Prefaces

Preface to the first edition

This manual has been assembled from different sources:

1. The spice3f5 manual,
2. the XSPICE user's manual,
3. the CIDER user's manual

and some original material needed to describe the new features and the newly implemented models. This cut and paste approach, while not being orthodox, allowed ngspice to have a full manual in a fraction of the time that writing a completely new text would have required. The use of LaTex and LyX instead of TeXinfo, which was the original encoding for the manual, further helped to reduce the writing effort and improved the quality of the result, at the expense of an on-line version of the manual but, due to the complexity of the software I hardly think that users will ever want to read an on-line text version.

In writing this text I followed the spice3f5 manual, both in the chapter sequence and presentation of material, mostly because that was already the user manual of SPICE.

Ngspice is an open source software, users can download the source code, compile, and run it. This manual has an entire chapter describing program compilation and available options to help users in building ngspice (see Chapt. 32). The source package already comes with all 'safe' options enabled by default, and activating the others can produce unpredictable results and thus is recommended to expert users only. This is the first ngspice manual and I have removed all the historical material that described the differences between ngspice and spice3, since it was of no use for the user and not so useful for the developer who can look for it in the Changelogs of in the revision control system.

I want to acknowledge the work done by Emmanuel Rouat and Arno W. Peters for converting the original spice3f documentation to TEXinfo. Their effort gave ngspice users the only available documentation that described the changes for many years. A good source of ideas for this manual came from the on-line spice3f manual written by Charles D.H. Williams (Spice3f5 User Guide), constantly updated and useful for its many insights.

As always, errors, omissions and unreadable phrases are only my fault.

Paolo Nenzi

Roma, March 24th 2001

Indeed. At the end of the day, this is engineering, and one learns to live within the limitations of the tools.

Kevin Aylward, Warden of the King's Ale

Preface to the current edition (as of Aug 2021)

Due to the wealth of new material and options in ngspice the actual order of chapters has been revised. Several new chapters have been added. The LyX text processor has allowed adding internal cross references. The PDF format has become the standard format for distribution of the manual. There is also a xhtml version available. Within each new ngspice distribution (starting with ngspice-21) a manual edition is provided reflecting the ngspice status at the time of distribution. At the same time, located at ngspice manuals, the manual is constantly updated. Every new ngspice feature should enter this manual as soon as it has been made available in the Git source code master branch.

Holger Vogt

Mülheim, 2021

Acknowledgments

ngspice contributors

Spice3 and CIDER were originally written at The University of California at Berkeley (USA).

XSPICE has been provided by Georgia Institute of Technology, Atlanta (USA).

Since then, there have been many people working on the software, most of them releasing patches to the original code through the Internet.

The following people have contributed in some way:

Vera Albrecht,
Cecil Aswell,
Giles Atkinson,
Giles C. Billingsley,
Phil Barker,
Steven Borley,
Stuart Brorson,
Alessio Cacciatori,
Mansun Chan,
Wayne A. Christopher,
Al Davis,
Glao S. Dezai,
Jon Engelbert,
Daniele Foci,
Noah Friedman,
David A. Gates,
Alan Gillespie,
John Heidemann,
Marcel Hendrix,
Jeffrey M. Hsu,
JianHui Huang,
S. Hwang,
Chris Inbody,
Gordon M. Jacobs,
Min-Chie Jeng,
Beorn Johnson,
Stefan Jones,

Kenneth H. Keller,
Francesco Lannutti,
Robert Larice,
Mathew Lew,
Robert Lindsell,
Weidong Liu,
Kartikeya Mayaram,
Richard D. McRoberts,
Manfred Metzger,
Jim Monte,
Wolfgang Muees,
Paolo Nenzi,
Gary W. Ng,
Hong June Park,
Stefano Perticaroli,
Arno Peters,
Serban-Mihai Popescu,
Georg Post,
Thomas L. Quarles,
Emmanuel Rouat,
Jean-Marc Routure,
Jaijeet S. Roychowdhury,
Lionel Sainte Cluque,
Takayasu Sakurai,
Amakawa Shuhei,
Kanwar Jit Singh,
Bill Swartz,
Hitoshi Tanaka,
Brian Taylor,
Steve Tell,
Andrew Tuckey,
Andreas Unger,
Holger Vogt,
Dietmar Warning,
Michael Widlok,
Charles D.H. Williams,
Antony Wilson,

and many others...

If someone helped in the development and has not been inserted in this list then this omission was unintentional. If you feel you should be on this list then please write to <ngspice-devel@lists.sourceforge.net>. Do not be shy, we would like to make a list as complete as possible.

Chapter 1

Introduction

Ngspice is a general-purpose circuit simulation program for nonlinear and linear analyses. Circuits may contain resistors, capacitors, inductors, mutual inductors, independent or dependent voltage and current sources, loss-less and lossy transmission lines, switches, uniform distributed RC lines, and the five most common semiconductor devices: diodes, BJTs, JFETs, MESFETs, and MOSFETs.

Some introductory remarks on how to use ngspice may be found in Chapt. 21.

Ngspice is an update of Spice3f5, the last Berkeley's release of Spice3 simulator family. Ngspice is being developed to include new features to existing Spice3f5 and to fix its bugs. Improving a complex software like a circuit simulator is a very hard task and, while some improvements have been made, most of the work has been done on bug fixing and code refactoring.

Ngspice has built-in models for the semiconductor devices, and the user need specify only the pertinent model parameter values.

Ngspice supports mixed-level simulation and provides a direct link between technology parameters and circuit performance. A mixed-level circuit and device simulator can provide greater simulation accuracy than a stand-alone circuit or device simulator by numerically modeling the critical devices in a circuit. Compact models can be used for all other devices. The mixed-level extensions to ngspice is CIDER, a mixed-level circuit and device simulator integrated into ngspice code.

Ngspice supports mixed-signal simulation through the integration of XSPICE code. XSPICE software, developed as an extension to Spice3C1 by GeorgiaTech, has been enhanced and ported to ngspice to provide 'board' level and mixed-signal simulation.

The XSPICE extension enables pure digital simulation as well.

New devices can be added to ngspice by several means: behavioral B-, E- or G-sources, the XSPICE code-model interface for C-like device coding, and the ADMS interface based on Verilog-A and XML.

Finally, numerous small bugs have been discovered and fixed, and the program has been ported to a wider variety of computing platforms.

1.1 Simulation Algorithms

Computer-based circuit simulation is often used as a tool by designers, test engineers, and others who want to analyze the operation of a design without examining the physical circuit. Simulation allows you to change quickly the parameters of many of the circuit elements to determine how they affect the circuit response. Often it is difficult or impossible to change these parameters in a physical circuit.

However, to be practical, a simulator must execute in a reasonable amount of time. The key to efficient execution is choosing the proper level of modeling abstraction for a given problem. To support a given modeling abstraction, the simulator must provide appropriate algorithms.

Historically, circuit simulators have supported either an analog simulation algorithm or a digital simulation algorithm. Ngspice inherits the XSPICE framework and supports both analog and digital algorithms and is a 'mixed-mode' simulator.

1.1.1 Analog Simulation

Analog simulation focuses on the linear and non-linear behavior of a circuit over a continuous time or frequency interval. The circuit response is obtained by iteratively solving Kirchhoff's Laws for the circuit at time steps selected to ensure the solution has converged to a stable value and that numerical approximations of integrations are sufficiently accurate. Since Kirchhoff's laws form a set of simultaneous equations, the simulator operates by solving a matrix of equations at each time point. This matrix processing generally results in slower simulation times when compared to digital circuit simulators.

The response of a circuit is a function of the applied sources. Ngspice offers a variety of source types including DC, sine-wave, and pulse. In addition to specifying sources, the user must define the type of simulation to be run. This is termed the 'mode of analysis'. Analysis modes include DC analysis, AC analysis, and transient analysis. For DC analysis, the time-varying behavior of reactive elements is neglected and the simulator calculates the DC solution of the circuit. Swept DC analysis may also be accomplished with ngspice. This is simply the repeated application of DC analysis over a range of DC levels for the input sources. For AC analysis, the simulator determines the response of the circuit, including reactive elements to small-signal sinusoidal inputs over a range of frequencies. The simulator output in this case includes amplitudes and phases as a function of frequency. For transient analysis, the circuit response, including reactive elements, is analyzed to calculate the behavior of the circuit as a function of time.

1.1.2 Device Models for Analog Simulation

There are three models for bipolar junction transistors, all based on the integral-charge model of Gummel and Poon; however, if the Gummel-Poon parameters are not specified, the basic model (BJT) reduces to the simpler Ebers-Moll model. In either case and in either models, charge storage effects, ohmic resistances, and a current-dependent output conductance may be included. The second bipolar model BJT2 adds dc current computation in the substrate diode. The third model (VBIC) contains further enhancements for advanced bipolar devices.

The semiconductor diode model can be used for either junction diodes or Schottky barrier diodes. There are two models for JFET: the first (JFET) is based on the model of Shichman and Hodges, the second (JFET2) is based on the Parker-Skellern model. All the original six MOSFET models are implemented: MOS1 is described by a square-law I-V characteristic, MOS2 [1] is an analytical model, while MOS3 [1] is a semi-empirical model; MOS6 [2] is a simple analytic model accurate in the short channel region; MOS9, is a slightly modified Level 3 MOSFET model - not to confuse with Philips level 9; BSIM 1 [3, 4]; BSIM2 [5] are the old BSIM (Berkeley Short-channel IGFET Model) models. MOS2, MOS3, and BSIM include second-order effects such as channel-length modulation, subthreshold conduction, scattering-limited velocity saturation, small-size effects, and charge controlled capacitances. The recent MOS models for submicron devices are the BSIM3 (Berkeley BSIM3 web page) and BSIM4 (Berkeley BSIM4 web page) models. Silicon-on-insulator MOS transistors are described by the SOI models from the BSIMSOI family (Berkeley BSIMSOI web page) and the STAG [18] one. There is partial support for a couple of HFET models and one model for MESA devices.

1.1.3 Digital Simulation

Digital circuit simulation differs from analog circuit simulation in several respects. A primary difference is that a solution of Kirchhoff's laws is not required. Instead, the simulator must only determine whether a change in the logic state of a node has occurred and propagate this change to connected elements. Such a change is called an 'event'.

When an event occurs, the simulator examines only those circuit elements that are affected by the event. As a result, matrix analysis is not required in digital simulators. By comparison, analog simulators must iteratively solve for the behavior of the entire circuit because of the forward and reverse transmission properties of analog components. This difference results in a considerable computational advantage for digital circuit simulators, which is reflected in the significantly greater speed of digital simulations.

1.1.4 Mixed-Signal Simulation

Modern circuits often contain a mix of analog and digital circuits. To simulate such circuits efficiently and accurately a mix of analog and digital simulation techniques is required. When analog simulation algorithms are combined with digital simulation algorithms, the result is termed 'mixed-mode simulation'.

Two basic methods of implementing mixed-mode simulation used in practice are the 'native mode' and 'glued mode' approaches. Native mode simulators implement both an analog algorithm and a digital algorithm in the same executable. Glued mode simulators actually use two simulators, one of which is analog and the other digital. This type of simulator must define an input/output protocol so that the two executables can communicate with each other effectively. The communication constraints tend to reduce the speed, and sometimes the accuracy, of the complete simulator. On the other hand, the use of a glued mode simulator allows the component models developed for the separate executables to be used without modification.

Ngspice is a native mode simulator providing both analog and event-based simulation in the same executable. The underlying algorithms of ngspice (coming from XSPICE and its Code Model Subsystem) allow use of all the standard SPICE models, provide a pre-defined collection of the most common analog and digital functions, and provide an extensible base on which to build additional models.

1.1.4.1 User-Defined Nodes

Ngspice supports creation of ‘User-Defined Node’ types. User-Defined Node types allow you to specify nodes that propagate data other than voltages, currents, and digital states. Like digital nodes, User-Defined Nodes use event-driven simulation, but the state value may be an arbitrary data type. A simple example application of User-Defined Nodes is the simulation of a digital signal processing filter algorithm. In this application, each node could assume a real or integer value. More complex applications may define types that involve complex data such as digital data vectors or even non-electronic data.

Ngspice digital simulation is actually implemented as a special case of this User-Defined Node capability where the digital state is defined by a data structure that holds a Boolean logic state and a strength value.

1.1.5 Mixed-Level Simulation (Electronic and TCAD)

Ngspice implements mixed-level simulation through the merging of its code with CIDER (details see Chapt. 30).

CIDER is a mixed-level circuit and device simulator that provides a direct link between technology parameters and circuit performance. A mixed-level circuit and device simulator can provide greater simulation accuracy than a stand-alone circuit or device simulator by numerically modeling the critical devices in a circuit. Compact models can be used for noncritical devices.

CIDER couples ngspice to a internal C-based device simulator, thus providing circuit analyses, compact models for semiconductor devices, and an interactive user interface. CIDER provides accurate, one- and two-dimensional numerical device models based on the solution of Poisson’s equation, and the electron and hole current-continuity equations. CIDER incorporates many of the same basic physical models found in the the Stanford two-dimensional device simulator PISCES [PINT85]. Input to CIDER consists of a SPICE-like description of the circuit and its compact models, and PISCES-like descriptions of the structures of numerically modeled devices. As a result, CIDER should seem familiar to designers already accustomed to these two tools.

The CIDER input format has great flexibility and allows increased access to physical model parameters. New physical models have been added to allow simulation of state-of-the-art devices. These include transverse field mobility degradation [GATE90] that is important in scaled-down MOSFETs and a polysilicon model for poly-emitter bipolar transistors. Temperature dependence has been included for most physical models over the range from -50°C to 150°C. The numerical models can be used to simulate all the basic types of semiconductor devices: resistors, MOS capacitors, diodes, BJTs, JFETs and MOSFETs. BJTs and JFETs can be modeled with or without a substrate contact. Support has been added for the management of device internal states. Post-processing of device states can be performed using the control language user interface of ngspice.

Previously computed states can be loaded into the program to provide accurate initial guesses for subsequent analyses.

Details of the basic semiconductor equations and the physical models used by CIDER are not provided in this manual. Unfortunately, no other single source exists that describes all of the relevant background material. Comprehensive reviews of device simulation can be found in [PINT90] and the book [SELB84]. CODECS (predecessor to CIDER) and its inversion-layer mobility model are described in [MAYA88] and [LGATE90], respectively. PISCES and its models are described in [PINT85]. Temperature dependencies for the PISCES models used by CIDER are available in [SOLL90].

For Linux users the cooperation of the TCAD software GSS with ngspice might be of interest, see https://ngspice.sourceforge.io/gss.html. This project is no longer maintained however, but has moved into the Genius simulator, still available as open source cogenda genius.

1.2 Supported Analyses

The ngspice simulator supports the following different types of analysis:

1. DC Analysis (Operating Point and DC Sweep)
2. AC Small-Signal Analysis
3. Transient Analysis
4. Pole-Zero Analysis
5. Small-Signal Distortion Analysis
6. Sensitivity Analysis
7. Noise Analysis

Applications that are exclusively analog can make use of all analysis modes with the exception of Code Model subsystem that do not implements Pole-Zero, Distortion, Sensitivity and Noise analyses. Event-driven applications that include digital and User-Defined Node types may make use of DC (operating point and DC sweep) and Transient only.

In order to understand the relationship between the different analyses and the two underlying simulation algorithms of ngspice, it is important to understand what is meant by each analysis type. This is detailed below.

1.2.1 DC Analysis

The DC analysis portion of ngspice determines the dc operating point of the circuit with inductors shorted and capacitors opened. DC analysis options are specified on the `.DC`, `.TF`, and `.OP` control lines.

DC analysis does not consider any time dependence on any of the sources within the system description. The simulator algorithm subdivides the circuit into those portions that require the analog simulator algorithm and those that require the event-driven algorithm. Each subsystem block is then iterated to solution, with the interfaces between analog nodes and event-driven nodes iterated for consistency across the entire system.

Once stable values are obtained for all nodes in the system, the analysis halts and the results may be displayed or printed out as you request them.

A dc analysis is automatically performed prior to a transient analysis to determine the transient initial conditions, and prior to an ac small-signal analysis to determine the linearized, small-signal models for nonlinear devices. If requested, the DC small-signal value of a transfer function (ratio of output variable to input source), input resistance, and output resistance is also computed as a part of the DC solution. DC analysis can also be used to generate DC transfer curves: a specified independent voltage, current source, resistor or temperature is stepped over a user-specified range and the DC output variables are stored for each sequential source value.

1.2.2 AC Small-Signal Analysis

AC analysis is limited to analog nodes and represents the small signal, sinusoidal solution of the analog system described at a particular frequency or set of frequencies. This analysis is similar to the DC analysis in that it represents the steady-state behavior of the described system with a single input node *at a given set of stimulus frequencies*.

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

1.2.3 Transient Analysis

Transient analysis is an extension of DC analysis to the time domain. A transient analysis first obtains a DC solution to provide a point of departure for simulating time-varying behavior. Once the DC solution is obtained, the time-dependent aspects of the system are reintroduced, and the two simulator algorithms incrementally solve for the time varying behavior of the entire system. Inconsistencies in node values are resolved by the two simulation algorithms such that the time-dependent waveforms created by the analysis are consistent across the entire simulated time interval. Resulting time-varying descriptions of node behavior for the specified time interval are accessible to you.

All sources that are not time dependent (for example, power supplies) are set to their dc value. The transient time interval is specified on a `.TRAN` control line.

1.2.4 Pole-Zero Analysis

Pole-zero analysis in ngspice computes the poles and/or zeros in the small-signal ac transfer function. Ngspice first computes the dc operating point and then determines the linearized, small-signal models for all the nonlinear devices in the circuit. The small-signal circuit model is then used to find the poles and zeros of the transfer function. Two types of transfer functions are allowed: one of the form (output voltage)/(input voltage) and the other of the form (output voltage)/(input current). These two types of transfer functions cover all the cases and one can find the poles/zeros of functions like input/output impedance and voltage gain. The input and output ports are specified as two pairs of nodes. The pole-zero analysis works with resistors, capacitors, inductors, linear-controlled sources, independent sources, BJTs, MOSFETs, JFETs and diodes. Transmission lines are not supported.

The method used in the analysis is a sub-optimal numerical search. For large circuits it may take a considerable time or fail to find all poles and zeros. Please note, that for some circuits, the method becomes “lost” and may find an excessive number of poles or zeros.

1.2.5 Small-Signal Distortion Analysis

Distortion analysis in ngspice computes steady-state harmonic and intermodulation products for small input signal magnitudes. If signals of a single frequency are specified as the input to the circuit, the complex values of the second and third harmonics are determined at every point in the circuit. If there are signals of two frequencies input to the circuit, the analysis finds out the complex values of the circuit variables at the sum and difference of the input frequencies, and at the difference of the smaller frequency from the second harmonic of the larger frequency. Distortion analysis is supported for the following nonlinear devices:

- Diodes (DIO),
- BJT,
- JFET (level 1),
- MOSFETs (levels 1, 2, 3, 9, and BSIM1),
- MESFET (level 1).

All linear devices are automatically supported by distortion analysis. If there are switches present in the circuit, the analysis continues to be accurate provided the switches do not change state under the small excitations used for distortion calculations.

If a device model does not support direct small signal distortion analysis, please use the Fourier of FFT statements and evaluate the output per scripting.

1.2.6 Sensitivity Analysis

Ngspice can calculate either the DC operating-point sensitivity or the AC small-signal sensitivity of an output variable with respect to all circuit variables, including model parameters. Ngspice calculates the difference in an output variable (either a node voltage or a branch current) by perturbing each parameter of each device independently. Since the method is a numerical approximation, the results may demonstrate second order effects in highly sensitive parameters, or may fail to show very low but non-zero sensitivity.

Since each variable is perturbed by a small fraction of its value, zero-valued parameters are not analyzed, reducing what is usually a very large amount of data.

1.2.7 Noise Analysis

Noise analysis in ngspice measures the device-generated noise for a given circuit. When provided with an input source and an output port, the analysis calculates the noise contributions of each device, and each noise generator within each device, as measured as a voltage at the output port. Noise analysis also calculates the equivalent input noise of the circuit, based on the output noise. This is done for every frequency point in a specified range - the calculated value of the noise corresponds to the spectral density of the circuit variable viewed as a stationary Gaussian stochastic process. After calculating the spectral densities, noise analysis integrates these values over the specified frequency range to arrive at the total noise voltage and current over this frequency range. The calculated values correspond to the variance of the circuit variables viewed as stationary Gaussian processes.

1.2.8 Periodic Steady State Analysis

Experimental code.

PSS is a radio frequency periodical large-signal dedicated analysis. The implementation is based on a time domain shooting method that make use of transient analysis. As it is in early development stage, PSS performs analysis only on autonomous circuits, meaning that it is able to predict fundamental frequency and (harmonic) amplitude(s) for oscillators, VCOs, etc.. The algorithm is based on a search of the minimum error vector defined as the difference of RHS vectors between two occurrences of an estimated period. Convergence is reached when the mean of this error vector decreases below a given threshold parameter. Results of PSS are the basis of periodical large-signal analyses like PAC or PNoise.

1.3 Analysis at Different Temperatures

1.3.1 Introduction

Temperature, in ngspice, is a property associated to the entire circuit, rather than an analysis option. Circuit temperature has a default (nominal) value of 27°C (300.15 K) that can be changed using the `TEMP` option in an `.option` control line (see 15.1.1) or by the `.TEMP` line (see 2.12), which has precedence over the `.option TEMP` line. All analyses are, thus, performed at circuit temperature, and if you want to simulate circuit behavior at different temperatures you should prepare a netlist for each temperature.

All input data for ngspice is assumed to have been measured at the circuit nominal temperature. This value can further be overridden for any device that models temperature effects by specifying the `TNOM` parameter on the `.model` itself. Individual instances may further override the circuit temperature through the specification of `TEMP` and `DTEMP` parameters on the instance. The two options are not independent even if you can specify both on the instance line, the `TEMP` option overrides `DTEMP`. The algorithm to compute instance temperature is described below:

Algorithm 1.1 Instance temperature computation

```
IF TEMP is specified THEN
instance_temperature = TEMP
ELSE
instance_temperature = circuit_temperature + DTEMP
END IF
```

Temperature dependent support is provided for all devices except voltage and current sources (either independent and controlled) and BSIM models. BSIM MOSFETs have an alternate temperature dependency scheme that adjusts all of the model parameters before input to ngspice.

For details of the BSIM temperature adjustment, see [6] and [7]. Temperature appears explicitly in the exponential terms of the BJT and diode model equations. In addition, saturation currents have a built-in temperature dependence. The temperature dependence of the saturation current in the BJT models is determined by:

$$I_S(T_1) = I_S(T_0)\left(\frac{T_1}{T_0}\right)^{XTI}\exp\left(\frac{E_g q(T_1 - T_0)}{k(T_1 T_0)}\right) \tag{1.1}$$

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

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

$$B(T_1) = B(T_0)\left(\frac{T_1}{T_0}\right)^{XTB} \tag{1.2}$$

where T_0 and T_1 are in degrees Kelvin, and XTB is a user-supplied model parameter. Temperature effects on beta are carried out by appropriate adjustment to the values of B_F, I_{SE}, B_R, and I_{SC} (SPICE model parameters BF, ISE, BR, and ISC, respectively).

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

$$I_S(T_1) = I_S(T_0)\left(\frac{T_1}{T_0}\right)^{\frac{XTI}{N}}\exp\left(\frac{E_g q(T_1 - T_0)}{Nk(T_1 T_0)}\right) \tag{1.3}$$

where N is the emission coefficient model parameter, and the other symbols have the same meaning as above. Note that for Schottky barrier diodes, the value of the saturation current temperature exponent, XTI, is usually 2. Temperature appears explicitly in the value of junction potential, U (in Ngspice PHI), for all the device models.

The temperature dependence is determined by:

$$U(T) = \frac{kT}{q}\ln\left(\frac{N_a N_d}{N_i(T)^2}\right) \tag{1.4}$$

where k is Boltzmann's constant, q is the electronic charge, N_a is the acceptor impurity density, N_d is the donor impurity density, N_i is the intrinsic carrier concentration, and E_g is the energy gap. Temperature appears explicitly in the value of surface mobility, M_0(or U_0), for the MOSFET model.

The temperature dependence is determined by:

$$M_0\left(T\right) = \frac{M_0\left(T_0\right)}{\left(\frac{T}{T_0}\right)^{1.5}} \tag{1.5}$$

The effects of temperature on resistors, capacitor and inductors is modeled by the formula:

$$R\left(T\right) = R\left(T_0\right)\left[1 + TC_1\left(T - T_0\right) + TC_2\left(T - T_0\right)^2\right] \tag{1.6}$$

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

1.3.2 Controlling the temperature

The default temperature is set to 27 °C.

```
.temp 40
```

will set the overall temperature to 40 °C (2.12). The command

```
.options temp=60
```

will set the overall temperature to 60 °C (15.1.1). Both commands are equivalent, however `.temp` will override `.options temp`.

The temperature of an individual device may be determined by the instance parameters `temp` or `dtemp`.

```
M1 d g s b MOSN temp=35
```

will set the temperature of the specific MOS device to 35 °C.

```
M2 d g s b MOSN dtemp=20
```

will set the temperature of device `M2` at a delta of 20° above the overall temperature.

The temperatures thus set are static throughout the simulation. It is possible, however, to sweep the temperature by a command like

```
.dc temp 25 49 2
```

starting at 25 °C, stopping at 49 °C with a step of 2° (see 15.3.2).

The current overall temperature may be assessed by the variable `TEMPER`, which can be used as part of an equation in B sources (5.1.2) or behavioral E, G, R, L, C sources (e.g. 5.2). A typical example may look like

```
Bt1 1 2 V='5 + TEMPER*TEMPER'
```

The nominal temperature, a reference temperature where device model parameters have been measured, is called `tnom`.

```
.options tnom=25
```

will set the nominal temperature for all devices to 25 °C (15.1.1). `Tnom` sometimes may be set as a model parameter in a `.model` line (3.2.2), depending on the specific class of devices and its model parameter set.

1.4 Convergence

Ngspice uses the Newton-Raphson algorithm to solve nonlinear equations arising from circuit description. The NR algorithm is interactive and terminates when both of the following conditions hold:

1. The nonlinear branch currents converge to within a tolerance of 0.1% or 1 picoamp (1.0e-12 Amp), whichever is larger.
2. The node voltages converge to within a tolerance of 0.1% or 1 microvolt (1.0e-6 Volt), whichever is larger.

1.4.1 Voltage convergence criterion

The algorithm has reached convergence when the difference between the last iteration k and the current one $(k+1)$

$$\left|v_n^{(k+1)} - v_n^{(k)}\right| \leq \texttt{RELTOL}\, v_{n_{max}} + \texttt{VNTOL}, \tag{1.7}$$

where

$$v_{n_{max}} = \max\left(\left|v_n^{(k+1)}\right|, \left|v_n^{(k)}\right|\right). \tag{1.8}$$

The `RELTOL` (RELative TOLerance) parameter, which default value is 10^{-3}, specifies how small the solution update must be, relative to the node voltage, to consider the solution to have converged. The `VNTOL` (absolute convergence) parameter, which has $1\mu V$ as default value, becomes important when node voltages have near zero values. The relative parameter alone, in such case, would need too strict tolerances, perhaps lower than computer round-off error, and thus convergence would never be achieved. `VNTOL` forces the algorithm to consider as converged any node whose solution update is lower than its value.

1.4.2 Current convergence criterion

Ngspice checks the convergence on the non-linear functions that describe the non-linear branches in circuit elements. In semiconductor devices the functions defines currents through the device and thus the name of the criterion.

Ngspice computes the difference between the value of the nonlinear function computed for the last voltage and the linear approximation of the same current computed with the actual voltage

$$\left|\widehat{i_{branch}^{(k+1)}} - i_{branch}^{(k)}\right| \leq \texttt{RELTOL}\, i_{br_{max}} + \texttt{ABSTOL}, \tag{1.9}$$

where

$$i_{br_{max}} = \max\left(\widehat{i_{branch}^{(k+1)}}, i_{branch}^{(k)}\right). \tag{1.10}$$

In the two expressions above, the $\widehat{i_{branch}}$ indicates the linear approximation of the current.

1.4.3 Convergence failure

Although the algorithm used in ngspice has been found to be very reliable, in some cases it fails to converge to a solution. When this failure occurs, the program terminates the job. Failure to converge in dc analysis is usually due to an error in specifying circuit connections, element values, or model parameter values. Regenerative switching circuits or circuits with positive feedback probably will not converge in the dc analysis unless the **`OFF`** option is used for some of the devices in the feedback path, `.nodeset` control line is used to force the circuit to converge to the desired state.

Chapter 2

Circuit Description

2.1 General Structure and Conventions

2.1.1 Input file structure

The circuit to be analyzed is described to ngspice by a set of element instance lines, which define the circuit topology and element instance values, and a set of control lines, which define the model parameters and the run controls. All lines are assembled in an input file to be read by ngspice. Two lines are essential:

- The first line in the input file must be the title, which is the only comment line that does not need any special character in the first place.
- The last line must be `.end`, plus a newline delimiter.

The order of the remaining lines is alomost arbitrary (except, of course, that continuation lines must immediately follow the line being continued, `.subckt` ... `.ends`, `.if` ... `.endif`, or `.control` ... `.endc` have to enclose their specific lines). Leading white spaces in a line are ignored, as well as empty lines.

The lines described in sections 2.1 to 2.12 are typically used in the core of the input file, outside of a `.control` section (see 16.4.3). An exception is the `.include includefile` line (2.7) that may be placed anywhere in the input file. The contents of `includefile` will be inserted exactly in place of the `.include` line.

2.1.2 Syntax check

A very preliminary syntax check has been added to the input parser.

2.1.2.1 Valid utf-8 characters

The input file will be scanned for valid utf-8 characters. If non-valid characters are found, reading the input is stopped.

2.1.2.2 Special characters leading a line

If the first character in a netlist or .control line is one of =[]?()&%$§\"!:, then ngspice replaces it by '*' and issues a warning. Command `set strict_errorhandling` will force ngspice to exit.

2.1.2.3 Dot command couple completion

Check for `.control ... .endc`, `.subckt ... .ends`, `.if ... .endif`.

2.1.3 Circuit elements (device instances)

Each element in the circuit is a device instance specified by an **instance line** that contains:

- the element instance name,
- the circuit nodes to which the element is connected,
- and the values of the parameters that determine the electrical characteristics of the element.

The first letter of the element instance name specifies the element type. The format for the ngspice element types is given in the following manual chapters. In the rest of the manual, the strings `XXXXXXX`, `YYYYYYY`, and `ZZZZZZZ` denote arbitrary alphanumeric strings.

For example, a resistor instance name must begin with the letter `R` and can contain one or more characters. Hence, `R`, `R1`, `RSE`, `ROUT`, and `R3AC2ZY` are valid resistor names. Details of each type of device are supplied in a following section 3. Table 2.1 lists the element types available in ngspice, sorted by their first letter.

First letter	Element description	Comments, links
A	XSPICE code model	12 analog (12.2) digital (12.4) mixed signal (12.3)
B	Behavioral (arbitrary) source	5.1
C	Capacitor	3.3.6
D	Diode	7
E	Voltage-controlled voltage source (VCVS)	linear (4.2.2), non-linear (5.2)
F	Current-controlled current source (CCCs)	linear (4.2.3)
G	Voltage-controlled current source (VCCS)	linear (4.2.1), non-linear (5.3)
H	Current-controlled voltage source (CCVS)	linear (4.2.4)
I	Current source	4.1
J	Junction field effect transistor (JFET)	9
K	Coupled (Mutual) Inductors	3.3.12
L	Inductor	3.3.10
M	Metal oxide field effect transistor (MOSFET)	11 BSIM3 (11.2.10) BSIM4 (11.2.11)
N	Numerical device for GSS	14
O	Lossy transmission line	6.2
P	Coupled multiconductor line (CPL)	6.4.2
Q	Bipolar junction transistor (BJT)	8
R	Resistor	3.3.1
S	Switch (voltage-controlled)	3.3.15
T	Lossless transmission line	6.1
U	Uniformly distributed RC line	6.3*
U	Basic digital building blocks using XSPICE	14*
V	Voltage source	4.1
W	Switch (current-controlled)	3.3.15
X	Subcircuit	2.5.3
Y	Single lossy transmission line (TXL)	6.4.1
Z	Metal semiconductor field effect transistor (MESFET)	10

Table 2.1: ngspice element types

*) For a disambiguation see chapter 14.1.3.

2.1.4 Some naming conventions

2.1.4.1 Lines

Fields on a line are separated by one or more blanks, a comma, an equal (=) sign, or a left or right parenthesis; extra spaces are ignored. A line may be continued by entering a '+' (plus) in column 1 of the following line; ngspice continues reading beginning with column 2. A name field must begin with a letter (A through Z) and cannot contain any delimiters.

2.1.4.2 Numbers

A number field may be an integer field (12, -44), a floating point field (3.14159), either an integer or floating point number followed by an integer exponent (1e-14, 2.65e3), or either an integer or a floating point number followed by one of the following scale factors:

Suffix	Name	Factor
T	Tera	10^{12}
G	Giga	10^{9}
Meg	Mega	10^{6}
K	Kilo	10^{3}
mil	Mil	25.4×10^{-6}
m	milli	10^{-3}
u	micro	10^{-6}
n	nano	10^{-9}
p	pico	10^{-12}
f	femto	10^{-15}
a	atto	10^{-18}

Table 2.2: Ngspice scale factors

2.1.4.3 Letters following a number

Letters immediately following a number that are not scale factors are ignored, and letters immediately following a scale factor are ignored. Hence, 10, 10V, 10Volts, and 10Hz all represent the same number, and M, MA, MSec, and MMhos all represent the same scale factor. Note that 1000, 1000.0, 1000Hz, 1e3, 1.0e3, 1kHz, and 1k all represent the same number. Note that *'M'* or *'m'* denote 'milli', i.e. 10^{-3}. Suffix *meg* has to be used for 10^{6}. If compatibility mode LT (16.14.6) is set, ngspice will accept the RKM notation for entering resistance or capacitance values, e.g. 2K7 or 100R.

2.1.4.4 Node names

Node names may be arbitrary character strings (exceptions see below) and are case insensitive, if ngspice is used in batch mode (16.4.1). If in interactive (16.4.2) or control (16.4.3) mode, node names may either be plain numbers or arbitrary character strings, **not** starting with a number. The following characters = % ( ) , [ ] < > ~ are not allowed in a node name, especially when XSPICE code models are used (they have their special meanings then and act as string delimiters).

2.1.4.5 Ground node

The ground node must be named '`0`' (zero). For compatibility reason `gnd` is accepted as ground node, and will internally be treated as a global node and be converted to '`0`'. If this is not feasible, you may switch the conversion off by setting `set no_auto_gnd` in one of the configuration files spinit or .spiceinit. *Each circuit has to have a ground node (gnd or 0)!* Note the difference in ngspice where the nodes are treated as character strings and not evaluated as numbers, thus '`0`' and `00` are distinct nodes in ngspice but not in SPICE2.

2.1.5 Topological constraints

Ngspice requires that the following topological constraints are satisfied:

- The circuit cannot contain a loop of voltage sources and/or inductors and cannot contain a cut-set of current sources and/or capacitors.
- Each node in the circuit must have a dc path to ground.
- Every node must have at least two connections except for transmission line nodes (to permit unterminated transmission lines) and MOSFET substrate nodes (which have two internal connections anyway).

2.2 Dot commands

This section summarizes all dot commands available in ngspice, with links to their detailed presentation, in alphabetical order. Control section (or interactive) commands are listed and explained in chapter 17.5.

`.AC` start an ac simulation (15.3.1).

`.CONTROL` start a .control section (16.4.3).

`.CSPARAM` define parameter(s) made available in a control section (2.11).

`.DC` start a dc simulation (15.3.2).

`.DISTO` start a distortion analysis simulation (15.3.3).

`.ELSE` conditional branching in the netlist (2.13).

`.ELSEIF` conditional branching in the netlist (2.13).

`.END` end of the netlist (2.3.2).

`.ENDC` end of the .control section (16.4.3).

`.ENDIF` conditional branching in the netlist (2.13).

`.ENDS` end of subcircuit definition (2.5.2).

`.FOUR` Fourier analysis of transient simulation output (15.6.4).

`.FUNC` define a function (2.10).

.GLOBAL define global nodes (2.6).

.IC set initial conditions (15.2.2).

.IF conditional branching in the netlist (2.13).

.INCLUDE include part of the netlist (2.7).

.LIB include a library (2.8).

.MEAS measurements during the simulation (15.4).

.MODEL list of device model parameters (2.4).

.NODESET set initial conditions (15.2.1).

.NOISE start a noise simulation (15.3.4).

.OP start an operating point simulation (15.3.5).

.OPTIONS set simulator options (15.1).

.PARAM define parameter(s) (2.9).

.PLOT printer plot during batch simulation (15.6.3).

.PRINT tabular listing during batch simulation (15.6.2).

.PROBE save device currents, voltages and differential voltages (15.6.5).

.PSS start a periodic steady state analysis (15.3.12).

.PZ start a pole-zero analysis simulation (15.3.6).

.SAVE name simulation result vectors to be saved (15.6.1).

.SENS start a sensitivity analysis (15.3.7).

.SP S parameter analysis (15.3.8).

.SUBCKT start of subcircuit definitions (2.5).

.TEMP set the ciruit temperature (2.12).

.TF start a transfer function analysis (15.3.9).

.TITLE title of the netlist (2.3.1).

.TRAN start a transient simulation (15.3.10).

.WIDTH width of printer plot (15.6.7).

2.3 Basic lines

2.3.1 .TITLE line

Examples:

```
POWER AMPLIFIER CIRCUIT
* additional lines following
*...

Test of CAM cell
* additional lines following
*...
```

The title line must be the first in the input file. Its contents are printed verbatim as the heading for each section of output.

As an alternative, you may place a `.TITLE <any title>` line anywhere in your input deck. The first line of your input deck will be overridden by the contents of this line following the .TITLE statement.

.TITLE line example:

```
******************************
* additional lines following
*...
.TITLE Test of CAM cell
* additional lines following
*...
```

will internally be replaced by

Internal input deck:

```
Test of CAM cell
* additional lines following
*...
*TITLE Test of CAM cell
* additional lines following
*...
```

2.3.2 .END Line

Examples:

```
.end
```

The `.end` line must always be the last in the input file. Note that the period is an integral part of the name.

2.3.3 Comments

General Form:

```
* <any comment>
```

Examples:

```
* RF=1K Gain should be 100
* Check open-loop gain and phase margin
```

The asterisk in the first column indicates that this line is a comment line. Comment lines may be placed anywhere in the circuit description.

2.3.4 End-of-line comments

General Form:

```
<any command> $ <any comment>
<any command> ; <any comment>
```

Examples:

```
RF2=1K $ Gain should be 100
C1=10p ; Check open-loop gain and phase margin
.param n1=1 //new value
```

ngspice supports comments that begin with double characters '`$ `' (dollar plus space) or '`//`'. For readability you should precede each comment character with a space. ngspice will accept the single character '`$`'.

Please note that the '$' character is not a valid end-of-line comment delimiter, if the PSPICE compatibility mode (16.14.5) has been chosen. Then '$' becomes an ordinary character.

2.3.5 Continuation lines

General Form:

```
<any command>
+ <continuation of any command> ; some comment
+ <further continuation of any command>
```

If input lines get overly long, they may be split into two or more lines (e.g. for better readability). Internally they will be merged into a single line. Each follow-up line starts with character '+' plus additional space. Follw-up lines have to follow immediately after each other. End-of-line comments will be ignored. The following lines do not allow using continuation lines: `.title`, `.lib`, and `.include`.

2.4 .MODEL Device Models

General form:

```
.model mname type(pname1=pval1 pname2=pval2 ... )
```

Examples:

```
.model MOD1 npn (bf=50 is=1e-13 vbf=50)
```

Most simple circuit elements typically require only a few parameter values. However, some devices (semiconductor devices in particular) that are included in ngspice require many parameter values. Often, many devices in a circuit are defined by the same set of device model parameters. For these reasons, a set of device model parameters is defined on a separate `.model` line and assigned a unique model name. The device element lines in ngspice then refer to the model name.

For these more complex device types, each device element line contains the device name, the nodes the device is connected to, and the device model name. In addition, other optional parameters may be specified for some devices: geometric factors and an initial condition (see the following section on Transistors (8 to 11) and Diodes (7) for more details). `mname` in the above is the model name, and type is one of the following fifteen types:

Code	Model Type
R	Semiconductor resistor model
C	Semiconductor capacitor model
L	Inductor model
SW	Voltage controlled switch
CSW	Current controlled switch
URC	Uniform distributed RC model
LTRA	Lossy transmission line model
D	Diode model
NPN	NPN BJT model
PNP	PNP BJT model
NJF	N-channel JFET model
PJF	P-channel JFET model
NMOS	N-channel MOSFET model
PMOS	P-channel MOSFET model
NMF	N-channel MESFET model
PMF	P-channel MESFET model
VDMOS	Power MOS model

Table 2.3: Ngspice model types

Parameter values are defined by appending the parameter name followed by an equal sign and the parameter value. Model parameters that are not given a value are assigned the default values given below for each model type. Models are listed in the section on each device along with the description of device element lines. Model parameters and their default values are given in Chapt. 31.

2.5 .SUBCKT Subcircuits

A subcircuit that consists of ngspice elements can be defined and referenced in a fashion similar to device models. Subcircuits are the way ngspice implements hierarchical modeling, but this is not entirely true because each subcircuit instance is flattened during parsing, and thus ngspice is not a hierarchical simulator.

The subcircuit is defined in the input deck by a grouping of element cards delimited by the `.subckt` and the `.ends` cards (or the keywords defined by the `substart` and `subend` options (see 17.7)); the program then automatically inserts the defined group of elements wherever the subcircuit is referenced. Instances of subcircuits within a larger circuit are defined through the use of an instance card that begins with the letter '`X`'. A complete example of all three of these cards follows:

Example:

```
* The following is the instance card:
*
xdiv1 10 7 0 vdivide

* The following are the subcircuit definition cards:
*
.subckt vdivide 1 2 3
r1 1 2 10K
r2 2 3 5K
.ends
```

The above specifies a subcircuit with ports numbered '1', '2' and '3':

- Resistor 'R1' is connected from port '1' to port '2', and has value 10 kOhms.
- Resistor 'R2' is connected from port '2' to port '3', and has value 5 kOhms.

The instance card, when placed in an ngspice deck, will cause subcircuit port '1' to be equated to circuit node '10', while port '2' will be equated to node '7' and port '3' will equated to node '`0`'.

There is no limit on the size or complexity of subcircuits, and subcircuits may contain other subcircuits. An example of subcircuit usage is given in Chapt. 21.6.

2.5.1 .SUBCKT Line

General form:

```
.SUBCKT subnam N1 <N2 N3 ...>
```

Examples:

```
.SUBCKT OPAMP 1 2 3 4
```

A circuit definition is begun with a `.SUBCKT` line. `subnam` is the subcircuit name, and N1, N2, ... are the external nodes, which cannot be zero. The group of element lines that immediately follow the `.SUBCKT` line define the subcircuit. The last line in a subcircuit definition is the `.ENDS` line (see below). Control lines may not appear within a subcircuit definition; however, subcircuit definitions may contain anything else, including other subcircuit definitions, device models, and subcircuit calls (see below). Note that any device models or subcircuit definitions included as part of a subcircuit definition are strictly local (i.e., such models and definitions are not known outside the subcircuit definition). Also, any element nodes not included on the `.SUBCKT` line are strictly local, with the exception of 0 (ground) that is always global. If you use parameters, the `.SUBCKT` line will be extended (see 2.9.3).

2.5.2 .ENDS Line

General form:

```
.ENDS <SUBNAM>
```

Examples:

```
.ENDS OPAMP
```

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

2.5.3 Subcircuit Calls

General form:

```
XYYYYYYY N1 <N2 N3 ...> SUBNAM
```

Examples:

```
X1 2 4 17 3 1 MULTI
```

Subcircuits are used in ngspice by specifying pseudo-elements beginning with the letter X, followed by the circuit nodes to be used in expanding the subcircuit. If you use parameters, the subcircuit call will be modified (see 2.9.3).

2.6 .GLOBAL

General form:

```
.GLOBAL nodename
```

Examples:

```
.GLOBAL gnd vcc
```

Nodes defined in the .GLOBAL statement are available to all circuit and subcircuit blocks independently from any circuit hierarchy. After parsing the circuit, these nodes are accessible from top level.

2.7 .INCLUDE

General form:

```
.INCLUDE filename
```

Examples:

```
.INCLUDE /users/spice/common/bsim3-param.mod
```

Frequently, portions of circuit descriptions will be reused in several input files, particularly with common models and subcircuits. In any ngspice input file, the `.INCLUDE` line may be used to copy some other file as if that second file appeared in place of the `.INCLUDE` line in the original file.

There is no restriction on the file name imposed by ngspice beyond those imposed by the local operating system.

2.8 .LIB

General form:

```
.LIB filename libname
```

Examples:

```
.LIB /users/spice/common/mosfets.lib mos1
```

The `.LIB` statement allows including library descriptions into the input file. Inside the *.lib file a library **libname** will be selected. The statements of each library inside the *.lib file are enclosed in `.LIB libname <...> .ENDL` statements.

If the compatibility mode (16.14) is set to `'ps'` by `set ngbehavior=ps` (17.7) in spinit (16.5) or .spiceinit (16.6), then a simplified syntax `.LIB filename` is available: a warning is issued and filename is simply included as described in Chapt. 2.7.

2.9 .PARAM Parametric netlists

Ngspice allows for the definition of parametric attributes in the netlists. This is an enhancement of the ngspice front-end that adds arithmetic functionality to the circuit description language.

2.9.1 .param line

General form:

```
.param <ident> = <expr>  <ident> = <expr> ...
```

Examples:

```
.param pippo=5
.param po=6 pp=7.8 pap={AGAUSS(pippo, 1, 1.67)}
.param pippp={pippo + pp}
.param p={pp}
.param pop='pp+p'
```

This line assigns numerical values to identifiers. More than one assignment per line is possible using a separating space. Parameter identifier names must begin with an alphabetic character. The other characters must be either alphabetic, a number, or `! # $ % [ ] _` as special characters. The variables **time**, **temper**, and **hertz** (see 5.1.1) are not valid identifier names. Other restrictions on naming conventions apply as well, see 2.9.6.

The `.param` lines inside subcircuits are copied per call, like any other line. All assignments are executed sequentially through the expanded circuit. Before its first use, a parameter name must have been assigned a value. Expressions defining a parameter should be put within braces `{p+p2}`, or alternatively within single quotes `'AGAUSS(pippo, 1, 1.67)'`. An assignment cannot be self-referential, something like `.param pip = 'pip+3'` will not work.

The current ngspice version does not always need quotes or braces in expressions, especially when spaces are used sparingly. However, it is recommended to do so, as the following examples demonstrate.

```
.param a = 123 * 3    b = sqrt(9) $ doesn't work, a <= 123
.param a = '123 * 3'  b = sqrt(9) $ ok.
.param c = a + 123   $ won't work
.param c = 'a + 123' $ ok.
.param c = a+123     $ ok.
```

Parameters may also have string values, but support is limited. String-valued parameters can be defined by `.param` and used in the same ways as numeric parameters. The only operation on string values is concatenation and that is possible only in top-level `.param` assignments.

```
.param str1="first" str2="second"
.param both={str1}" and "str2
```

2.9.2 Brace expressions in circuit elements:

General form:

```
{ <expr> }
```

Examples:

These are allowed in `.model` lines and in device lines. A SPICE number is a floating point number with an optional scaling suffix, immediately glued to the numeric tokens (see Chapt. 2.9.5). Brace expressions ({..}) cannot be used to parameterize node names or parts of names. All identifiers used within an `<expr>` must have known values at the time when the line is evaluated, else an error is flagged.

2.9.3 Subcircuit parameters

General form:

```
.subckt <identn> node node ...  <ident>=<value> <ident>=<value> ..
```

Examples:

```
.subckt myfilter in out rval=100k cval=100nF
```

<identn> is the name of the subcircuit given by the user. **node** is an integer number or an identifier, for one of the external nodes. The first **<ident>=<value>** introduces an optional section of the line. Each **<ident>** is a formal parameter, and each **<value>** is either a SPICE number or a brace expression. Inside the `.subckt` ... `.ends` context, each formal parameter may be used like any identifier that was defined on a `.param` control line. The **<value>** parts are default values of the parameters.

The syntax of a subcircuit call (invocation) is:

General form:

```
X<name> node node ... <identn> <ident>=<value> <ident>=<value> ...
```

Examples:

```
X1 input output myfilter rval=1k
```

Here **<name>** is the symbolic name given to that instance of the subcircuit, **<identn>** is the name of a subcircuit defined beforehand. **node node ...** is the list of actual nodes where the subcircuit is connected. **<value>** is either a SPICE number or a brace expression **{ <expr> }** .

Subcircuit example with parameters:

```
* Param-example
.param amplitude= 1V
*
.subckt myfilter in out rval=100k  cval=100nF
Ra in p1   {2*rval}
Rb p1 out  {2*rval}
C1 p1 0    {2*cval}
Ca in p2   {cval}
Cb p2 out  {cval}
R1 p2 0    {rval}
.ends myfilter
*
X1 input output myfilter rval=1k cval=1n
V1 input 0 AC {amplitude}
.end
```

2.9.4 Symbol scope

All subcircuit and model names are considered global and must be unique. The `.param` symbols that are defined outside of any `.subckt` ... `.ends` section are global. Inside such a section, the pertaining `params:` symbols and any `.param` assignments are considered local: they mask any global identical names, until the `.ends` line is encountered. You cannot reassign to a global number inside a `.subckt`, a local copy is created instead. Scope nesting works up to a level of 10. For example, if the main circuit calls A that has a formal parameter xx, A calls B that has a param. xx, and B calls C that also has a formal param. xx, there will be three versions of 'xx' in the symbol table but only the most local one - belonging to C - is visible.

2.9.5 Syntax of expressions

```
<expr> ( optional parts within [...] )
```

An expression may be one of:

```
<atom> where <atom> is either a spice number or an identifier
<unary-operator> <atom>
<function-name> ( <expr> [ , <expr> ...] )
<atom> <binary-operator> <expr>
( <expr> )
```

As expected, atoms, built-in function calls and stuff within parentheses are evaluated before the other operators. The operators are evaluated following a list of precedence close to the one of the C language. For equal precedence binary ops, evaluation goes left to right. Functions operate on real values only!

Operator	Alias	Precedence	Description
`-`		1	unary -
`!`		1	unary not
`**`	`^`	2	power, like pwr
`*`		3	multiply
`/`		3	divide
`%`		3	modulo
`\`		3	integer divide
`+`		4	add
`-`		4	subtract
`==`		5	equality
`!=`	`<>`	5	non-equal
`<=`		5	less or equal
`>=`		5	greater or equal
`<`		5	less than
`>`		5	greater than
`&&`		6	boolean and
`||`		7	boolean or
`c?x:y`		8	ternary operator

The evaluation of the power functions ** or ^ depends on the compatibility mode (16.14.1) chosen.

Power function source code implementation:

```
compatmode hs: x>0 pow(x, y); x<0 pow(x, round(y)); X=0 0
compatmode lt: x>0 pow(x, y); x<0 pow(x, y) if y is close to intege
```

The number zero is used to represent boolean False. Any other number represents boolean True. The result of logical operators is 1 or 0. An example input file is shown below:

Example input file with logical operators:

```
* Logical operators

v1or    1 0  {1 || 0}
v1and   2 0  {1 && 0}
v1not   3 0  {! 1}
v1mod   4 0  {5 % 3}
v1div   5 0  {5 \ 3}
v0not   6 0  {! 0}

.control
op
print allv
.endc

.end
```

Built-in function	Notes
sqrt(x)	`y = sqrt(x)`
sin(x), cos(x), tan(x)	
sinh(x), cosh(x), tanh(x)	
asin(x), acos(x), atan(x)	
asinh(x), acosh(x), atanh(x)	
arctan(x)	`atan(x)`, kept for compatibility
exp(x)	
ln(x), log(x)	
abs(x)	
nint(x)	Nearest integer, half integers towards even
int(x)	Nearest integer rounded towards 0
floor(x)	Nearest integer rounded towards -∞
ceil(x)	Nearest integer rounded towards +∞
pow(x,y)	x raised to the power of y (pow from C runtime library)
pwr(x,y)	`pow(fabs(x), y)`
min(x, y)	
max(x, y)	
sgn(x)	`1.0 for x > 0, 0.0 for x == 0, -1.0 for x < 0`
ternary_fcn(x, y, z)	`x ? y : z`
gauss(nom, rvar, sigma)	nominal value plus variation drawn from Gaussian distribution with mean 0 and standard deviation rvar (relative to nominal), divided by sigma
agauss(nom, avar, sigma)	nominal value plus variation drawn from Gaussian distribution with mean 0 and standard deviation avar (absolute), divided by sigma
unif(nom, rvar)	nominal value plus relative variation (to nominal) uniformly distributed between +/-rvar
aunif(nom, avar)	nominal value plus absolute variation uniformly distributed between +/-avar
limit(nom, avar)	nominal value +/-avar, depending on random number in [-1, 1[being `> 0` or `< 0`

The scaling suffixes (any decorative alphanumeric string may follow):

suffix	value
g	1e9
meg	1e6
k	1e3
m	1e-3
u	1e-6
n	1e-9
p	1e-12
f	1e-15

Note: there are intentional redundancies in expression syntax, e.g. `x^y` , `x**y` and `pwr(x,y)` all have nearly the same result.

2.9.6 Reserved words

In addition to the above function names and to the verbose operators (`not and or div mod`), other words are reserved and cannot be used as parameter names: `or`, `defined`, `sqr`, `sqrt`, `sin`, `cos`, `exp`, `ln`, `log`, `log10`, `arctan`, `abs`, `pwr`, `time`, `temper`, `hertz`.

2.9.7 A word of caution on the three ngspice expression parsers

The historical parameter notation using `&` as the first character of a line as equivalence to `.param`. is deprecated and will be removed in a coming release.

Confusion may arise in ngspice because of its multiple numerical expression features. The `.param` lines and the brace expressions (see Chapt. 2.10) are evaluated in the front-end, that is, just after the subcircuit expansion. (Technically, the X lines are kept as comments in the expanded circuit so that the actual parameters can be correctly substituted). Therefore, after the netlist expansion and before the internal data setup, all number attributes in the circuit are known constants. However, there are circuit elements in Spice that accept arithmetic expressions *not* evaluated at this point, but only later during circuit analysis. These are the arbitrary current and voltage sources (B-sources, 5), as well as E- and G-sources and R-, L-, or C-devices. The syntactic difference is that 'compile-time' expressions are within braces, but 'run-time' expressions have no braces. To make things more complicated, the back-end ngspice scripting language accepts arithmetic/logic expressions that operate only on its own scalar or vector data sets (17.2). Please see Chapt. 2.14.

It would be desirable to have the same expression syntax, operator and function set, and precedence rules, for the three contexts mentioned above. In the current Numparam implementation, that goal is not achieved.

2.10 .FUNC

This keyword defines a function. The syntax of the expression is the same as for a `.param` (2.9.5).

General form:

```
    .func <ident> { <expr> }
    .func <ident> = { <expr> }
```

Examples:

```
    .func icos(x) {cos(x) - 1}
    .func f(x,y) {x*y}
    .func foo(a,b) = {a + b}
```

`.func` will initiate a replacement operation. After reading the input files, and before parameters are evaluated, all occurrences of the `icos(x)` function will be replaced by `cos(x)-1`. All occurrences of `f(x,y)` will be replaced by `x*y`. Function statements may be nested to a depth of t.b.d..

2.11 .CSPARAM

Create a constant vector (see 17.8.2) from a parameter in `plot` (17.3) `const`.

General form:

```
.csparam <ident> = <expr>
```

Examples:

```
.param pippo=5
.param pp=6
.csparam pippp={pippo + pp}
.param p={pp}
.csparam pap='pp+p'
```

In the example shown, vectors pippp, and pap are added to the constants that already reside in `plot const`, having length one and real values. These vectors are generated during circuit parsing and thus cannot be changed later (same as with ordinary parameters). They may be used in ngspice scripts and `.control` sections (see Chapt. 17).

The use of `.csparam` is still experimental and has to be tested. A simple usage is shown below.

```
* test csparam
.param TEMPS = 27
.csparam newt = {3*TEMPS}
.csparam mytemp = '2 + TEMPS'
.control
echo $&newt $&mytemp
.endc
.end
```

2.12 .TEMP

Sets the circuit temperature in degrees Celsius.

General form:

```
.temp value
```

Examples:

```
.temp 27
```

This card overrides the circuit temperature given in an `.option` line (15.1.1).

2.13 .IF Condition-Controlled Netlist

A simple `.IF-.ELSE(IF)` block allows condition-controlling of the netlist. `boolean expression` is any expression according to Chapt. 2.9.5 that evaluates parameters and returns a boolean 1 or 0. The netlist block in between the **.if** ... **.endif** statements may contain device instances or `.model` cards that are selected according to the logic condition.

General form:

```
.if(boolean expression)
...
.elseif(boolean expression)
...
.else
...
.endif
```

Example 1:

```
* device instance in IF-ELSE block
.param ok=0 ok2=1

v1 1 0 1
R1 1 0 2

.if (ok && ok2)
R11 1 0 2
.else
R11 1 0 0.5   $ <-- selected
.endif
```

Example 2:

```
* .model in IF-ELSE block
.param m0=0 m1=1

M1 1 2 3 4 N1 W=1 L=0.5

.if(m0==1)
.model N1 NMOS level=49 Version=3.1
.elseif(m1==1)
.model N1 NMOS level=49 Version=3.2.4  $ <-- selected
.else
.model N1 NMOS level=49 Version=3.3.0
.endif
```

Nesting of `.IF-.ELSE(IF)-.ENDIF` blocks is possible. Several `.elseif` (but of course only one `.else`) are allowed per block (please see example ngspice/tests/regression/misc/if-elseif.cir). However some restrictions apply, as the following netlist components are *not* supported within the `.IF-.ENDIF` block: `.SUBCKT`, `.INC`, `.LIB`, and `.PARAM`.

2.14 Parameters, functions, expressions, and command scripts

In ngspice there are several ways to describe functional dependencies. In fact there are three independent function parsers, being active before, during, and after the simulation. So it might be due to have a few words on their interdependence.

2.14.1 Parameters

Parameters (Chapt. 2.9.1) and functions, either defined within the `.param` statement or with the `.func` statement (Chapt. 2.10) are evaluated **before** any simulation is started, that is during the setup of the input and the circuit. Therefore these statements may not contain any simulation output (voltage or current vectors), because it is simply not yet available. The syntax is described in Chapt. 2.9.5. During the circuit setup all functions are evaluated, all parameters are replaced by their resulting numerical values. Thus it will not be possible to get feedback from a later stage (during or after simulation) to change any of the parameters.

2.14.2 Nonlinear sources

During the simulation, the B source (Chapt. 5) and their associated E and G sources, as well as some devices (R, C, L) may contain expressions. These expressions may contain parameters from above (evaluated immediately upon ngspice start up), numerical data, predefined functions, but also node voltages and branch currents resulting from the simulation. The source or device values are continuously updated **during** the simulation. Therefore the sources are powerful tools to define non-linear behavior, you may even create new ‘devices’ by yourself. Unfortunately the expression syntax (see Chapt. 5.1) and the predefined functions may deviate from the ones for parameters listed in 2.9.1.

2.14.3 Control commands, Command scripts

Commands, as described in detail in Chapt. 17.5, may be used interactively, but also as a command script enclosed in `.control ... .endc` lines. The scripts may contain expressions (see Chapt. 17.2). The expressions may work upon simulation output vectors (of node voltages, branch currents), as well as upon predefined or user defined vectors and variables, and are invoked **after** the simulation. Parameters from 2.9.1 defined by the `.param` statement are not allowed in these expressions. However you may define such parameters with `.csparam` (2.11). Again the expression syntax (see Chapt. 17.2) will deviate from the one for parameters or B sources listed in 2.9.1 and 5.1.

If you want to use parameters from 2.9.1 inside your control script, you may use `.csparam` (2.11) or apply a trick by defining a voltage source with the parameter as its value, and then have it available as a vector (e.g. after a transient simulation) with a then constant output (the parameter). A feedback from here back into parameters (2.14.1) is never possible. Also you cannot access non-linear sources of the preceding simulation. However you may start a first simulation inside your control script, then evaluate its output using expressions, change some of the element or model parameters with the `alter` and `altermod` statements (see Chapt. 17.5.3) and then automatically start a new simulation.

Expressions and scripting are powerful tools within ngspice, and we will enhance the examples given in Chapt. 21 continuously to describe these features.