factor for the parameter.

The SPICE style input functions also work, but are considered to be part of the circuit, instead of part of the test equipment.

The parameters available are designed to emulate the controls on a function generator. There are actually two generators here: sine wave and pulse. If both are on (by setting non-zero parameters) the sine wave is modulated by the pulse, but either can be used alone.

Unless you change it, it is a unit-step function at time 0. The purpose of the command is to change it.

This command does not affect `AC` or `DC` analysis in any way. It is only for `transient` and `Fourier` analysis. In `AC` analysis, the input signal is always a sine wave at the analysis frequency.

Typical usage is the name of the control followed by its value, or just plain `Generator` to display the present values.

The actual time when switching takes place is ambiguous by one time step. If precise time switching is necessary, use the `Skip` option on the transient analysis command, to force more resolution. This ambiguity can usually be avoided by specifying finite rise and fall times.

2.20.4 Parameters

`Frequency` The frequency of the sine wave generator for a transient analysis. The sine wave is modulated by the pulse generator. A frequency of zero puts the pulse generator on line directly.

`Amplitude` The overall amplitude of the pulse and sine wave. A scale factor. It applies to everything except the *offset* and *init* values.

`Phase` The phase of the sine wave, at the instant it is first turned on.

`MAx` The amplitude of the pulse, when it is 'on'. (During the *width* time) If the sine wave is on (frequency not zero) this is the amplitude of the sine wave during the first part of the period. The *max* is scaled by *ampl*.

`MIn` The amplitude of the pulse, when it is 'off'. (After it falls, but before the next period begins.) Although we have called these *min* and *max*, there is no requirement that *max* be larger than *min*. If the sine wave is on, this is its amplitude during the second part of the period. The *min* is scaled by *ampl*.

`Offset` The DC offset applied to the entire signal, at all times after the initial delay. The *offset* is **not** scaled by *ampl*.

`Init` The initial value of the pulse generator output. It will have this value starting at time 0, until *Delay* time has elapsed. It will never return to this value, unless you restart at time 0.

`Rise` The rise time, or the time it takes to go from *MIn* to *MAx*, or for the first rise, *Init* to *MAx*. The rise is linear.

`Fall` The fall time. (The time required to go from *MAx* back to *MIn*.)

`Delay` The waiting time before the first rise.

`Width` The length of time the output of the generator has the value *Max*. A width of zero means that the output remains high for the remainder of the period. If you really want a width of zero, use a very small number, less than the step size.

`Period` The time for repetition of the pulse. It must be greater than the sum of rise + fall + width. A period of zero means that the signal is not periodic and so will not repeat.

2.20.5 Examples

The generator command ...

`gen` Display the present settings.

`gen Freq=1k` Sets the sine wave to 1 kHz. All other parameters are as they were before.

`gen Freq=0` Turns off the sine wave, leaving only the pulse.

`gen Ampl=0` Sets the amplitude to zero, which means the circuit has no input, except for possibly a DC offset.

`gen Period=.001 Freq=1m` Sets the period back to 1 millisecond. Applies 1 mHz modulation to the pulse, resulting in a pulsed sine wave. In this case, a 100 microsecond 10 volt burst, repeating every millisecond. Between bursts, you will get 2.5 volts, with reversed phase. The old values, in this case from 2 lines back (above) are kept. (`Ampl 5 Rise 10u Fall 10u` ...)

`gen Freq=60 Phase=90 Delay=.1` The sine wave frequency is 60 Hertz. Its phase is 90 degrees when it turns on, at time .1 seconds. It turns on sharply at the peak.

A component using it ...

`V12 1 0 generator(1)` Use the generator as the circuit input through this voltage source. The DC and AC values are 0.

`V12 1 0 tran generator(1) ac 10 dc 5` Same as before, except that the AC value is 10 and DC value is 5.

`Rinput 1 0 tran generator(1)` Unlike SPICE, the functions can be used on other components. The resistance varies in time according to the "generator".

2.21 `GET` command

2.21.1 Syntax

`GET` *filename*

2.21.2 Purpose

Gets an existing circuit file, after clearing memory.

2.21.3 Comments

The first comment line of the file being read is taken as the 'title'. See the `title` command.

Comments in the circuit file are stored, unless they start with `*+` in which case they are thrown away.

'Dot cards' are set up, but not executed. This means that variables and options are changed, but simulation commands are not actually done. As an example, the `options` command is actually performed, since it only sets up variables. The `ac` card is not performed, but its parameters are stored, so that a plain `ac` command will perform the analysis specified in the file.

Any circuit already in memory will be erased before loading the new circuit.

2.21.4 Examples

`get amp.ckt` Get the circuit file `amp.ckt` from the current directory.

`get /usr/foo/ckt/amp.ckt` Get the file `amp.ckt` from the `/usr/foo/ckt` directory.

`get npn.mod` Get the file `npn.mod`.

2.22 IC command

The Spice `IC` command is not implemented. Similar functionality is not available.

2.23 INSERT command

2.23.1 Syntax

`INsert` *node*
`INsert` *node, count*

2.23.2 Purpose

Open up node numbers inside a circuit.

2.23.3 Comments

To open up an internal node, enter `insert` followed by the number and how many. All node numbers higher than the first number will be raised by the second. The second (how many) is optional. If omitted, 1 will be assumed.

2.23.4 Examples

`insert 8 3` Insert 3 nodes before node 8. Adds 3 nodes (8,9,10) with no connections. Old node numbers 8 and higher have 3 added to them to make room. Old node 8 is now 11, 9 is now 12, 10 is now 13, 11 is 14, etc.

`insert 6` Insert one node at 6. Old nodes 6 and higher are incremented by 1. Old node 6 is now 7, 7 is 8, etc.

2.24 LIST command

2.24.1 Syntax

`List` {*label ...*}
`List` {*label - label*}

2.24.2 Purpose

Lists the circuit in memory.

2.24.3 Comments

Plain `List` will list the whole circuit on the console.

`List` with a component label asks for that one only. Wildcards are supported: `?` matches any character, once. `*` matches zero or more of any character.

For several components, list them.

For a range, specify two labels separated by a dash.

2.24.4 Examples

`list` List the entire circuit to the console.

`list R11` Show the component `R11`.

`list D12 - C5` List the part of the netlist from `M12` to `C5`, inclusive. `D12` must be before `C5` in the list.

`list D* C*` List all diodes and capacitors.

2.25 LOG command

2.25.1 Syntax

`LOg` *file*
`LOg >>` *file*
`LOg`

2.25.2 Purpose

Saves a copy of your keyboard entries in a file.

2.25.3 Comments

The '>>' option appends to an existing file, if it exists, otherwise it creates one.

Files can be nested. If you open one while another is already open, both will contain all the information.

A bare LOg closes the file. Because of this, the last line of this file is always LOg. Ordinarily, this will not be of any consequence, but if a log file is open when you use this file as command input, this will close it. If more than one LOg file is open, they will be closed in the reverse of the order in which they were opened, maintaining nesting.

See also: '>' and '<' commands.

2.25.4 Bugs

The file is an exact copy of what you type, so it is suitable for `gnucap <file` from the shell. It is NOT suitable for the < command in gnucap or the Spice-like mode `gnucap file` without <.

2.25.5 Examples

`log today` Save the commands in a file `today` in the current directory. If `today` already exists, the old one is gone.

`log >> doit` Save the commands in a file `doit`. If `doit` already exists, it is kept, and the new data is added to the end.

`log runit.bat` Use the file `runit.bat`.

`log` Close the file. Stop saving.

2.26 MARK command

2.26.1 Syntax

```
MArk
```

2.26.2 Purpose

Remember circuit voltages and currents.

2.26.3 Comments

After the `mark` command, the `transient` and `fourier` analysis will continue from the values that were kept by the `mark` command, instead of progressing every time.

This allows reruns from the same starting point, which may be at any time, not necessarily 0.

2.26.4 Examples

`transient 0 1 .01` A transient analysis starting at zero, running until 1 second, with step size .01 seconds. After this run, the clock is at 1 second.

`mark` Remember the time, voltages, currents, etc.

`transient` Another transient analysis. It continues from 1 second, to 2 seconds. (It spans 1 second, as before.) This command was not affected by the `mark` command.

`transient` This will do exactly the same as the last one. From 1 second to 2 seconds. If it were not for `mark`, it would have started from 2 seconds.

`transient 1.5 .001` Try again with smaller steps. Again, it starts at 1 second.

`unmark` Release the effect of `mark`.

`transient` Exactly the same as the last time, as if we didn't `unmark`. (1 to 1.5 seconds.)

`transient` This one continues from where the last one left off: at 1.5 seconds. From now on, time will move forward.

2.27 `MERGE` command

2.27.1 Syntax

`MErge` *filename*

2.27.2 Purpose

Gets an existing circuit file, without clearing memory.

2.27.3 Comments

The first comment line of the file being read is the new title, and replaces the existing title.

Comments in the circuit file are stored, unless they start with `*+` in which case they are thrown away.

'Dot cards' are set up, but not executed. This means that variables and options are changed, but simulation commands are not actually done. As an example, the `options` command is actually performed, since it only sets up variables. The `ac` command is not performed, but its parameters are stored, so that a plain `ac` command will perform the analysis specified in the file.

Any circuit already in memory is kept. New elements with duplicate labels replace the old ones. New elements that are not duplicates are added to the end of the list, as if the files were appended.

2.27.4 Examples

`merge amp.ckt` Get the circuit file `amp.ckt` from the current directory. Use it to change the circuit in memory.

`merge npn.mod` Include the file `npn.mod`.

2.28 `MODIFY` command

2.28.1 Syntax

`MOdify` *partlabel=value ...*

2.28.2 Purpose

Quickly change a component value.

2.28.3 Comments

This command quickly changes the value of a component. It is restricted to simply changing the value.

If several components have the same label or if wildcard characters are used, all are changed.

2.28.4 Example

`modify R66=1k` R66 now has a value of 1k, regardless of what it was before.

`modify C12=220p L1=1u` C12 is 220 pf and L1 is 1 uH.

`mod R*=22k` All resistors are now 22k.

2.29 NODESET command

The Spice `NODESET` command is not implemented. Similar functionality is not available.

2.30 NOISE command

The Spice `NOIse` command is not implemented. Similar functionality is not available.

2.31 OP command

2.31.1 Syntax

`OP` *start stop stepsize* {*options ...*}

2.31.2 Purpose

Performs a nonlinear DC steady state analysis, with no input. If a temperature range is given, it sweeps the temperature.

2.31.3 Comments

There are substantial extensions beyond the capabilities of the SPICE `op` command.

If there are numeric arguments, they represent a temperature sweep. They are the start and stop temperatures in degrees Celsius, and the step size, in order. They are saved between commands, so no arguments will repeat the previous sweep.

This command will use the `op` probe set, instead of automatically printing all nodes and source currents, so you must do "`print op` ...." before running op. We did it this way because we believe that printing everything all the time is usually unnecessary clutter. All of the information available from SPICE and more is available here. See the `print` command and the device descriptions for more details.

A single parameter represents a single temperature. Two parameters instruct the computer to analyze for those two points only.

This command also sets up the quiescent point for subsequent `AC` analysis. It is necessary to do this for nonlinear circuits. The last step in the sweep determines the quiescent point for the AC analysis.

2.31.4 Options

`*` *multiplier* Log sweep. Multiply the *absolute* temperature by *multiplier* to get the next step. The fact that it is offset to absolute zero may make the step sizes look strange.

`>` *file* Send results of analysis to *file*.

`>>` *file* Append results to *file*.

`BY` *stepsize* Linear sweep. Add *stepsize* to get the next step.

`Continue` Use the last step of a `OP`, `DC` or `Transient` analysis as the first guess.

`Input` *volts* Apply *volts* input to the circuit, instead of zero.

`LOop` Repeat the sweep, backwards.

`PLot` Graphic output, when plotting is normally off.

`Print` Send results to printer.

`Quiet` Suppress console output.

`REverse` Sweep in the opposite direction.

`TAble` Tabular output. Override default plot.

`TEmperature` *degrees* Temperature, degrees C. Override the sweep.

`TImes` *multiplier* Log sweep. Multiply the **absolute** temperature by *multiplier* to get the next step.

`TRace` *n* Show extended information during solution. Must be followed by one of the following:

- `Off` No extended trace information (default, override .opt)
- `Warnings` Show extended warnings
- `Iterations` Show every iteration.
- `Verbose` Show extended diagnostics.

2.31.5 Examples

`op 27` Do a DC operating point simulation at temperature 27 degrees Celsius.

`op -50 200 25` Sweep the temperature from -50 to 200 in 25 degree steps. Do a DC operating point simulation at each step.

`op` With no parameters, it uses the same ones as the last time. In this case, from -50 to 200 in 25 degree steps.

`op 200 -50 -25` You can sweep downward, by asking for a negative increment.

`op Input 2.3` Apply an input to the circuit of 2.3 volts. This overrides the default of no input.

`op TEmperature 75` Simulate at 75 degrees, this time. This isn't remembered for next time.

`op` Since the last time used the `TEmperature` option, go back one more to find what the sweep parameters were. In this case, downward from 200 to -50 in 25 degree steps. (Because we did it 3 commands ago.)

2.32 OPTIONS command

2.32.1 Syntax

`OPTions`
`OPTions` *option-name value ...*

2.32.2 Purpose

Sets options, iteration parameters, global data.

2.32.3 Comments

Typical usage is the name of the item to set followed by the value.

The bare command '`OPTions`' displays the values.

These options control the simulation by specifying how to handle marginal circumstances, how long to wait for convergence, etc.

Most of the SPICE options are supported, more have been added.

2.32.4 Parameters

`ACCT` Turns on accounting. When enabled, print the CPU time used after each command, and a summary on exit in batch more. This does not affect accounting done by the `status` command.

`NOACCT` Turns off accounting. (Not in SPICE.)

`LIST` Turns on echo of files read by `get` and `merge` commands, and in batch mode. (SPICE option accepted but not implemented.)

`NOLIST` Turns off list option. (Not in SPICE.)

`MOD` Enable printout of model parameters. (Accepted, but not implemented, to complement `NOMOD`.)

`NOMOD` Suppress printout of model parameters. (SPICE option accepted but not implemented.)

`PAGE` Enable page ejects at the beginning of simulation runs. (Accepted, but not implemented, to complement `NOPAGE`.)

`NOPAGE` Turn off page ejects. (SPICE option accepted but not implemented.)

`NODE` Enable printing of the node table. (SPICE option accepted but not implemented.)

`NONODE` Disable printing of the node table. (Accepted, but not implemented, to complement `NODE`.)

`OPTS` Enable printing of option values on every options command.

`NOOPTS` Disable automatic printing of option values. Option values are only printed on a null options command.

`GMIN` = x Minimum conductance allowed by the program. (Default = 1e-12 or 1 picomho.) Every node must have a net minimum conductance of `GMIN` to ground. If effective open circuits are found during the solution process (leading to a singular matrix) a conductance of `GMIN` is forced to ground, after printing an "open circuit" error message.

`RELTOL` = x Relative error tolerance allowed. (Default =.001 or .1%.) If the ratio of successive values in iteration are within `RELTOL` of one, this value is considered to have converged.

`ABSTOL` = x Absolute error tolerance allowed. (Default = 1e-12) If successive values in iteration are within `ABSTOL` of each other, this value is considered to have converged.

`VNTOL` = x Absolute voltage error required to force model re-evaluation. (Default = 1e-12 or 1 microvolt.) If the voltage at the terminals of a model is within `VNTOL` of the previous iteration, the model is not re-evaluated. The old values are used directly.

`TRTOL` = x Transient error "tolerance". (Default = 7.) This parameter is an estimate of the factor by which the program overestimates the actual truncation error.

`CHGTOL` = x Charge tolerance. (Default = 1e-14) It is used in step size control in transient analysis.

`PIVTOL` = x Pivot tolerance. (Default = 1e-13) SPICE option accepted but not implemented.

`PIVREL` = x Pivot ratio. (Default = 1e-3) SPICE option accepted but not implemented.

`NUMDGT` = x Number of significant digits to print for analysis results. (Default = 5.) It is silently limited to 3 to 20.

`TNOM` = x Nominal temperature. (Default = 27° C.) All components have their nominal value at this temperature.

`ITL1` = x DC iteration limit. (Default = 100.) Sets the maximum number of iterations in a DC, OP, or initial transient analysis allowed before stopping and reporting that it did not converge.

`ITL2` = x DC transfer curve iteration limit. (Default = 50.) SPICE option accepted but not implemented. Use ITL1 instead.

`ITL3` = x Lower transient iteration limit. (Default = 4.) If the number of iterations is more than `ITL3` the step size will not increase beyond its present size. Otherwise, it can grow by trstepgrow.

`ITL4` = x Upper transient iteration limit. (Default = 10.) Sets the maximum number of iterations on a step in transient analysis. If the circuit fails to converge in this many iterations the step size is reduced (by option trstepshrink), time is backed up, and the calculation is repeated.

`ITL5` = x Transient analysis total iteration limit. (Default = 5000.) SPICE option accepted but not implemented. Actual behavior is the same as `ITL5` = 0, in SPICE, which omits this test.

`ITL6` = x Source stepping iteration limit. (Default = 0.) SPICE option accepted but not implemented. Source stepping is not available.

`ITL7` = x Worst case analysis iteration limit. (Default = 1.) Sets the maximum number of iterations for the individual element trials in a DC or bias worst case analysis. If more iterations than this are necessary, the program silently goes on to the next step, as if nothing was wrong, which is usually the case.

ITL8 = x Convergence diagnostic iteration threshold. (Default = 100.) If the iteration count on a step exceeds ITL8 diagnostic messages are printed in an attempt to aid the user in solving the convergence problem.

CPTIME = x Total CPU job time limit. (Default = 30000.) SPICE option accepted but not implemented. There is no limit imposed.

LIMTIM = x CPU time reserved for plotting. (Default = 2.) SPICE option accepted but not implemented.

LIMPTS = x Max number of points printed. (Default = 201.) SPICE option accepted but not implemented.

LVLCOD = x Matrix solution and allocation method. (Default = 2, generate machine language.) SPICE option not implemented.

LVLTIM = x Time step control method. (Default = 2, truncation error.) SPICE option not implemented.

METHOD = x Integration method. (Default = TRAPezoidal.) Possible values are:

EULER backward Euler, unless forced to other

EULERONLY backward Euler only

TRAP usually trap, but Euler where better

TRAPONLY always trapezoid

DEFL = x MOSFET default channel length in meters. (Default = 100u.)

DEFW = x MOSFET default channel width in meters. (Default = 100u.)

DEFAD = x MOSFET default drain diffusion area in square meters. (Default = 0.)

DEFAS = x MOSFET default source diffusion area in square meters. (Default = 0.)

SEED = x Seed used by the random number generator. (Default = 1.) (ECA-2 equivalent = Random.) (Not available in SPICE.) The same random numbers will be used every time, determined by this seed number. Setting this to zero is a special case, causing each run to start from a random point.

WCZERO = x Worst case zero window. (Default = 1e-9) (Not available in SPICE.) Sets a window for the difference in a DC or bias worst case analysis. Differences less than this are assumed to be zero, for purposes of setting direction flags. This prevents cluttering up the screen with very small numbers that are essentially zero.

DAMPMAX = x Normal Newton damping factor. (Default = 1.) Sets the damping factor for iteration by damped Newton's method, used when all is well. It must be between 0 and 1, as close to 1 as possible and still achieve convergence. The useful range is from .9 to 1. Setting DAMPMAX too low will probably cause convergence to a nonsense result.

DAMPMIN = x Newton damping factor in problem cases. (Default = .5) Sets the damping factor for iteration by damped Newton's method, used when there are problems. It must be between 0 and 1, and is usually set somewhat less than DAMPMAX. The useful range is from .5 to .9. Setting it lower than .5 may cause convergence to a nonsense result. Aside from that, a lower value (but less than DAMPMAX) tends to improve robustness at the expense of convergence speed.

`DAMPSTrategy` = x Damping strategy. (Default = 0) The actual damping factor to use is determined by heuristics. Normally the damping factor is `DAMPMAX`. It is reduced to `DAMPMIN` when certain conditions occur, then it drifts back up on subsequent iterations. This parameter turns the various heuristics on or off. The number to use is the sum of the following flags.

1 the second iteration on any voltage or time step. (usually helps robustness, but always increases iteration count.)

2 if the voltage at any nonlinear node exceeds the range determined by `VMIN`, `VMAX`, and `LIMIT`. (usually not desirable.)

4 if any device limiting algorithm is activated. (usually not desirable.)

10 when any device crosses a region boundary. (usually desirable and has little cost.)

20 when a FET or BJT is reversed. (usually helps robustness. sometimes increases iteration count.)

`FLOOR` = x Effective zero value. (Default = 1e-21) Results values less than `FLOOR` are shown as zero. Other small numbers are rounded to the nearest `FLOOR`.

`VFLOOR` = x Effective zero value for voltage probes. (Default = 1e-15) Results values less than `VFLOOR` are shown as zero. Other small numbers are rounded to the nearest `VFLOOR`.

`TEMPAMB` = x Simulation temperature. (Default = 27° C.) Sets the ambient temperature, in degrees Celsius. This is the temperature at which the simulation takes place, unless changed by some other command.

`Short` = x Resistance of voltage source or short. (Default = 1e-7 or 10 $\mu\Omega$.) Sets the default resistance of voltage sources. In some cases, inductors are replaced by resistors, if so, this is the value. It is also the resistance used to replace short circuits anywhere they are not allowed and the program finds one.

`TRansits` = x Mixed mode transition count. (Default = 2) Sets the number of "good" transitions for a supposedly digital signal to be accepted as digital.

`IN` = x Input width. (Default = 80.) Sets the last column read from each line of input. Columns past this are ignored. This option is present only for SPICE compatibility, through the `width` command, which is an alias for `options`.

`OUT` = x Output width. (Default = 80.) Sets the output print width, for tables and character graphics.

`XDivisions` = x X axis divisions. (Default = 4) Sets the number of divisions on the X axis for plotting.

`YDivisions` = x Y axis divisions. (Default = 4) Sets the number of divisions on the Y axis for plotting.

`ORder` = x Equation ordering. (Default = auto.) Determines how external node numbers are mapped to internal numbers. The values are `FORward`, `REVerse`, and `AUTo`.

`MODe` = x Simulation mode selection. (Default = mixed.) Values are `ANAlog`, `DIGital`, and `MIXed`. In analog mode, logic elements (type U) are replaced by their subcircuits as if they were type X. In digital mode, logic elements are simulated as digital regardless of whether the signals are proper or not, as in traditional mixed-mode simulation. In mixed mode, logic elements may be simulated as analog or digital depending on the signals present.

`BYPass` Bypass model evaluation if appropriate. If the last two iterations indicate that an element is converged or dormant, do not evaluate it but use its old values directly. (Default)

VBYpass Check only voltage to bypass model evaluation. This produces a faster but possibly less accurate simulation.

NOBYpass Do not bypass model evaluation.

LUBypass Bypass parts of LU decomposition if appropriate. If only a few elements of the matrix were changed solve only those parts of the LU matrix that depend on them. (Default)

NOLUbypass Do not bypass parts of LU decomposition. Solve the entire LU matrix whenever a matrix solution is called for regardless of whether it is actually needed.

INCmode Incrementally update the matrix. Instead of rebuilding the matrix on every iteration, keep as much of the old matrix as possible and make incremental changes. (Default)

NOIncmode Do not incrementally update the matrix. This eliminates a possible cause of roundoff error at the expense of a slower simulation.

TRACELoad Use a queue to only load changed elements to the matrix. This results in faster loading and has no known drawbacks. (Default)

NOTRACELoad Do not use a queue to only load changed elements to the matrix. Instead, load all elements, even if they are unchanged or zero. This is always slower, and is forced if "noincmode".

LIMIT = x Internal differential branch voltage limit. (Default = 1e10, essentially disabled.) All circuit branch voltages may be limited to $\pm x$ to aid in convergence. This is intended as a convergence aid only. It may or may not help.

VMIN = x Negative node voltage limit. (Default = -30) All node voltages may be limited to $-x$ to aid in convergence and prevent numeric overflow. This is intended as a convergence aid only. It may or may not help.

VMAX = x Positive node voltage limit. (Default = 30) All node voltages may be limited to $+x$ to aid in convergence and prevent numeric overflow. This is intended as a convergence aid only. It may or may not help.

DTMIn = x Minimum time step. (Default = 1e-12.) The smallest internal time step in transient analysis. The transient command dtmin option and the dtratio option override it if it is bigger.

DTRatio = x The ratio between minimum and maximum time step. (Default = 1e9).

RSTray Include series resistance in device models. This creates internal nodes and results in a significant speed and memory penalty. It also makes convergence characteristics worse.

NORSTray Do not include series resistance in device models. This results in faster simulations and better numerical accuracy at the expense of model accuracy. Differences between rstray and norstray have been observed to be insignificant most of the time. Some popular commercial versions of SPICE do not implement series resistance at all, so norstray may be more consistent with other simulators.(Default)

CSTray Include capacitance in device models. This may create internal nodes and result in a significant speed and memory penalty. It also may make convergence characteristics worse. (Default)

NOCSTray Do not include capacitance in device models. This results in faster simulations and better numerical accuracy at the expense of model accuracy. Differences between cstray and nocstray are usually significant, since often the strays are the dominant reactive elements.

`Harmonics` = x Harmonics in Fourier analysis. (Default = 9) The number of harmonics to display in a Fourier analysis, unless specified otherwise.

`TRSTEPGrow` = x The maximum internal step size growth in transient analysis. (Default = 2.)

`TRSTEPShrink` = x The amount to decrease the transient step size by when convergence fails. (Default = 8.)

`TRReject` = x Transient error rejection threshold. (Default = .5) Controls how bad the truncation error must be to reject a time step. A value of .5 means that if the step reqested is smaller than .5 times the step size used, the current step will be rejected. If the new step is .8 times the old step size it will be adjusted but the step just calculated will not be rejected.

2.32.5 Examples

`options` Display the present settings.

`options itl1=50` Allows 50 iterations in a `dc` or `op` analysis.

2.33 `PAUSE` command

2.33.1 Syntax

`PAuse` *comment*

2.33.2 Purpose

Suspend batch mode. Wait for the user to hit a key.

2.33.3 Status

This command does not work on all systems, due to buffering of console i/o.

2.33.4 Comments

Prints `Continue?` and waits for a key hit. Type '`n`', '`N`', escape or control-c to terminate the batch mode. Type anything else to continue.

Any *comment* is ignored.

2.33.5 Examples

```
pause Try more gain
```

`pause` These both work the same. Ask to continue, wait for a key hit, then go on.

2.34 PLOT command

2.34.1 Syntax

PLot
PLot *mode points ... ...*
PLot *mode + points ... ...*
PLot *mode - points ... ...*
PLot *mode* CLEAR

2.34.2 Purpose

Select points in the circuit for graphic output. Select graphic output.

2.34.3 Status

The plotting leaves something to be desired. Only two signals can be plotted at a time. The output file is corrupt when plotting is on.

2.34.4 Comments

The '`plot`' command selects where to look at the circuit, or where to hook the oscilloscope probe.

There are separate lists of probe points for each type of analysis.

To list the probe points, use the bare command '`plot`'.

Syntax for each point is *parameter(node)(limits)*, *parameter(componentlabel)(limits)*, or *parameter(index)(limits)*. Some require a dummy index.

For more information on the data available see the `print` command.

You must set the scaling. If you do not, the default range is fixed at -5 to 5. Gnucap cannot auto-scale because it generates the plot during simulation, so the necessary information is not available yet. Spice can auto-scale only because it waits for the simulation to complete before producing any output.

`Plot` uses the same variables as `print`. See the print command for a list of what is available.

The options `plot` and `noplot` on any analysis command turn plotting on and off a single run. The `plot` command turns plotting on and tabular output off. The `print` command turns plotting off and tabular output on.

You can add to or delete from an existing list by prefixing with `+` or `-`. `plot ac + v(3)` adds v(3) to the existing set of AC probes. `plot ac - q(c5)` removes q(c5) from the list. You can use the wildcard characters `*` and `?` when deleting.

Plotting is limited to 2 items.

2.34.5 Examples

`plot ac vm(12)(0,5) vm(13)(-5,5)` The magnitude of the voltage at node 12 with a range of 0 to 5, and node 13 with a range of -5 to 5 for AC analysis.

`plot dc v(r26)` The voltage across `R26` for DC analysis. Since there is no range, default values will be used.

`plot tran v(r83)(0,5) p(r83)(0,1u)` Plot the voltage and power of `R83` in the next transient analysis. The voltage scale is 0 to 5. The power scale is 0 to 1 microwatt.

`plot` List all the probes for all modes.

`plot dc` Display the DC plot list.

`plot ac CLear` Clear the AC list.

2.35 PRINT command

2.35.1 Syntax

`PRint`
`PRint` *mode points ... ...*
`PRint` *mode + points ... ...*
`PRint` *mode - points ... ...*
`PRint` *mode* `CLEAR`

2.35.2 Purpose

Select points in the circuit for tabular output. Select tabular output.

2.35.3 Comments

The '`print`' command selects where to look at the circuit, or where to hook the voltmeter (ammeter, watt meter, ohm meter, etc.) probe.

There are separate lists of probe points for each type of analysis.

To list the probe points, use the bare command '`print`'.

On start-up, probes are not set. You must do the command '`print op v(nodes)`' or put '`.print op v(nodes)`' in the circuit file to get any output from the `op` command.

Syntax for each point is *parameter(node)*, *parameter(componentlabel)*, or *parameter(index)*. Some require a dummy index.

You can access components in subcircuits by connecting the names with dots. For example: `R56.X67.Xone` is `R56` in `X67` in `Xone`. Some built-in elements, including diodes, transistors, and mosfets, contain subcircuits with internal elements. `Cgd.M12` is the gate to drain capacitor of mosfet `M12`.

If the component does not exist, you will get an error message. If the component exists but the parameter is not valid for that type, there will be no error message but the value printed will be obviously bogus.

The options `plot` and `noplot` on any analysis command turn plotting on and off a single run. The `plot` command turns plotting on and tabular output off. The `print` command turns plotting off and tabular output on.

You can add to or delete from an existing list by prefixing with `+` or `-`. `print ac + v(3)` adds v(3) to the existing set of AC probes. `print ac - q(c5)` removes q(c5) from the list. You can use the wildcard characters `*` and `?` when deleting.

For AC analysis, by adding a suffix letter to the parameter, you can get the magnitude `M`, phase `P`, real part `R`, or imaginary part `I`. Adding `DB` gives the value in decibels, relative to 1. For example, `VRDB(R13)` gives you the real part of the voltage across R13, in decibels.

2.35.4 Node probes

Several parameters are available at each node.

All modes

`V` Voltage.

All except Transient

`Z` Impedance looking into the node.

Transient, DC, OP only

`Logic` A numeric interpretation of the logic value at the node. The value is displayed encoded in a number of the form *a.bc* where *a* is the logic state: 0 = logic 0, 1 = rising, 2 = falling, 3 = logic 1. *b* is an indication of the quality of the digital signal. 0 is a fully valid logic signal. Nonzero indicates it does not meet the criteria for logic simulation. *c* indicates how the node was calculated: 0 indicates logic simulation. 1 indicates analog simulation of a logic device. 2 indicates analog simulation of analog devices.

AC only

In addition to those listed here, you can add a suffix (`M, P, R, I` and `DB`) for magnitude, phase, real part, imaginary part, and decibels, to any valid probe.

`VI` Imaginary part of the voltage.

`VDB` Decibels relative to 1 v.

`ZI` Imaginary part of the impedance looking into the node.

`ZP` Impedance phase (angle between voltage and current).

2.35.5 Status probes

There are several status variables that can be probed.

All modes

`Temperature(0)` The simulation temperature in degrees Celsius.

`TIme(0)` The current time in a transient analysis. In AC analysis it shows the time at which the bias point was set, 0 if it was set in a DC or OP analysis, or -1 if it is the bias was not set (power off).

Transient, DC, OP only

`GEnerator` The output of the "signal generator". In a transient analysis, it shows the output of the signal generator, as set up by the `generator` command. In a DC analysis, it shows the DC input voltage (not the power supply). In an OP analysis, it shows the DC input, normally zero.

`ITer(0)` The number of iterations needed for convergence for this printed step including any hidden steps.

`ITer(1)` The number of iterations needed for convergence for this printed step not including any hidden steps.

`ITer(2)` The total number of iterations needed since startup including check passes.

`Control(0)` A number indicating why the simulator chose this time to simulate at.

1 The user requested it. One of the steps in a sweep.

2 A discrete event. An element required a solution at this time.

3 The effect of the "`skip`" parameter.

4 The iteration count exceeded `ITL4` so the last step was rejected and is being redone at a smaller time step.

5 The iteration count exceeded `ITL3` so the time interval is the same as it was last time.

6 Determined by local truncation error or some other device dependent approximation in hopes of controlling accuracy.

7,8 The step size was limited to twice the previous step size.

9 The step size was reduced to half the interval to an event to avoid a tiny next step.

10 + x The previous step was rejected.

20 + x A zero time step was replaced by *mrt*.

30 + x The required step size less than *mrt*, so it was replaced by *mrt*.

`Control(1)` The number internal time steps. (1 if all steps are printed. One more than the number of hidden steps.)

2.35.6 Element probes

Each element type has several parameters that can be probed. In general, the form is `Parameter(element)`. Wild cards are allowed in element names to allow probing the same parameter of a group of elements.

For components in a subcircuit, the names are connected with dots. For example `R12.X13` is `R12` in the subcircuit `X13`.

Most two node elements (capacitors, inductors, resistors, sources) and four terminal elements (controlled sources) have at least the following parameters available. Others are available for some elements.

All modes

`V` Branch voltage. The first node in the net list is assumed positive. This is the same as "output voltage".

`Vout` Output voltage. The voltage across the "output" terminals.

`VIN` Input voltage. The voltage across the "input" terminals. For two terminal elements, input and output voltages are the same.

`I` Branch current. It flows into the first node in the net list, out of the second.

`P` Branch power. Positive power indicates dissipation. Negative power indicates that the part is supplying power. Its value is the same as (PD - PS). In AC analysis, it is the real part only.

`NV` Nominal value. In most cases, this is just the value which is constant, but it can vary for internal elements of complex devices.

`EV` The effective value of the part, in its units. If the part is ordinary, it will just show its value, but if it is time variant or nonlinear, it shows what it is now.

R Resistance. The effective resistance of the part, in ohms. In AC analysis, shows the magnitude of the self impedance. In OP, DC or TRansient analysis, shows its incremental resistance. In TRansient analysis, it shows the effective Z-domain resistance of inductors and capacitors.

Y Admittance. $1/R$.

All except Transient

Z Impedance at a port. The port impedance seen looking into the circuit across the branch. It does not include the part itself. In transient analysis, it shows the effective Z-domain impedance, which is a meaningless number if there are capacitors or inductors in the circuit.

ZRAW Impedance at a port, raw. This is the same as "Z" except that it includes the part itself.

Transient, DC, OP only

These parameters are available in addition to the above.

PD Branch power dissipated. The power dissipated in the part. It is always positive and does not include power sourced.

PS Branch power sourced. The power sourced by the part. It is always positive and does not consider its own dissipation.

F The result of evaluating the function related to the part. It is the voltage across a resistor, the charge stored in a capacitor, the flux in an inductor, etc.

INput The "input" of the device. It is the current through a resistor or inductor, the voltage across a capacitor or admittance, etc. It is the value used to evaluate nonlinearities.

IOffset The offset current in the device. The current through a nonlinear device can be considered to have two parts: a passive part and an offset.

IPassive The passive part of the current.

AC only

In addition to those listed here, you can add a suffix (M, P, R, I and DB) for magnitude, phase, real part, imaginary part, and decibels, to any valid probe. Negative phase is capacitive. Positive phase is inductive.

PI Reactive (imaginary) power, volt-amps reactive.

PIDB Decibels relative to 1 va reactive.

PM Volt amps, complex power.

PMDB Decibels relative to 1 va.

PP Power phase (angle between voltage and current).

2.35.7 Examples

`print ac v(12) v(13) v(14)` The voltage at nodes 12, 13, and 14 for AC analysis.

`print dc v(r26)` The voltage across `R26`, for DC analysis.

`print tran v(r83) p(r83)` Voltage and power of `R83`, for transient analysis.

`print dc i(c8) p(r5) z(r5)` The current through `C8`, power dissipated in `R5`, and the impedance seen looking into the circuit across `R5`.

`print op v(nodes)` The voltage at all nodes for the `op` command.

`print` List all the probes, for all modes.

`print op` Display the `OP` probe list.

`print ac clear` Clear the AC list.

2.36 `QUIT` command

2.36.1 Syntax

`Quit`

2.36.2 Purpose

Terminates the program.

2.36.3 Comments

'`exit`' also works.

Be sure you have saved everything you want to!

2.37 `SAVE` command

2.37.1 Syntax

`SAve` *filename* {*options ...*}

2.37.2 Purpose

Saves the circuit on the disk.

2.37.3 Comments

The file is in an ASCII format, so the list may be used as part of a report. It is believed to be compatible with other simulators such as Berkeley Spice to the extent that the capabilities are the same. Compatibility with commercial Spice derivatives may be a problem because they all have proprietary extensions and are incompatible with each other.

If the file name specified already exists, the old file is deleted and replaced by a new file of the same name, after asking you for permission.

You can save a part of a circuit. See the `list` command for more details.

2.37.4 Examples

`save works.ckt` Save the circuit in the file `works.ckt`, in the current directory.

`save` Save the circuit. Since you did not specify a file name, it will ask for one.

`save partof.ckt R*` Save a partial circuit, just the resistors, to the file `partof.ckt`. (See the `List` command.)

`save /client/sim/ckt/no33` You can specify a path name.

2.38 SENS command

The Spice `SENS` command is not implemented. Similar functionality is not available.

2.39 STATUS command

2.39.1 Syntax

`STatus`

2.39.2 Purpose

Shows information on how the system resources are being utilized.

2.40 SWEEP command

2.40.1 Syntax

`SWeep` {*stepcount*} *partlabel=range ...*

2.40.2 Purpose

Sweep a component (or group of components) over a range. Set up a loop for iteration.

2.40.3 Comments

This command begins a loop which will sweep a component or group of components.

When this command is given, the only apparent actions will be a change in the prompt from '`-->`' to '`>>>`', and some disk action.

The different prompt means that commands are not executed immediately, but are stored in a temporary file.

The bare command will repeat the same command sequence as the last time `sweep` was run, and not prompt for anything else.

Additional components can be swept at the same time by entering a '`FAult`' command at the '`>>>`' prompt. The '`fault`' behaves differently here: It accepts a range, which is the sweep limits.

The '`go`' command will end the entry sequence, and make it all happen. After this, the values are restored. (Also, all `fault`s are restored, as if by the '`restore`' command.)

All commands can be used in this mode. Of course, some of them are not really useful (`quit`) because they work as usual.

Only linear, ordinary parts can be swept. (No semiconductor devices, or elements using behavioral modeling.) The tolerance remains unchanged. If you attempt to sweep a nonlinear or otherwise strange part, it becomes ordinary and linear during the sweep.

2.40.4 Example

```
-->sweep  5   R14=1,100k   R15=100k,1
>>>list
>>>ac 500 2k oct
>>>go
```

This sequence of commands says to simultaneously sweep R14 and R15 in 5 steps, in opposite directions, list the circuit and do an AC analysis for each step.

Assuming the circuit was:

```
R14  1   0   50k
R15  2   0   50k
```

The result of this sequence would be:

```
R14  1   0   1
R15  2   0   100k
```

an AC analysis

```
R14  1   0   25.75k
R15  2   0   75.25k
```

an AC analysis

```
R14  1   0   50.5k
R15  2   0   50.5k
```

an AC analysis

```
R14  1   0   75.25k
R15  2   0   25.75k
```

an AC analysis

```
R14  1   0   100k
R15  2   0   1
```

an AC analysis

After all this is done, the circuit is restored, so `list` would show:

```
R14  1   0   50k
R15  2   0   50k
```

You could accomplish the same thing by entering `fault` commands at the '`>>>`' prompt.

```
-->sweep  5
>>>fault  R14=1, 100k
>>>fault  R15=100k, 1
>>>list
>>>ac 500 2k oct
>>>go
```

2.41 TEMP command

The Spice `TEmp` command is not implemented. Similar functionality is available by sweeping the `op` command.

2.42 TF command

The Spice `TF` command is not implemented. Similar functionality is not available.

2.43 TITLE command

2.43.1 Syntax

`TItle`
`TItle` *a line of text*

2.43.2 Purpose

View and create the heading line for printouts and files.

2.43.3 Comments

There is a header line at the beginning of every file, to help you identify it in the future. This command sets up what it says. It also sets up a heading for printouts and graphs.

When you use the '`get`' command to bring in a new circuit, it replaces the title with the one in the file. The '`title`' command lets you change it, for the next time it is written out.

2.43.4 Examples

`title This is a test.` Sets the file heading to '`This is a test.`' In the future, all files written will have '`This is a test.`' as their first line.

`title` Displays the file heading. In this case, it prints '`This is a test.`'

2.44 `TRANSIENT` command

2.44.1 Syntax

`Transient` *start stop stepsize* {*options ...*}
`Transient` *stepsize stop start* {*options ...*}

2.44.2 Purpose

Performs a nonlinear time domain (transient) analysis.

2.44.3 Comments

The nodes to look at must have been previously selected by the `Print` or `Plot` command.

Three parameters are normally needed for a Transient analysis: start time, stop time and step size, in this order. The SPICE order (step size, stop, start) is also acceptable. An optional fourth parameter is the maximum internal time step.

If all of these are omitted, the simulation will continue from where the most recent one left off, with the same step size, unless the circuit topology has been changed. It will run for the same length of time as the previous run.

Do not use a step size too large as this will result in errors in the results. If you suspect that the results are not accurate, try a larger argument to 'Skip'. This will force a smaller internal step size. If the results are close to the same, they can be trusted. If not, try a still larger 'Skip' argument until they appear to match close enough.

The most obvious error of this type is aliasing. You must select sample frequency at least twice the highest signal frequency that exists anywhere in the circuit. This frequency can be very high, when you use the default step function as input. The signal generator does **not** have any filtering.

2.44.4 Options

`>` *file* Send results of analysis to *file*.

`>>` *file* Append results to *file*.

`Cold` Zero initial conditions. Cold start from power-up.

`DTMIn` = x Minimum time step. (Default = from `options`) The smallest internal time step in transient analysis. The `transient` command `dtmin` option and the `dtratio` option override it if it is bigger.

`DTRatio` = x The ratio between minimum and maximum time step. (Default = from `options`).

`NOPlot` Suppress plotting.

`PLot` Graphic output, when plotting is otherwise off.

`Quiet` Suppress console output.

`Skip` *count* Force at least *count* simulation steps for each one displayed. If the output is a table or ASCII plot, the extra steps are hidden.

`TEmperature` *degrees* Temperature, degrees C.

`TRace` *n* Show extended information during solution. Must be followed by one of the following:

`Off` No extended trace information (default, override .opt)

`Warnings` Show extended warnings

`Alltime` Show all accepted internal time steps.

`Rejected` Show all internal time steps including rejected steps.

`Iterations` Show every iteration.

`Verbose` Show extended diagnostics.

`UIC` Use initial conditions. Use the values specified with the `IC =` options on the various elements.

2.44.5 Examples

`transient 0 100u 10n` Start at time 0, stop after 100 micro-seconds. Simulate using 10 nanosecond steps.

`transient` No parameters mean to continue from the last run. In this case it means to step from 100 us to 200 us in 10 ns steps. (The same step size and run length, but offset to start where the last one stopped.

`transient skip 10` Do 10 extra steps internally for every step that would be done otherwise. In this case it means to internally step at 1 nanosecond. If the output is in tabular form, the extra steps are hidden.

`transient 0` Start over at time = 0. Keep the same step size and run length.

`transient cold` Zero initial conditions. This will show the power-on transient.

`transient >arun` Save the results of this run in the file `arun`.

2.45 UNFAULT command

2.45.1 Syntax

`UNFault`

2.45.2 Purpose

Undo any action from `fault` commands.

2.45.3 Comments

This command reverses the action of all `fault` commands.

It will also clean up any side effects of an aborted `sweep` command.

`Unfault` is automatically invoked on any `clear` command.

If you change the circuit in any other way, `unfault` will bring back the old on top of the changes. This can bring on some surprises.

2.45.4 Example

`fault R66=1k` R66 now has a value of 1k, regardless of what it was before.

`unfault` Clears all faults. In this case, R66 has its old value again.

`unfault` can bring on surprises. Consider this sequence ...

```
V1   1   0    ac  1
C3   1   2    1u
R4   2   0    10k
```

`fault C3=100p` C3 is 100 picofarads, for now.

`modify C3=220p` C3 is 220 pf, for now. It will be restored.

`modify R4=1k` R4 is 1k. It will not be restored.

`restore` C3 back to 1 uf, but R4 still 1k.

2.46 UNMARK command

2.46.1 Syntax

```
UNMark
```

2.46.2 Purpose

Forget remembered circuit voltages and currents. Undo the '`mark`' command.

2.46.3 Comments

Allow time to proceed. It has been held back by the '`mark`' command.

2.46.4 Examples

`transient 0 1 .01` A transient analysis starting at zero, running until 1 second, with step size .01 seconds. After this run, the clock is at 1 second.

`mark` Remember the time, voltages, currents, etc.

`transient` Another transient analysis. It continues from 1 second, to 2 seconds. (It spans 1 second, as before.) This command was not affected by the `mark` command.

`transient` This will do exactly the same as the last one. From 1 second to 2 seconds. If it were not for `mark`, it would have started from 2 seconds.

`transient 1.5 .001` Try again with smaller steps. Again, it starts at 1 second.

`unmark` Release the effect of `mark`.

`transient` Exactly the same as the last time, as if we didn't `unmark`. (1 to 1.5 seconds.)

`transient` This one continues from where the last one left off: at 1.5 seconds. From now on, time will move forward.

2.47 WIDTH command

2.47.1 Syntax

`Width` {`IN=`*value*} {`OUT=`*value*}

2.47.2 Purpose

Set input and output width.

2.47.3 Comments

The '`width`' command is the same as the '`options`' command. It is provided for SPICE compatibility. SPICE uses `width` to set two parameters: `in` and `out`, which we set with the `options` command.

Chapter 3

Circuit description

3.1 Summary

To describe a circuit, you must provide a 'netlist'. The netlist is simply a list of the components with their connections and values. The format is essentially the same as the standard SPICE format.

Before doing this, number the nodes on your schematic. (A node is a place where parts connect together.) Then, each part gets a line in the netlist (circuit description). In its simplest form, which you will use most of the time, it is just the type, such as 'R' for resistor, or a label, like 'R47', followed by the two nodes it connects to, then its value.

Example: '`R29 6 8 22k`' is a 22k resistor between nodes 6 and 8.

Node 0 is used as a reference for all calculations and is assumed to have a voltage of zero. (This is the ground, earth or common node.) Nodes must be nonnegative integers, but need not be numbered sequentially.

There should be a DC path through the circuit to node 0 from every node that is actually used. The circuit cannot contain a cutset of current sources and/or capacitors. If either of these cases occurs, it will be discovered during analysis. The program will attempt to correct the error, issue an 'open circuit' error message and continue. This is rarely a problem with real circuits. Most circuits have such a path, however indirect.

Semiconductor devices require both a device statement, and a `.model` statement (or "card"). The device statement, described in the Circuit description chapter, defines individual devices as variations from a prototype, as is required for different devices on the same substrate. The model statement, described in this chapter, defines process dependent parameters, which usually apply to all devices on a substrate.

The `.model` card syntax is:

`.model` *mname type* {*args*}

Mname is the model name, which elements will use to refer to this model. *Type* is one of several types of built-in models. *Args* is a list of the parameters, of the form *name*=*value*.

`D` Diode model

`NMOS` N-channel MOSFET model

`PMOS` P-channel MOSFET model

`LOGIC` Logic family description

`SW` Voltage controlled switch

`CSW` Current controlled switch

`C` Semiconductor capacitor

`R` Semiconductor resistor

`TABLE` y/x table of values

3.2 `C`: Capacitor

3.2.1 Syntax

`C`*xxxxxx n+ n– value*
`C`*xxxxxx n+ n– expression*
`C`*xxxxxx n+ n– value* {`IC=`*initial-voltage*}
`C`*xxxxxx n+ n– model* {`L=`*length*} {`W=`*width*} {`TEMP=`*temperature*} {`IC=`*initial-voltage*}
`.CAPacitor` *label n+ n– expression*

3.2.2 Purpose

Capacitor, or general charge storage element.

3.2.3 Probes

The following probes (Transient, DC, and OP analysis) are available in addition to those available for all basic elements.

`DT` Time step. The internal time step used for this device for numerical integration. It is not necessarily the same as the global time step.

`TIME` Time. The time of the most recent calculation of this device. It is not necessarily the same as the global time.

`TIMEOLD` The time of the previous calculation of this device. It is not necessarily the same as the global time.

`TIMEFuture` The latest recommended time for the next sample, as determined by this device. The actual time will probably be sooner than this.

`CHarge` The charge stored in this capacitor.

`Q` The same as `Charge`.

`Capacitance` The effective capacitance of this device. For a fixed capacitor, it is constant. It will vary if this device is nonlinear.

`DQDT` The time derivative of charge. Hopefully this is the same as current, but it is calculated a different way and can be used as an accuracy check.

`DQ` The change in charge compared to the previous sample. Its primary use is in debugging models and numerical problems.

3.2.4 Comments

N+ and *n–* are the positive and negative element nodes, respectively. *Value* is the capacitance in Farads.

The (optional) initial condition is the initial (time = 0) value of the capacitor voltage (in Volts). Note that the initial conditions (if any) apply only if the `UIC` option is specified on the `transient` command.

You may specify the *value* in one of three forms:

1. A simple value. This is the capacitance in Farads.
2. An expression, as described in the behavioral modeling chapter. The expression can specify the charge as a function of voltage, or the capacitance as a function of time.
3. A *model*, which calculates the capacitance as a function of length and width, referencing a `.model` statement of type `C`. This is compatible with the Spice-3 "semiconductor capacitor".

3.2.5 Model statement

A model statement may be used,, with model type `C` or `Cap`. The parameters are:

`CJ` = *x* Junction bottom capacitance. (Farads / meter squared). (Default = 0.)

`CJSW` = *x* Junction sidewall capacitance. (Farads / meter). (Default = 0.)

`DEFW` = *x* Default width. (meters). (Default = 1e-6)

`NARROW` = *x* Narrowing due to side etching. (meters). (Default = 0.)

`TC1` = *x* First order temperature coefficient. (Farads / degree C). (Default = 0.) (Not in Spice.)

`TC2` = *x* Second order temperature coefficient. (Farads / degree C squared). (Default = 0.) (Not in Spice.)

`TNOM` = *x* Parameter measurement temperature. (degrees C.). (Default = 27.) (Not in Spice.)

Capacitance is computed by the formula:

```
capacitance = CJ * (L - NARROW) * (W - NARROW)
  + 2 * CJSW * (L + L - 2 * NARROW)
```

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

```
value *= (1 + TC1 * (T-T0) + TC2 * (T-T0)^2)
```

3.3 Trans-capacitor

3.3.1 Syntax

`.TCAPacitor` *label n+ n– nc+ nc– expression*
`.TCAPacitor` *label n+ n– nc+ nc– value* {`IV=`*initial-voltage*}
`.TCAPacitor` *label n+ n– model* {`L=`*length*} {`W=`*width*} {`IC=`*initial-voltage*}

3.3.2 Purpose

Trans-capacitor, or charge transfer device.

3.3.3 Probes

All probes that apply to ordinary capacitors also apply here.

3.3.4 Comments

N+ and *n–* are the positive and negative element nodes, respectively. *Nc+* and *nc–* are the positive and negative controlling nodes, respectively.

This device places a charge between the output nodes that depends on the voltage on its input nodes. If you parallel the input with the output, it becomes an ordinary capacitor. While the use of this device may appear straightforward, be careful. It is easy to use it in an unstable way.

All options, expressions, models, and probes that apply to ordinary capacitors can also be used here.

It is used internally in some transistor models.

3.4 `D`: Diode

3.4.1 Syntax

`D`*xxxxxx n+ n– mname* {*area*} {*args*}
`.DIOde` *xxxxxx n+ n– mname* {*area*} {*args*}

3.4.2 Purpose

Junction diode.

3.4.3 Comments

N+ and *n–* are the positive and negative element nodes, respectively. *Mname* is the model name. *Area* is the area factor. If the area factor is omitted, a value of 1.0 is assumed. *Args* is a list of additional arguments. The parameters available are a superset of those available in SPICE.

A diode can also use a MOSFET model (type `NMOS` or `PMOS`) to represent the equivalent of the source-bulk or drain-bulk diodes.

When the element is printed out, by a `list` or `save` command, the the computed values of `IS`, `RS`, `CJ`, and `CJSW` are printed as a comment if they were not explicitly entered.

3.4.4 Element Parameters

`Area` = *x* Area factor. (Default = 1.0) If optional parameters `IS`, `RS`, and `CJO` are not specified, the `.model` value is multiplied by `area` to get the actual value.

`Perim` = *x* Perimeter factor. (Default = 1.0) If optional parameter `CJSW` is not specified, the `.model` value is multiplied by `perim` to get the actual value.

`IC` = *x* Initial condition. The initial voltage to use in transient analysis, if the `UIC` option is specified. Default: don't use initial condition. This is presently ignored, but accepted for compatibility.

`OFF` Start iterating with this diode off, in DC analysis.

`IS` = *x* Saturation current. This overrides `IS` in the `.model`, and is not affected by `area`. Default: use `IS` from `.model` * `area`.

RS = x Ohmic (series) resistance. This overrides RS in the .model, and is not affected by area. Default: use RS from .model * area.

CJ = x Zero-bias junction capacitance. This overrides CJ in the .model, and is not affected by area. Default: use CJ from .model * area.

CJSW = x Zero-bias sidewall capacitance. This overrides CJSW in the .model, and is not affected by perim. Default: use CJSW from .model * perim.

GParallel = x Parallel conductance. This overrides GParallel in the .model, and is not affected by area. Default: use GParallel from .model * area.

3.4.5 Model Parameters

IS = x Normalized saturation current. (Amperes). (Default = 1.0e-14) IS is multiplied by the *area* in the element statement to get the actual saturation current. It may be overridden by specifying IS in the element statement.

RS = x Normalized ohmic resistance. (Ohms) (Default = 0.) RS is multiplied by the *area* in the element statement to get the actual ohmic resistance. It may be overridden by specifying RS in the element statement.

N = x Emission coefficient. (Default = 1.0) In ECA-2 the default value was 2.

TT = x Transit time. (Default = 0.) The diffusion capacitance is given by: $c_d = TTg_d$ where g_d is the diode conductance.

VJ = x Junction potential. (Default = 1.0) Used in computation of capacitance. For compatibility with older versions of SPICE, PB is accepted as an alias for VJ.

CJo = x Normalized zero-bias depletion capacitance. (Default = 0.) CJo is multiplied by the *area* in the element statement to get the actual zero-bias capacitance. It may be overridden by specifying CJ in the element statement.

Mj = x Grading coefficient. (Default = 0.5)

PBSw = x Sidewall junction potential. (Default = PB)

CJSw = x Normalized zero-bias sidewall capacitance. (Default = 0.) CJSw is multiplied by the *perimeter* in the element statement to get the actual zero-bias capacitance. It may be overridden by specifying CJSW in the element statement.

MJSw = x Sidewall grading coefficient. (Default = 0.33)

EG = x Activation energy. (electron Volts) (Default = 1.11, silicon.) For other types of diodes, use:

1.11 ev. Silicon (default value)
0.69 ev. Schottky barrier
0.67 ev. Germanium
1.43 ev. GaAs
2.26 ev. GaP

XTI = x Saturation current temperature exponent. (Default = 3.0) For Schottky barrier, use 2.0.

`KF` = x Flicker noise coefficient. (Default = 0.) SPICE parameter accepted but not implemented.

`AF` = x Flicker noise exponent. (Default = 1.0) SPICE parameter accepted but not implemented.

`FC` = x Coefficient for forward bias depletion capacitance formula. (Default = 0.5)

`BV` = x Reverse breakdown voltage. (Default = ∞.) SPICE parameter accepted but not implemented.

`IBV` = x Current at breakdown voltage. (Default = 1 ma.) SPICE parameter accepted but not implemented.

`GParallel` = x Parallel conductance. (Default = 0.)

3.4.6 Probes

`Vd` Voltage. The first node (anode) is assumed positive.

`Id` Total current. It flows into the first node (anode), out of the second (cathode). I(Dxxxx) is the same as IJ(Dxxxx) + IC(Dxxxx).

`VJ` Junction voltage. The voltage across the junction, excluding the series resistance.

`VSR` Resistive voltage. The voltage across the series resistance, excluding the junction voltage.

`IJ` Junction current. The current through the junction. IJ(Dxxxx) is the same as I(Yj.Dxxxx).

`IC` Capacitor current. The current through the parallel capacitor. IC(Dxxxx) is the same as I(Cj.Dxxxx).

`P` Power. P(Dxxxx) is the same as PJ(Dxxxx) + PC(Dxxxx).

`PD` Power dissipated. The power dissipated as heat. It is always positive and does not include power sourced. It should be the same as P because the diode is passive.

`PS` Power sourced. The power sourced by the part. It is always positive and does not consider its own dissipation. It should be 0 because the diode is passive.

`PJ` Junction power. PJ(Dxxxx) is the same as P(Yj.Dxxxx).

`PC` Capacitor power. PC(Dxxxx) is the same as P(Cj.Dxxxx).

`Capacitance` Effective capacitance. C(Dxxxx) is the same as Capacitance(Cj.Dxxxx).

`Req` Effective resistance. R(Dxxxx) is the same as R(Yj.Dxxxx).

`Z` Impedance at a port. The port impedance seen looking into the circuit across the branch. It does not include the part itself. In transient analysis, it shows the effective Z-domain impedance, which is a meaningless number if there are capacitors or inductors in the circuit. (DC only)

`ZRAW` Impedance at a port, raw. This is the same as "Z" except that it includes the part itself. (DC only)

`REgion` Region code. A numeric code that represents the region it is operating in. +1 = forward, -1 = reversed, 0 = unknown, -2 = assumed off.

All parameters of the internal elements Yj and Cj are available. To access them, concatenate the labels for the internal element with the diode, separated by a dot. Yj.D6 is the admittance (Yj) element of the diode D6.

In this release, there are no probes available in AC analysis except for the internal elements.

The general element probes do not apply to diodes.

3.5 E: Voltage Controlled Voltage Source

3.5.1 Syntax

E*xxxxxx n+ n– nc+ nc– value*
E*xxxxxx n+ n– nc+ nc– expression*
.VCVS *label n+ n– nc+ nc– expression*

3.5.2 Purpose

Voltage controlled voltage source, or voltage gain block.

3.5.3 Comments

N+ and *n–* are the positive and negative element (output) nodes, respectively. *Nc+* and *nc–* are the positive and negative controlling nodes, respectively. *Value* is the voltage gain.

3.6 F: Current Controlled Current Source

3.6.1 Syntax

F*xxxxxx n+ n– ce value*
F*xxxxxx n+ n– ce expression*
.CCCS *label n+ n– ce expression*

3.6.2 Purpose

Current controlled current source, or current gain block.

3.6.3 Comments

N+ and *n–* are the positive and negative element (output) nodes, respectively. Current flow is from the positive node, through the source, to the negative node. *Ce* is the name of an element through which the controlling current flows. The direction of positive controlling current is from the positive node, through the element, to the negative node of *ce*. *Value* is the current gain.

The controlling element can be any simple two terminal element. Unlike SPICE, it does not need to be a voltage source.

3.7 G: Voltage Controlled Current Source

3.7.1 Syntax

G*xxxxxx n+ n– nc+ nc– value*
G*xxxxxx n+ n– nc+ nc– expression*
.VCCS *label n+ n– nc+ nc– expression*

3.7.2 Purpose

Voltage controlled current source, or transconductance block.

3.7.3 Comments

N+ and *n–* are the positive and negative element (output) nodes, respectively. Current flow is from the positive node, through the source, to the negative node. *Nc+* and *nc–* are the positive and negative controlling nodes, respectively. *Value* is the transconductance in mhos.

The letter G can also be used to select the `vccap`, `vcr`, and `vcg` devices using a syntax compatible with some other simulators.

3.8 Voltage Controlled Capacitor

3.8.1 Syntax

Preferred syntax:
`.VCCAP` *label n+ n– nc+ nc– expression*
Alternate syntax:
`G`*xxxxxx n+ n–* VCCAP *nc+ nc– expression*

3.8.2 Purpose

Voltage controlled capacitor.

3.8.3 Probes

All probes that apply to ordinary capacitors also apply here.

3.8.4 Comments

N+ and *n–* are the positive and negative element (output) nodes, respectively. *Nc+* and *nc–* are the positive and negative controlling nodes, respectively. *Value* is the transfactor in Farads per volt.

The simulator will faithfully give you a negative capacitor if it seems appropriate. Usually, this part is used with a behavioral modeling function, like PWL, which allows you to specify a table of capacitance vs. voltage.

3.9 Voltage Controlled Admittance

3.9.1 Syntax

Preferred syntax:
`.VCG` *label n+ n– nc+ nc– expression*
Alternate syntax:
`G`*xxxxxx n+ n–* VCG *nc+ nc– expression*

3.9.2 Purpose

Voltage controlled admittance.

3.9.3 Comments

N+ and *n–* are the positive and negative element (output) nodes, respectively. *Nc+* and *nc–* are the positive and negative controlling nodes, respectively. *Value* is the transfactor in mhos per volt.

The simulator will faithfully give you a negative admittance if it seems appropriate. Usually, this part is used with a behavioral modeling function, like PWL, which allows you to specify a table of admittance vs. voltage.

3.10 Voltage Controlled Resistor

3.10.1 Syntax

Preferred syntax:
`.VCR` *label n+ n– nc+ nc– expression*
Alternate syntax:
`G`*xxxxxx n+ n–* VCR *nc+ nc– expression*

3.10.2 Purpose

Voltage controlled resistor.

3.10.3 Comments

N+ and *n–* are the positive and negative element (output) nodes, respectively. *Nc+* and *nc–* are the positive and negative controlling nodes, respectively. *Value* is the transfactor in ohms per volt.

The simulator will faithfully give you a negative resistor if it seems appropriate. Usually, this part is used with a behavioral modeling function, like PWL, which allows you to specify a table of resistance vs. voltage.

3.11 `H`: Current Controlled Voltage Source

3.11.1 Syntax

`H`*xxxxxx n+ n– ce value*
`H`*xxxxxx n+ n– ce expression*
`.CCVS` *label n+ n– ce expression*

3.11.2 Purpose

Current controlled voltage source, or transresistance block.

3.11.3 Comments

N+ and *n–* are the positive and negative element (output) nodes, respectively. *Ce* is the name of an element through which the controlling current flows. The direction of positive controlling current is from the positive node, through the element, to the negative node of *ce*. *Value* is the transresistance in Ohms.

The controlling element can be any simple two terminal element. Unlike SPICE, it does not need to be a voltage source.

3.12 I: Independent Current Source

3.12.1 Syntax

I*xxxxxxx n+ n– value*
I*xxxxxxx n+ n– expression*
.ISOurce *label n+ n– expression*

3.12.2 Purpose

Independent current source.

3.12.3 Comments

N+ and *n–* are the positive and negative element nodes, respectively. Positive current flow is from the positive node, through the source, to the negative node. *Value* is the current in Amperes.

All of the SPICE time dependent functions (`pulse`, `sin`, `exp`, `pwl`, and `sffm` are supported. An additional function `generator` emulates a laboratory type function generator, for a more convenient signal input to the circuit.

3.13 J: Junction Field-Effect Transistor

3.13.1 Syntax

J*xxxxxxx nd ng ns mname* {*area*} {*args*}

3.13.2 Purpose

Junction Field Effect Transistor.

3.13.3 Comments

Not implemented. Plans are to implement it as in SPICE.

3.14 K: Coupled (Mutual) Inductors

3.14.1 Syntax

K*xxxxxxx* L*yyyyyyy* L*zzzzzzz value*
.MUTual_inductor *label* L*yyyyyyy* L*zzzzzzz value*

3.14.2 Purpose

Coupled mutual inductance.

3.14.3 Comments

K couples two inductors. The value is the coefficient of coupling. Using the dot convention, place a dot on the first node of each inductor.

The coefficient of coupling is given by $K = \frac{M_{ij}}{\sqrt{L_i L_j}}$.

3.14.4 Bugs

This release does not support multiple coupled inductors.

3.15 L: Inductor

3.15.1 Syntax

L*xxxxxx n+ n– value*
L*xxxxxx n+ n– expression*
L*xxxxxx n+ n– value* {II=*initial-current*}
.INDuctor *label n+ n– expression*

3.15.2 Purpose

Inductor, or general flux storage element.

3.15.3 Probes

The following probes are available in addition to those available for all basic elements.

DT Time step. The internal time step used for this device for numerical integration. It is not necessarily the same as the global time step.

TIME Time. The time of the most recent calculation of this device. It is not necessarily the same as the global time.

TIMEOLD The time of the previous calculation of this device. It is not necessarily the same as the global time.

3.15.4 Comments

N+ and *n–* are the positive and negative element nodes, respectively. *Value* is the inductance in Henries.

The (optional) initial condition is the initial (time = 0) value of the inductor current (in Amperes). Note that the initial conditions (if any) apply only if the UIC option is specified on the transient command.

3.16 M: MOSFET

3.16.1 Syntax

M*xxxxxx nd ng ns nb mname* {*args*}
M*xxxxxx nd ng ns nb mname* {*width*/*length*} {*args*}
.MOSfet *label nd ng ns nb mname* {*args*}
.MOSfet *label nd ng ns nb mname* {*width*/*length*} {*args*}

3.16.2 Purpose

MOSFET.

3.16.3 Comments

Nd, *ng*, *ns*, and *nb* are the drain, gate, source, and bulk (substrate) nodes, respectively. *Mname* is the model name.

Length and *width* are the drawn channel length and width, in microns. Note that the notation W/L has units of microns, but the same parameters, in the argument list (`W` and `L`) have units of meters. All other dimensions are in meters.

The options `rstray` and `norstray` determines whether or not series resistances are included. `rstray` is the default. Experience has shown that the effect of series resistance is often not significant, it can significantly degrade the simulation time, and it often increases roundoff errors. `rstray` is the default for Spice compatibility, and because it usually is significant for the BJT model. `Norstray` is the equivalent of setting the model parameters `rd`, `rs`, and `rsh` all to zero.

Entering a parameter value of 0 is not the same as not specifying it. This behavior is not compatible with SPICE. In SPICE, a value of 0 is often interpreted as not specified, with the result being to calculate it some other way. If you want it to be calculated, don't specify it.

Another subtle difference from SPICE is that Gnucap may omit some unnecessary parts of the model, which may affect some reported values. It should not affect any voltages or currents. For example, if the gate and drain are tied, Cgs will be omitted from the model, so the printed value for Cgdovl and Cgd will be 0, which will disagree with SPICE. It doesn't matter because a shorted capacitor can store no charge.

Levels 1, 2, 3, 4, 5, 6, 7 are implemented.

3.16.4 Element Parameters

Basic Spice compatible parameters

`L` = *x* Drawn channel length. (Default = `DEFL` parameter from `options. DEFL default = 100`μ)

`W` = *x* Drawn channel width. (Default = `DEFW` parameter from `options. DEFW default = 100`μ)

`AD` = *x* Area of drain diffusion. (Default = `DEFAD` parameter from `options. DEFAD default = 0`)

`AS` = *x* Area of source diffusion. (Default = `DEFAS` parameter from `options. DEFAS default = 0`)

`PD` = *x* Perimeter of drain junction. (Default = 0.)

`PS` = *x* Perimeter of source junction. (Default = 0.)

`NRD` = *x* Number of squares of drain diffusion. (Default = 1.)

`NRS` = *x* Number of squares of source diffusion. (Default = 1.)

3.16.5 Model Parameters

Basic selection – required for all models

`LEVEL` = *x* Model index. (Default = 1) Selects which of several models to use. The choices supported are 1-7, corresponding to Spice 3f5.

Extended control (not in Spice) – all models

`CMODEL` = x Capacitance model selector (Default = 1 for level 4,5,7. Default = 2 for level 1,2,3. Default = 3 for level 6.) The only valid values are 1, 2 and 3. 2 selects Meyer capacitance calculations compatible with Spice 2. 3 selects Meyer's model compatible with Spice 3. 1 selects not to use Meyer's model.

Binning (not in Spice) – all models

Gnucap supports "binning". You can specify any number of models as a family. These models must have the selection parameters `WMAX`, `WMIN`, `LMAX`, and `LMIN`.

To use "binning", define a set of models with the same name, except for a numeric extension, beginning at 1. The models must be numbered consecutively. For example, you might have a set of models: `NM3U.1`, `NM3U.2`, `NM3U.3`, `NM3U.4`, `NM3U.5`, `NM3U.6`. For the device, you would specify the model `NM3U`. The first model meeting the requirements that length is between `LMIN` and `LMAX`, and width is between `WMIN` and `WMAX` will be used. They will be tried in numerical order.

If there is a gap in the numbering, only those below the gap will be used. If you want a specific model from a set, disabling binning, you can specify its full name.

`WMAX` = x Maximum width. (Default = Infinity.) The maximum device width that may be used with this model.

`WMIN` = x Maximum width. (Default = 0.) The minimum device width that may be used with this model.

`LMAX` = x Maximum length. (Default = Infinity.) The maximum device length that may be used with this model.

`LMIN` = x Maximum length. (Default = 0.) The minimum device length that may be used with this model.

Substrate coupling – all models

`IS` = x Bulk junction saturation current. If not input, it is calculated from `JS`. If both are input, a warning is issued, and the calculated value (from `JS`) is used, if `AD` and `AS` are also input. If neither `IS` or `JS` is input, a default value of 1e-14 is used.

`JS` = x Bulk junction saturation current per square-meter of junction area. May be used to calculate `IS`. If a conflict exists, a warning is issued.

`FC` = x Coefficient for forward bias depletion capacitance formula. (Default = 0.5)

`PB` = x Bulk junction potential. (Default = 0.8)

`CJ` = x Zero bias bulk junction bottom capacitance per square-meter of junction area. If not input, but `NSUB` is, it is calculated, otherwise a default value of 0 is used.

`MJ` = x Bulk junction bottom grading coefficient. (Default = 0.5)

`PBSW` = x Sidewall Bulk junction potential. (Default = PB)

`CJSW` = x Zero bias bulk junction sidewall capacitance per meter of junction perimeter. (Default = 0.)

`MJSW` = x Bulk junction sidewall grading coefficient. (Default = 0.33)

Strays – all models

RSH = x Drain and source diffusion sheet resistance. If not input, use RS and RD directly. If a conflict exists, a warning is issued. The resistance is only used if the option rstray is set.

RD = x Drain ohmic resistance (unscaled). If RS is input, the default value of RD is 0. If RD and RS are both not input, and RSH is input, they are calculated from RSH. If any conflict exists, a warning is issued, indicating the action taken, which is believed to be compatible with SPICE. The resistance is only used if the option rstray is set.

RS = x Source ohmic resistance (unscaled). If RD is input, the default value of RS is 0. If RD and RS are both not input, and RSH is input, they are calculated from RSH. If any conflict exists, a warning is issued, indicating the action taken, which is believed to be compatible with SPICE. The resistance is only used if the option rstray is set.

CBD = x Zero bias B-D junction capacitance (unscaled). If CBD is not specified, it is calculated from CJ.

CBS = x Zero bias B-S junction capacitance (unscaled). If CBS is not specified, it is calculated from CJ.

CGSO = x Gate-source overlap capacitance, per channel width. (Default = 0.)

CGDO = x Gate-drain overlap capacitance, per channel width. (Default = 0.)

CGBO = x Gate-bulk overlap capacitance, per channel length. (Default = 0.)

Accepted and ignored – all models

KF = x Flicker noise coefficient. SPICE parameter accepted but not implemented.

AF = x Flicker noise exponent. SPICE parameter accepted but not implemented.

Level 1,2,3,6 shared parameters

VTO = x Zero bias threshold voltage. If not input, but NSUB is, it is calculated, otherwise a default value of 0 is used.

KP = x Transconductance parameter. If not input, it is calculated by UO * COX.

GAMMA = x Bulk threshold parameter. If not input, but NSUB is, it is calculated, otherwise a default value of 0 is used.

PHI = x Surface potential. If not input, but NSUB is, it is calculated, otherwise a default value of 0.6 is used. A warning is issued if the calculated value is less than 0.1, in which case 0.1 is used.

LAMBDA = x Channel length modulation. If not input, it is calculated dynamically during simulation. If the value input is larger than 0.2, a warning is issued, but no correction is made. (accepted but ignored for level 3)

TOX = x Oxide thickness. (meters) (Default = 1e-7)

NSUB = x Substrate doping. (atoms / cm^3) Used in calculation of VTO, GAMMA, PHI, and CJ. If not input, default values are used.

NSS = x Surface state density. (atoms / cm^2) (Default = 0.) Used, with NSUB in calculation of VTO.

`XJ` = x Metallurgical junction depth. (meters) Used to calculate short channel effects. If not input, do not model short channel effects, effectively defaults to 0.

`LD` = x Lateral diffusion. (Default = 0.) Effective channel length is reduced by 2 * `LD`.

`UO` = x Surface mobility. (cm^2/V-s) (Default = 600.)

`DELTA` = x Width effect on threshold voltage. (Default = 0.) (Level 2 and 3 only.)

`TPG` = x Type of gate material. (Default = 1.)

+1 opposite to substrate
–1 same as substrate
0 Aluminum

Level 1

The Level 1 model has no additional parameters.

Level 2

`NFS` = x Fast surface state density. (atoms / cm^2) Used in modeling sub-threshold effects. If not input, do not model sub-threshold effects.

`VMAX` = x Maximum drift velocity of carriers. (m/s) Used in calculating vdsat, and lambda. If not input, use a different method. `VMAX` does not always work, if the method fails, the alternate method is used and the warning "Baum's theory rejected" is issued if the error threshold is set to `debug` or worse.

`NEFF` = x Total channel charge (fixed and mobile) coefficient. (Default = 1.) Used in internal calculation of lambda.

`UCRIT` = x Critical field for mobility degradation. (V/cm) (Default = 1e4)

`UEXP` = x Critical field exponent in mobility degradation. If not input, do not model mobility degradation, effectively defaulting to 0.

`UTRA` = x Transverse field coefficient. SPICE parameter accepted but not implemented. It is also not implemented in most versions of SPICE.

Level 3

`NFS` = x Fast surface state density. (atoms / cm^2) Same as Level 2.

`VMAX` = x Maximum drift velocity of carriers. (m/s) Used in calculating vdsat. If not input, use a different method.

`THETA` = x Mobility modulation.

`ETA` = x Static feedback.

`KAPPA` = x Saturation field vector.

Level 6

`KV` = x Saturation voltage factor.

`NV` = x Saturation voltage coefficient.

`KC` = x Saturation current factor.

`NC` = x Saturation current coefficient.

`NVTH` = x Threshold voltage coefficient.

`PS` = x Sat. current modification par.

`GAMMA1` = x Bulk threshold parameter 1.

`SIGMA` = x Static feedback effect par.

`LAMBDA1` = x Channel length modulation parameter. 1.

Level 4, 5, 7, 8 (BSIM models) general comments

The BSIM models have additional parameters for length, width, and product (length * width) dependency. To get the name, prefix the listed parameter with L, W, or P, respectively. Spice supports the "P" parameter only for BSIM3, but Gnucap supports it for all 3 models. For example, VFB is the basic parameter with units of Volts, and LVFB, WVFB, and PVFB also exist. The units of LVFB and WVFB are Volts * micron. The units of PVFB are Volts * micron * micron. The real parameter is calculated by $P = P_0 + P_L/L + P_W/W + P_P/(L * W)$, where L and W are the effective length and width in microns.

The parameter s are not listed here, but they are the same as Spice 3f5, with the same defaults.

The "levels" are the same as Spice.

`4` BSIM 1.

`5` BSIM 2.

`7` BSIM 3v3.1.

The following are reserved for future use:

`8` BSIM 3v3.2.

`9` BSIM-SOI.

`10` BSIM 4.

3.16.6 Probes

`VDS` Drain-source voltage.

`VGS` Gate-source voltage.

`VBS` Bulk-source voltage.

`VDSInt` Drain-source internal voltage.

`VGSInt` Gate-source internal voltage.

`VBSInt` Bulk-source internal voltage.

`VGD` Gate-drain voltage.

`VBD` Bulk-drain voltage.

`VSD` Source-drain voltage.

`VDM` Drain-midpoint voltage.

`VGM` Gate-midpoint voltage.

`VBM` Bulk-midpoint voltage.

`VSM` Source-midpoint voltage.

`VDG` Drain-gate voltage.

`VBG` Bulk-gate voltage.

`VSG` Source-gate voltage.

`VDB` Drain-bulk voltage.

`VGB` Gate-bulk voltage.

`VSB` Source-bulk voltage.

`VD` Drain-ground voltage.

`VG` Gate-ground voltage.

`VB` Bulk-ground voltage.

`VS` Source-ground voltage.

`Id` Drain current.

`IS` Source current.

`IG` Gate current.

`IB` Bulk current.

`CGSO` Gate-source overlap capacitance.

`CGDO` Gate-drain overlap capacitance.

`CGBO` Gate-bulk overlap capacitance.

`CGSm` Gate-source Meyer capacitance.

`CGDm` Gate-drain Meyer capacitance.

`CGBm` Gate-bulk Meyer capacitance.

`CGST` Gate-source total capacitance.

`CGDT` Gate-drain total capacitance.

`CGBT` Gate-bulk total capacitance.

`CBD` Bulk-drain junction capacitance.

`CBS` Bulk-source junction capacitance.

`CGATE` Nominal gate capacitance.

`GM` Transconductance.

`GDS` Drain-source conductance.

`GMB` Body effect transconductance.

`VDSAT` Saturation voltage.

`VTH` Threshold voltage.

`IDS` Drain-source current, not including strays.

`IDSTray` Drain current due to strays.

`IError` Estimated drain current error bound.

`P` Power.

`PD` Power dissipated. The power dissipated as heat. It is always positive and does not include power sourced. It should be the same as P because the mosfet cannot generate energy.

`PS` Power sourced. The power sourced by the part. It is always positive and does not consider its own dissipation. It should be 0 because the mosfet cannot generate energy.

`REgion` Region code. A numeric code that represents the region it is operating in. The number is the sum of several factors. A negative code indicates the source and drain are reversed.

1 Active. (Not cut off.)

2 Not sub-threshold.

4 Saturated.

10 Source to bulk is forward biased.

20 Drain to bulk is forward biased.

40 Punch through.

All parameters of the internal elements (Ids, Gmr, Gmf, Yds, Gmbr, Gmbf, Cgb, Cgd, Cgs, Dsb, Ddb, Rd, Rs) are available. To access them, concatenate the labels for the internal element with this device, separated by a dot. Cgd.M6 is the gate to drain capacitance of M6.

In this release, there are no probes available in AC analysis except for the internal elements.

3.17 Q: Bipolar Junction Transistor

3.17.1 Syntax

Q*xxxxxx nc nb ne ns mname* {*area*} {*args*}
.BJT *label nc nb ne ns mname* {*area*} {*args*}

3.17.2 Purpose

Bipolar junction transistor,

3.17.3 Comments

Nc, *nb*, *ne*, and *ns* are the collector, base, emitter, and substrate nodes, respectively. *Mname* is the model name.

Area is a unit-less multiplier for the area.

The options `rstray` and `norstray` determines whether or not series resistances are included. `rstray` is the default. `Norstray` is the equivalent of setting the model parameters `rc`, `re`, and `rb` all to zero.

Entering a parameter value of 0 is not the same as not specifying it. This behavior is not compatible with SPICE. In SPICE, a value of 0 is often interpreted as not specified, with the result being to calculate it some other way. If you want it to be calculated, don't specify it.

Another subtle difference from SPICE is that Gnucap may omit some unnecessary parts of the model, which may affect some reported values. It should not affect any voltages or currents. For example, if the gate and drain are tied, Cgs will be omitted from the model, so the printed value for Cgdovl and Cgd will be 0, which will disagree with SPICE. It doesn't matter because a shorted capacitor can store no charge.

3.17.4 Element Parameters

Basic Spice compatible parameters

`Area` = x Junction area. (Default = `1`) This is a scaling parameter, with no relevant actual units.

`OFF` (Default = not specified) If this word is specified, the initial guess will assume the device is off.

`TEMP` = x Junction temperature. (Default = the global temperature.)

`ICVBE` = x Initial condition, Vbe. (Default = NA) Use this as the initial condition, when the `UIC` option is specified. The syntax is different from Spice, but the function is the same.

`ICVCE` = x Initial condition, Vce. (Default = NA) Use this as the initial condition, when the `UIC` option is specified. The syntax is different from Spice, but the function is the same.

3.17.5 Model Parameters

Basic DC parameters

`BF` = x Ideal maximum forward beta. (Default = 100) Alternate name is `BFM`.

`BR` = x Ideal maximum reverse beta. (Default = 1) Alternate name is `BRM`.

`IBC` = x BC Transport saturation Current per area. (Default = IS) If omitted, `IS` is used. You should specify either `IS` or `IBC`, not both.

IBE = x BE Transport saturation Current per area. (Default = IS) If omitted, IS is used. You should specify either IS or IBE, not both.

IS = x Transport saturation Current per area. (Default = 1e-16) If IBE and IBC are specified, they are used instead. Do not specify both.

NF = x Forward current emission coefficient. (Default = 1)

NR = x Reverse current emission coefficient. (Default = 1)

Base width modulation

VAF = x Forward Early voltage. (Default = Infinite) Alternate names are VA and VBF.

VAR = x Reverse Early voltage. (Default = Infinite) Alternate name is VB.

Low current beta degeneration

ISC = x B-C leakage saturation current. (Default = c4 * is)

C4 = x B-C leakage scale factor. (Default = 0) Alternate name is JLC.

NC = x B-C leakage emission coefficient. (Default = 2)

ISE = x B-E leakage saturation current. (Default = c2 * is)

C2 = x B-E leakage scale factor. (Default = 0) Alternate name is JLE.

NE = x B-E leakage emission coefficient. (Default = 1.5)

High current beta degeneration

IKF = x Forward beta roll-off corner current. (Default = Infinite) Alternate names are JBF and IK.

IKR = x Reverse beta roll-off corner current. (Default = Infinite) Alternate name is JBR.

Parasitic resistance

IRB = x Current for base resistance=(rb+rbm)/2". (Default = Infinite) Current where base resistance falls halfway to its minimum value. Alternate name is JRB.

RB = x Zero bias base resistance. (Default = 0)

RBM = x Minimum base resistance at high current. (Default = rb)

RE = x Emitter resistance. (Default = 0)

RC = x Collector resistance. (Default = 0)

Junction capacitance

CJC = x Zero bias B-C depletion capacitance. (Default = 0)

CJE = x Zero bias B-E depletion capacitance. (Default = 0)

CJS = x Zero bias C-S capacitance. (Default = 0) Alternate name is CCS.

FC = x Coefficient for forward-bias depletion capacitance formula. (Default = .5)

MJC = x B-C junction grading coefficient. (Default = .33) Alternate names are MJ and MC.

MJE = x B-E junction grading coefficient. (Default = .33) Alternate name is ME.

MJS = x Substrate junction grading coefficient. (Default = 0) Alternate names are MS and MSUB.

VJC = x B-C built in potential. (Default = .75) Alternate name is PC.

VJE = x B-E built in potential. (Default = .75) Alternate name is PE.

VJS = x Substrate junction built in potential. (Default = .75) Alternate name is PS.

XCJC = x Fraction of B-C capacitance connected to internal base node. (Default = 1)

Parasitic capacitance

CBCP = x External B-C constant parasitic capacitance. (Default = 0)

CBEP = x External B-E constant parasitic capacitance. (Default = 0)

CBSP = x External B-S constant parasitic capacitance for lateral transistors. (Default = 0)

CCSP = x External B-C constant parasitic capacitance for vertical transistors. (Default = 0)

Transit time

ITF = x High current dependence of TF. (Default = 0)

PTF = x Excess phase at freq=1.0/(TF*2PI) Hz. (Default = 0)

TF = x Ideal forward transit time. (Default = 0)

TR = x Ideal reverse transit time. (Default = 0)

VTF = x Voltage giving VBC dependence of TF. (Default = Infinite)

XTF = x Coefficient for bias dependence of TF. (Default = 0)

Temperature effects

XTB = x Forward and reverse beta temperature exponent. (Default = 0)

XTI = x Temperature exponent for effect on IS. (Default = 3)

EG = x Energy gap for IS temperature dependency. (Default = 1.11)

TNOM = x Parameter measurement temperature, Celsius. (Default = 27)

3.17.6 Probes

`VBEInt` Base-emitter internal voltage.

`VBCInt` Base-collector internal voltage.

`VBXInt` External base to internal base voltage.

`VCSInt` Collector-substrate internal voltage.

`VBS` Base-substrate voltage.

`VBE` Base-emitter voltage.

`VBC` Base-collector voltage.

`VCS` Collector-substrate voltage.

`VCB` Collector-base voltage.

`VCE` Collector-emitter voltage.

`VES` Emitter-substrate voltage.

`VEB` Emitter-base voltage.

`VEC` Emitter-collector voltage.

`VB` Base-ground voltage.

`VC` Collector-ground voltage.

`VE` Emitter-ground voltage.

`VS` Substrate-ground voltage.

`VBI` Internal Base-ground voltage.

`VCI` Internal Collector-ground voltage.

`VEI` Internal Emitter-ground voltage.

`ICE` Collector-emitter current.

`ICEOffset` Offset part of ICE.

`GO` Output (collector-emitter) conductance.

`GM` Transconductance.

`IPI` Base-emitter current.

`IPIOffset` Offset part of IPI.

`GPI` Base-emitter conductance.

`IMU` Base-collector current.

`IMUOffset` Offset part of IMU.

`GMU` Base-collector conductance.

`IB` Base current.

`GX` Conductance of base spreading resistance.

`RX` Base spreading resistance.

`IC` Collector current.

`IE` Emitter current.

`QBX` External Base-collector charge.

`CQBX` External Base-collector capacitance.

`CBX` External Base-collector capacitance (CQBX).

`QBC` Internal Base-collector charge.

`CQBC` Internal Base-collector capacitance.

`CBC` Internal Base-collector capacitance (CQBC).

`CMU` Internal Base-collector capacitance (CQBC).

`QCS` Collector-substrate charge.

`CQCS` Collector-substrate capacitance.

`CCS` Collector-substrate capacitance (CQCS).

`QBE` Base-emitter charge.

`CQBE` Base-emitter capacitance.

`CBE` Base-emitter capacitance. (CQBE).

`CPI` Base-emitter capacitance. (CQBE).

`P` Power.

`PD` Power dissipated. The power dissipated as heat. It is always positive and does not include power sourced. It should be the same as P because transistors cannot generate energy.

`PS` Power sourced. The power sourced by the part. It is always positive and does not consider its own dissipation. It should be 0 because transistors cannot generate energy.

All parameters of the internal elements (Ice, Ipi, Imu, Rc, Re, Yb, Cbx, Cbc, Ccs, Cbe, Cbcp, Cbep, Cbsp, Ccsp) are available. To access them, concatenate the labels for the internal element with this device, separated by a dot. Cbe.Q6 is the base to emitter capacitance of Q6.

In this release, there are no probes available in AC analysis except for the internal elements.

3.18 R: Resistor

3.18.1 Syntax

R*xxxxxxx n+ n– value*
R*xxxxxxx n+ n– expression*
R*xxxxxxx n+ n– model* {L=*length*} {W=*width*} {TEMP=*temperature*}
.RESistor *label n+ n– expression*

3.18.2 Purpose

Resistor, or general current controlled dissipative element.

3.18.3 Comments

N+ and *n–* are the positive and negative element nodes, respectively. *Value* is the resistance in Ohms.

The resistor (type R) differs from the admittance (type Y) in that the resistor is a current controlled element, and the conductance is a voltage controlled element, in addition to the obvious use of conductance ($1/R$) instead of resistance.

You may specify the *value* in one of three forms:

1. A simple value. This is the resistance in Ohms.
2. An expression, as described in the behavioral modeling chapter. The expression can specify the voltage as a function of current, or the resistance as a function of time.
3. A *model*, which calculates the resistance as a function of length and width, referencing a .model statement of type R. This is compatible with the Spice-3 "semiconductor resistor".

3.18.4 Model statement

A model statement may be used,, with model type R or Res. The parameters are:

RSH = x Sheet resistance. (Ohms / square). (Required)

CJSW = x Junction sidewall capacitance. (Farads / meter). (Default = 0.)

DEFW = x Default width. (meters). (Default = 1e-6)

NARROW = x Narrowing due to side etching. (meters). (Default = 0.)

TC1 = x First order temperature coefficient. (Farads / degree C). (Default = 0.)

TC2 = x Second order temperature coefficient. (Farads / degree C squared). (Default = 0.)

TNOM = x Parameter measurement temperature. (degrees C.). (Default = 27.)

Resistance is computed by the formula:

```
resistance = RSH * (L - NARROW) / (W - NARROW)
```

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

```
value *= (1 + TC1 * (T-T0) + TC2 * (T-T0)^2)
```

3.19 S: Voltage Controlled Switch

3.19.1 Syntax

S*xxxxxxx n+ n– nc+ nc– mname* {*ic*}
.VSWitch *label n+ n– nc+ nc– mname* {*ic*}

3.19.2 Purpose

Voltage controlled switch.

3.19.3 Comments

N+ and *n–* are the positive and negative element nodes, respectively. *Nc+* and *nc–* are the controlling nodes. *Mname* is the model name. A switch is a resistor between *n+* and *n–*. The value of the resistor is determined by the state of the switch.

The resistance between *n+* and *n–* will be *RON* when the controlling voltage (between *nc+* and *nc–*) is above *VT + VH*. The resistance will be *ROFF* when the controlling voltage is below *VT - VH*. When the controlling voltage is between *VT - VH* and *VT + VH*, the resistance will retain its prior value.

You may specify ON or OFF to indicate the initial state of the switch when the controlling voltage is in the hysteresis region.

RON and ROFF must have finite positive values.

3.19.4 Model Parameters

VT = *x* Threshold voltage. (Default = 0.)

VH = *x* Hysteresis voltage. (Default = 0.)

RON = *x* On resistance. (Default = 1.)

ROFF = *x* Off resistance. (Default = 1e12)

3.20 T: Transmission Line

3.20.1 Syntax

T*xxxxxxx n1+ n1– n2+ n2–* {*args*}
.TLIne *xxxxxxx n1+ n1– n2+ n2–* {*args*}

3.20.2 Purpose

Lossless transmission line.

3.20.3 Comments

N1+ and *n1–* are the nodes at one end. *N2+* and *n2–* are the nodes at the other end.

The parameters TD, Freq, and NL determine the length of the line. Either TD or Freq and NL must be specified. If only Freq is specified, NL is assumed to be .25. The other will be calculated based on the one you specify. If you specify too much, Freq and NL dominate, and a warning is issued.

3.20.4 Element Parameters

`Z0` = x Characteristic impedance. (Default = 50.)

`TD` = x Time delay.

`Freq` = x Frequency for NL.

`NL` = x Number of wavelengths at *Freq*.

3.21 U: Logic Device

3.21.1 Syntax

U*xxxxxx out gnd vdd enable in1 in2 ... family gatetype*

3.21.2 Purpose

Logic element for mixed or logic mode simulation.

3.21.3 Comments

A sample 2 input nand gate might be: `U102 5 0 34 34 2 3 cmos nand`. The input pins are connected to nodes 2 and 3. The output is at node 5. Node 34 is the power supply.

The logic element behaves differently depending on the options `analog`, `mixed`, or `digital`. You set one of these with the `options` command. `Analog` mode substitutes a subcircuit for the gate for full analog simulation. `Digital` mode simulates the gate as a digital device as in an event driven gate level logic simulator. `Mixed` mode applies heuristics to decide whether to use analog or digital for each gate.

In `analog` mode the logic (`U`) device is almost the same as a subcircuit (`X`). The subcircuit is user defined for each gate type used. A `.subckt` defines the analog equivalent of a logic element. The name of the subcircuit is made by concatenating the *family*, *gatetype*, and the number of inputs. For example, if the *family* is `cmos` and the *gatetype* is `nand` and it has two inputs, the name of the subcircuit is `cmosnand2`. So, the gate in the first paragraph becomes equivalent to: `X 5 0 34 34 2 3 cmosnand2`. You then need to define the subcircuit using the standard `.subckt` notation. You can probe the internal elements the same as an ordinary subcircuit.

The `digital` mode uses simple boolean expressions to compute the output, just like a gate level logic simulator. In this case the output is computed by L(5) = not(L(2) and L(3)) where L(2) is the logic state at node 2. The simulator exploits latency so it will only compute the output if one of the inputs changes. The output actually changes after a delay, specified in the `.model` statement. There are no conversions between digital and analog where gates connect together. There will be an automatic conversion from analog to digital for any input that is driven by an analog device. There will be an automatic conversion from digital to analog for any output that drives an analog device. These conversions will only be done if they are needed. You can probe the analog value at any node. The probe will automatically request the conversion if it needs it. There is no internal subcircuit so it is an error to probe the internal elements.

The `mixed` mode is a combination of analog and digital modes on a gate by gate basis. Some gates will be analog. Some will be digital. This will change as the simulation runs based on the quality of the signals. You need to specify a `.subckt` as you do for the analog mode, but the simulator may not use it. You can usually not probe the elements inside the subcircuit because they come and go.

3.21.4 Element Parameters

Family refers to the logic family `.model` statement.

Gatetype is the type of logic gate:

`AND`

`NAND`

`OR`

`NOR`

`XOR`

`INV`

3.21.5 Model Parameters

Parameters used in digital mode

`DElay` = x Propagation delay. (Seconds) (Default = 1e-9) The propagation delay of a simple gate when simulated in logic mode.

Parameters used in conversion both ways

`VMAx` = x Nominal logic 1. (Volts) (Default = 5.) The nominal value for a logic 1.

`VMIn` = x Nominal logic 0. (Volts) (Default = 0.) The nominal value for a logic 0.

`Unknown` = x Nominal logic unknown. (Volts) (Default = (vmax+vmin)/2) The output voltage for a logic unknown. In a real circuit, this voltage is unknown, but a simulator needs something, so here it is.

Digital to Analog conversion

`RIse` = x Rise time. (Seconds) (Default = delay / 2) The nominal rise time of a logic signal. This will be the rise time when a logic signal is converted to analog.

`FAll` = x Fall time. (Seconds) (Default = delay / 2) The nominal fall time of a logic signal. This will be the fall time when a logic signal is converted to analog.

`RS` = x Series resistance, strong. (Ohms) (Default = 100.) The resistance in series with the output when a logic gate drives analog circuitry.

`RW` = x Series resistance, weak. (Ohms) (Default = 1e9) The output resistance in a high impedance state.

Analog to Digital conversion

`THH` = x Threshold high. (Unitless) (Default = .75) The threshold for the input to cross from transition to high expressed as a fraction of the difference between high and low values. (Low = 0. High = 1.)

`THL` = x Threshold low. (Unitless) (Default = .25) The threshold for the input to cross from transition to low expressed as a fraction of the difference between high and low values. (Low = 0. High = 1.)

Mode decision parameters

`MR` = x Margin rising. (Unitless) (Default = 5) How much worse than nominal the analog input rise time can be and still be accepted as clean enough for logic simulation.

`MF` = x Margin falling. (Unitless) (Default = 5) How much worse than nominal the analog input fall time can be and still be accepted as clean enough for logic simulation.

`OVer` = x Overshoot limit. (Unitless) (Default = .1) How much overshoot can a signal have and still be accepted as clean enough for logic simulation, expressed as a fraction of the difference between high and low values. (Low = 0. High = 1.)

3.21.6 Probes

`V` Output voltage.

In this release, there are no probes available in AC analysis except for the internal elements. Internal elements in the analog model are available, but they come and go so they may be unreliable. More parameters will be added.

You can probe the logic value at any node. See the `print` command for details.

3.22 `V`: Independent Voltage Source

3.22.1 Syntax

`V`*xxxxxx n+ n– value*
`V`*xxxxxx n+ n– expression*
`.VSOurce` *label n+ n– expression*

3.22.2 Purpose

Independent voltage source.

3.22.3 Comments

N+ and *n–* are the positive and negative element nodes, respectively. *Value* is the voltage in Volts.

All of the SPICE time dependent functions (`pulse`, `sin`, `exp`, `pwl`, and `sffm` are supported. An additional function `generator` emulates a laboratory type function generator, for a more convenient signal input to the circuit.

3.23 `W`: Current Controlled Switch

3.23.1 Syntax

`W`*xxxxxx n+ n– ce mname* {*ic*}
`.ISWitch` *label n+ n– ce mname* {*ic*}

3.23.2 Purpose

Current controlled switch.

3.23.3 Comments

N+ and *n–* are the positive and negative element nodes, respectively. *Ce* is the name of an element through which the controlling current flows. *Mname* is the model name. A switch is a resistor between *n+* and *n–*. The value of the resistor is determined by the state of the switch.

The resistance between *n+* and *n–* will be *RON* when the controlling current (through *ce*) is above *IT* + *IH*. The resistance will be *ROFF* when the controlling current is below *IT* - *IH*. When the controlling current is between *IT* - *IH* and *IT* + *IH*, the resistance will retain its prior value.

You may specify `ON` or `OFF` to indicate the initial state of the switch when the controlling current is in the hysteresis region.

`RON` and `ROFF` must have finite positive values.

The controlling element can be any simple two terminal element. Unlike SPICE, it does not need to be a voltage source.

3.23.4 Model Parameters

`IT` = *x* Threshold current. (Default = 0.)

`IH` = *x* Hysteresis current. (Default = 0.)

`RON` = *x* On resistance. (Default = 1.)

`ROFF` = *x* Off resistance. (Default = 1e12)

3.24 X: Subcircuit Call

3.24.1 Syntax

`X`*xxxxxx n1 {n2 n3 ...} subname*

3.24.2 Purpose

Subcircuit call

3.24.3 Comments

Subcircuits are used by specifying pseudo-elements beginning with `X`, followed by the connection nodes.

3.24.4 Probes

`V`*x* Port (terminal node) voltage. *x* is which port to probe. 1 is the first node in the "X" statement, 2 is the second, and so on.

`P` Power. The sum of the power probes for all the internal elements.

`PD` Power dissipated. The total power dissipated as heat.

`PS` Power sourced. The total power generated.

In this release, there are no probes available in AC analysis except for the internal elements. More parameters will be added. Internal elements can be probed by concatenating the internal part label with the subcircuit label. R5.X7 is R5 inside X7.

3.25 Y: Admittance

3.25.1 Syntax

Y*xxxxxx n+ n– value*
Y*xxxxxx n+ n– expression*
.ADMittance *label n+ n– expression*

3.25.2 Purpose

Admittance, or general voltage controlled dissipative element.

3.25.3 Comments

N+ and *n–* are the positive and negative element nodes, respectively. *Value* is the admittance in Mhos.

The resistor (type R) differs from the admittance (type Y) in that the resistor is a current controlled element, and the conductance is a voltage controlled element, in addition to the obvious use of conductance $(1/R)$ instead of resistance.

Chapter 4

Behavioral modeling

Gnucap behavioral modeling is in a state of transition, so this is subject to change in a future release.

Basically, all simple components can have a behavioral description, with syntax designed as an extension of the Spice time dependent sources. They are not necessarily physically realizeable. Some only work on particular types of analysis, or over a small range of values. Some can be used together, some cannot.

In general, all simple components are considered to have simple transformations. A function returns one parameter as a function of one other, as an extension of their linear behavior.

Linear behavior:

`Capacitor` $q = Cv$

`Inductor` $\phi = Li$

`Resistor` $v = Ir$

`Admittance` $i = Yv$

`VCVS` $v_o = Ev_i$

`VCCS` $i_o = Gv_i$

`CCVS` $v_o = Ei_i$

`CCCS` $i_o = Gi_i$

Sources are defined as functions of time:

`Voltage source` $v = f(t)$

`Current source` $i = f(t)$

For behavioral modeling / nonlinear values, replace the constant times input by an arbitrary function:

`Capacitor` $q = f(v)$

`Inductor` $\phi = f(i)$

`Resistor` $v = f(r)$

`Admittance` $i = f(v)$

`VCVS` $v_o = f(v_i)$

`VCCS` $i_o = f(v_i)$

`CCVS` $v_o = f(i_i)$

`CCCS` $i_o = f(i_i)$

Conditionals

`AC` AC analysis only.

`DC` DC (steady state) value.

`OP` OP analysis.

`TRAN` Transient analysis.

`FOUR` Fourier analysis only.

`ELSE` Anything not listed.

`ALL` All modes.

Functions

`COMPLEX` Complex (re, im) value.

`EXP` Spice Exp source. (time dependent value).

`FIT` Fit a curve with splines.

`GENERATOR` Value from Generator command.

`POLY` Polynomial (Spice style).

`POSY` Posynomial (Like poly, non-integer powers).

`PULSE` Spice Pulse source. (time dependent value).

`PWL` Piece-wise linear.

`SFFM` Spice Frequency Modulation (time dependent value).

`SIN` Spice Sin source. (time dependent value).

`TANH` Hyperbolic tangent xfer function.

Model Functions

`TABLE` Fit a curve with splines.

`Cap` Spice semiconductor “capacitor” model.

`Res` Spice semiconductor “resistor” model.

4.1 Conditionals

Gnucap behavioral modeling conditionals are an extension of the "AC" and "DC" Spice source parameters. The extensions ...

1. There are more choices, including an "else".
2. They apply to all elements (primitive components).
3. Each section can contain functions and options.

The following are available:

`AC` AC analysis only.

`DC` DC (steady state) value.

`OP` OP analysis.

`TRAN` Transient analysis.

`FOUR` Fourier analysis only.

`ELSE` Anything not listed.

`ALL` Anything not listed.

A value or function with no conditional keyword is equivalent to `ALL`. For SPICE compatibility, use only `DC`, `AC`, or nothing.

They are interpreted like a "switch" statement. In case of a conflict, the last one applies. A set of precedence rules applies when some keys are missing. It is SPICE compatible, to the extent the features overlap.

OP analysis OP, DC, ALL, TRAN, 0

DC analysis DC, ALL, OP, TRAN, 0

Transient analysis TRAN, ALL, DC, OP, 0

Fourier analysis FOUR, TRAN, ALL, DC, OP, 0

AC analysis, fixed sources AC, 0

AC analysis, other elements AC, ALL, 0

4.1.1 Examples

`V12 1 0 AC 1 DC 3` This voltage source has a value of 1 for AC analysis, 3 for DC. OP, Transient, and Fourier inherit the DC value.

`R44 2 3 OP 1 ELSE 1g` This resistor has a value of 1 ohm for the "OP" analysis, 1 gig-ohm for anything else. This might be useful as the feedback resistor on an op-amp. Set it to 1 ohm to set the operating point, then 1 gig to measure its open loop characteristics, hiding the fact that the op-amp would probably saturate if it was really left open loop.

4.2 Functions

Gnucap behavioral modeling functions are an extension of the Spice source time dependent values.

4.2.1 The extensions

They apply to all elements (primitive components).

All accept either Spice compatible order dependent parameters, or easier keyword=value notation.

The syntax is identical for all supported components.

4.2.2 Fixed sources

Time dependent functions are voltage or current as a function of time. They are mostly Spice compatible, with extensions.

Nonlinear transfer functions use time as the independent variable. Some may not make sense, but they are there anyway.

4.2.3 Capacitors and inductors

Time dependent functions are capacitance or inductance as a function of time. They are voltage/current conserving, not charge/flux conserving.

Nonlinear transfer functions are charge or flux as a function of input (voltage or current). Charge and flux are conserved, and can be probed.

4.2.4 Resistors and conductances

Time dependent functions are resistance or conductance as a function of time.

Nonlinear transfer functions are current or voltage as a function of input (voltage or current). Resistors define voltage as a function of current. Conductances define current as a function of voltage.

4.2.5 Controlled sources

Time dependent functions are gain (v/v, transconductance, etc) function of time.

Nonlinear transfer functions are output (voltage or current) as a function of input (voltage or current).

4.2.6 Available functions

`COMPLEX` Complex (re, im) value.

`EXP` Spice Exp source. (time dependent value).

`FIT` Fit a curve with splines.

`GENERATOR` Value from Generator command.

`POLY` Polynomial (Spice style).

`POSY` Posynomial (Like poly, non-integer powers).

`PULSE` Spice Pulse source. (time dependent value).

`PWL` Piece-wise linear.

`SFFM` Spice Frequency Modulation (time dependent value).

`SIN` Spice Sin source. (time dependent value).

`TANH` Hyperbolic tangent transfer function.

In addition, you may name a "function" defined by a `.model` statement. The following `.model` types may be used here:

`TABLE` Fit a curve with splines.

`Cap` Spice semiconductor "capacitor" model.

`Res` Spice semiconductor "resistor" model.

4.2.7 Parameters that apply to all functions

These parameters are available with all functions. Some may not make sense in some cases, but they are available anyway.

`Bandwidth` = x AC analysis bandwidth. (Default = infinity.) The transfer function is frequency dependent, with a 3 DB point at this frequency. There is frequency dependent phase shift ranging from 0 degrees at low frequencies to 90 degrees at high frequencies. The phase shift is 45 degrees at the specified frequency. AC ANALYSIS ONLY.

`Delay` = x AC analysis delay. (Default = 0.) The signal is delayed by x seconds, effectively by a frequency dependent phase shift. AC ANALYSIS ONLY.

`Phase` = x AC analysis phase. (Default = 0.) A fixed phase shift is applied. This is primarily intended for sources, but applies to all elements. AC ANALYSIS ONLY.

`IOffset` = x Input offset. (Default = 0.) A DC offset is added to the "input" of the element, before evaluating the function.

`OOffset` = x Output offset. (Default = 0.) A DC offset is added to the "output" of the element, after evaluating the function.

`Scale` = x Transfer function scale factor. (Default = 1.) The transfer function is multiplied by a constant.

`TNOM` = x Nominal temperature. (Default = .option TNOM) The nominal values apply at this temperature.

`TC1` = x First order temperature coefficient. (Default = 0.)

`TC2` = x Second order temperature coefficient. (Default = 0.)

`IC` = x Initial condition. An initial value, to force at time=0. The actual parameter applied depends on the component. (Capacitor voltage, inductor current. All others ignore it.) You must use the "UIC" option for it to be used.

Temperature adjustments and scaling use the following formula:

```
value *= _scale * (1 + _tc1*tempdiff
             + _tc2*tempdiff*tempdiff)
```

where `tempdiff` is `t - _tnom`.

4.3 COMPLEX: Complex value

4.3.1 Syntax

COMPLEX *realpart imaginarypart options*

4.3.2 Purpose

Complex component value, using a real and imaginary part. AC only.

4.3.3 Comments

Strictly, this adds no functionality over the polar option on any function, except notational convenience.

4.3.4 Example

`V12 2 0 complex 1,2` A voltage source with a value of 1 + j2 volts.

4.4 EXP: Exponential time dependent value

4.4.1 Syntax

EXP *args*
EXP *iv pv td1 tau1 td2 tau2 period*

4.4.2 Purpose

The component value is an exponential function of time.

4.4.3 Comments

For voltage and current sources, this is the same as the Spice EXP function, with some extensions.

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

```
ev = _iv;
for (reltime=time; reltime>=0; reltime-=_period){
  if (reltime > _td1){
    ev += (_pv - _iv)
          * (1. - Exp(-(reltime-_td1)/_tau1));
  }
  if (reltime > _td2){
    ev += (_iv - _pv)
          * (1. - Exp(-(reltime-_td2)/_tau2));
  }
}
```

4.4.4 Parameters

`IV` = x Initial value. (required)

`PV` = x Pulsed value. (required)

`TD1` = x Rise time delay. (Default = 0.)

`TAU1` = x Rise time constant. (Default = 0.)

`TD2` = x Fall time delay. (Default = 0.)

`TAU2` = x Fall time constant. (Default = 0.)

`Period` = x Repeat period. (Default = infinity.)

4.5 FIT: Fit a curve

4.5.1 Syntax

`FIT` *x1,y1 x2,y2 ... args*

4.5.2 Purpose

Fits a set of data using piecewise polynomials, or splines.

4.5.3 Comments

This function fits a set of piecewise polynomials to a set of data.

For capacitors, this function defines *charge* as a function of voltage. For inductors, it defines *flux* as a function of current.

For fixed sources, it defines voltage or current as a function of time.

The values of x must be in increasing order.

If *order* is 1, it is the same as `PWL`. If *order* is 3, it will use cubic splines. The result and its first two derivatives are continuous.

Outside the specified range, it uses linear extrapolation. The behavior depends on the parameters *below* and *above*. The value of *below* or *above* is the derivative to use, which is a resistance for resistors, voltage gain for a VCVS, and so on. If it is not specified, the value is automatically determined.

The properties are determined by the value of *order*.

Order = 3 (cubic splines)

The default is to use "natural" splines, which sets the second derivative to zero at the boundary. If a value of *below* or *above* is specified, "clamped" splines will be used. In any case, there will be a smooth transition at the boundaries. When using "clamped" splines, the second derivative may have a discontinuity at the boundaries

Order = 2 (quadratic splines)

By default, the derivative at the upper end is determined by the slope of the last segment. This is also the derivative above the range. Below the range, the derivative determined at the lower bound is used. It is recommended that only one of *below* and *above* be specified. If both are specified, the splines are determined using *above*, and there will be a discontinuity in the derivative at the lower bound.

Order = 1 (piecewise linear interploation)

For first order (linear) interpolation, the default slope outside the range is the extension of the slope in the end segments. The parameters *below* and *above* have no effect inside the range.

Order = 0 (piecewise constant interploation)

The resulting value is constant over the interval, and has discontinuities at the specified points. The parameters *below* and *above* are ignored. The slope is always 0.

4.5.4 Parameters

`Order` = x The order of the polynomial to fit, within the supplied data. (Default = 3) Legal values are 0, 1, 2, and 3, only.

`Below` = x The value of the derivative to use below or before the specified range.

`Above` = x The value of the derivative to use above or after the specified range.

4.5.5 Example

`C1 2 0 fit -5,-5u 0,0 1,1u 4,2u 5,2u order=1` This "capacitor" stores 5 microcoulombs at -5 volts (negative, corresponding to the negative voltage, as expected). The charge varies linearly to 0 at 0 volts, acting like a 1 microfarad capacitor. (C = dq/dv). This continues to 1 volt. The 0,0 point could have been left out. The charge increases only to 2 microcoulombs at 4 volts, for an incremental capacitance of 1u/3 or .3333 microfarads. The same charge at 5 volts indicates that it saturates at 2 microcoulombs. For negative voltages, the slope continues.

4.6 `GENERATOR`: Signal Generator time dependent value

4.6.1 Syntax

`GENERATOR` *scale*

4.6.2 Purpose

The component "value" is dependent on a "signal generator", manipulated by the "generator" command.

4.6.3 Comments

For transient analysis, the "value" is determined by a signal generator, which is considered to be external to the circuit and part of the test bench. See the "generator" command for more information.

For AC analysis, the value here is the amplitude.

Strictly, all of the functionality and more is available through the Spice-like behavioral modeling functions, but this one provides a user interface closer to the function generator that an analog designer would use on a real bench. It is mainly used for interactive operation.

It also provides backward compatibility with predecessors to Gnucap, which used a different netlist format.

4.7 POLY: Polynomial nonlinear transfer function

4.7.1 Syntax

POLY *c0 c1 c2 c3 ...*
POLY *c0 c1 c2 c3 ... args*

4.7.2 Purpose

Defines a transfer function by a one dimensional polynomial.

4.7.3 Comments

For capacitors, this function defines *charge* as a function of voltage. For inductors, it defines *flux* as a function of current. If you have the coefficients defining capacitance or inductance, prepending a "0" to the list will turn it into the correct form for Gnucap.

For fixed sources, it defines voltage or current as a polynomial function of time.

The transfer function is defined by:

```
out = c0 + (c1*in) + (c2*in^2) + ....
```

4.7.4 Parameters

MIN = *x* Minimum output value (clipping). (Default = -infinity.)

MAX = *x* Maximum output value (clipping). (Default = infinity)

ABS Absolute value, truth value. (Default = false). If set to true, the result will be always positive.

4.8 POSY: Polynomial with non-integer powers

4.8.1 Syntax

POSY *c1,p1 c2,p2 ...*
POSY *c1,p1 c2,p2 ... args*

4.8.2 Purpose

Defines a transfer function by a one dimensional "posynomial", like a polynomial, except that the powers are arbitrary, and usually non-integer.

4.8.3 Comments

There is no corresponding capability in any SPICE that I know of.

For capacitors, this function defines *charge* as a function of voltage. For inductors, it defines *flux* as a function of current.

For fixed sources, it defines voltage or current as a function of time.

Normal use of this function required positive input (voltage or current). The result is zero if the input is negative. Raising a negative number to a non-integer power would produce a complex result, which implies a non-causal result, which cannot be represented in a traditional transient analysis.

The transfer function is defined by:

```
if (in >= 0){
  out = (c1*in^p1) + (c2*in^p2) + ....
}else{
  out = 0.
}
```

4.8.4 Parameters

`MIN` = x Minimum output value (clipping). (Default = -infinity.)

`MAX` = x Maximum output value (clipping). (Default = infinity)

`ABS` Absolute value, truth value. (Default = false). If set to true, the result will be always positive.

`ODD` Make odd function, truth value. (Default = false). If set to true, negative values of x will be evaluated as out = -f(-x), giving odd symmetry.

`EVEN` Make even function, truth value. (Default = false). If set to true, negative values of x will be evaluated as out = f(-x), giving even symmetry.

4.8.5 Example

`E1 2 0 1 0 posy 1 .5` The output of `E1` is the square root of its input.

4.9 `PULSE`: Pulsed time dependent value

4.9.1 Syntax

`PULSE` *args*
`PULSE` *iv pv delay rise fall width period*

4.9.2 Purpose

The component value is a pulsed function of time.

4.9.3 Comments

For voltage and current sources, this is the same as the Spice PULSE function, with some extensions.

The shape of a single pulse is described by the following algorithm:

```
if (time > _delay+_rise+_width+_fall){
  // past pulse
  ev = _iv;
}else if (time > _delay+_rise+_width){
  // falling
  interp=(time-(_delay+_rise+_width))/_fall;
  ev = _pv + interp * (_iv - _pv);
}else if (time > _delay+_rise){
  // pulsed value
  ev = _pv;
}else if (time > _delay){
  // rising
  interp = (time - _delay) / _rise;
  ev = _iv + interp * (_pv - _iv);
}else{
  // initial value
  ev = _iv;
}
```

4.9.4 Parameters

IV = *x* Initial value. (required)

PV = *x* Pulsed value. (required)

DELAY = *x* Rise time delay, seconds. (Default = 0.)

RISE = *x* Rise time, seconds. (Default = 0.)

FALL = *x* Fall time, seconds. (Default = 0.)

WIDTH = *x* Pulse width, seconds. (Default = 0.)

PERIOD = *x* Repeat period, seconds. (Default = infinity.)

4.10 PWL: Piecewise linear function

4.10.1 Syntax

PWL *x1,y1 x2,y2 ...*

4.10.2 Purpose

Defines a piecewise linear transfer function or time dependent value.

4.10.3 Comments

This is similar to, but not exactly the same as, the Berkeley SPICE PWL for fixed sources.

For capacitors, this function defines *charge* as a function of voltage. For inductors, it defines *flux* as a function of current.

For fixed sources, it defines voltage or current as a function of time.

The values of x must be in increasing order.

Outside the specified range, the behavior depends on the type of element. For fixed sources, the output (voltage or current) is constant at the end value. This is compatible with SPICE. For other types, the last segment is extended linearly. If you want it to flatten, specify an extra point so the slope of the last segment is flat.

4.10.4 Parameters

There are no additional parameters, beyond those that apply to all.

4.10.5 Example

`C1 2 0 pwl -5,-5u 0,0 1,1u 4,2u 5,2u` This "capacitor" stores 5 microcoulombs at -5 volts (negative, corresponding to the negative voltage, as expected. The charge varies linearly to 0 at 0 volts, acting like a 1 microfarad capacitor. (C = dq/dv). This continues to 1 volt. The 0,0 point could have been left out. The charge increases only to 2 microcoulombs at 4 volts, for an incremental capacitance of 1u/3 or .3333 microfarads. The same charge at 5 volts indicates that it saturates at 2 microcoulombs. For negative voltages, the slope continues.

4.11 `SFFM`: Frequency Modulation time dependent value

4.11.1 Syntax

`SFFM` *args*
`SFFM` *offset amplitude carrier modindex signal*

4.11.2 Purpose

The component value is a sinusoid, frequency modulated by another sinusoid.

4.11.3 Comments

For voltage and current sources, this is the same as the Spice `SFFM` function, with some extensions.

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

```
mod = (_modindex * sin(2*PI*_signal*time));
ev = _offset + _amplitude
        * sin(2*PI*_carrier*time + mod);
```

4.11.4 Parameters

`Offset` = x Output offset. (Default = 0.)

`Amplitude` = x Amplitude. (Default = 1.)

`Carrier` = x Carrier frequency, Hz. (required)

`Modindex` = x Modulation index. (required)

`Signal` = x Signal frequency. (required)

4.12 SIN: Sinusoidal time dependent value

4.12.1 Syntax

`SIN` *args*
`SIN` *offset amplitude frequency delay damping*

4.12.2 Purpose

The component value is a sinusoidal function of time, with optional exponential decay.

4.12.3 Comments

For voltage and current sources, this is the same as the Spice `SIN` function, with some extensions.

It generates either a steady sinusoid, or a damped sinusoid.

If *delay* and *damping* are both zero, you get a steady sine wave at the specified *frequency*. Otherwise, you get a damped pulsed sine wave, starting after *delay* and damping out with a time constant of $1/$*damping*.

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

```
reltime = time - _delay
if (reltime > 0.){
  ev = _amplitude * sin(2*PI*_freq*reltime);
  if (_damping != 0.){
    ev *= exp(-reltime*_damping);
  }
  ev += _offset;
}else{
  ev = _offset;
}
```

4.12.4 Parameters

`Offset` = x DC offset. (Default = 0.)

`Amplitude` = x Peak amplitude. (Default = 1.)

`Frequency` = x Frequency, Hz. (required)

`Delay` = x Turn on delay, seconds. (Default = 0.)

`Damping` = x Damping factor, 1/seconds. (Default = 0.)

4.13 TANH: Hyperbolic tangent transfer function

4.13.1 Syntax

TANH *gain limit*
TANH *args*

4.13.2 Purpose

Defines a hyperbolic tangent, or soft limiting, transfer function.

4.13.3 Comments

There is no corresponding capability in any SPICE that I know of, but you can get close with POLY.

For capacitors, this function defines *charge* as a function of voltage. For inductors, it defines *flux* as a function of current.

For fixed sources, it defines voltage or current as a function of time, which is probably not useful.

This function describes a hyperbolic tangent transfer function similar to what you get with a single stage push-pull amplifier, or a simple CMOS inverter acting as an amplifier.

4.13.4 Parameters

GAIN = x The small signal gain at 0 bias. (Required)

LIMIT = x Maximum output value (soft clipping). (Required)

4.13.5 Example

`E1 2 0 1 0 tanh gain=-10 limit=2 ioffset=2.5 ooffset=2.5` This gain block has a small signal gain of -10. The input is centered around 2.5 volts. The output is also centered at 2.5 volts. It "clips" softly at 2 volts above and below the output center, or at .5 volts $(2.5 - 2)$ and 4.5 volts $(2.5 + 2)$.

4.14 .model TABLE: Fit a curve

4.14.1 Syntax

.model *name* TABLE *x1,y1 x2,y2 ... args*

4.14.2 Purpose

Fits a table of data using piecewise polynomials, or splines.

4.14.3 Comments

This function fits a set of piecewise polynomials to a set of data.

It differs from the FIT function in that the TABLE form uses a .model statement containing the actual data, while the FIT form has all of the data on the instance line.

See the comments section of FIT for more detail on the options.

4.14.4 Parameters

`Order` = x The order of the polynomial to fit, within the supplied data. (Default = 3) Legal values are 0, 1, 2, and 3, only.

`Below` = x The value of the derivative to use below or before the specified range.

`Above` = x The value of the derivative to use above or after the specified range.

4.14.5 Example

```
.model nlcap table -5,-5u 0,0 1,1u 4,2u 5,2u order=1
C1 2 0 nlcap
```

This "capacitor" stores 5 microcoulombs at -5 volts (negative, corresponding to the negative voltage, as expected). The charge varies linearly to 0 at 0 volts, acting like a 1 microfarad capacitor. (C = dq/dv). This continues to 1 volt. The 0,0 point could have been left out. The charge increases only to 2 microcoulombs at 4 volts, for an incremental capacitance of 1u/3 or .3333 microfarads. The same charge at 5 volts indicates that it saturates at 2 microcoulombs. For negative voltages, the slope continues. See the example under `FIT` for a comparison.

Chapter 5

Installation

5.1 The easy way

For this version, you can use either the GNU style "configure;make" style build process, or the old ACS style. If it works for you, use the GNU style.

For the GNU style build, just type "./configure" then "make" from the project's root directory. This will configure both the model compiler and the simulator, and then build the model compiler first, then use it to build the simulator. That should be all that is needed. You do not to read any further.

5.2 If that doesn't work

This version requires a two-step build. First you build the model compiler, then you build the simulator. You can usually get away with only building the simulator.

So .. cd to modelgen, type make (as below) then go back down and cd to src, type make (as below)

If it fails, go into its build directory (the one containing the .o files) and manually create a symbolic link to the model compiler.

"Type make" really means .....

Usually, you can just type "make". This will make a "release" version, with optimization on and extra debug code out. It will build in the O subdirectory. This assumes you have g++ in a reasonable configuration.

To make a "debug" version (slow, with additional error checking), type "make debug". If you have a recent g++ compiler, this should build it in the O-DEBUG subdirectory.

If your compiler is not g=++, but called by "CC", try "make CC". This is believed to work with some compilers. Some of them do not implement the full language, so they cannot be used. Try it. There is a special one "sun4-CC" for a Sun running Solaris with the most recent version of Sun's compiler. It will not work with older versions.

To make a "release" version for a particular system, type make followed by your system type, such as "make linux". This will build it in a subdirectory (in this case LINUX). With this method, you can build for multiple systems in the same directory.

Look at "Makefile" for a list of supported systems, and clues of how to do it on others. Most of them have not been tried in years.

If it doesn't work, edit only a "Make2.*" file, and possibly `md.h` or `md.cc`. All nonportabilities are confined to these files.

It does require a recent and proper C++ compiler with a proper library, including STL. Gnu compilers older than 2.8 will probably not work. Anything else that old will also probably not work. Any high quality C++ compiler available today should work.

To install ....

Just move or copy the executable to where you want it.

5.3 Details, custom compilation

Read this section if you have problems or want to know more. It is not necessary most of the time.

Most of the development of Gnucap was done on a PC running Linux. I have also compiled it successfully on several other systems, listed at the end of this file. Other users have ported it to several other systems. Some of the files are included in the distribution. They may not have been tested in the latest release. It should compile with any "standard" C++ compiler. It should produce no warnings when compiled with the switches in the supplied makefiles and g++, except those due to the system supplied header files being defective. It requires templates, but not exceptions.

All source files are in the src and modelgen directories. I use subdirectories for the .o files each supported machine. This makes it possible to install it on several different machines all sharing the same file system.

To avoid maintaining multiple versions of Makefiles, I have broken them up to parts that must be concatenated: Make1.*, Make2.*, Make3.*. In general, to make a Makefile for your system, cat one of each. See the Makefile for details. I have automated this for some systems. Just "make your-machine", if it is one that is supported. In some cases, the Makefile will compile both a "release" and "debug" version. In these cases, type "make your-machine-release" or "make your-machine-debug" depending on which you want. This will make the appropriate Makefile, cd to where the .o's go and run make from there. For porting information for specific machines, read its `Make2.*` file.

I assume that make will follow "VPATH" to find the sources. This system makes it possible to manage several platforms on a single file system which may be NFS mounted to all the supported machines. If your make does not support VPATH, there are three options. The preferred method on unix based systems is to cd to where the .o's go and type `ln -s ../*.cc ../*.h .` (The command ends with a dot.) This will set up links so the Makefiles will work as intended. In some cases we have set up the Makefile to do this automatically. The second method, which may be needed on systems that don't have symbolic links is to copy the .c and .h files to satisfy make. The third option, where you have only one computer, is to move the machine specific Makefile to the src directory and run make from there.

If you have g++ on a unix type system that is not directly supported, try to compile it by just typing `make`. In most cases this will do it, but you may get a few warnings. If it doesn't work, look in the file `md.h` for hints. Just plain `make` will build a guess at a release version, assuming a Linux-like system with GNU tools.

If you want a development version with additional debugging enabled, type `make debug`. This results in a significant speed penalty.

Then make the installation version, select the machine you have from the make file and make that. The machine specific versions will build in their own directory, have debugging code disabled, and options are set for best speed. The general purpose `make g++` builds a version that is optimized as much as it can be in the general case.

If you have a cfront-type compiler, called `CC`, and your system is not directly supported, try it first by typing `make CC`. Again, you may get a few warnings but it should work. Look in the file `md.h` for hints, if it doesn't work, or if the warnings look serious.

Since C++ is an evolving language, there are some known portability problems. All of them are due to compilers that do not implement the standard correctly. Since the problems will go away in time, I

have chosen either not to burden the code with them, except where a few mainstream systems fail. All dependencies should be confined to the two files `md.h` and `md.cc`, if possible.

Here are some possible problems that are no longer supported:

bool The C++ language includes a type `bool`, which is not implemented in older compilers. Older compilers just use `int`, and fake it with a `typedef` or `#define`, neither of which work correctly.

Here are some problems that you will need to deal with creatively:

missing files or functions Another cause of a port to fail is missing header files or missing function prototypes. Sometimes missing functions can be a problem. The solution to these problems is to supply what is missing. The `md_*` files exist for this purpose. You should make a copy of the appropriate Make2.___ file, patch it to define something to identify the system, then patch the `md.h` and `md.cc` as appropriate. You should not use any `#ifdef`'s except in these file.

bad header files In some cases, the header files that come with the system or compiler are defective and generate warnings without anything wrong with the program being compiled. This slips by in the distribution because most developers compile with warnings off. Usually, these can be ignored.

Here are some problems that have work-arounds:

const C++ uses an abstract notion of constant, meaning that the external appearance of an object declared const must not change, but there can be internal changes like reference counters. The keyword `mutable` means that a member variable can change even if it is declared const. As a work around, we use CONST, which is either defined to nothing or const. For any good compiler, the line `#define CONST const` will give correct behavior. For a bad compiler, the line `#define CONST` will turn it off. There is no harm in treating all compilers as "bad" except for the loss of compile diagnostics.

complex The evolving standard shows complex to be a template class, so instead of having a type `complex`, there is `complex<double>`, `complex<float>`, and so on. Older compilers have only `complex`. The line `typedef std::complex<double> COMPLEX;` in `md.h` works for a correct compiler. You may need to change it of an older one.

template instantiation There are three common ways to instantiate templates in common use. Unfortunately, they are incompatible and none of the methods are available in all compilers. Gnucap requires templates, so will not work with many older compilers.

- **Link time** The entire program is compiled and linked without templates, resulting in some unresolved externals. The files defining the templates are compiled again to fill the need. This is the preferred way, if you have it. It is supported by CFRONT derivatives such as the Sun CC compiler. Define `LINK_TEMPLATES` to force it.
- **Compile time** All parts of templates must be compiled as if in-line, requiring all code to be in the .h file, or included by the .h file. Header files must include .cc files. The duplicates are supposed to be thrown away by the linker. This is the only style supported by Borland 3.1 or 4.0. It is supported inefficiently by the GNU compiler starting at version 2.6. Since no mainstream compiler requires this, and it is inefficient, it is no longer supported.
- **manual** Templates must be instantiated manually. This is the preferred way for the GNU and Microsoft compilers. It is a nuisance, but it generates the best code. Define `MANUAL_TEMPLATES` to force it.

template resolution The second inconsistency with templates is how the type matching is resolved. Some compilers require an exact match. Some will make trivial conversions, such as int to const int. The language definition allows for templates to be "specialized" by providing a specific implementation for a specific type, resorting to the template for others. Some compilers (Sun) do not support this. Since this is common, there are work arounds in the code for it in the simulator but not the model compiler. If you want to compile the model compiler, you will need to get a better C++ compiler.

The files starting with `plot` contain plotting drivers are generally bogus.

There should be NO non-portable code anywhere but the `md_*` files. If a fix is absolutely necessary elsewhere, `#define` some symbol in `md.h` and refer to it elsewhere. Then consider it to be temporary.

Chapter 6

Adding models

Gnucap has three distinct styles of adding models:

Model Compiler is the easiest way to add models, but the least flexible. The model compiler generates .cc and .h files using the *enhanced subcircuit* mode. It is possible to develop models with almost no knowledge of the simulator internals. In most cases, this is the preferred way. The standard MOSFET and diode models are done this way.

Enhanced subcircuit is less efficient than *primitive* but has other advantages that make it preferable to *primitive* when you can use it. The model is defined as a combination of equations and topology. The AC and pole-zero code is inherited from a base class, so you don't need to to it. You need to understand the simulator's internals, and it is not likely to be portable to other simulators.

Primitive should be used only when absolutely necessary. If it is done correctly, it will result in the fastest execution, but you need to do everything. It requires thorough knowledge of the simulator internals, including how Gnucap is different from other simulators. If you miss some of the details, it is possible that your model will work but slow down the simulator significantly. Most of the primitive devices (resistors, sources) are done this way. A few device types that have special considerations, like gates and transmission lines, are also done this way.

6.1 Using the model compiler

This section is a first cut at documentation. If you actually want to install a model, please ask for more information. Your questions will help me write the more complete documentation. (aldavis@ieee.org)

To create a model using this method, you create one file, with the extension `.model`. A separate program, `modelgen` processes this file to generate the appropriate `.cc` and `.h` files. The resulting files are equivalent to the subcircuit method of creating models.

There are two primary sections, `device` and `model`. Most models have both, but a device can use several different `model`s as long as they are derived from a common base and designed to work together. It is standard practice to share like this. For example, all of the MOS models use the same `device` section.

Any `model` can inherit from another `model`, thus reducing the need for repetition when code or parameters are the same in different models, and allowing several to use the same `device` section.

This model compiler has restrictions that will be removed in future releases. Not all device types can be fully done with it, due to missing features. Often, it is necessary to finish the job manually. In this release (0.30), the diode fully uses it, but not the way I want to. The MOSFET uses the `cc_direct` to finish the

job. Two functions `do_tr` and `tr_needs_eval` must be provided this way. Code placed here is simply copied out.

As a general rule, when using the name = value form, the value is delimited by whitespace, or possibly other tokens. If you want blanks in the value string, put it in quotes. It is recommended to quote any value string that is comprised of more than one word, even if there are no blanks.

6.1.1 Device section

`parse_name` (required) This is the name of the device, to be recognized by the parser, in Gnucap native format. Example: `diode`.

`id_letter` (required) This is the letter used to identify the device, when the `parse_name` is omitted (Spice format). Example: `D` identifies a diode.

`model_type` (required) This is the name of the model type associated with this device type. It can be the name of the one matching type, or the name of a base from which a family can be derived.

`circuit` (required) This is a subsection containing a netlist representing the internal structure of the device. See the section *circuit subsection* for details.

`tr_probe` (optional) This is a subsection containing a list of internal probes to be made available to the user. See the section *Tr_probe subsection* for details.

`device` (required) This is a subsection describing the non-shared data relating to the device. Information here is unique to this device. It is primarily state information. See the section *Device subsection* for details.

`common` (optional) This is a subsection describing the shared data relating to the device. Information here may be shared between similar devices. It is primarily information that is read from the circuit description. See the section *Common subsection* for details.

`tr_eval` (optional) This is a subsection which will eventually contain evaluation code for the whole device. For now, it is a stub, which is used as a flag. See the section *Tr_eval subsection* for details. If this subsection is omitted, it is considered to be a pure subcircuit.

`eval` (optional) There may be any number of `eval` subsections, which are specific evaluators for internal elements. See the section *Evaluators* for details.

Circuit subsection

The circuit subsection has 5 parts, in order.

1. The optional keyword "`sync`" says that the entire subcircuit representing the device must be evaluated synchronously. Without this keyword, it is treated as a subcircuit made of independent elements.

2. The port list "`ports`", which is a list of the nodes interfacing to the outside.

3. A list of local internal nodes, not visible outside.

   Each local node may have two optional attributes:

   `short_if` This specifies a conditional which if true will result in this node being omitted.

`short_to` This specifies another node which will be substituted for this node if the `short_if` condition is true.

4. Any number of named and typed `args` sections. Each section contains name = value pairs which assign values to elements in the subcircuit. It is used only for more complex elements like diodes.

5. A list of circuit elements that comprise the model. Each has three required fields, then a list of optional key = value pairs.

   The required fields, in order, are:

   - The type of element. This is usually one of resistance, capacitance, admittance, or poly_g, but can be any device type, including those created by modelgen.
   - The label, a string used to refer to it.
   - A node list, in curly braces.

   After that, optional fields are used to assign attributes. Not all are legal or appropriate with all element types.

   `value` This must evaluate to a constant, which is interpreted as the nominal value of the element.

   `eval` This is the name of an `eval` section, which specifies nonlinear, and depenndent characteristics.

   `args` This refers to an `args` section, described previously. It is used only for advanced element types, as those created by modelgen.

   `reverse` This specifies an expression, that when evaluated will tell whether to reverse the element. If it evaluates to true, the node pairs are interchanged. If there are two nodes, they are interchanged. If there are four, the first pair are interchanged and the second pair are interchanges. The pattern repeats for as many nodes as there are. This is used with diodes, which may be reversed depending on whether the device being defined is N-type or P-type.

   `omit` This specifies an expression, that when evaluated will tell whether to omit the element. If it evaluates to true, the element is omitted.

   `state` This specifies a the name of a device state variable that is applied to this element. This state variable must be one of those defined under `calculated parameters`. It is primarily intended for the `poly_g` and `poly_cap` element types. In the `calculated parameters` section, the following parameters are its derivatives. If the poly element has N node pairs, the following N parameters are the derivatives, with respect to each voltage, in order.

Tr_probe subsection

The `tr_probe` subsection is where you list the probes for transient and DC analysis. It is a list of name = value pairs, where the value is an expression that calculates or looks up the value.

You can reference any device parameter directly, or others with the appropriate struct prefixing.

You can reference probes on internal elements with the syntax "`@`", followed by the element label, followed by the probe name in square brackets. For example, "`@Cj[Capacitance]`" refers to the probe named "Capacitance" on the element "Cj". It is your responsibility to see that the element actually exists, and that it has a probe with that name.

You can reference node voltages with the same syntax, but the "device" name is formed by prefixing the node name with "`n_`".

You can also call functions, and make arbitrary expressions. In general, the code is just copied over, with the exception of the probes, which are modified to the internal format.

The probe name in the generated model will be non-case-sensitive. To the model compiler, upper case letters must match exactly, and lower case letters are optional. For example, "`CGSOvl`" in the model file can be referred to as `cgso`, `cgsov`, or `cgsovl`, or any variants differing only in case.

Device subsection

This subsection defines information that is not shared between instances. In general, that which must be maintained as different, even though devices are identical, is placed here.

Calculated_parameters subsubsection This subsubsection lists all of the "calculated parameters". In this case, it means that which is calculated during simulation, the state information.

The format for each item is: type, name, comment, attributes, semicolon.

The only attribute appropriate is "`default`", which is the default value set by the constructor.

Common subsection

This section defines information that the simulator may share between instances. Most parameters specified by the user are placed here, allowing the simulator to share data for identical devices.

Unnamed subsubsection You can designate one of the `raw_parameters` to be the "value". When a number is given without a name, it is assigned to this one.

Raw_parameters subsubsection This subsubsection lists all of the "raw parameters", the parameters supplied by the user on the instance line. The format and available attributes are described in the "Parameter lists" section, which follows.

Calculated_parameters subsubsection This subsubsection lists all of the "calculated parameters", the parameters not supplied by the user on the instance line. Instead, the are calculated based on other input. The format and available attributes are described in the "Parameter lists" section, which follows.

Tr_eval subsection

In this release, this section is a dummy. Put a stub here if you define a `do_tr` later. Otherwise leave it out. This will change in a future release.

Evaluators

The `eval` sections are evaluators that turn the primitive resistors and capacitors into advanced behavioral elements.

The body is the core of a C++ function, which is copied over directly after attaching some headers. Given some "x", this function computes "f(x)" and its derivative with respect to x. The primary communication is through the structure "`d->_y0`". The input is "`d->_y0.x`". You must evaluate the function, and place the result in "`d->_y0.f0`" and its derivative in "`d->_y0.f1`". The exact meaning of these values depends on what type of element it is.

For the primitives ...

resistance x is current, y0 is voltage, y1 is resistance

admittance x is voltage, y0 is current, y1 is admittance

capacitance x is voltage, y0 is charge, y1 is capacitance

inductance x is current, y0 is flux, y1 is inductance

vccs x is voltage, y0 is current, y1 is transconductance

In addition, all relevant parameters are available with the appropriate prefix. See the section *accessing data in code blocks*. Most are read-only.

The prefix `d->` refers to the element being processed. This data is read-write.

6.1.2 Model section

`base` (optional) The keyword `BASE` is used as a flag to say this is a base for other models. When the base flag is set, others can be derived from it and used interchangeably with the same device type.

`dev_type` (required, all) This is the name of the device type associated with this model type.

`level` (optional, final only) When several models are derived from a base, the numeric level is used as a parameter to select which one to use.

`inherit` (optional) The model being defined inherits from the named base model. There is no limit to the depth if inheritance.

`keys` (required for base, optional otherwise) This is a list of the keywords that are used to identify the model, and assign attributes. See the section *Keys subsection* for details.

`independent` (optional) This is a section describing parameters that are not dependent on size or temperature. See the section *Independent subsection* for details.

`size_dependent` (optional) This is a section describing parameters that are dependent on size. See the section *Size_dependent subsection* for details.

`temperature_dependent` (optional) This is a section describing parameters that are dependent on temperature. See the section *Temperature_dependent subsection* for details.

`tr_eval` (required once in hierarchy) This is a section containing evaluation code for the whole device. See the section *Tr_eval subsection* for details.

Keys subsection

The `keys` subsection consists of a number of keywords that are used in the `.model` statement to identify this model. Different keys can be used to represent variants, such as "`NMOS`" and "`PMOS`" to represent the N and P channel devices. Each one is followed by an assignment to be made when the key is present.

It is required, at least once in the hierarchy. Additional keys can be used to select a particular model, as an alternative to the `level` parameter.

Independent subsection

The `independent` subsection list all of the "independent" parameters supplied by the user in the `.model` statement.

Raw_parameters subsubsection This subsubsection lists all of the "raw parameters", the parameters supplied by the user on the `.model` line. The format and available attributes are described in the "Parameter lists" section, which follows.

Calculated_parameters subsubsection This subsubsection lists all of the "calculated parameters", the parameters not supplied by the user. Instead, they are calculated based on other input. The format and available attributes are described in the "Parameter lists" section, which follows.

Override subsubsection This subsubsection lists parameters that have already been defined in base classes, that need a change for this particular type. You can override most attributes, giving the benefit of defining it locally, while retaining most from the base. The format and available attributes are described in the "Parameter lists" section, which follows.

Code_pre and Code_post subsections These subsubsections define C++ code that is inserted into the function that calculates values, scales, and checks limits. The block `code_pre` is inserted before the automatically generated code. The block `code_post` is inserted after the automatically generated code.

Size_dependent subsection

The `size_dependent` subsubsection is similar to the `independent` subsubsection except that it defines a base paramater and scale factors so a custom value can be generated based on the device size.

Every parameter in this subsubsection actually generates a set of four. The first is the base, as in the `independent` subsubsection. In addition, the same name prefixed by "`L`" is the length dependency, the name prefixed by "`W`" is the width dependency, and the name prefixed by "`P`" is the product (length * width) dependency.

You must provide a `code_pre` section, which must declare and define values for "`L`" (length) and "`W`" (width).

The actual value is calculated by: `nom + ld/L + wd/W + pd/(W*L);`, where `nom` is the nominal value, `ld` is the length dependency (key name has the "`L`" prefix), `wd` is the width dependency (key name has the "`W`" prefix), and `pd` is the product dependency (key name has the "`P`" prefix).

Temperature_dependent subsection

The `temperature_dependent` subsubsection contains a list of parameters that are calculated based on temperature, and two code blocks (`code_pre` and `code_post` to make the calculations.

This code is evaluated at run time, possibly every time step, whenever temperature changes. Some Spice models throw calculations not related to temperature into the temperature block. This is very bad practice. In Gnucap, temperature is local and time variant.

Tr_eval subsection

The `tr_eval` subsubsection is the actual model evaluation code for nonlinear DC and transient analysis. This code must calculate all state variables (data listed as "`calculated`" in the `device` section, except those that are part of one of the subcircuit elements. Inputs and outputs are through the `d->` structure.

This function only needs to fill in the `calculated` data. The details, like differentiating charge in capacitors, is left to the subcircuit elements. It is also not necessary to check convergence. This, too, is left to the subcircuit elements.

6.1.3 Accessing data in code blocks

Most parameters are available, usually read-only, in any code block, with the appropriate prefix:

`p->` The parent device, usually the device being defined by the `.model` file. This is usually the "calculated parameters" under "device" in the `.model` file.

`c->` The "common" belonging to the parent device. This consists of all of the parameters in the `common` section of the `.model` file.

`m->` This is the "model" parameters, all of the parameters in the `model` section of the `.model` file, except those listed as "size dependent".

`b->` This is the sized value of the size dependent parameters in the `model` section. "B" is for "bin", which is derived from the concept of "binning" of models.

`d->` This is the device parameters. In evaluation functions, it is read-write.

`t.` This is the device values, scaled by temperature.

6.1.4 Parameter lists

The format for each item is: type, name, comment, attributes, semicolon.
The available attributes are:

`name` This is the name to be used for input in the data file. It is also the name this parameter is listed as when the internal data is printed.

`alt_name` This is an alternate name used for input.

`default` This is the initial default value, set by the constructor.

`calculate` If no value is supplied, the program will calculate it using this formula.

`quiet_min` If the input or calculated value is less than this number, substitute this number without warning.

`quiet_max` If the input or calculated value is more than this number, substitute this number without warning.

`final_default` This is the final default value, supplied after all attempts to fill or calculate it fail.

`offset` Add this number to the input value to get the value actually stored in memory. Example: `double temperature ‘‘’’ offset=273;`. This sample allows data entry in degrees Celsius, but storage in Kelvin.

`scale` Multiply the input value by this number to get the value actually stored in memory. Example: `double length ‘‘’’ scale=1e6;`. This sample allows data entry in microns, but storage in meters.

`positive` This number is always positive. The magnitude of the entered value is stored.

`octal` The number read is interpreted as octal (base 8), instead of the usual base 10.

`print_test` This is a test to determine whether the value is printed in a standard listing or not. The value is printed only if this test evaluates to true at run time. If `print_test` is omitted, it is always printed.

`calc_print_test` This is a test to determine whether the value is printed as a comment in a standard listing or not. The value is printed only if this test evaluates to true at run time. If `calc_print_test` is omitted, it is never printed.

Chapter 7

Technical Notes

7.1 Architecture

7.1.1 File organization

Gnucap source files are organized into groups by the name prefix as follows:

ap "Argparse". Generic parser and lexical analysis library.

bm Behavioral modeling.

c Commands.

d Devices and models.

e Device and model base classes. ("e" comes from "electrical" and is retained because of inertia.)

io Input and output library, raw, generic.

l Library. General purpose functions and classes that do not fit elsewhere.

m Math library.

plot Obsolete plotting that should be replaced.

s Simulation engine.

u Utility functions and classes. Gnucap Specific.

The files `ap_`, `io_`, `l_`, `m_` are not Gnucap specific. Although they were created for Gnucap, they are public domain and may be used by anyone for any purpose.

The remaining files `bm_`, `c_`, `d_`, `e_`, `s_`, `u_` are Gnucap specific, and reuse is subject to the Gnu Public License.

Some of the `d_` files are automatically generated during compilation. Do not change them, because your changes may be lost in a recompile. For licensing and distribution legal purposes, these files are considered to be "object" code, even though they are readable C++.

The files `d_.model`, where present, contain the actual model descriptions as input for `modelgen`, the model compiler. These files are the source that is used to generate the corresponding `.cc` and `.h` files. All changes should be done to the `.model` file. For GPL purposes, these files are considered to be "source".

7.1.2 Building, Makefiles

Gnucap uses a 4 part Makefile, designed for simultaneous builds on several systems. A true Makefile is built by selecting and catenating the four pieces. A master Makefile switches to a subdirectory and builds a specialized Makefile there.

Make1 The file list. Specific to this program.

Make2 Compiler and system dependencies. Specific to the compiler. In some cases, hardware dependencies are here. There are several provided. Choose the one that matches your system.

Make3 Basic "make" targets. Generic.

Make.depend List of dependencies.

7.1.3 Program flow

It all starts at "`main`", in `main.cc`. The function "`main`" has a loop that gets input and calls "`CMD::cmdproc`" to dispatch the command.

Batch mode is done in "`process_cmd_line`", by using "`CMD::cmdproc`" to execute the commands "`get`" or "`<`" which is passed to "`CMD::cmdproc`" as text.

The function "`CMD::cmdproc`" dispatches the command to its handler. The handlers are located in the "`CMD`" namespace, and the "`c_`" files.

7.2 Transient analysis

7.2.1 The "CPOLY" and "FPOLY" classes

Before beginning a discussion of the evaluation and stamp methods, it is necessary to understand the "CPOLY" and "FPOLY" classes.

These classes represent polynomials. At present, only the first order versions are used, but consider that they could be extended to any order.

When evaluating a function $f(x)$, there are several possible representations for the result. The "CPOLY" and "FPOLY" represent two of them.

The "CPOLY" classes represent the result in a traditional polynomial form. Consider a series of terms, c0, c1, c2, ... These represent the coefficients of a Taylor series of the function expanded about 0. (Maclauran series). Thus $f(x) = c_0 + c_1 x + c_2 x^2 + c_3 x^3 + ...$ In most cases, only the c0 and c1 terms are used, hence the "CPOLY1" class. The series is truncated, so it is exact only at one point. The value "x" in the "CPOLY" class is the point at which the truncated series is exact, so it is not truly a series expanded about 0.

The other "FPOLY" classes represent the same polynomial as a Taylor series expanded about a point "x". Again, consider a series of terms, f0, f1, f2, ... This time the terms represent the function evaluated at x and its derivatives. Therefore, f0 is $f(x)$, f1 is the first derivative, f2 is the second derivative, and so on. To evaluate this for some t near x, $f(t) = f_0 + f_1(t-x) + f_2(t-x)^2 + f_3(t-x)^3 + ...$ Again, in most cases, only the f0 and f1 terms are used, hence the "FPOLY1" class.

Both of these are equivalent in the sense that they represent the same data, and there are functions (constructors) that convert between them. The "FPOLY" form is usually most convenient for function evaluation used in behavioral modeling and device modeling. The "CPOLY" form is most suitable for stamping into the admittance matrix and current vector for the final solution.

7.2.2 The basic solution algorithm

In simplified form, the algorithm looks like this ...
before doing anything ....

```
expand()
precalc()
```

on issuing the "tran" command ..

```
tr_begin() // get ready
for (each time step) {
  tr_advance() // precalculate and propagate
  for (each iteration) {
    tr_queue_eval() // decide which models need evaluating
    do_tr() // evaluate models
    tr_load() // build the matrix of equations
    solve the resulting system of equations
  }
  if (converged) {
    tr_review() // how are we doing? suggest time step
  }
  if (no problems) {
    tr_accept() // postcalculate and accept data
  }
}
```

The functions referred to above are actually loops that call that function for all devices in the circuit.

For all of them, it is possible that they may not be called. If there is evidence that the result will not change from the last time it was called, it probably will not be called. Since this algorithm is not perfect, it is possible that any particular function may be called twice, so they are written so calling more than once is equivalent to calling once.

expand

The *expand* functions expand subcircuits and models, as needed. Unlike Spice, it does not flatten the circuit. It allocates space for the additional storage, attaches models, and related tasks. It does not compute any values. It is called once after reading the circuit, and possibly later when the topology of the circuit is changed.

Most simple elements do not have expand functions. Most advanced components do.

Expanding a subcircuit makes a copy of it, and remaps the nodes. Most components use a shallow copy. That is, if something is attached through a pointer, the value of the pointer is copied, not the attachment. Commons are never copied when the owner components are copied.

It is ok to expand a component more than once. Either it frees then re-expands, or it keeps what it can and checks to make sure it is correct.

precalc

The *precalc* functions attempt to pre-calculate anything that will remain constant during a simulation run. This includes size dependent transistor parameters and the stamp values for linear elements.

The actual evaluation of constant linear elements is done here. For nonlinear elements, it computes a first guess.

dc_begin, tr_begin, tr_restore

These functions are called once on issuing a simulation command. The *dc_begin* functions are called on starting a DC or OP anaylsis. The *tr_begin* functions are called on starting a transient analysis from time = 0, or the first time. The *tr_restore* functions are called on starting a transient analysis in such a way that the analysis continues from where a previous transient analysis left off.

The purpose is to make sure that the initial guesses and numbers for prior iterations that don't exist are properly set up. For linear elements, the values are set up here and are not computed later.

dc_advance, tr_advance

These functions are called before beginning a new time or voltage step.

For basic storage elements like capacitors, they store the data from the previous step. They may also attempt to predict a new value, in hopes of speeding up the real solution.

For delay elements like logic devices and transmission lines, this function does the real work. It takes previous results and applies them, generating data that will be later loaded into the matrix.

tr_needs_eval

This function returns true if the component needs to be evaluated on this iteration. It should return false if it has already been queued, but some do not do this.

tr_queue_eval

This function queues the component to be evaluated, if it needs it. If *tr_queue_eval* is not called, it will not be evaluated.

do_tr

In most cases, the *do_tr* functions do the real work, or call the *tr_eval* function to do it. It evaluates the model, checks convergence, and queues it for loading. Calling this function more than once on an iteration is harmless, except for the waste of time.

Usually, it calculates the function and derivative. It may also do integration, interpolation, iteration, or whatever is required. The result is a set of values ready to stamp into the admittance matrix and current vector.

There are several distinct steps within this function.

1. The first step is to gather the information necessary to make the computations. Usually, this is the node voltages, but it could be currents, temperature, charge, or something else.

2. The next step is to evaluate any attached function. This could be done in line, or by a call to *tr_eval*. The result of this evaluation is stored in *_y0* (of type *FPOLY1*. The *tr_eval* function reads the value of x from *_y0*, and fills in the *f0* with the result of function evaluation, and *f1* with its derivative. The *tr_eval* function must also check for convergence by comparing the new *_y0* with the old value, *_y1*. This attached function is generic in the sense that it is the same for all device types. This is the $y = f(x)$ that is referred to in the behavioral modeling documentation.

3. These values are stored for convergence checking and probing.

4. After that, it must be converted to a current and admittance so it can be used in the system of nodal equations. This step is dependent on what type of device it is. For a conductance element, *tr_eval* directly returns the correct information, so nothing needs to be done here. For a capacitor, this step does numerical integration. Capacitors store this in *_i0*. Most other elements do not store this result directly.

5. Then, it must be converted into *CPOLY* form to meet the requirements of the system of equations.

6. The device is queued for loading. Unlike Spice, Gnucap does not actually load the matrix here.

tr_load

This function gives the appearance of loading the admittance matrix and current vector with the values calculated in *do_tr*.

Actually, it does much more. In most cases, it actually loads a correction factor, assuming the old values are already loaded. To do this, it keeps track of what values are actually loaded. Whether it loads a correction or the actual value is determined first by the option *incmode*, then by status information about the solution. If it is suspected that correcting would cause too much roundoff error, it loads the actual value. The decision of whether to do a full load or an update is global.

In addition, it may apply damping in hopes of improving convergence. This means to load a value somewhere between the new and old values, in effect taking a partial step. The decision to damp is semi-global. Groups of elements are adjusted together.

The actual loading is done by one or more of a small group of general functions, depending on whether the element is active, passive, poly, or a source. Only certain patterns can be stamped. Complex devices use a combination of these patterns.

WARNING to model developers: DO NOT stamp the matrix directly!

tr_review

The *tr_review* function checks errors and signal conditions after a time step has converged. It makes entries into the event queue, makes mode decisions for mixed-mode simulation, and evaluates time step dependent errors. It returns an approximate time that the element wants for the next step. The actual next time step will probably be sooner than the value returned.

tr_accept

This function is called after the solution at a time step has been accepted. For most devices, it does nothing. For devices having storage and delayed propagation, it evaluates what signal will be propagated. For a transmission line, it calculates and sends on the reflections.

tr_unload

This function removes the component from the matrix, possibly by subtracting off what was loaded. Usually, it sets the current values to 0 and calls *tr_load*.

7.2.3 Step control

The basic algorithm

The basis of it is in the files “s_tr_swp.cc” and “s_tr_rev.cc”.

The function TRANSIENT::review sets two variables: “approxtime” and “control”.

The variable “approxtime” is a suggestion of what the next time should be. Note that this is a time, not a difference. Also note that the simulator may override this suggestion. Another “control” is an enum that shows how the time was selected. You can probe control(0) to find this code, or control(1) to see how many steps (not iterations) it calculated internally.

This time may be in the future, past, or again at the present time, depending on conditions. A time in the future means all is well, and the simulation can proceed as expected. A time in the past indicates something is wrong, such as convergence failure, excessive truncation error, or a missed event. In this case, the step is rejected, and time backed up. A repeat at the present time usually means a latency check failed. A portion of the circuit that was thought to be latent was found to be active. This usually indicates a model problem.

First, it attempts to suggest a time “rtime” based on iteration count and options.

There are several “options” that control the stepping:

- iterations > itl4 ... reduce by option ”trstepshrink”.
- iterations > itl3 ... suggest the same step as last time.
- else (iterations <= itl3) ... increase step size. Try the max as per userstepsize/skip limit to larger of (rdt*trstepgrow) where “rdt” is the old “review” estimate or (oldstep*trstepgrow) where oldstep is what was actually used last time and trstepgrow is an option, from the options command.

Second it makes another suggestion “tetime” based on truncation error, etc. It does this by calling the “review” function for all components, and taking the minimum. Any component can suggest a time for its next evaluation with its review function. Most components return a very large number, letting the capacitors and inductors dominate, but it is not required for it to be so. This time should be in the future, but errors could produce a time in the past.

Then, the earliest time of the above two methods is selected. A time in the past means to reject the most recent time step and back up, but note that this time is only a suggestion that may not be used.

The function “TRANSIENT::sweep” essentially processes the loop “for (first(); notpastend; next())”. The function “TRANSIENT::next()” actually advances (hopefully) to the next step. It may go backwards.

The actual time step depends on the suggestion by the review function (approxtime), the event queue (which includes what Spice calls “breakpoints”), the user step size (nexttick), and some tricks to minimize changes.

Some considerations ...

- Changing the step size is an expensive operation, because it usually forces a full LU decomposition and matrix reload. If the step can be kept constant, changes are limited to the right-side, eliminating the need for the full evaluation and LU.
- The simulator will place a time step exactly at any step for which the user has requested output, or Fourier analysis needs a point, or at any event from the event queue.

So, here it is ...

Assume we want it at the time the user requested. If the event queue says to do it sooner, take it, else take the user time. Note that this time is needed exactly, either now or later. If the “approxtime” is sooner than the exact time, inject a time step as follows... if the time step is less than half of the time to when an exact time is needed, take the approxtime, else take half of the exact interval, in hopes that the next step will use up the other half.

After that, there are some checks ....

"Very backward time step" means that the suggested new step is earlier than the PREVIOUS step, meaning that both the current step and its predecessor are rejected, thus it should back up two steps. Since it can back up only one step, it rejects the most recent step and tries again at the minimum step size. This usually means there is a bug in the software.

"Backwards time step" means to reject the most recent step, but the one before that is ok. It will reject the step and try again at the smaller interval. This happens fairly often, usually due to slow convergence.

"Zero time step" means that the new time is the same as the previous time, which usually means there is a bug in the software. Something is requesting a re-evaluation at the same time.

The combination of "zero time step" and "very backward time step" means that the re-evaluation didn't work.

Now, accept the new time and proceed.

The "review" function

Every component can have a "review" function, in which it can determine whether to accept or reject the solution. It will accept by suggesting a time in the future, or reject by suggesting a time in the past. It returns the suggested time. It can call new_event to request an exact time.

For capacitors and inductors, the review function attempts to estimate truncation error using a divided difference method, and it suggests a time for the next solution that will result in meeting the error requirement. Occasionally, it will discover that the step just computed fails to meet the requirement, so it will reject it.

Truncation error is related to the third derivative of charge or flux. Since current is the first derivative of charge, it would seem that second derivative of current should produce the same results faster. In practice, the current based method tends to estimate high leading to smaller steps, and the charge based method tends to estimate low, leading to larger steps. The conservative approach would suggest using the current based method, but that sometimes led to unreasonably small steps ans slow simulations, so I chose (as Spice did) the other method. Either method is ok when the step size used is close to being reasonable, but when the trial step is unreasonably large, either approach gives a very poor estimate. Taking a step much too small will make the simulator run much slower, as it takes many steps, then the step size is allowed to grow slowly. This is slower both because of the many unnecessary steps, and because of many adjustments. Taking a step that is much too large will result in a choice that is better than the first trial, which will make a better estimate and be rejected. It is rare to get more than one rejection based on truncation error.

Conclusion

Gnucap will usually do more time steps than Spice will, due to 2 factors. Gnucap will force calculations at print points and fourier points, and can reject a bad step. It is usually a little more, but could be as much as twice as many steps.

7.3 Data Structures

7.3.1 Parts list

Main parts list

The primary data storage is in a list of "cards". A card is anything that can appear in a net list. Cards live here, primarily, but there are some other auxilary lists that also contain pointers to cards.

The list stores pointers, rather than actual objects, because there are many types of cards. All are derived from the "card", through several levels of inheritance.

Usually, they are stored in the order they are read from the file, except for subcircuits, which are stored in separate lists to preserve the hierarchy.

As of release 0.24, the main list is in static storage, so there can be only one. This will change. New cards can be inserted anywhere in the list, but usually they are inserted at the end. The mechanism for marking the location is a hybrid of STL and a 15 year old pointer scheme, which will also change someday.

The "Common" and "Eval" classes

The "common" serves two distinct purposes. The first is to share storage for similar devices. The second is to attach "evaluators" to otherwise simple components for special behavior.

Most circuits have many identical elements. The "common" enables them to share storage. One "common" can be attached to many devices. When a new device is created, even if it is parses separately, an attempt is made to find an appropriate device to share with.

Simple elements like resistors and capacitors can have "evaluators" attached as commons. These evaluators calculate a function and its derivative, and return it in a standard form. Some evaluators are used internally, such as in the diode and mosfet models. Some are used explicitly, such as in behavioral modeling.

7.4 Performance

This section gives some notes on some of the performance issues in simulators. It is not intended to be complete or well organized.

7.4.1 Virtual functions

There is a question of the impact on speed from the use of virtual functions. The experiment used here is to use the circuit `eq4-2305.ckt` from the examples directory, and try several modified versions of the program. I used a 100 point dc sweep, a version between 0.20 and 0.21, and made several modifications for testing purposes. I chose this circuit because it has little to mask the effect, and therefore is sort of a worst case.

I added an int `foo` to the element class. I made the function `il_trload_source` call a virtual function `virtual_test` and stored the result. The local version body has a print call, which should not show, to make sure it calls the other. These functions simply return a constant, determined by which version of the function is called. Run time is compared, with and without this.

With 1 virtual function call (included in load)

	user	sys	total
evaluate	13.45	0.11	13.56
load	13.40	0.06	13.47
lu	1.91	0.09	2.00
back	22.35	0.27	22.61
review	0.00	0.00	0.00
output	0.11	0.11	0.22
overhead	0.23	0.19	0.42
total	51.45	0.83	52.28

With 10 virtual function calls (included in load)

	user	sys	total
evaluate	13.47	0.09	13.57
load	24.69	0.17	24.87

```
        lu     2.09     0.02     2.11
      back    22.17     0.35    22.51
    review     0.00     0.00     0.00
    output     0.14     0.11     0.25
  overhead     0.25     0.25     0.50
     total    62.82     0.99    63.81
```

No extra function calls (included in load)

```
              user      sys    total
  evaluate   13.41     0.09    13.50
      load   11.75     0.05    11.79
        lu    2.04     0.03     2.07
      back   22.51     0.33    22.84
    review    0.00     0.00     0.00
    output    0.08     0.11     0.19
  overhead    0.31     0.25     0.56
     total   50.10     0.86    50.96
```

My conclusion is that in this context, even a single virtual function call is significant (10-15% of the load time), but not so significant as to prohibit their use. The load loop here calls one virtual function inside a loop. The virtual function calls an ordinary member function. Therefore, about 30% of the load time is function call overhead.

The impact should be less significant for complex models like transistors because the calculation time is much higher and would serve to hide this better.

Spice uses a different architecture, where a single function evaluates and loads all elements of a given type. This avoids these two calls.

7.4.2 Inline functions

For this test, `il_trload_source` is not inline. Contrast to "No extra function calls" and "1 virtual function" above, in which this function is inline.

```
              user      sys    total
  evaluate   13.44     0.15    13.60
      load   13.85     0.14    13.99
        lu    1.73     0.02     1.75
      back   22.89     0.35    23.24
    review    0.00     0.00     0.00
  overhead    0.45     0.17     0.63
     total   52.50     0.94    53.44
```

This shows (crudely) that the overhead of an ordinary private member function call (called from another member function in the same class) is significant here. The cost of a virtual function call is comparable to the cost of an ordinary private member function call.