

Translations of this page are also available in the following languages: [Русский](#).

Circuit Simulation using gEDA and SPICE - HOWTO

Stuart Brorson
Electroniscript, inc.
sdb@electroniscript.com

5th January 2006

abstract

Linux will become an increasingly popular engineering platform in the future. Professional-quality CAD applications for circuit design are becoming available from programmers within the free-software community. For electronics, the gEDA suite is the preferred tool set for circuit design. Analog circuit simulation using SPICE is also now available on Linux. This HOWTO describes the design flow employed to perform SPICE simulations using gEDA tools on Linux.

Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 2 or any later version published by the Free Software Foundation with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts. You may obtain a copy of the GNU Free Documentation License from the Free Software Foundation by visiting their Web site (<http://www.fsf.org/> [<http://www.fsf.org/>]) by writing to: Free Software Foundation, Inc., 51 Franklin Street, Fifth Floor, Boston, MA 02110-1301 USA. ¹⁾

Introduction

Modern engineering is a computer-intensive discipline. Like professionals in other engineering disciplines, electrical engineers and electronics designers are heavy users of all kinds of CAD software, including software for circuit design and simulation, as well as PCB and chip production. Electrical engineers have a special name for the CAD software they use: EDA, which stands for "Electronic Design Automation". Under this rubric fall many different kinds of CAD software. For example, during the front-end stages of a design, an electrical engineer will use a program called a "schematic capture" package to enter his design into the computer. A schematic capture program is basically a specialized drawing program incorporating symbols used in creating a circuit design. After drawing his schematic, the electrical engineer may choose to simulate the behavior of his circuit in order to verify that his design will work as desired. The most popular program for this purpose is SPICE (Simulation Program with Integrated Circuit Emphasis), which was developed at Berkeley starting in the 1970s, and is widely available in multiple forms today. SPICE is now considered a fundamental engineering tool, and is an essential part of the repertoire of most practicing engineers.

The gEDA project [<http://www.geda-project.org/>] is an open-source effort to create a GPL'ed EDA suite running on Linux. GEDA has developed to the point where the power and quality of the tools is quite high; using the gEDA suite, you can now create complex SPICE netlists (files) incorporating vendor model files. You can then use various simulators running on Linux to perform SPICE simulations of your netlists. The purpose of this document is to explain how to use the gEDA tools (typically running on GNU/Linux) to perform SPICE simulations. In particular, this HOWTO documents the usage of **spice-sdb**, which is an advanced backend for the gEDA netlister (**gnetlist**) used to create SPICE netlists. **spice-sdb** is bundled with the gEDA tool suite; if you have installed gEDA, you are ready to create SPICE netlists. This HOWTO also provides advice about using ngspice/tclspice and/or LTSpice to simulate a circuit netlisted with **spice-sdb**.

Target audience for this HOWTO

This HOWTO is not a tutorial about circuit design or SPICE simulation. Rather, it is designed to help the practicing engineer begin using gEDA to perform SPICE simulations on the Linux platform. Therefore, I assume that you are already familiar with electronic design, the mechanics of schematic capture using EDA tools, and SPICE simulation in general. I also assume that you are reasonably familiar with the GNU/Linux operating system and its development environment. Finally, I assume that you have already installed gEDA, and know how to use it. If you need to come up to speed on any of these subjects, further information is available at the following websites:

- The gEDA project: <http://www.geda-project.org/> [<http://www.geda-project.org/>]
- [faq-simulation](#)
- SPICE3 syntax and commands: <http://newton.ex.ac.uk/teaching/CDHW/Electronics2/userguide/> [<http://newton.ex.ac.uk/teaching/CDHW/Electronics2/userguide/>]
- Ngspice: <http://ngspice.sourceforge.net/> [<http://ngspice.sourceforge.net/>]
- Tclspice: <http://tclspice.sourceforge.net/> [<http://tclspice.sourceforge.net/>]
- LTSpice: <http://www.linear.com/software/> [<http://www.linear.com/software/>]
- Spice on Linux resources: <http://www.brorson.com/gEDA/SPICE/> [<http://www.brorson.com/gEDA/SPICE/>]
- Starting with gEDA - slightly out of date, but a great resource http://www-mdp.eng.cam.ac.uk/web/CD/engapps/geda/starting_gEDA_long.pdf [http://www-mdp.eng.cam.ac.uk/web/CD/engapps/geda/starting_gEDA_long.pdf]

Acknowledgements

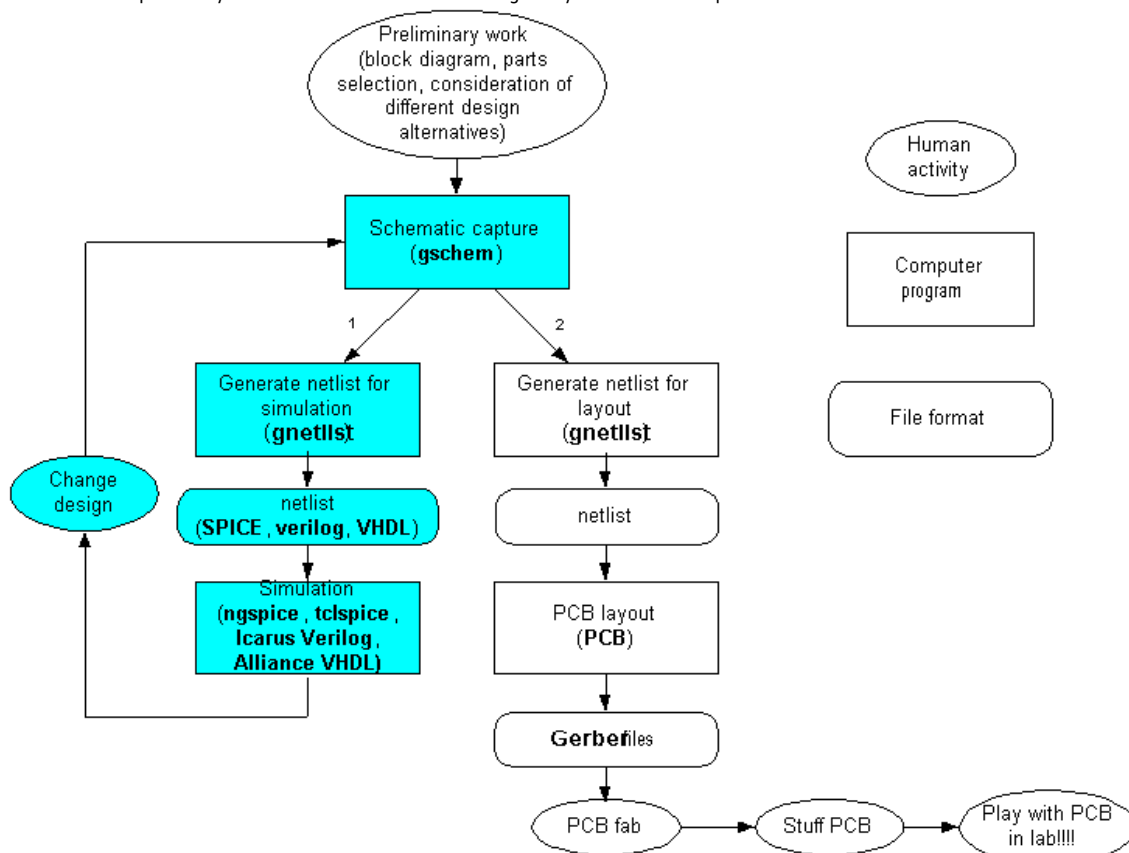
This document does not live in isolation. Several active members of the free EDA community were instrumental in helping me to create this HOWTO. First and foremost, Paolo Nenzi, the author of ngspice, took my original HOWTO and turned it into a LyX

document which I could then make a DocBook. Thanks, Paolo, for helping with this HOWTO, and more importantly, thanks for all the great work on ngspice! Also at the top of the list stands Ales Hvezda, who is the driving force behind the gEDA project. Without Ales, none of this would have been possible; his contribution of **gschem** is invaluable. Thanks, Ales, for creating gEDA and distributing it worldwide under the GPL - you've started a revolution! Stefan Jones deserves a deep thank-you for his work on tclspice, and his gracious support and integration efforts when I submitted patches to the tclspice project. I should also thank W. Kazubski and S. Gieltjes - they wrote the original SPICE netlisters upon which I based gnet-spice-sdb.scm. I also want to thank Ken Healy for contributing the netlist sorting patch, and Peter Kaiser for pushing me to include some features useful for chip simulation. Peter also deserves thanks for writing some of the device-oriented sections of this document. Finally, I should acknowledge the contributions and suggestions I receive from readers of the gEDA-user e-mail list. The beauty of free software is that it encourages collaboration, which means that the end product is greater than what one individual could achieve alone.

The big picture: the design flow in gEDA

In EDA, the concept of "design flow" is important. GEDA is a suite of tools used to do electronic design - it is not a single application. "Design flow" refers to the order in which you use the tools to achieve your goal. Depending upon whether you are doing analog or digital design, designing boards or chips, the type of files required by the manufacturer of your boards, and a number of other factors, you will use different tools from the gEDA suite to achieve your goal.

A flow diagram of the gEDA design flow is shown in the figure below. The diagram shows a simple flow suitable for designing, simulating, and laying out PC boards. As can be seen, the simulation activity (blue blocks) is a loop. That is, you create your design and simulate it repeatedly until it behaves according to your desired specifications.



The design flow used in gEDA. Shown under "simulation" are several different types of simulator available. In this HOWTO, we are interested only in the SPICE variants (e.g. ngspice, tclspice).

Overview of SPICE usage with gEDA

Conceptually, SPICE simulation in gEDA proceeds via the following steps:

1. Creation and gathering of schematic symbols and SPICE model files. Often, the SPICE model files are obtained from your component vendor. You can generally find most models by checking the component vendor's website.
2. Schematic capture using symbols and SPICE models created in step 1.
3. Netlist generation from the schematic created in step 2.
4. SPICE simulation of the circuit described by the netlist created in step 3.

These steps are illustrated by the blue boxes in the flow diagram above.

To create a SPICE netlist, the netlister (**gnetl1st**) iterates through the entire schematic and looks at several parts of each component's symbol in order to create a blob of SPICE code. In general, each component can generate one or more lines of SPICE code. Component information needed by the netlister is held in two places:

1. The symbol itself, in the **device** attribute, which is attached when the symbol is created, and is typically accessed through the symbol editor.

2. In attributes manually attached to the component during schematic capture using **gschem**.

Since there are two places the netlister looks for information, *you must make sure that the required information is available in both places.*

Detailed design/simulation flow summary

The detailed steps required to design and simulate a circuit using gEDA look like this:

1. Schematic symbol creation with correct **device** attribute. (Usually, the symbols have already been created with the correct **device** attribute, but if you are having problems, it doesn't hurt to check them.)
2. Schematic capture using **gschem**.
3. Assignment of SPICE attributes (**value**, **model**, **file**, **type**, etc.) to components using **gschem** or **gattrib**.
4. Assignment of **refdes** using e.g. **refdes_renum**.
5. Creation of netlist using: **gnetlist -g spice-sdb**
6. Check netlist for correctness (manually open and inspect netlist).
7. Run spice using a simulator such as **LTSpice**, **ngspice**, or **tclspice**.
8. Plot/analyze results (often plotting/analysis tools are incorporated in the simulator).
9. If you are not happy with your circuit's performance as revealed by simulation, go back to step 2, fix it using **gschem** and iterate.

The purpose of this HOWTO is to provide the detailed understanding necessary to successfully navigate this process.

Preliminary work: preparing your symbols and SPICE files

When you create schematic symbols for inclusion into your schematic, you must make sure that certain built-in attributes are correctly configured. The steps outlined below are done by editing the symbol itself using the symbol editor in **gschem**, or by editing the symbol file itself using a text editor.

Configuring your symbols

Identifying the component to the netlister

The SPICE netlister can recognize any particular symbol in two ways:

1. The symbol's **device** attribute, and
2. The symbol's **refdes**.

Both of these attributes are attached to the symbol when the symbol is created.

Each symbol has a **device** attribute attached to it. The **device** attribute is the first thing the netlister examines when processing the symbol. There are a number of devices which are native to the netlister, meaning that the netlister knows exactly how to deal with these types of devices. Native device types include **RESISTOR**, **CAPACITOR**, **NPN_TRANSISTOR**, etc. The entire list of native devices is present in [Appendix A -- Native components and their attributes](#).

The **device** attribute is hidden during normal use of **gschem**. Most often, the symbol's creator has already given the symbol the correct **device** attribute. However, because the **device** attribute is hidden from the ordinary user, it can sometimes cause problems with SPICE netlist creation when it is set to an unexpected value. To view the **device** attribute, go into the symbol editor (select the symbol to edit, and do *Hierarchy* → *down symbol*), and turn on invisible attributes (*Edit* → *show/hide inv text*). If the **device** attribute is incorrect, you may change it by editing the symbol itself using a text editor.

If a symbol is not native (i.e. the netlister doesn't recognize it as a built-in type), the netlister relies upon the first letter of the **refdes** to determine how to process the symbol. The **refdes** prefix is also built into the symbol when it is created. Example **refdes** prefixes are **R** for resistors, **C** for capacitors, **Q** for transistors, etc. **refdes** prefixes correct for SPICE are listed in [Appendix A -- Native components and their attributes](#). Note that relying upon the **refdes** to identify the component for SPICE is not foolproof – for example, the netlister cannot distinguish between NPN and PNP transistors based upon the **refdes**. Therefore, it is always best to use a native **device** in your symbols.

Setting the pin order

The netlister emits component pins in the order set by the **pinseq** attribute. Note that this is not the same as the physical pin location. To set the **pinseq** attribute, first determine the pin ordering you want. SPICE uses a specific pin order for many components, including diodes and transistors. For example, a bipolar transistor's pins listed in CBE order. Another example: if your symbol is meant to represent an IC modeled with a vendor's **.subckt**, the order of the connections to the subcircuit is set by the **.subckt** line in the file.

Once you know the order in which to emit the pins, simply set the **pinseq** attribute with the correct order for the part. This will ensure that the part's pins are emitted in the correct order.

Configuring your SPICE files

Files holding complicated SPICE models or other SPICE code may be incorporated into the final SPICE netlist by including appropriate symbols into the schematic. SPICE model files are usually obtained from component vendors. Dealing with these files is straightforward. However, some issues should be kept in mind when preparing models for use during schematic capture:

- It is usually prudent to place these files into a dedicated directory distinct from the symbol directories.
- *Make sure that the SPICE files pin assignments correctly correspond to the pins as defined in the component's symbol!* This is hard to over-emphasize. The order in which pins are listed in a .subckt file do not necessarily correspond to the physical pin ordering of the part. As described above, pins are emitted from the netlist in the order given by the **pinseq** attribute.
- *Make sure that the last character in a SPICE model file is a carriage return.* If no carriage return exists, then the next component listed in the netlist may be placed on the same line as the last line of the SPICE model file.

Creating your circuit: schematic capture

Schematic capture is the process by which one uses a special-purpose drawing program to draw a schematic diagram of the circuit under design. In the gEDA environment, the schematic capture program is called **gschem**. I assume you already know how to use **gschem**. If not, consult the documentation available at the gEDA website: <http://www.geda-project.org/> [<http://www.geda-project.org/>]. For the purposes of creating SPICE netlists, you must use **gschem** to attach attributes to components, and possibly also incorporate other SPICE directives into your netlist. After you are done with schematic capture, you create the SPICE netlist by running gEDA's netlister **gnetlist** on your design.

Gschem attributes for spice netlisting

There are several ways that spice attributes may be associated with a component using **gschem**. The way you choose to do this depends upon many factors, including the type of component, and the size and format of the SPICE model.

Component attributes and meanings

The following attributes are meaningful for SPICE netlisting, and may be attached from within **gschem**:

- **refdes**: The reference designator of the component. Valid values depend upon the component type and are given in [Appendix A](#).
- **value**: For passives, this is the component value. For actives, this is the type (model no.) of the component (e.g. 2N3904, uA741). When a model for an active is instantiated separately from the component itself, the **value** attribute holds the name of the spice model.
- **model**: This holds a one line spice model for the component.
- **file**: This holds the name of a file. Typically, this is a file holding e.g. a SPICE .MODEL, .SUBCKT, or other SPICE code.
- **model-name**: This holds the name of the spice model referred to in a .MODEL or .SUBCKT statement. **model-name** is mainly used to identify the spice model name in the symbol **spice-model-1.sym**. Active components should call out this name in the **device** attribute to associate the component with its particular spice model or subcircuit.
- **type**: This specifies the type of component and is used by spice when interpreting the model parameters. Valid values depend upon the device being modeled.

refdes conventions

As a prerequisite to handling SPICE-related attributes, the SPICE netlister requires that all components must have a **refdes** attached to them. The **refdes** may be attached either by hand (which is laborious), or using the program **refdes_renum** included in the gEDA distribution.

Note that the first letter of the **refdes** must correspond to the appropriate letter for spice simulation. The **refdes** convention is given in [Appendix A](#).

Passives

Basic passives

The most basic components which one encounters in SPICE are passive components like resistors and capacitors which have numeric values, but no other modeling attributes. In this case the following attributes must be filled in:

- **refdes**: The correct **refdes** for the component.
- **value**: For passives, this is the numeric value of the component (e.g. 100pF). For actives, this attribute may be filled in, but if no model attribute is available elsewhere in the schematic, the value is not used (in SPICE netlisting, anyway).

If only a **refdes** and **value** attribute are encountered, the netlister will write a single line into the output file.

Example resistor:

- **refdes** = R2
- **value** = 220

SPICE line generated: **R2 0 4 220**

(note that "0" and "4" correspond to the net nodes connected to the component, and are generated automatically by **gnetlist**.)

Example capacitor:

- **refdes** = C22
- **value** = 1UF

SPICE line generated: **C22 4 23 1UF**

Passives with additional attributes

Oftentimes, passive components have additional attributes attached to them for spice simulation. Examples of such attributes are temperature coefficients (for resistors) and initial conditions (for reactive components). These additional attributes may be incorporated into the SPICE file by simply attaching them to the component's **model** attribute. Specifically, the required attributes are:

- **refdes**: Correct component **refdes**.
- **value**: Numerical component **value**, as always.
- **model**: One line string holding additional parameters, formatted as a valid SPICE string.

This string is placed after the component value in the line generated by **gnetlist**. Therefore, it is important to format the string placed in the **model** line to be valid SPICE code. Otherwise, you will risk causing the SPICE simulator to barf.

Example resistor:

- **refdes** = R5
- **value** = 1MEG
- **model** = TC=0.001,0.015

SPICE line generated: **R5 0 2 1MEG TC=0.001,0.015**

Passives for semiconductor design

The values for resistors and capacitors are often given as dimensions in an ASIC design. SPICE takes from the technology library the typical value per square and calculates the actual value in Ohm or Farad by itself. Therefore the following attributes are required:

- **refdes**: The correct refdes for the component.
- **model-name**: corresponds to the model in the technology library.
- **w, l**: dimensions of the device.

The technology library must be included with an **.include** line in the SPICE input file.

Example semiconductor resistor:

- **refdes** = R6
- **model-name** = rpoly
- **w** = 3u
- **l** = 100u

SPICE line generated: **R6 0 5 rpoly w=3u l=100u**

Example semiconductor resistor model:

- model rpoly R rsh=300

This should be part of the technology library from your ASIC vendor.

Transistors and diodes

Transistors and diodes are generally accompanied by a device-specific model. Each model attempts to capture the detailed nonlinear dynamics of its particular device; otherwise, SPICE simulation is pointless. The SPICE model may be either a short, one-line string of parameters, or a multi-line set of SPICE parameters. A typical one-line parameter string is a short list of parameters describing a small-signal diode. Typical multi-line models come from component vendors, who often provide models for their components in a text file. Since there are two broad formats of SPICE information, there are two approaches to incorporating these parameters into the schematic:

One line string of SPICE parameters

To incorporate a one line string of SPICE parameters into the netlist, the following attributes must be attached to the component:

- **refdes**: Correct component **refdes**.
- **value**: The model number or part number of the component.
- **model-name**: The name you wish to give the SPICE model. This is usually the model number or part number of the component. If you have already attached a **value** attribute to the component, this parameter is optional.
- **model**: One line string holding additional parameters. Do not place the model parameters in parentheses - **gnetlist** will do this for you.

Example diode:

- **refdes** = D5
- **model-name** = 1N1004
- **model** = IS=0.5UA RS=6 BV=5.20

SPICE lines generated: **D5 2 6 1N1004 MODEL 1N1004 D (IS=0.5UA RS=6 BV=5.20)**

SPICE model file

To incorporate a file-full of SPICE parameters into the netlist, the following attributes must be attached to the component:

- **refdes**: Correct component **refdes**.
- **value**: The model number or part number of the component.
- **model-name**: The name you wish to give the SPICE model. This is usually the model number or part number of the component. If you have already attached a **value** attribute to the component, this parameter is optional.
- **file**: The file name of the SPICE model which you wish to incorporate into the netlist. This file name may specify either a relative or an absolute path, but it is probably better to use an absolute path to avoid problems if you ever move your schematic directory.

Note that you need to make sure that the model name held in your SPICE model file is the same as the **value** or **model-name** attributes you attached to the component. It is also a good idea to verify that the pin assignments in the model file correspond to the pin assignments made by the component symbol.

Actives -- integrated circuits

Integrated circuits are incorporated into the netlist similarly to transistors and diodes. As such, you may incorporate the spice information either as a one-line parameter string, or as a model file.

One line string of SPICE parameters

To incorporate a one line string of SPICE parameters into the netlist, the following attributes must be attached to the component:

- **refdes**: Correct component **refdes**.
- **value**: The model number or part number of the component.
- **model-name**: the name you wish to give the SPICE model. This is usually the model number or part number of the component. If you have already attached a **value** attribute to the component, this parameter is optional.
- **model**: One line string holding additional parameters. Do not place the model parameters in parentheses - **gnetlist** will do this for you.

SPICE .MODEL or .SUBCKT file

To incorporate a file-full of SPICE parameters into the netlist, the following attributes must be attached to the component:

- **refdes**: Correct component **refdes**. *Note that if the file holds a .MODEL, the **refdes** should start with U; if the file holds a .SUBCKT, the **refdes** should start with X.* The netlister checks for the file type and tries to "do the right thing", but problems can arise if you don't follow this rule.
- **value**: The model number or part number of the component.
- **model-name**: The name you wish to give the SPICE model. This is usually the model number or part number of the component. If you have already attached a **value** attribute to the component, this parameter is optional.
- **file**: The name of the file holding the SPICE .MODEL or .SUBCKT which you wish to incorporate into the netlist. This file name may specify either a relative or an absolute path, but it is probably better to use an absolute path to avoid problems if you ever move your schematic directory.

Note that you need to make sure that the model name held in your SPICE model file is the same as the **value** or **model-name** attributes you attached to the component. It is also a good idea to verify that the pin assignments in the model file correspond to the pin assignments made by the component symbol.

Independent sources

There are two independent sources: voltage sources and current sources. For incorporation into a SPICE netlist, they both work the same way. To incorporate an independent source into your SPICE netlist, do the following:

1. Place the independent source on your schematic. (Do **Add** → **Component** → **spice** → **<independent source name>.sym**)
2. Double click on the block and add/edit the following attributes:
 - **refdes:** V? or I?
 - **value:** A one line string in SPICE format describing the source.

Dependent sources

There are four dependent sources:

- current controlled voltage source
- current controlled current source
- voltage controlled voltage source
- voltage controlled current source

For incorporation into a SPICE netlist, they all work the same way. To incorporate a dependent source into your SPICE netlist, do the following:

1. Place the dependent source on your schematic. (Do **Add** → **Component** → **spice** → **<dependent source name>.sym**). Appropriate symbol names are abbreviations of the source type (i.e. **ccvs-1.sym**, **cccs-1.sym**, **vcvs-1.sym**, and **vccs-1.sym**)
2. Double click on the block and add/edit the following attributes:
 - **refdes:** H?, F?, E?, or G?. Correct **refdes** prefixes for each source are listed in [Appendix A -- Native components and their attributes](#).
 - **value:** A one line string in SPICE format describing the source. Typically the **value** attribute represents the gain of the source given in appropriate measuring units.

Nullor

A nullor [<http://en.wikipedia.org/wiki/nullor>] is an ideal element composed of a *nullator* and a *norator*. It has zero input resistance and infinite output resistance, as well as infinite current, voltage, transconductance and transimpedance gain and transmission parameters equal to zero. It is a universal active element, that is, ideally it can be used for implementation of any linear and nonlinear functions, if a suitable set of linear and nonlinear passive elements is available. In particular the nullor, resistors and capacitors form a complete set for linear circuits.

Depending on connections of the nullor terminals, the nullor can be used to analyze and synthesize real circuits, which is achieved by replacing of real opamps, current conveyors, amplifying triodes (vacuum tubes and transistors) with the nullor and a small set of passives reflecting their parameters. Nullor based ideal transistors have been successfully used in ac modeling for synthesis of various composite transistor configurations and composite transistors. Nullor based operational amplifier circuits have been used for filter implementations. There are also methods using the nullor for verification, automatic fault diagnosis, automatic biasing analog circuits and so on.

Nullor in SPICE

In the general-purpose circuit simulators which have no nullor model, the nullor element can be modeled using a [dependent source](#) with a large gain, for example 10^9 . The controlled source can be of type VCVS, VCCS, CCVS, or CCCS; the choice depends on the sensitivity issue and what output you want to have. An infinite-gain controlled source of any of the four types of dependent sources is exactly equivalent to a nullor.

A three terminal nullor allows ac modeling of *ideal transistors* and other *triodes*. An *ideal operational amplifier* is realized by a voltage controlled voltage source having an infinite (actually, large enough) gain. *Current conveyor* (CCII) is equivalent to the already mentioned three terminal nullor.

Usually the nullor is used for simulation in AC small-signal analysis (in the frequency domain). When a negative feedback is used, the nullor can be used as an ideal opamp even in transient simulation (see the [example](#) below).

Nullor in gEDA

ngspice and **gnucap** have no special models for the nullor. Therefore to represent the nullor, the **spice-sdb** backend uses VCVS with a high gain.

To incorporate a nullor into your SPICE netlist, do the following:

1. Place the nullor on your schematic. (Do **Add** → **Component** → **spice** → **nullor-1.sym**).
2. Double click on the block and add/edit the following attributes:
 - **refdes:** N?
 - **value:** the voltage gain of the nullor, typically 1000Meg (not needed since gEDA-gaf 1.9.2)

Example: nullor

- **refdes=N1**
- **value=1000Meg**

SPICE lines generated:

```
E_N1 1 2 3 4 1000Meg
IMeasure_N1 3 4 dc 0
IOut_N1 1 2 dc 0
```

This code contains:

- The controlled voltage source E_N1.
- The voltage measuring current source IMeasure_N1.
- The output current source IOut_N1.

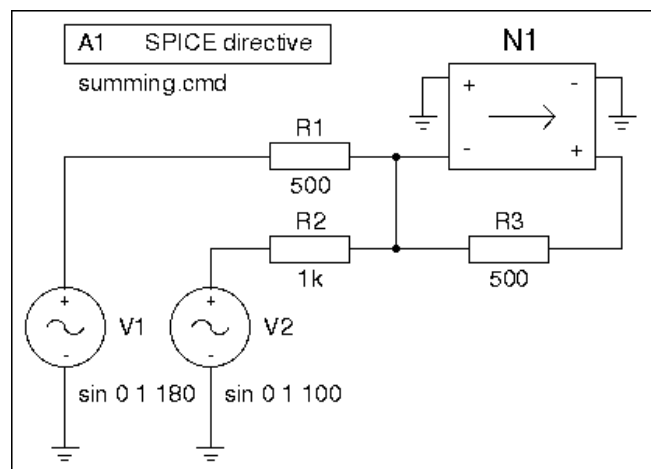
So the nullor in **gnet-spice-sdb** is just a voltage controlled voltage source with two zero current sources connected to its input and output in order to prevent fails of the simulator program when the nullor input or output has no connections to anything.

Please note: after some experiments I (vzh) have found that the presence or absence of those current sources doesn't affect simulation and doesn't solve the mentioned issue in modern versions of **ngspice**. **ngspice** still outputs an error in the case where either the nullor input or its output is floating, that is, not somehow connected to the ground. To avoid such an error you can connect one of those nullor terminals to the ground using a high ohm resistor. **gnucap** always works well; however, in such a case the nodes not connected to the ground will have an arbitrary varying large voltage, so you have to measure not the potential on the separate nullor terminals but the voltage just across the input or the output.

If you want to make your own nullor component (with another type of sensitivity), you can use the controlled source symbol you want (one of **vcvs-1.sym**, **vccs-1.sym**, **cccs-1.sym**, **ccvs-1.sym**) and just change its **value** attribute to a large value, say **1000Meg**.

Example schematic - summing amplifier

In this example, the nullor is used as a model of an ideal opamp.



Schematic file for **gschem**: summing.sch

Command file for simulation in **gnucap** and/or **ngspice**:

summing.cmd

```
.print tran v(nodes)
.tran .0001 1 0 >summing.dat
```

Note the **>summing.dat** thing in the command file. It is ignored by **ngspice** while **gnucap** uses it to output data to the specified file in the batch mode (using shell redirection would output **gnucap** prompt together with data, which is not what we want).

Command line to make netlist (note the **sort_mode** flag, we need it to make **gnucap** work right):

```
gnetlist -g spice-sdb -0 sort_mode -o summing.net summing.sch
```

Command line to simulate using **ngspice**:

```
ngspice -b -r summing.dat summing.net
```

Command line to simulate using **gnucap**:


```
gnucap -b summing.net
```

Command line to see the output waveforms:

```
gwave summing.dat
```

SPICE components

Spice model block

In certain situations, you may wish to embed a spice model block directly into your schematic. This is done when you have several devices with a **value** attribute calling out for a spice model. Depending upon whether the spice block is one line or multi-line, you may embed the code in one of two ways:

One line SPICE model:

1. Place a spice model block on your schematic. (Do *Add* → *Component* → *spice* → *spice-model-1.sym*)
2. Double click on the block and add/edit the following attributes:
 - **refdes**: A?
 - **model-name**: model name (i.e. the model name used in the components being modeled)
 - **type**: One of the valid spice component types defined in the spice spec.
 - **model**: The corresponding one-line spice model

Multi-line SPICE model:

1. Place a spice model block on your schematic. (Do *Add* → *Component* → *spice* → *spice-model-1.sym*)
2. Double click on the block and add/edit the following attributes:
 - **refdes**: A?
 - **model-name**: model name
 - **file**: Name of file holding SPICE model code (i.e. .MODEL or .SUBCKT).

Include block

The include block places a .INCLUDE directive into your netlist.

1. Place a spice model block on your schematic. (Do *Add* → *Component* → *spice* → *spice-include-1.sym*)
2. Double click on the block and add/edit the following attributes:
 - **refdes**: A?
 - **file**: Name of file to include.

SPICE directive block

Placing a SPICE directive block into your schematic creates an arbitrary block of SPICE code in the netlist. The directive may be either statements held in a file, or a one-line string held in the **value** attribute. The netlister will simply dump the contents of the string or the file into your netlist verbatim. Examples of situations where this is useful include:

- .TEMP statement
- .IC statement
- Other SPICE statements for which **gschem** has no symbol.

To place a SPICE directive on your schematic, do:

1. Place a SPICE directive block on your schematic. (Do *Add* → *Component* → *spice* → *spice-directive-1.sym*)
2. Double click on the block and add/edit the following attributes:
 - **refdes**: A?
 - **file**: Name of file to include.

Handling hierarchical models

In SPICE modeling, there are often situations where you wish to create a schematic representation of some particular component as a .SUBCKT, and then embed that component's model in a higher level schematic. A common example might be as follows: You are doing a microwave simulation, and want to use a capacitor model which includes parasitic inductances and resistances, as well as the capacitance. Capacitor manufacturers often supply a printed schematic showing a circuit topology incorporating parasitics, and specify values for the parasitics. You would like to draw the capacitor model using **gschem**, netlist it to create a .SUBCKT, and then use the .SUBCKT to model capacitors in a higher level schematic.

Since this kind of task is very common in SPICE simulation, **gnet-spice-sdb** now supports it (starting with rev 20030331). To create a lower level .SUBCKT and use it in a higher level schematic, do the following:

1. Draw the schematic of the lower level component (e.g. the capacitor + parasitics) using **gschem**.
2. On the lower level schematic, place a **spice-subcircuit-LL** block (**spice-subcircuit-LL-1.sym**). This alerts the netlister that the schematic is a Lower Level .SUBCKT. Attach the following attributes to the symbol:
 - **model-name** = cap_with_parasitics
(Of course, "cap_with_parasitics" is the example we use here. Use your own model name in your schematic.) Upon netlisting, this schematic symbol will cause the netlist to insert ".SUBCKT cap_with_parasitics" into the first line of the netlist file.
3. On the lower level schematic, attach a **spice-subcircuit-IO** symbol (**spice-subcircuit-IO-1.sym**) to each IO net (i.e. connection to the upper level). Number the refdeses of the IO symbols in the same order as you would like the IO nets to be listed in the .SUBCKT line in the output file. (i.e. P1 = first, P2 = second, etc.)
4. When you are done with the lower level schematic, netlist it in the usual way. For example, if your schematic is called **cap_with_parasitics.sch**, netlist it by saying:

```
gnetlist -g spice-sdb -o cap_with_parasitics.cir cap_with_parasitics.sch
```

This will dump the SPICE netlist into the file called "**cap_with_parasitics.cir**". Visually inspect the .cir file to make sure that netlisting worked correctly.

5. Next, create a symbol for the upper level schematic which will point to the .SUBCKT. Note that the symbol must have a **refdes** starting with the letter "X". To ensure that this happens, do the following:
 - Use **gschem** to draw the symbol. I usually draw a box around a model symbol to distinguish it from a normal component. Make any other annotations desired.
 - In the symbol, make sure that the pins are ordered identically to the order in which you have placed the pins in the .SUBCKT. This is done by editing the symbol with a text editor and setting the **pinseq** attribute. The netlister will output the pins in the order determined by the **pinseq** attribute.
 - Using a text editor, give the symbol a **device** attribute like "capacitor-model". Do **not** assign the symbol one of the native device types listed in the [appendix](#)! The goal is to create a symbol whose **refdes** starts with "X", and if the **device** is a recognized type, this will not happen.
 - Using a text editor, give the symbol the **refdes** attribute "X?"
6. Create the upper level schematic. Place your newly created symbol on the schematic as many times as required and wire up the schematic in the usual way.
7. To point your symbol to the lower level .SUBCKT, double click on the symbol and set the following attributes:
 - **file** = cap_with_parasitics.cir
 - **model-name** = cap_with_parasitics
as well as any other attributes required (e.g. **refdes**).
8. Now netlist your upper level schematic the usual way. The contents of each .SUBCKT file is dumped into the main netlist. Inspect your netlist visually using a text editor to ensure that it is correct. It is a good idea to pay particular attention to the following:
 - Verify that the ordering of the nets connecting the upper level netlist to the lower level .SUBCKT is correct.
 - Make sure that the upper level model-name and the lower level model name (on the .SUBCKT declaration line) are the same.

Once the netlist is created, you may simulate your design using any SPICE simulator desired. Some simulators running on Linux are covered below.

SPICE netlist generation

Using gnetlist

Once the schematic is captured, a SPICE netlist can be generated running gEDA's command-line program **gnetlist** on the schematic files. **gnetlist** is architected in two sections: a front-end processor written in C which reads in the .sch file and creates from it an internal, generic representation of your design, and a back-end netlister written in SCHEME. Using this architecture, **gnetlist** is highly customizable; different SCHEME backends are used to write out different netlist formats. The beauty of this scheme (pun intended) is that gEDA users can easily write their own netlisters to suit their own applications. The back-end Scheme file which implements advanced SPICE netlisting is called **gnet-spice-sdb.scm**, and it lives in the **\${PREFIX}/geda/share/gEDA/scheme** directory.

gnetlist with **spice-sdb** is invoked from the command line in the following way: **gnetlist [OPTIONS] -g spice-sdb filename1 ... filenameN**. Among other options described in the [gnetlist User Guide](#), **gnetlist** supports using of backend specific options. A backend specific option can be enabled using the **-O OPTION** switch. The following specific options are available with **spice-sdb**:

- **include_mode**: put .INCLUDE <filename> in output file instead of model file's contents
- **embedd_mode**: force embedding of .include file contents
- **nomunge_mode**: do not automatically correct component refdes
- **sort_mode**: sort output netlist

Creating the netlist using gnetlist and spice-sdb

Creating a netlist from a schematic is easy. To generate a SPICE netlist, just do the following:

- Save your schematic to **<filename.sch>**
- Create the SPICE netlist by doing **"gnetlist -g spice-sdb <filename.sch>"**. The output is a netlist held in the file **output.net**. Alternatively, if you wish to give your output file a different name, set the output name using the **-o** switch. For example:

```
gnetlist -g spice-sdb -o amplifier.cir amplifier.sch
```

takes the design schematic called **"amplifier.sch"** and outputs a SPICE netlist named **"amplifier.cir"**.

- Inspect your SPICE netlist using a text editor. Verify that there are no missing attributes or other netlist problems.

Common netlisting problems

The following list attempts to catalog common problems with the netlist and the associated fixes:

- **ERROR_INVALID_PIN:**
This can happen if the symbol's **pinseq** attributes don't start at 1, or have gaps in the numbering. This must be fixed by editing the symbol itself in a text editor.
- **ERROR: In procedure caddr:**
This error is quite common. It usually occurs when you forget to add a mandatory attribute. To rectify the problem, try running gnetlist in verbose mode (**gnetlist -v -g spice-sdb <filename.sch>**). The netlister will stop processing and bomb out at the part with the missing attribute. Having therefore identified the offending part, you can re-open the schematic in **gschem** and fix the attributes.

Finally, remember that it is important to manually inspect your SPICE netlist prior to using it in simulation. Please keep in mind that the netlister is still "beta" quality, and some problems may still exist in netlist generation.

SPICE simulation

There are several options for doing SPICE simulations under GNU/Linux; I will highlight three:

- **LTSpice**, which is a freeware SPICE simulator originally released by Linear Technologies as a component selection/design tool running under Windows. Because its SPICE engine is very fast and powerful, it has become a popular SPICE simulator amongst hobbyists and design engineers who prefer to use free tools. Originally written for Windows, LTSpice has been tweaked to run under GNU/Linux using wine; I recommend using it if you need a robust, professional-quality SPICE simulator.
- **Ngspice**, which is the "official" SPICE simulator of the gEDA suite. Ngspice is a revival of the SPICE 3 code for Linux. It provides a simulation engine, a command-line driven front-end, and the capability to plot simulation results graphically under the X Windows System. Ngspice is Linux-native and open-source. It is the SPICE of choice for those who want to do SPICE simulations easily on Linux, or want to hack and improve SPICE's internals.
- **Tclspice**, is a fork off the ngspice development path. Tclspice is a superset of ngspice which (in theory) exports the SPICE command set to a TCL API, allowing you to embed SPICE analyses into a TCL program. This is useful for automating a design optimization, amongst other things. Tclspice is the simulator to use if you are interested in advanced, scripted design.

There is also a GPL'ed simulator called **gnucap**, which is based upon (or is the descendant of) Al's Circuit Simulator (**ACS**). I haven't used it very much; information about gnucap is therefore TBD.

LTSpice

LTSpice was written by Mike Englehardt and others at Linear Technologies, and is given away by LinearTech as a design aid for engineers wishing to simulate the performance of LinearTech's switch mode power supply controllers. The package incorporates a schematic capture front end, fast and powerful SPICE engine, and the capability for plotting the results of many different types of SPICE analysis. Personally, I think the schematic capture front-end is hard to use and clunky; **gschem** knocks its socks off for ease of use and features. However, the SPICE engine and analysis stuff in LTSpice is simply great.

LTSpice was originally developed to run under Windows, but Mike has tweaked it so that it runs fairly well on GNU/Linux under wine. (Only the help menu system is broken - the rest of the package runs well). Another good feature of LTSpice is that it is well supported - Mike reads the newsgroup **sci.electronics.cad** regularly and is generally happy to help people who experience problems with it. Therefore, despite its Windoze heritage, I recommend LTSpice as a powerful, professional-quality simulation and analysis back end for gEDA.

Installation and configuration of LTSpice

To install and configure LTSpice, do the following:

1. Download and install wine. I have had success using Wine-20030219. Later versions probably also work.

2. Download LTSpice. It is available under <http://www.linear.com/software> [<http://www.linear.com/software>] under the name SwitcherCAD-III.
3. Run the LTSpice installer under wine.

Running LTSpice with gEDA designs

LTSpice can read a file holding a gEDA SPICE netlist. I have had success doing LTSpice simulations in the following way:

1. First of all, make sure that you are logged in as a normal user - Wine doesn't like to run when invoked by root.
2. Create a file in your project directory called "Simulation.cmd". In this file place your spice analysis commands (e.g. .OP, .AC, .DC, etc.)
3. Place a SPICE include block into your schematic. For the file attribute, type in "Simulation.cmd".
4. Netlist your design.
5. Create a link from your netlist **output.net** and a netlist in the directory in which SwCADIII lives. Make the netlist suffix **.cir**. For example:

```
ln -s ${DESIGN_HOME}/output.net ${WINE_HOME}/.wine/fake_windows/Program Files/LTC/SwCADIII/MyDesign.cir
```

6. Run LTSpice: cd into the directory where SwCADIII lives and say

```
wine scad3.exe
```

7. From the SwCADIII GUI, do: **File** → **Open** → (**files of type netlist [.cir]**), and select your file.
8. Run the simulator by clicking on the run button, or doing: **Simulate** → **Run**.
9. Select the variables to graph, and then click OK. SwCADIII does the rest of the work.

Naturally, it is very important to play around with LTSpice to understand how to use it effectively, but the above description should suffice to get you started.

Ngspice

Ngspice was started at the University of Rome "La Sapienza" by Paolo Nenzi as an attempt to create a GPL'ed version of the standard Berkeley SPICE version 3 by re-writing the entire SPICE package. Plans were also laid to create better, more robust computational algorithms for the simulation engine. More information is available at the ngspice website: <http://ngspice.sourceforge.net/> [<http://ngspice.sourceforge.net/>]. In light of his lofty plans, what Paolo did, however, was a little different: He took the SPICE 3 code which had been floating around the internet for many years, refactored it, and hacked the build system so that it would compile using the normal GNU make procedure. This was a major achievement for which Paolo deserves great praise. Unfortunately, from the look of the webpage, development on **ngspice** seems to have ceased at the end of 2001. Indeed, development did slow down considerably after 2001, but recently Paolo has been working on **ngspice** again. He released the latest version, **ngspice-rework-15**, in February 2004. This version is available only on the Sourceforge download page; Paolo hasn't updated the rest of the project's website.

Installation and configuration of ngspice

I generally find it best to download, configure, and compile the source of **ngspice** instead of trying to install a binary package. That's the approach I outline here.

Downloading the source code

Get the latest distribution from: <http://sourceforge.net/projects/ngspice> [<http://sourceforge.net/projects/ngspice>]. Make sure that you get the latest version for best performance and the most features. As of May 2004, the latest release is **ngspice-rework-15**. Install the source in the place you typically put your sources. I like to keep my gEDA sources in a separate directory, for example **/usr/local/geda/sources/ngspice**. You might adopt a similar system.

Extracting the source code

The source code you downloaded is distributed in a "tarball", a compressed archive. You have to extract archived files by doing:

```
user@host:~$ cd <directory where you want to extract the source>
user@host:~sources$ tar -xvzf </path/to/package.tar.gz>
user@host:~sources$ cd <extracted dir>
```

At this point you are in the top level directory of ngspice. Read the usual files, like **README**, and **INSTALL**, to learn about the simulator and the installation process. Reading **NOTES** file is also a good idea; it holds information valuable if you want to hack or debug features present in ngspice.

Configuration and compilation of ngspice.

Ngspice uses the typical "**configure && make && make install**" sequence used by other GNU software. There are numerous configure time options available for ngspice. A complete listing with attendant documentation is TBD; the best way to see them all

is to look at **configure.ac** itself. Many of the configure time options pertain to debugging the simulator, or are to enable experimental analyses. For newbies, three configure time options are worth mentioning:

- **--enable-xspice**: This flag compiles in support for XSpice extensions. These extensions allow you to define devices whose behavior is given by arbitrary “code models”. Arguably, the most important code model is **spice2poly**, which is a model which translates SPICE2 style POLY constructs into an XSpice model usable by SPICE 3.
- **--with-readline**: This flag compiles GNU readline support into **ngspice**, which means that you can use emacs-style key commands, as well as the arrow keys to move around in the command line interface (CLI). Without this feature, the command line interface can be hostile, meaning that if you make a mistake in typing a long command, you have no choice but to type it all over again. Paolo discourages use of the readline feature because it mixes GPL code (readline) with BSD code (**ngspice**), but he left the option open to other to decide for themselves how pure they wanted to be.
- **--prefix**: This flag point to the base directory where you want your binaries to be installed.

Before you run configure, you should check the options you want to include, a brief description is given in appendix TBD. Once ready type:

```
user@host:~sources/<tld>$ ./configure --enable-xspice --with-readline --prefix=/usr/local/geda <other configure options>
```

Of course, “**--prefix=**” should point to the place where you put **your** gEDA stuff. After issuing the command, your simulator is configured and ready to be compiled. Compilation is straightforward:

```
user@host:~sources/<tld>$ make && make install
```

As always, you will probably need to be root in order to install the packages in a public directory, in such case you should do:

```
user@host:~sources/<tld>$ make
user@host:~sources/<tld>$ su -c make install
```

Testing the installation

At this point, you should be able to use ngspice. You can test your installation by trying one of the test circuits held in the tests directory. I recommend running the TransImpedanceAmp test, since it tests the SPICE2 POLY functionality.

Using ngspice

Running ngspice is very simple. Just issue the command:

```
user@host:~$ ngspice filename.net
```

at the unix command prompt, and ngspice will load the SPICE netlist called **filename.net** into its workspace, and leave you at an ngspice command prompt. You can run the simulator by saying “run”. Your results will be stored in SPICE vectors for later printing or plotting. The command set available to you is documented at: <http://newton.ex.ac.uk/teaching/CDHW/Electronics2/userguide/sec5.html#5> [<http://newton.ex.ac.uk/teaching/CDHW/Electronics2/userguide/sec5.html#5>].

To make use of the SPICE2 POLY codemodel, you need to load it into **ngspice before** you load your netlist. (If you load it after loading your netlist, POLYs in your netlist are not translated, and therefore won't be simulated correctly.) To load the codemodel, just say:

```
codemodel /usr/local/geda/lib/spice/spice2poly.cm
```

(or wherever you put your codemodels) at the ngspice prompt. Note that you must provide the **absolute path** to the location of the codemodel; ngspice isn't smart enough to look for it in any default locations. (Also note that you should specify the location where **spice2poly.cm** lives on your machine; the path above is for mine.)

A better way to read in the **spice2poly** codemodel is to include it in the ngspice initialization file, **spinit**. The initialization file lives in the directory **/usr/local/geda/share/ng-spice-rework/scripts** (or where ever you placed your gEDA installation). Other ngspice customizations may also be placed into the **spinit** file.

Tclspice

While the main branch of ngspice development hibernated in 2002, some friendly people at MultiGig Ltd. (<http://www.multigig.com/> [<http://www.multigig.com/>]) were busy developing a branch of ngspice which they called **tclspice**. Tclspice is a superset of ngspice in which much of the SPICE command set is exported as an API to TCL. The purpose of this is to facilitate scripting of SPICE analyses. This is a very powerful tool: With tclspice you can write a TCL script which runs a loop, tweaks component values, runs an analysis, and then evaluates the circuit performance with the tweaked components before looping again. Obviously, this ability can be used to perform automated, multi-dimensional circuit optimization. When complete, tclspice might possibly become a “killer-app” for open-source EDA.

Downloading, installing, and building tclspice

Tclspice's project homepage is at: <http://tclspice.sourceforge.net/> [<http://tclspice.sourceforge.net/>]. The tclspice source lives at <http://sourceforge.net/projects/tclspice> [<http://sourceforge.net/projects/tclspice>]. Download and installation of tclspice follow the same steps as those detailed for ngspice above. Since tclspice is a superset of ngspice, you can install ngspice alone from the tclspice sources if desired. To build the entire package requires a couple of extra steps. Here, I present a series of steps which will build both ngspice (the stand-alone, CLI driven program) and the TCL API from the tclspice source.

Before building tclspice, you need to have the following packages already installed:

- TclX (tclx8.3.5 works for me.)
- tclreadline (tclreadline-2.1.0 works for me.)
- BLT for TCL (blt2.4z works for me.)
- TCL/Tk (8.4.3. works for me)

If you don't have these packages already on your Linux box, you need to get and build them. Note that building TclX requires having the sources for TCL and Tk, so you will also need to get those sources if you don't have them installed already. I am running successfully with TCL/Tk 8.4.3, although 8.3.X versions are also supposed to work. Also, if you want to run spice in the background you need to recompile TCL and Tk to enable thread support if they haven't got it enabled already (redhat packages haven't).

Assuming you have downloaded and installed the additional packages mentioned above, the following steps will build both ngspice and the TCL API on your machine:

```
user@host:~sources/<tld>$ ./configure --enable-xspice --with-readline --prefix=/usr/local/geda
user@host:~sources/<tld>$ make && make install (this makes and installs regular old ngspice)
user@host:~sources/<tld>$ ./configure --enable-xspice --prefix=/usr/local/geda --enable-tcl --enable-experimental --disable-shared
user@host:~sources/<tld>$ make tcl && make install-tcl
```

As always, you will probably need to be root in order to install the packages in a public directory, in such case you should do:

```
user@host:~sources/<tld>$ su -c make install
user@host:~sources/<tld>$ su -c make install-tcl
```

to install your packages. Now you will be ready to write TCL scripts which incorporate SPICE commands. Information about using tclspice is given below. Finally, if you are interested in hacking tclspice (or even if you are not), it's a good idea to read the **NOTES** file living in the top source directory for a couple of useful pointers.

Use of tclspice

Tclspice is designed to export SPICE commands to TCL programs. To use tclspice, you just need to say “**package require spice**” at the beginning of your TCL program. Thereafter, to invoke a SPICE command, you just call it in the spice namespace. For example, the following TCL program will read in a SPICE netlist, command a transient analysis, run the simulation, and then plot the voltage observed over time on net Vout:

```
#!/ tclsh
package require spice
spice::codemodel /usr/local/src/tclspice-0.2.12/src/xspice/icm/spice2poly.cm
spice::source netlistname.cir
spice::tran 0.1ns 40ns
spice::run
spice::plot Vout
puts "All done now!"
```

Note that since tclspice doesn't read the ngspice initialization file **spinit**, you will need to put any initialization commands directly into the TCL program. For example, in the above example we read the spice2poly codemodel directly into the workspace. Many other commands are also available; the entire tclspice commandset is documented at: http://tclspice.sourceforge.net/docs/tclspice_com.html [http://tclspice.sourceforge.net/docs/tclspice_com.html].

Tclspice problems

A major problem with tclspice (which was inherited from ngspice) is that it leaks memory. Therefore, the time over which you may run a simulation is limited. This means that if you want to do an optimization by looping through a circuit many, many times, you may run out of memory before your program has completed its optimization. This is a known issue with tclspice, and efforts are underway to plug the leaks.

Meanwhile, there are some workarounds which can be used on moderate-sized designs to facilitate long optimization runs. One method I have employed is to have the optimizer write its current state into a file after every circuit analysis, and read its starting state from the same file. The optimizer also stores the current list of best components in another file, and reads this file at the start of every run. Then, I have a TCL program called **TaskMgr.tcl** which runs in a loop; at each iteration of the loop it forks a child process to run the optimizer. Meanwhile, the parent process waits for 5 minutes (a heuristically determined time), and then issues a “KILL” signal to the child before looping and starting the optimizer again. This way, the optimizer never runs long enough to consume all the memory in my machine. The **TaskMgr.tcl** program is shown here:

```
#!/ tclsh
```

```
package require Tclx
while {1} {
    set PID [fork]
    if {$PID} {
        # Parent
        after 300000
        puts "About to kill child PID = $PID . . . ."
        kill $PID
        wait $PID
    } else {
        # Child
        source Optimize.tcl
        # If we ever get through this, we can print out the following:
        error "We are done now!!!!!"
    }
}
```

Note that **TaskMgr.tcl** needs the TclX package you already installed to run tclspice. Also, you may want to change the wait time to a different value depending upon the memory and speed of your machine. Finally, the parent has to wait on \$PID because that causes the child process's corpse to be taken off the Linux kernel's task list when it dies. Otherwise, you will end up with a lot of zombie processes lurking around your machine as the optimizer runs - a long optimization could turn your system into "the night of the living dead"!

This method of waiting a specific amount of time for the child process is preferable if a single analysis run takes a relatively short time compared to the time required to eat all memory in the machine. If the analysis time is comparable to the time taken to eat all memory in the machine, a better approach is to have the parent keep track of the analysis state, kick off a single analysis run, and then have the run terminate after every iteration. Whether this is preferable depends upon the size and complexity of your design; you may want to experiment with your analysis to see just how long it takes and how much memory it consumes. I have found that a design comprised of six op amps (with corresponding vendor models) and 50 or so passives will run in under 10 seconds on a PIII 333MHz with 128MB RAM. Therefore, your design must be very big before a single analysis will eat a significant amount of RAM.

Appendix A -- Native components and their attributes

Presented in table 1 are the devices and associated attributes used with spice-sdb. **Bold faced** attributes are **required**, normal typeface attributes are optional. Note that the **device** attribute is invisible, and is normally attached to the symbol when it is created. The other attributes are attached to the symbol during schematic capture using **gschem**.

When dealing with simple actives (diodes, transistors) having SPICE models held in files, you only need to set the **model-name** and **file** attributes; you don't need to set the **model** attribute. However, if your simple active has a one-line SPICE model which you wish to enter directly into the schematic, then set the **model** and **model-name** attributes; you don't need to set the **file** attribute.

Table 1: Attributes required for SPICE netlisting

| device | refdes | value | model | file | model-name | type | Comment |
|---------------------|--------|-------------|----------------------|-----------------|----------------------|------|-----------------------------------|
| RESISTOR | R? | (4) | (2) | - | Name of model | - | (11) |
| CAPACITOR | C? | (4) | (3) | - | Name of model | - | (11) |
| POLARIZED_CAPACITOR | C? | (4) | (3) | - | Name of model | - | (11) |
| INDUCTOR | L? | (4) | (3) | - | Name of model | - | (11) |
| SPICE-ccvs | H? | (5) | - | - | - | - | Current controlled voltage source |
| SPICE-cccs | F? | (5) | - | - | - | - | Current controlled current source |
| SPICE-vcvs | E? | (5) | - | - | - | - | Voltage controlled voltage source |
| SPICE-vccs | G? | (5) | - | - | - | - | Voltage controlled current source |
| SPICE-nullor | N? | (15) | - | - | - | - | |
| DIODE | D? | Part number | One line SPICE model | Model file name | Name of model | - | (12) |
| PMOS_TRANSISTOR | M? | Part number | One line SPICE model | Model file name | Name of model | - | (12) |
| NMOS_TRANSISTOR | M? | Part number | One line SPICE model | Model file name | Name of model | - | (12) |
| PNP_TRANSISTOR | Q? | Part number | One line SPICE model | Model file name | Name of model | - | (12) |
| | | Part | One line SPICE | | | | |

2015/1/11
geda:csygas [gEDA Project Wiki]

| | | | | | | | |
|-------------------|----|-------------|----------------------|--------------------|-----------------|------|----------------------------|
| NPN_TRANSISTOR | Q? | number | model | Model file name | Name of model | - | (12) |
| PFET_TRANSISTOR | J? | Part number | One line SPICE model | Model file name | Name of model | - | (12) |
| NFET_TRANSISTOR | J? | Part number | One line SPICE model | Model file name | Name of model | - | (12) |
| MESFET_TRANSISTOR | B? | Part number | One line SPICE model | Model file name | Name of model | - | (12) |
| IC | U? | Part number | - | .model file name | Name of model | - | For IC with .model file |
| IC | X? | Part number | - | .subckt file name | Name of .subckt | - | For IC with .subckt file |
| model | A? | - | One line SPICE model | .model file name | (9) | (10) | (12) |
| include | A? | - | - | .include file name | - | - | (13) |
| options | A? | (8) | - | - | - | - | (14) |
| directive | A? | (1) | - | - | - | - | (12) |
| VOLTAGE_SOURCE | V? | (6) | - | - | - | - | Independent voltage source |
| CURRENT_SOURCE | I? | (7) | - | - | - | - | Independent current source |

- (1) One line string holding SPICE statements for inclusion in netlist
(2) One line of SPICE model parameters (e.g. IC, etc.)
(3) One line of SPICE model parameters (e.g. IC, POLY, etc.)
(4) Component numeric value
(5) String describing source behavior
(6) One line string holding voltage source behavior
(7) One line string holding current source behavior
(8) line of options to include
(9) Name of model pointed to by other components
(10) Corresponding SPICE model type (valid types given below)
(11) Model parameters are placed inside parentheses after component value
(12) For modeling, one must include either model or file
(13) Places .INCLUDE directive in SPICE netlist
(14) Places .OPTIONS directive in SPICE netlist
(15) A large enough gain value, e.g. 1000Meg; it is not required in the recent geda-gaf versions (since 1.9.2)

“Native to the netlister” means that there is a corresponding blob of scheme code which knows exactly how to handle these components and is guaranteed (almost) to generate correct spice code. Symbols having “device” attributes not on the above list are handled using the scheme function “spice-sdb:write-default-component”, which looks at the refdes of the component to make a decision about how to treat the component. In general, this function will “do the right thing” when generating spice code, but it is not guaranteed. In particular, this function cannot distinguish between N and P type transistors, and will generate an <unknown> type for the .MODEL string in the netlist. This will probably cause your SPICE simulator to barf. Therefore, it is best to make sure that all devices used have the proper “device” attribute.

Appendix B -- Valid "type" values

The “type” attribute is a flag signaling the spice engine the component type, and prepares it to accept model parameters specific to that component type. The following values are valid SPICE “type”s:

Table 2: Valid “type” attributes for components

| Component | “type” | Comment |
|---------------------|--------|---------|
| RESISTOR | RES | |
| CAPACITOR | CAP | |
| POLARIZED_CAPACITOR | CAP | |
| INDUCTOR | IND | |
| DIODE | D | |
| PMOS_TRANSISTOR | PMOS | |
| NMOS_TRANSISTOR | NMOS | |
| PNP_TRANSISTOR | PNP | |
| NPN_TRANSISTOR | NPN | |
| PFET_TRANSISTOR | PJF | |
| NFET_TRANSISTOR | NJF | |
| MESFET_TRANSISTOR | - | |

1) This HOWTO is released under the GNU Free Documentation License thanks to the generosity of Electroniscript, inc. The most recent copy can always be found at <http://www.brorson.com/gEDA/HOWTO/> [<http://www.brorson.com/gEDA/HOWTO/>]

geda/csygas.txt · Last modified: 2014/07/28 08:05 by vzh

Except where otherwise noted, content on this wiki is licensed under the following license: CC Attribution-Share Alike 3.0
Unported [<http://creativecommons.org/licenses/by-sa/3.0/>]