XSPICE

SOFTWARE USER'S MANUAL

December 1992

Prepared by:

F.L. Cox, W.B. Kuhn, H.W. Li, J.P. Murray, S.D. Tynor, M.J. Willis

Computer Science and Information Technology Laboratory
Georgia Tech Research Institute
Georgia Institute of Technology
Atlanta, Georgia 30332

XSPICE Simulator
Software User's Manual

Contents

List of Figures

List of Tables

1 Scope

1.1 Identification

This Software User's Manual describes the operation of the XSPICE simulator. XSPICE is an enhanced and extended version of the University of California at Berkeley's SPICE3 analog circuit simulator.

XSPICE was developed under contract to the United States Air Force as part of the Automatic Test Equipment Software Support Environment (ATESSE), version 2.0. The ATESSE is an integrated set of software tools designed to assist engineers who develop software that controls Automatic Test Equipment (ATE) to test and diagnose faults in modern analog and hybrid (analog/digital) systems. XSPICE provides the board-level and system-level simulation and modeling capabilities needed to make detailed analysis of such systems possible.

1.2 System Overview

XSPICE consists of two major components:

- The XSPICE Simulator
- The XSPICE Code Model Subsystem

1.2.1 The XSPICE Simulator

The XSPICE simulator is the main software program that performs mathematical simulation of a circuit specified by you, the user. It takes input in the form of commands and circuit descriptions and produces output data (e.g. voltages, currents, digital states, and waveforms) that describe the circuit's behavior.

Unlike SPICE3, which is designed for analog simulation and is based exclusively on matrix solution techniques, XSPICE includes both analog and event-driven simulation capabilities. Thus, designs that contain significant portions of digital circuitry can be efficiently simulated together with the analog components. XSPICE also includes a new "User-Defined Node" capability that allows event-driven simulations to be carried out with any type of data.

1.2.2 The XSPICE Code Model Subsystem

The second major component of XSPICE, the Code Model Subsystem, provides the tools needed to model the various parts of your system. While SPICE3 is targeted primarily at integrated circuit (IC) analysis, XSPICE includes features to model and simulate board-level and system-level designs as well. The Code Model Subsystem is central to this new capability, providing XSPICE with an extensive set of models to use in designs and allowing you to add your own models to this model set.

The SPICE3 simulator at the core of XSPICE includes built-in models for discrete components commonly found within integrated circuits. These "model primitives" include components such as resistors, capacitors, diodes, and transistors. The XSPICE Code Model Subsystem extends this set of primitives in two ways. First, it provides a library of over 40 additional primitives, including summers, integrators, digital gates, controlled oscillators, s-domain transfer functions, and digital state machines. Second, it provides a set of programming utilities to make it easy for you to create your own models by writing them in the C programming language.

1.2.3 XSPICE User Interfaces

The approach you will use to enter commands and circuit descriptions to XSPICE and to examine output data produced by XSPICE depends on the interface you are using. XSPICE currently supports two different user interfaces:

- The ATESSE Simulator Interface (SI)
- Nutmeg

If you are using the ATESSE system, you will not be communicating directly with the XSPICE simulator. Instead, you will enter circuit descriptions in schematic form and issue

commands by making menu selections from the ATESSE SI. Similarly you will examine results produced by XSPICE through the ATESSE SI. In this case, this document serves as a reference manual for using simulation models supplied with the XSPICE simulator and for creating new models of your own. Please refer to the ATESSE SI Software User's Manual for details on running simulations through this user interface.

If you do not have access to the ATESSE SI software, you can use the XSPICE simulator through the built-in Nutmeg software developed by the University of Calironia at Berkeley as part of the SPICE3 program on which XSPICE is based. Nutmeg is a command-line interface for controlling the simulator and examining results. With this user interface, you create circuit descriptions as text files and enter commands by typing their names and arguments at the XSPICE prompt after invoking the program. Results may be viewed by typing appropriate commands and examining the resulting textual or graphical displays.

1.2.4 XSPICE Top-Level Diagram

A top-level diagram of the XSPICE system outlined in the paragraphs above is shown in Figure 1.1. The XSPICE Simulator is made up of the SPICE3 core, the event-driven algorithm, circuit description syntax parser extensions, code model device routines, the Nutmeg user interface, and interprocess communications code used to integrate XSPICE with the ATESSE SI. The XSPICE Code Model Subsystem consists of the Code Model Toolkit, the Code Model Library, the Node Type Library, and interfaces to User-Defined Code Models and to User-Defined Node Types.

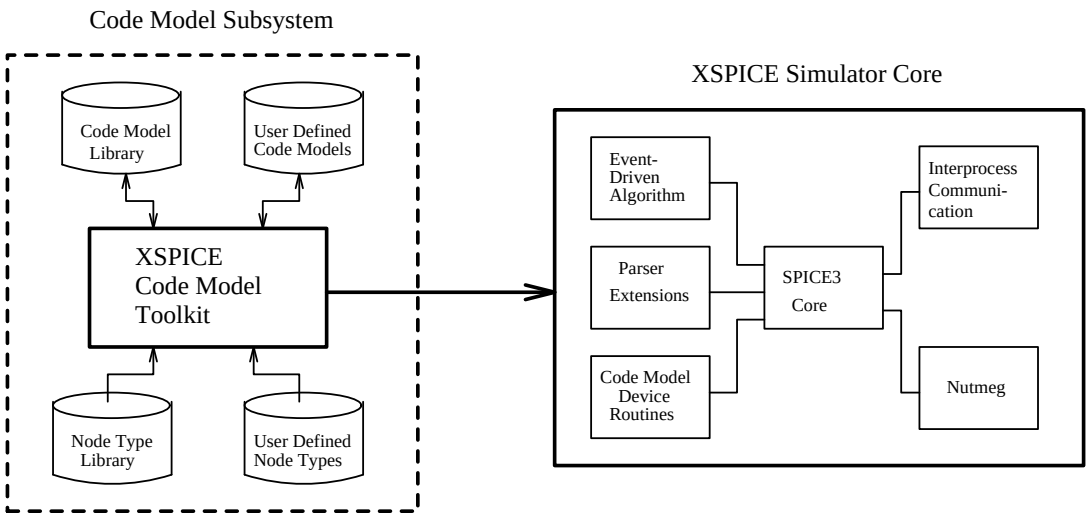

Figure 1.1 XSPICE Top-Level Diagram

1.3 Document Overview

The remainder of this document describes the steps involved in using the various components of XSPICE. Chapter 2 lists specifications, standards and other documents applicable to this manual and to XSPICE. Chapter 3 describes the execution procedures for the simulator, including the circuit description (SPICE deck) syntax [1], and the steps involved in creating Code Models and User-Defined Nodes. This chapter also includes descriptions of the libraries of predefined Code Models and predefined Node Types provided with the simulator. Error messages that may be encountered when using the simulator or associated tools are documented in Chapter 4. Chapter 5 provides a glossary and list of acronyms used in the document.

Finally, a number of appendices are included. These appendices cover requirements for installing XSPICE on your system, type definitions used in creating code models, and tutorial examples of using the simulator through the Nutmeg user inteface.

1.4 Acknowledgments

The XSPICE simulator is based on the SPICE3 program developed by the Electronics Research Laboratory, Department of Electrical Engineering and Computer Sciences, University of California at Berkeley. The authors of XSPICE gratefully acknowledge UC Berkeley's development and distribution of this software, and their licensing policies which promote further improvements to simulation technology.

We also gratefully acknowledge the participation and support of our U.S. Air Force sponsors, the Aeronautical Systems Center and the Warner Robins Air Logistics Command, without which the development of XSPICE would not have been possible.

[1] "SPICE deck" is the historical name for a file used to describe a circuit and the desired analysis to be performed. This document generally uses more modern terms such as "circuit description".

2 Referenced Documents

1. SPICE3C.1 Nutmeg Programmer's Manual, Department of Electrical Engineering and Computer Sciences, University of California, Berkeley, California, April, 1987.

2. SPICE3 Version 3C1 User's Guide, Thomas L. Quarles, Department of Electrical Engineering and Computer Sciences, University of California, Berkeley, California, April, 1989.

3. The C Programming Language, Second Edition, Brian Kernighan and Dennis Ritchie, Prentice-Hall, Englewood Cliffs, New Jersey, 1988.

4. "Code-Level Modeling in XSPICE," F.L. Cox, W.B. Kuhn, J.P. Murray, and S.D. Tynor, published in the Proceedings of the 1992 International Symposium on Circuits and Systems, San Diego, CA, May 1992, vol 2, pp. 871-874.

3 Execution Procedures

This chapter covers operation of the XSPICE simulator and the Code Model Subsystem. It begins with background material on simulation and modeling and then discusses the analysis modes supported in XSPICE and the circuit description syntax used for modeling. Detailed descriptions of the predefined Code Models and Node Types provided in the XSPICE libraries are also included.

3.1 Simulation and Modeling Overview

This section introduces the concepts of circuit simulation and modeling. It is intended primarily for users who have little or no previous experience with circuit simulators, and also for those who have not used circuit simulators recently. However, experienced SPICE users may wish to scan the material presented here since it provides background for the capabilities of XSPICE's new Code Model and User-Defined Node capabilities.

3.1.1 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. Many newer simulators such as XSPICE support two or more such algorithms and are referred to as "mixed-mode" simulators.

3.1.1.1 Analog Simulation

The simulation of analog circuitry is typically accomplished using one of the many versions of a program called "Simulation Program with Integrated Circuit Emphasis", or SPICE. As the name implies, SPICE was originally developed as an aid in the design and analysis of integrated circuit devices. However, its utility in the analysis of discrete circuits, subsystems, and systems quickly became apparent. It is now the standard for simulation of all types of analog circuits.

Analog simulation focuses on the linear and non-linear behavior of a circuit over a continuous time interval. The circuit response is obtained by iteratively solving Kirchoff'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 Kirchoff'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. SPICE offers a variety of source types including DC, sinewave, 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 SPICE. 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.

3.1.1.2 Digital Simulation

Digital circuit simulation differs from analog circuit simulation in several respects. A primary difference is that a solution of Kirchoff'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.

3.1.1.3 Mixed-Mode 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.

XSPICE is a native mode simulator providing both analog and event-based simulation in the same executable. The underlying algorithms of XSPICE and the associated 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.

3.1.1.4 XSPICE User-Defined Nodes

In addition to the Code Model features of XSPICE that support traditional analog and digital modeling, XSPICE 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.

XSPICE's 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.

3.1.2 Describing the Circuit

This section provides an overview of the circuit description syntax expected by the XSPICE simulator.

If you are using the ATESSE SI user interface to the simulator, you do not need to create these circuit descriptions yourself. The ATESSE SI will automatically create a circuit description from your schematic and other information entered through menu selections. However, a general understanding of circuit description syntax will still be helpful to you should you encounter problems with your circuit and need to examine the simulator's error messages, or should you wish to develop your own models.

If you are using the Nutmeg user interface, this section will introduce you to the creation of circuit description input files.

In either case, note that this material is presented in an overview form. Details of circuit description syntax are given later in this chapter and in the references.

3.1.2.1 Example Circuit Description Input

Although different SPICE-based simulators may include various enhancements to the basic version from the University of California at Berkeley, most use a similar approach in describing circuits. This approach involves capturing the information present in a circuit schematic in the form of a text file that follows a defined format. This format requires the assignment of alphanumeric identifiers to each circuit node, the assignment of component identifiers to each circuit device, and the definition of the significant parameters for each device. For example, the circuit description below shows the equivalent input file for the circuit shown in Figure 3.1.

```
Small Signal Amplifier
*
*This circuit simulates a simple small signal amplifier.
*
Vin          Input 0                 0 SIN(0 .1 500Hz)
R_source     Input Amp_In            100
C1           Amp_In 0                1uF
R_Amp_Input  Amp_In 0                1MEG
E1           (Amp_Out 0) (Amp_In 0)  -10
R_Load       Amp_Out 0               1000
```

```
*
.Tran 1.0u 0.01
*
.end
```

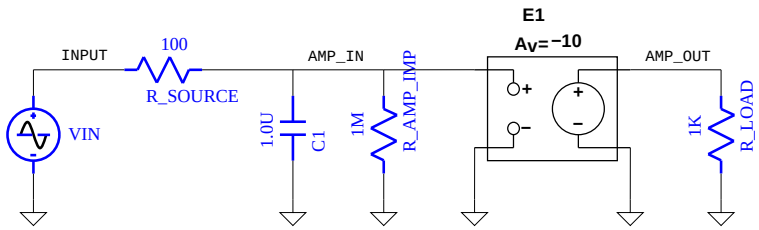

Figure 3.1 Example Circuit 1

This file exhibits many of the most important properties common to all SPICE circuit description files including the following:

- The first line of the file is always interpreted as the title of the circuit. The title may consist of any text string.
- Lines which provide user comments, but no circuit information, are begun by an asterisk.
- A circuit device is specified by a device name, followed by the node(s) to which it is connected, and then by any required parameter information.
- The first character of a device name tells the simulator what kind of device it is (e.g. R = resistor, C = capacitor, E = voltage controlled voltage source).
- Nodes may be labeled with any alphanumeric identifier. The only specific labelling requirement is that 0 must be used for ground.
- A line that begins with a dot is a "control directive". Control directives are used most frequently for specifying the type of analysis the simulator is to carry out.
- An ".end" statement must be included at the end of the file.
- With the exception of the Title and .end statements, the order in which the circuit file is defined is arbitrary.
- All identifiers are case insensitive - the identifier 'npn' is equivalent to 'NPN' and to 'nPn'.
- Spaces and parenthesis are treated as white space.
- Long lines may be continued on a succeeding line by beginning the next line with a '+' in the first column.

In this example, the title of the circuit is 'Small Signal Amplifier'. Three comment lines are included before the actual circuit description begins. The first device in the circuit is voltage source 'Vin', which is connected between node 'Input' and '0' (ground). The parameters after the nodes specify that the source has an initial value of 0, a waveshape of 'SIN', and a DC offset, amplitude, and frequency of 0, .1, and 500 respectively. The next device in the circuit is resistor 'R_Source', which is connected between nodes 'Input' and 'Amp_In', with a value of 100 Ohms. The remaining device lines in the file are interpreted similarly.

The control directive that begins with '.Tran' specifies that the simulator should carry out a simulation using the Transient analysis mode. In this example, the parameters to the transient analysis control directive specify that the maximum timestep allowed is 1 microsecond and that the circuit should be simulated for 0.01 seconds of circuit time.

Other control cards are used for other analysis modes. For example, if a frequency response plot is desired, perhaps to determine the effect of the capacitor in the circuit, the following statement will instruct the simulator to perform a frequency analysis from 100 Hz to 10 MHz in decade intervals with ten points per decade.

```
.ac dec 10 100 10meg
```

To determine the quiescent operating point of the circuit, the following statement may be inserted in the file.

```
.op
```

A fourth analysis type supported by XSPICE is swept DC analysis. An example control statement for the analysis mode is

```
.dc Vin -0.1 0.2 .05
```

This statement specifies a DC sweep which varies the source Vin from -100 millivolts to +200 millivolts in steps of 50 millivolts.

3.1.2.2 Models and Subcircuits

The file discussed in the previous section illustrated the most basic syntax rules of a circuit description file. However, XSPICE (and other SPICE-based simulators) include many other features which allow for accurate modelling of semiconductor devices such as diodes and transistors and for grouping elements of a circuit into a macro or 'subcircuit' description which can be reused throughout a circuit description. For instance, the file shown below is a representation of the schematic shown in Figure 3.2.

```
Small Signal Amplifier with Limit Diodes
*
*This circuit simulates a small signal amplifier
*with a diode limiter.
*
.dc Vin  -1  1 .05
*
Vin        Input 0           DC 0
R_source   Input Amp_In      100
*
D_Neg      0 Amp_In          1n4148
D_Pos      Amp_In 0          1n4148
*
C1         Amp_In 0          1uF
X1         Amp_In 0 Amp_Out  Amplifier
R_Load     Amp_Out 0         1000
*
.model 1n4148 D (is=2.495E-09 rs=4.755E-01 n=1.679E+00
+  tt=3.030E-09 cjo=1.700E-12 vj=1 m=1.959E-01 bv=1.000E+02
+  ibv=1.000E-04)
*
.subckt  Amplifier  Input Common Output
E1         (Output Common) (Input Common)  -10
R_Input    Input Common    1meg
.ends  Amplifier
*
.end
```

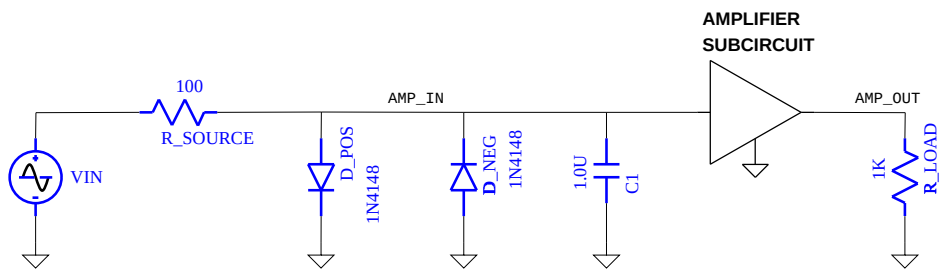

Figure 3.2 Example Circuit 2

This is the same basic circuit as in the initial example, with the addition of two components and some changes to the simulation file. The two diodes have been included to illustrate the use of device models, and the amplifier is implemented with a subcircuit. Additionally, this file shows the use of the swept DC control card.

3.1.2.2.1 Device Models

Device models allow you to specify, when required, many of the parameters of the devices being simulated. In this example, model statements are used to define the silicon diodes. Electrically, the diodes serve to limit the voltage at the amplifier input to values between about +/- 700 millivolts. The diode is simulated by first declaring the "instance" of each diode with a device statement. Instead of attempting to provide parameter information separately for both diodes, the label "1n4148" alerts the simulator that a separate model statement is included in the file which provides the necessary electrical specifications for the device ("1n4148" is the part number for the type of diode the model is meant to simulate).

The model statement that provides this information is:

```
.model 1n4148 D (is=2.495E-09 rs=4.755E-01 n=1.679E+00
+  tt=3.030E-09 cjo=1.700E-12 vj=1 m=1.959E-01 bv=1.000E+02
+  ibv=1.000E-04)
```

The model statement always begins with the string ".model" followed by an identifier and the model type (D for diode, NPN for NPN transistors, etc).

The optional parameters ('is', 'rs', 'n', 'etc.') shown in this example configure the simulator's mathematical model of the diode to match the specific behavior of a particular part (e.g. a 1n4148).

3.1.2.2.2 Subcircuits

In some applications, describing a device by imbedding the required elements in the main circuit file, as is done for the amplifier in Figure 3.1, is not desirable. A hierarchical approach may be taken by using subcircuits. An example of a subcircuit statement is shown in the second circuit file:

```
X1  Amp_In 0 Amp_Out  Amplifier
```

Subcircuits are always identified by a device label beginning with "X". Just as with other devices, all of the connected nodes are specified. Notice, in this example, that three nodes are used. Finally, the name of the subcircuit is specified. Elsewhere in the circuit file, the simulator looks for a statement of the form:

```
.subckt  <Name>  <Node1>  <Node2>  <Node3> ...
```

This statement specifies that the lines that follow are part of the Amplifier subcircuit, and that the three nodes listed are to be treated wherever they occur in the subcircuit definition as referring, respectively, to the nodes on the main circuit from which the subcircuit was called. Normal device, model, and comment statements may then follow. The subcircuit definition is concluded with a statement of the form:

```
.ends  <Name>
```

3.1.2.3 XSPICE Code Models

In the previous example, the specification of the amplifier was accomplished by using a SPICE Voltage Controlled Voltage Source device. This is an idealization of the actual amplifier. Practical amplifiers include numerous non-ideal effects, such as offset error voltages and non-ideal input and output impedances. The accurate simulation of complex, real-world components can lead to cumbersome subcircuit files, long simulation run times, and difficulties in synthesizing the behavior to be modeled from a limited set of internal devices known to the simulator.

To address these problems, XSPICE allows you to create Code Models which simulate complex, non-ideal effects without the need to develop a subcircuit design. For example, the following file provides simulation of the circuit in Figure 3.2, but with the subcircuit amplifier replaced with a Code Model called 'Amp' that models several non-ideal effects including input and output impedance and input offset voltage.

```
Small Signal Amplifier
*
*This circuit simulates a small signal amplifier
*with a diode limiter.
*
.dc Vin  -1  1 .05
*
Vin       Input 0           DC 0
R_source  Input Amp_In      100
*
D_Neg     0 Amp_In          1n4148
D_Pos     Amp_In 0          1n4148
*
C1        Amp_In 0          1uF
A1        Amp_In 0 Amp_Out  Amplifier
R_Load    Amp_Out 0         1000
*
.model 1n4148 D (is=2.495E-09 rs=4.755E-01 n=1.679E+00
```

```
+  tt=3.030E-09 cjo=1.700E-12 vj=1 m=1.959E-01 bv=1.000E+02
+  ibv=1.000E-04)
*
.model Amplifier Amp (gain = -10  in_offset = 1e-3
+  rin = 1meg  rout = 0.4)
*
.end
```

A statement with a device label beginning with "A" alerts the simulator that the device uses a Code Model. The model statement is similar in form to the one used to specify the diode. The model label 'Amp' directs XSPICE to use the code model with that name. Parameter information has been added to specify a gain of -10, an input offset of 1 millivolt, an input impedance of 1 meg ohm, and an output impedance of 0.4 ohm. Subsequent sections of this document detail the steps required to create such a Code Model and include it in the XSPICE simulator.

3.1.2.4 Node Bridge Models

When a mixed-mode simulator is used, some method must be provided for translating data between the different simulation algorithms. XSPICE's Code Model support allows you to develop models that work under the analog simulation algorithm, the event-driven simulation algorithm, or both at once.

In XSPICE, models developed for the express purpose of translating between the different algorithms or between different User-Defined Node types are called "Node Bridge" models. For translations between the built-in analog and digital types, predefined node bridge models are included in the XSPICE Code Model Library.

3.1.2.5 Practical Model Development

In practice, developing models often involves using a combination of SPICE passive devices, device models, subcircuits, and XSPICE Code Models. XSPICE's Code Models may be seen as an extension to the set of device models offered in standard SPICE. The collection of over 40 predefined Code Models included with XSPICE provides you with an enriched set of modeling primitives with which to build subcircuit models. In general, you should first attempt to construct your models from these available primitives. This is often the quickest and easiest method.

If you find that you cannot easily design a subcircuit to accomplish your goal using the available primitives, then you should turn to the code modeling approach. Because they are written in a general purpose programming language (C), code models enable you to simulate virtually any behavior for which you can develop a set of equations or algorithms.

3.2 Supported Analysis Modes

The XSPICE simulator supports four different types of analysis:

1. DC Analysis
2. Swept DC Analysis
3. Transient Analysis
4. AC Analysis

Applications that are exclusively analog can make use of all four analysis modes, whereas event-driven applications that include digital and User-Defined Node types may make use of the first three types only. AC analysis as defined in standard SPICE usage is not applicable to event-driven simulation.

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

DC and Swept DC analyses are steady-state forms of system modeling. There is assumed to be no time dependence on any of the sources within the system description. The simulator algorithm subdivides the circuit into those portions which require the analog simulator algorithm and those which 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. The difference between DC and Swept DC analyses is that the latter involves multiple DC analyses performed by "sweeping" a particular input across a given range of values. Running a swept DC analysis is equivalent to running multiple DC analyses separately, changing the value of an input slightly for each run.

Transient analysis is an extension of DC analysis to the time domain. A transient analysis begins by obtaining 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. Inconsistancies 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.

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 input stimulus is incrementally swept across a given frequency interval, and the magnitude and phase responses of the system are calculated at each frequency. Results of the analysis include magnitude and phase values for each analog node in the system as a result of the single input stimulus.

3.3 Circuit Description Syntax

If you are using the ATESSE system, you will enter most of the information about a circuit through graphical means, and the ATESSE Simulator Interface will automatically handle the translation of the graphical schematic into the XSPICE simulator's circuit description input language. Because of this, the circuit description language used by the simulator will rarely be visible to you unless you are developing models or need to examine error messages output by the simulator.

If you need to debug a simulation, if you are planning to develop your own models, or if you are using the XSPICE simulator through the Nutmeg user interface, you will need to become familiar with the circuit description language.

The previous sections presented example circuit description input files. The following sections provide more detail on XSPICE circuit descriptions with particular emphasis on the syntax for creating and using models. First, the language and syntax of the SPICE3 simulator are described and references to additional information are given. Next, XSPICE extensions to the SPICE3 syntax are detailed. Finally, various enhancements to SPICE operation are discussed including polynomial sources, arbitrary phase sources, supply ramping, matrix conditioning, convergence options, and debugging support.

3.3.1 SPICE3 Syntax

XSPICE is built around the U.C. Berkeley SPICE3C1 simulator and derives much of its SPICE deck syntax from that simulator. Consequently, a thorough discussion of simulator syntax must begin with a review of SPICE3C1 syntax. An in-depth discussion of SPICE3C1 syntax may be found in the SPICE3 Version 3C1 User's Guide (see Referenced Documents). The following is a brief overview of the SPICE3C1 syntax designed to acquaint the new user with its form.[1] Notes are included in this section describing those SPICE3C1 capabilities that are supported by XSPICE and those that are not. If you are in doubt as to whether

[1] Much of the material found in this section was abstracted from the SPICE3 Version 3C1 User's Guide, pages 2-38.

a particular SPICE3C1 feature is supported in the simulator, the following sections should be consulted.

3.3.1.1 Analysis Modes

SPICE3C1 provides several analysis modes. Of these, DC analysis, swept DC analysis, Transient analysis and AC small-signal analysis are currently supported in XSPICE. The DC analysis mode of SPICE3C1 determines the DC operating point of a circuit with inductors shorted and capacitors opened. The swept DC analysis mode of SPICE3C1 allows for multiple DC operating point solutions to be obtained across a range of DC input voltage values. The Transient analysis mode of SPICE3C1 computes transient output values as a function of time over a user-specified time interval. The AC small-signal mode of SPICE3C1 computes AC output values as a function of frequency.

3.3.1.2 Input Format

The input format for the simulator is of the free format type. Fields on a card are separated by white-space characters, which for SPICE3C1 include blanks, commas, equal signs (=), and left or right parentheses. A card[2] may be continued by entering a plus (+) sign in the first column of the following line; SPICE3C1 continues reading beginning with column 2.

Floating point numbers and integers in SPICE3C1 may be scaled by appending one of the following scale factors to the number:

```
T=1E12    G=1E9     MEG=1E6   K=1E3     MIL=25.4E-6
M=1E-3    U=1E-6    N=1E-9    P=1E-12   F=1E-15
```

Warning: You must use "MEG" to indicate 1E6. SPICE is case insensitive and always interprets the single letter "M" or "m" as "milli" (1E-3). Thus, the following is the correct way to specify a one million ohm resistor in a SPICE deck:

```
rlarge 0 13 1.0MEG
```

[2] "card" is the traditional name for a single logical line of information in a SPICE deck. See the glossary for more details.

In contrast, the values of the resistors specified in the following lines will be interpreted identically despite the intent of the user to specify the first resistor as one million ohms and the second as one thousandth of an ohm:

```
rlarge 0 13 1.0M
rtiny 0 14 1.0m
```

3.3.1.3 Title, Comment and .END Cards

The title card is always the first card in a SPICE3C1 input deck. Comment cards begin with an asterisk (*) in the first column of the line and may contain any information after the asterisk. The .END card is always the last card in a SPICE3C1 deck. It defines the end of the input to the simulator.

3.3.1.4 Control Cards

Table 3.1 lists SPICE3C1 control cards supported by XSPICE, along with a description of their required form. For more detail on how each of these control cards affects the simulation, you should refer to the SPICE3 Version 3C1 User's Guide.

Values separated by a forward slash (/) represent groups of values, only one of which may be selected for any single occurrence of the control card.

3.3.1.5 Element Cards

3.3.1.5.1 Element Descriptions

Table 3.2 lists standard SPICE3C1 elements supported by XSPICE, along with a description of their required form. All SPICE3C1 model names mentioned in the SPICE3 Version 3C1 User's Guide are supported by the ATESSE system. These include R, C, URC, D, NPN, PNP, NJF, PJF, NMOS, PMOS, NMF, PMF, SW and CSW models. Only MOSFET level 1, 2 and 3 model types are supported.

3.3.1.5.2 The .MODEL Card

Table 3.2 lists two types of standard SPICE elements: those which may be completely described using one logical element card (R, C, VCVS, etc.), and those which require more than one logical element card (diode, BJT, JFET and MOSFET). The latter are declared through the use of two cards: an instance card and a .MODEL card. This method is generally used by transistors, etc., which require a large number of parameters.

Control Card	Form	Description
.op	.op	Requests an operating point analysis. This is the way you request a DC analysis.
.dc	.dc srcnam start stop incr	Requests a swept-DC analysis with name of source to sweep, starting and stopping values of sweep, and sweep increment.
.tran	.tran tstep tstop	Requests a transient analysis with specified maximum timestep and specified end time.
.ac	.ac dec/oct/lin nd/no/np fstart fstop	Requests an AC analysis. First argument should be dec, oct, or lin to specify type of sweep (dec="decade", oct="octave", lin="linear"). Second argument specifies number of points (nd="# of points per decade", no="# of points per octave", np="# of points in full linear sweep"). Third and fourth arguments specify starting and stopping frequencies of sweep range.
.options	.options opt1 opt2 ... (or opt=optval)	Allows you to set various simulation options

Table 3.1 Standard SPICE3C1 Control Cards

The .MODEL card specifies a set of model parameters that will be used by one or more elements. Individual elements are specified through the use of an instance card (refer to the SPICE3 Version 3C1 User's Guide for details of standard SPICE instance types which require this method of declaring an instance). The form of a .MODEL card is as follows:

.MODEL MODELNAME TYPE(PNAME1=PVAL1 PNAME2=PVAL2...)

The MODELNAME entry on the card must agree with that specified on the instance card that references the model. As a complete example, the following card pair completely describes a SPICE3C1 call to an NPN transistor:

```
Q75A 11 16 13 BJTMOD1
.MODEL BJTMOD1 NPN (IS=1.5E-16 NF=1.2 BF=175 ISE=1E-13)
```

The transistor instance is named "q75a", and it is this name which distinguishes it from another transistor. Node number "11" is the collector node of the transistor, node "16" is the base node and "13" is the emitter node. The parameter names on the .MODEL card

are specific to the SPICE BJT model, and their definition can be found in the SPICE3 Version 3C1 User's Guide. Also in the above, the ordering of the cards is not important. Section 3.3.3.1 describes the use of .MODEL cards for code models. It will be shown that code models use a similar syntax to that used for diodes, BJTs, JFETS, and MOSFETS (i.e. instance and .model cards together describing an element).

Standard SPICE3C1 Elements	
Type	Form
RESISTOR	RXXXXXXX N1 N2 VALUE
CAPACITOR	CXXXXXXX N+ N- VALUE
INDUCTOR	LXXXXXXX N+ N- VALUE
DIODE	DXXXXXXX N+ N- MODELNAME
BIPOLAR TRANSISTOR	QXXXXXXX NC NB NE <NS> MODELNAME1
JUNCTION FET	JXXXXXXX ND NG NS MODELNAME1
MOSFET	MXXXXXXX ND NG NS NB MODELNAME
VOLTAGE SOURCE	VXXXXXXX N+ N- <<DC> DC/TRANSIENT VALUE> <AC <ACMAG <ACPHASE>>>
	VXXXXXXX N+ N- PULSE(VINITIAL, VFINAL, TD, TR, TF, PW, PER)
	VXXXXXXX N+ N- SIN(OFFSET, AMPL, FREQ, TD, DAMPFAC)
	VXXXXXXX N+ N- EXP(VINITIAL, VFINAL, TDRISE, TAURISE, TDFALL, TAUFALL)
	VXXXXXXX N+ N- PWL(T1 V1 <T2 V2 T3 V3...>)
	VXXXXXXX N+ N- SFFM(OFFSET, AMPL, FCARRIER,MODINDEX,FSIGNAL)
CURRENT SOURCE	IXXXXXXX N+ N- <<DC> DC/TRANSIENT VALUE> <AC <ACMAG <ACPHASE>>>
	IXXXXXXX N+ N- PULSE(IINITIAL, IFINAL, TD, TR, TF, PW, PER)
	IXXXXXXX N+ N- SIN(OFFSET, AMPL, FREQ, TD, DAMPFAC)
	IXXXXXXX N+ N- EXP(IINITIAL, IFINAL, TDRISE, TAURISE, TDFALL, TAUFALL)
	IXXXXXXX N+ N- PWL(T1 I1 <T2 I2 T3 I3...>)
	IXXXXXXX N+ N- SFFM(OFFSET, AMPL, FCARRIER, MODINDEX, FSIGNAL)
VCVS	EXXXXXXX N+ N- NC+ NC- VALUE
CCCS	FXXXXXXX N+ N- VNAM VALUE
VCCS	GXXXXXXX N+ N- NC+ NC- VALUE
CCVS	HXXXXXXX N+ N- VNAM VALUE
In the above, values bounded by angle brackets (<>) are considered optional.	

Table 3.2 Standard SPICE3C1 Elements

3.3.1.5.3 Subcircuit Cards

A subcircuit that consists of SPICE3C1 elements can be defined and referenced in a fashion similar to device models. The subcircuit is defined in the input deck by a grouping of element cards delimited by the .SUBCKT and the .ENDS cards; 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 which begins with the letter "X". A complete example of all three of these cards follows:

```
*  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 a SPICE 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".

3.3.2 XSPICE Syntax Extensions

In the preceding discussion, SPICE3C1 syntax was reviewed, and those features of SPICE3C1 that are specifically supported by the XSPICE simulator were enumerated. In addition to these features, there exist extensions to the SPICE3C1 capabilities that provide much more flexibility in describing and simulating a circuit. The following sections describe these capabilities, as well as the syntax required to make use of them.

3.3.2.1 Code Model Element & .MODEL Cards

XSPICE includes a library of predefined "Code Models" that can be placed within any circuit description in a manner similar to that used to place standard SPICE3C1 device models. Code model instance cards always begin with the letter "A", and always make use of a .MODEL card to describe the code model desired. Section 3.4 of this document goes into greater detail as to how a code model similar to the predefined models may be developed, but once any model is created and linked into the simulator it may be placed using one instance card and one .MODEL card (note here we conform to the SPICE custom of referring to

a single logical line of information as a "card"). As an example, the following uses the predefined "gain" code model which takes as an input some value on node 1, multiplies it by a gain of 5.0, and outputs the new value to node 2. Note that, by convention, input ports are specified first on code models. Output ports follow the inputs.

```
a1 1 2 amp
.model amp gain(gain=5.0)
```

In this example the numerical values picked up from single-ended (i.e. ground referenced) input node 1 and output to single-ended output node 2 will be voltages, since in the Interface Specification File for this code model (i.e., gain), the default port type is specified as a voltage (more on this later). However, if you didn't know this, the following modifications to the instance card could be used to insure it:

```
a1 %v(1) %v(2) amp
.model amp gain(gain=5.0)
```

The specification "%v" preceding the input and output node numbers of the instance card indicate to the simulator that the inputs to the model should be single-ended voltage values. Other possibilities exist, as described later.

Some of the other features of the instance and .MODEL cards are worth noting. Of particular interest is the portion of the .MODEL card which specifies `gain=5.0`. This portion of the card assigns a value to a parameter of the "gain" model. There are other parameters which can be assigned values for this model, and in general code models will have several. In addition to numeric values, code model parameters can take non-numeric values (such as TRUE and FALSE), and even vector values. All of these topics will be discussed at length in the following pages. In general, however, the instance and .MODEL cards which define a code model will follow the abstract form described below. This form illustrates that the number of inputs and outputs and the number of parameters which can be specified is relatively open-ended and can be interpreted in a variety of ways (note that angle-brackets "<" and ">" enclose optional inputs):

```
AXXXXXXX <%v,%i,%vd,%id,%g,%gd,%h,%hd, or %d>
+ <[> <~><%v,%i,%vd,%id,%g,%gd,%h,%hd, or %d><NIN1 or +NIN1 -NIN1 or "null">
+ <~>...<NIN2.. <]> >
+ <%v,%i,%vd,%id,%g,%gd,%h,%hd,%d or %vname>
+ <[> <~><%v,%i,%vd,%id,%g,%gd,%h,%hd, or %d><NOUT1 or +NOUT1 -NOUT1>
```

```
+ <~>...<NOUT2.. <]>>
+ MODELNAME


.MODEL MODELNAME MODELTYPE
+ <( PARAMNAME1= <[> VAL1 <VAL2... <]>> PARAMNAME2..>)>
```

Square brackets ([ ]) are used to enclose vector input nodes. In addition, these brackets are used to delineate vectors of parameters.

The literal string "null", when included in a node list, is interpreted as no connection at that input to the model. "Null" is not allowed as the name of a model's input or output if the model only has one input or one output. Also, "null" should only be used to indicate a missing connection for a code model; use on other XSPICE component is not interpreted as a missing connection, but will be interpreted as an actual node name.

The tilde, "~", when prepended to a digital node name, specifies that the logical value of that node be inverted prior to being passed to the code model. This allows for simple inversion of input and output polarities of a digital model in order to handle logically equivalent cases and others that frequently arise in digital system design. The following example defines a NAND gate, one input of which is inverted:

```
a1 [~1 2] 3 nand1
.model nand1 d_nand (rise_delay=0.1 fall_delay=0.2)
```

The optional symbols %v, %i, %vd, etc. specify the type of port the simulator is to expect for the subsequent port or port vector. The meaning of each symbol is given in Table 3.3.

Port Type Modifiers	
Modifier	Interpretation
%v	represents a single-ended voltage port - one node name or number is expected for each port.
%i	represents a single-ended current port - one node name or number is expected for each port.
%g	represents a single-ended voltage-input, current-output (VCCS) port - one node name or number is expected for each port. This type of port is automatically an input/output.
%h	represents a single-ended current-input, voltage-output (CCVS) port - one node name or number is expected for each port. This type of port is automatically an input/output.
%d	represents a digital port - one node name or number is expected for each port. This type of port may be either an input or an output.
%vnam	represents the name of a voltage source, the current through which is taken as an input. This notation is provided primarily in order to allow models defined using SPICE2G6 syntax to operate properly in XSPICE.
%vd	represents a differential voltage port - two node names or numbers are expected for each port.
%id	represents a differential current port - two node names or numbers are expected for each port.
%gd	represents a differential VCCS port - two node names or numbers are expected for each port.
%hd	represents a differential CCVS port - two node names or numbers are expected for each port.

Table 3.3 Port Type Modifiers

The symbols described in Table 3.3 may be omitted if the default port type for the model is desired. Note that non-default port types for multi-input or multi-output (vector) ports must be specified by placing one of the symbols in front of EACH vector port. On the other hand, if all ports of a vector port are to be declared as having the same non-default type, then a symbol may be specified immediately prior to the opening bracket of the vector. The following examples should make this clear:

```
Example 1: - Specifies two differential voltage connections, one
             tonodes 1 & 2, and one to nodes 3 & 4.

    %vd [1 2 3 4]
```

```
Example 2: - Specifies two single-ended connections to node 1 and
             at node 2, and one differential connection to
             nodes 3 & 4.

    %v [1 2 %vd 3 4]

Example 3: - Identical to the previous example...parenthesis
             are added for additional clarity.

    %v [1 2 %vd(3 4)]

Example 4: - Specifies that the node numbers are to be treated in the
             default fashion for the particular
             model.  If this model had ''%v'' as a default for this
             port, then this notation would represent four single-ended
             voltage connections.

    [1 2 3 4]
```

The parameter names listed on the .MODEL card must be identical to those named in the code model itself. The parameters for each predefined code model are described in detail in Section 3.5. The steps required in order to specify parameters for user-defined models are described in Section 3.4.

The following is a list of instance card and associated .MODEL card examples showing use of predefined models within an XSPICE deck:

```
a1 1 2 amp
.model amp gain(in_offset=0.1  gain=5.0  out_offset=-0.01)


a2 %i[1 2] 3 sum1
.model sum1 summer(in_offset=[0.1 -0.2]   in_gain=[2.0 1.0]
+                           out_gain=5.0  out_offset=-0.01)


a21 %i[1 %vd(2 5) 7 10] 3 sum2
.model sum2 summer(out_gain=10.0)
```

```
a5 1 2 limit5
.model limit5 limit(in_offset=0.1  gain=2.5  out_lower_limit=-5.0
+                   out_upper_limit=5.0  limit_domain=0.10
+                   fraction=FALSE)


a7 2 %id(4 7) xfer_cntl1
.model xfer_cntl1 pwl(x_array=[-2.0 -1.0 2.0 4.0 5.0]
+                     y_array=[-0.2 -0.2 0.1 2.0 10.0]
+                     input_domain=0.05  fraction=TRUE)


a8 3 %gd(6 7) switch3
.model switch3 aswitch(cntl_off=0.0  cntl_on=5.0  r_off=1e6
+                      r_on=10.0  log=TRUE)
```

3.3.2.2 Polynomial Source Compatibility

Dependent polynomial sources available in SPICE2G6 (the version of UCB SPICE on which the ATESSE Version 1.0 system was built) are fully supported in the XSPICE. Dependent polynomial sources are not supported in SPICE3 but were reinstated in XSPICE to allow existing third party models to be incorporated readily into XSPICE. Polynomial source syntax is also supported for compatibility with the previous version of ATESSE to allow circuit descriptions written for that system or for the SPICE2G6 simulator to operate in XSPICE. The form used to specify these sources is shown in Table 3.4.

Dependent Polynomial Sources	
Source Type	Instance Card
POLYNOMIAL VCVS	EXXXXXXX N+ N- (POLY (ND)) NC1+ NC1- P0 (P1...)
POLYNOMIAL VCCS	GXXXXXXX N+ N- (POLY (ND)) NC1+ NC1- P0 (P1...)
POLYNOMIAL CCCS	FXXXXXXX N+ N- (POLY (ND)) VNAM1 <VNAM2...> P0 (P1...)
POLYNOMIAL CCVS	HXXXXXXX N+ N- (POLY (ND)) VNAM1 <VNAM2...> P0 (P1...)

Table 3.4 Dependent Polynomial Sources

3.3.2.3 General Enhancements

The following general enhancements are intended to provide better support for electronic board-level simulation but, in general, may be viewed as enhancements which allow you to

have more control over the simulation process regardless of the system simulated. These capabilities are not provided by the SPICE3 Version 3C1 simulator.

3.3.2.3.1 Arbitrary Phase Sources

The XSPICE simulator supports arbitrary phase independent sources that output at TIME=0.0 a value corresponding to some specified phase shift. Other versions of SPICE use the TD (delay time) parameter to set phase-shifted sources to their time-zero value until the delay time has elapsed. The XSPICE phase parameter is specified in degrees and is included after the SPICE3 parameters normally used to specify an independent source. Partial XSPICE deck examples of usage for pulse and sine waveforms are shown below:

```
* Phase shift is specified after Berkeley defined parameters
* on the independent source cards. Phase shift for both of the
* following is specified as +45 degrees
*
v1 1 0 0.0 sin(0 1 1k 0 0 45.0)
r1 1 0 1k
*
v2 2 0 0.0 pulse(-1 1 0 1e-5 1e-5 5e-4 1e-3 45.0)
r2 2 0 1k
*
```

3.3.2.3.2 Initial Conditions

The simulator supports the specification of voltage and current initial conditions on capacitor and inductor models, respectively. **These models are not the standard ones supplied with SPICE3, but are in fact code models which can be substituted for the SPICE models when realistic initial conditions are required.**Partial XSPICE deck examples of usage of these models are shown below:

```
*
* This circuit contains a capacitor and an inductor with
* initial conditions on them.  Each of the components
* has a parallel resistor so that an exponential decay
* of the initial condition occurs with a time constant of
* 1 second.
*
a1 1 0 cap
.model cap capacitor (c=1000uf ic=1)
r1 1 0 1k
*
a2 2 0 ind
.model ind inductor (l=1H ic=1)
r2 2 0 1.0
*
```

3.3.2.3.3 Supply Ramping

A supply ramping function is provided by the simulator as an option to a transient analysis to simulate the turn-on of power supplies to a board level circuit. The supply ramping function linearly ramps the values of all independent sources and the XSPICE capacitor and inductor code models with initial conditions toward their final value at a rate which you define. A complete XSPICE deck example of usage of the ramptime option is shown below:

```
Supply ramping option
*
* This circuit demonstrates the use of the option
* "ramptime" which ramps independent sources and the
* capacitor and inductor initial conditions from
* zero to their final value during the time period
* specified.
*
*
.tran 0.1 5
.option ramptime=0.2
*
a1 1 0 cap
.model cap capacitor (c=1000uf ic=1)
r1 1 0 1k
*
a2 2 0 ind
.model ind inductor (l=1H ic=1)
r2 2 0 1.0
*
v1 3 0 1.0
r3 3 0 1k
*
i1 4 0 1e-3
r4 4 0 1k
*
v2 5 0 0.0  sin(0 1 0.3 0 0 45.0)
r5 5 0 1k
*
.end
```

3.3.2.3.4 Matrix Conditioning

In most SPICE-based simulators (including SPICE3C1), problems can arise with certain circuit topologies. One of the most common problems is the absense of a DC path to ground at some node. This may happen, for example, when two capacitors are connected in series with no other connection at the common node or when certain code models are cascaded. The result is an ill-conditioned or nearly singular matrix that prevents the simulation from completing.

XSPICE introduces a new "rshunt" option to help eliminate this problem. When used, this option inserts resistors to ground at all the analog nodes in the circuit. In general, the value of "rshunt" should be set to some very high resistance (e.g. 1000 Meg Ohms or greater) so that the operation of the circuit is essentially unaffected, but the matrix problems are corrected. If you should encounter a "no DC path to ground" or a "matrix is nearly singular" error message with your circuit, you should try adding the following .option card to your circuit description deck.

```
.option  rshunt = 1.0e12
```

Usually a value of 1.0e12 is sufficient to correct the matrix problems. However, if you still have problems, you may wish to try lowering this value to 1.0e10 or 1.0e9.

3.3.2.3.5 DC Convergence Options

The following options are provided to allow you to have more control over simulator execution. In general, these options are useful in cases where standard simulator execution defaults result in failure of the simulator to converge to a solution.

An option is provided by XSPICE to allow you to disable the simulator's GMIN stepping mode, making source stepping the default DC convergence algorithm. In addition, the source stepping algorithm can be modified to use a variable step size to improve the convergence probability for difficult circuits. Options also exist for setting the maximum number of analog/event alternations that the simulator can use in solving a hybrid circuit, for setting the number of event iterations that are allowed at an analysis point, for disallowing alternations between the analog and event-driven simulator during a DC operating point analysis, and for setting a limit on the absolute and relative size of steps seen by the code models in a circuit. All of these may be of value in getting difficult circuits to converge (although none can guarantee convergence for all cases).

In the following partial XSPICE deck example, the GMIN stepping algorithm is disabled by setting gminsteps to zero and the number of steps to use in the source stepping algorithm is set to 1000. In addition, the maximum number of analog/event alternations is set to 1000, the maximum number of event iterations is set to 2000, and analog/event alternations are enabled. The final two statements control automatic convergence assistance on code models by establishing relative and absolute step size limits to be applied to code model inputs in solving for the DC operating point.

```
*
.option gminsteps=0
.option srcsteps=1000
.option maxopalter=1000
.option maxevtiter=2000
```

```
.option noopalter=FALSE
.option convstep=0.01
.option convabsstep=1e-6
*
```

3.3.2.3.6 Convergence Debugging Support

When a simulation is failing to converge, the simulator can be asked to return convergence diagnostic information that may be useful in identifying the areas of the circuit in which convergence problems are occurring. When running the simulator with the ATESSE SI, these messages are included in the simulator text output. When running through the Nutmeg user interface, these messages are written directly to the terminal.

3.3.2.3.7 Digital Nodes

Support is included for digital nodes that are simulated by an event-driven algorithm. Because the event-driven algorithm is faster than the standard SPICE matrix solution algorithm, and because all "digital", "real", "int" and User-Defined Node types make use of the event-driven algorithm, reduced simulation time for circuits that include these models can be anticipated compared to simulation of the same circuit using analog code models and nodes.

3.3.2.3.8 User-Defined Nodes

Support is provided for User Defined Nodes that operate with the event-driven algorithm. These nodes allow the passing of arbitrary data structures among models. The real and integer node types supplied with XSPICE are actually predefined User-Defined Node types.

3.4 Code Models and User-Defined Nodes

The following sections explain the steps required to create code models and User-Defined Nodes (UDNs) and link them into the simulator. The XSPICE simulator includes libraries of predefined models and node types that span the analog and digital domains. These are detailed later in this document (see Section 3.5, Predefined Code Models). However, the real power of the XSPICE simulator is in its support for extending these libraries with new models written by users. In order to provide this capability, XSPICE includes a "Code Model Toolkit" that enables you to define new models and node data types to be passed between them. These models are handled by XSPICE in a manner analogous to its handling of SPICE devices and XSPICE Predefined Code Models.

The basic steps required to create code models or User-Defined Nodes and link them into the XSPICE simulator are similar. They consist of 1) creating the code model or User-Defined Node (UDN) directory and its associated model or data files, and 2) creating a simulator directory (or returning to the existing simulator directory) and linking the new files into a new XSPICE simulator executable. Once the simulator executable has been created, instances of models, can be placed into any simulator deck that describes a circuit of interest and simulated along with all of the other components in that circuit.

3.4.1 Creating Code Models

The first step in creating a new code model within XSPICE is to create a model directory containing a UNIX 'Makefile' and the following template files: 'Makefile'.

- Interface Specification File
- Model Definition File

After this, the template Interface Specification File (ifspec.ifs) is edited to define the model's inputs, outputs, parameters, etc. You then edit the template Model Definition File (cfunc.mod) to include the C-language source code that defines the model behavior. Once this is done, the files are preprocessed by the XSPICE Code Model Toolkit under the direction of the UNIX Makefile and then compiled into object files ready to be linked into a simulator executable.

The first step in the process of producing a code model, that of creating the model directory and the associated template files, is handled automatically. You simply execute the "mkmoddir" command in a UNIX shell as follows:

```
mkmoddir <directory name>
```

This command creates the named directory, a "Makefile", and the two template files ifspec.ifs and cfunc.mod. You then edit the ifspec.ifs and cfunc.mod files to define your code model. A complete list of the steps taken to create a model follows:

1. In a UNIX shell, execute the command "mkmoddir" to create a directory containing a "Makefile" and templates for an ifspec.ifs file and a cfunc.mod file.
2. Move into the newly created directory using the UNIX command "cd".
3. Edit the Interface Specification template file (ifspec.ifs) to specify the model's name, ports, parameters, and static variables.

4. Edit the model definition template file (cfunc.mod) to include the C-language source code that defines the model's behavior.

5. Execute the UNIX command "make" to preprocess and compile the Interface Specification and Model Definition files.

The Interface Specification File is a text file that describes, in a tabular format, information needed for the code model to be properly interpreted by the simulator when it is placed with other circuit components into a SPICE deck. This information includes such things as the parameter names, parameter default values, and the name of the model itself. The specific format presented to you in the Interface Specification File template must be followed exactly, but is quite straightforward. A detailed description of the required syntax, along with numerous examples, is included in Section 3.4.2.

The Model Definition File contains a C programming language function definition. This function specifies the operations to be performed within the model on the data passed to it by the simulator. Special macros are provided that allow the function to retrieve input data and return output data. Similarly, macros are provided to allow for such things as storage of information between iteration timepoints and sending of error messages. Section 3.4.3 describes the form and function of the Model Definition File in detail and lists the support macros provided within the simulator for use in code models.

3.4.2 Creating User-Defined Nodes

In addition to providing the capability of adding new models to the simulator, a facility exists which allows node types other than those found in standard SPICE to be created. Models may be constructed which pass information back and forth via these nodes. Such models are constructed in the manner described in the previous sections, with appropriate changes to the Interface Specification and Model Definition Files.

Because of the need of electrical engineers to have ready access to both digital and analog simulation capabilities, the "digital" node type is provided as a built-in node type along with standard SPICE analog nodes. For "digital" nodes, extensive support is provided in the form of macros and functions so that you can treat this node type as a standard type analogous to standard SPICE analog nodes when creating and using code models. In addition to "analog" and "digital" nodes, the node types "real" and "int" are also provided with the simulator. These were created using the User-Defined Node (UDN) creation facilities described below.

The first step in creating a new node type within XSPICE is to set up a node type directory along with the appropriate template files needed. After this, the UDN Definition File (cfunc.udn) is edited to provide the node-specific functions which will be needed to support code models using this node type.

The first step in the process of producing a UDN type, that of creating the UDN directory and the associated template files, is handled automatically. You simply execute the "mkudndir" command in a UNIX shell as follows:

```
mkudndir <directory name>
```

This command creates the named directory, a "Makefile", and the template file udnfunc.c. You may then edit the template file as described below. A complete list of the steps necessary to create a User-Defined Node type follows:

1. In a UNIX shell, execute the command "mkudndir" to create a directory containing a "Makefile" and a template for a udnfunc.c file.
2. Move into the newly created directory using the UNIX command "cd".
3. Edit the udnfunc.c template file to code the required C functions (see Section 3.4.4 for details).
4. Execute the UNIX command "make" to preprocess and compile the node functions.

The UDN Definition File contains a set of C language functions. These functions perform operations such as allocating space for data structures, initializing them, and comparing them to each other. Section 3.4.4 describes the form and function of the User-Defined Node Definition File in detail and includes an example UDN Definition File.

3.4.3 Compiling and Linking the Simulator

The concept of creating a new "version" of XSPICE whenever a code model needs to be added is probably foreign to most users. However, the advantages gained from taking this approach are considerable. Most mixed-mode simulators are closed systems; the set of models they provide cannot be extended by the average user. In many cases, even the creators of the original software are not in a position to easily add to the set of models. Consequently, when the need arises for new models they must be defined as subcircuits based on the built-in models.

For simple devices, the synthesis of models from the set of built-in models does not necessarily lead to a degradation of simulator performance. However, if you wish to build up a model that does not lend itself readily to such a synthesis, penalties in the form of increased simulation time and lower modeling accuracy can result.

With this in mind, XSPICE was constructed so that you can readily add to the lowest level of simulator functionality simply by creating a new model or User-Defined Node type

and linking it into a new simulator executable. This process is described in the following paragraphs.

The first step in creating a new instatnce of the XSPICE simulator is to set up a simulation directory along with the appropriate model-list template file and User-Defined Node-list template file. After this, the model-list template and User-Defined Node list template files should be edited to specify the path names to directories containing the code models to be incorporated into the simulator, and the path names to any User-Defined Node type definitions required by the simulator, respectively. Once this is done, the files are preprocessed by the Code Model Toolkit and the executable is built.

The first step in the process of producing a working version of the simulator, that of creating the simulator directory and the associated template files, is handled automatically for you. You simply execute the "mksimdir" command in a UNIX shell as follows:

```
mksimdir <directory name>
```

This command creates the named directory, a "Makefile", and the model-list and User-Defined Node-list template files. The template files may then be edited as described below. A complete list of the steps necessary to create a new version of the simulator follows:

1. In a UNIX shell, execute the command "mksimdir" to create a directory containing a "Makefile" and the template files for the model list file and the UDN-list file.
2. Move into the newly created directory using the UNIX command "cd".
3. Edit the model list template file to indicate the path names to directories containing the desired code models.
4. Edit the User-Defined Node list template file to indicate the path names to directories containing the required User-Defined Node types.
5. Type "make xspice" or "make atesse_xspice" to preprocess the list files, link the specified models and node types, and create the desired simulator executable.

Making "xspice" will create a stand-alone simulator that incorporates the Nutmeg user interface. Making "atesse_xspice" creates a version of the simulator suitable for use with the ATESSE SI user interface.

3.4.4 Interface Specification File

The Interface Specification (IFS) file is a text file that describes the model's naming information, its expected input and output ports, its expected parameters, and any variables

within the model that are to be used for storage of data across an entire simulation. These four types of data are described to the simulator in IFS file sections labeled NAME_TABLE, PORT_TABLE, PARAMETER_TABLE and STATIC_VAR_TABLE, respectively. An example IFS file is given below. The example is followed by detailed descriptions of each of the entries, what they signify, and what values are acceptable for them. Keywords are case insensitive.

```
NAME_TABLE:
C_Function_Name:    ucm_xfer
Spice_Model_Name:   xfer
Description:        "arbitrary transfer function"

PORT_TABLE:
Port_Name:          in              out
Description:        "input"         "output"
Direction:          in              out
Default_Type:       v               v
Allowed_Types:      [v,vd,i,id]     [v,vd,i,id]
Vector:             no              no
Vector_Bounds:      -               -
Null_Allowed:       no              no

PARAMETER_TABLE:
Parameter_Name:     in_offset                gain
Description:        "input offset"           "gain"
Data_Type:          real                     real
Default_Value:      0.0                      1.0
Limits:             -                        -
Vector:             no                       no
Vector_Bounds:      -                        -
Null_Allowed:       yes                      yes

PARAMETER_TABLE:
Parameter_Name:     num_coeff
Description:        "numerator polynomial coefficients"
Data_Type:          real
Default_Value:      -
Limits:             -
Vector:             yes
Vector_Bounds:      [1 -]
Null_Allowed:       no

PARAMETER_TABLE:
Parameter_Name:     den_coeff
Description:        "denominator polynomial coefficients"
Data_Type:          real
Default_Value:      -
Limits:             -
Vector:             yes
Vector_Bounds:      [1 -]
Null_Allowed:       no
```

```
PARAMETER_TABLE:
Parameter_Name:     int_ic
Description:        "integrator stage initial conditions"
Data_Type:          real
Default_Value:      0.0
Limits:             -
Vector:             yes
Vector_Bounds:      den_coeff
Null_Allowed:       yes

STATIC_VAR_TABLE:

Static_Var_Name:    x
Data_Type:          pointer
Description:        "x-coefficient array"
```

3.4.4.1 The Name Table

The name table is introduced by the "Name_Table:" keyword. It defines the code model's C function name, the name used on a .MODEL card, and an optional textual description. The following sections define the valid fields that may be specified in the Name Table.

3.4.4.1.1 C Function Name

The C function name is a valid C identifier which is the name of the function for the code model. It is introduced by the "C_Function_Name:" keyword followed by a valid C identifier. To reduce the chance of name conflicts, it is recommended that user-written code model names use the prefix "ucm_" for this entry. Thus, in the example given above, the model name is "xfer", but the C function is "ucm_xfer". Note that when you subsequently write the model function in the Model Definition File, this name must agree with that of the function (i.e., "ucm_xfer"), or an error will result in the linking step.

3.4.4.1.2 SPICE Model Name

The SPICE model name is a valid SPICE identifier that will be used on SPICE .MODEL cards to refer to this code model. It may or may not be the same as the C function name. It is introduced by the "Spice_Model_Name:" keyword followed by a valid SPICE identifier.

3.4.4.1.3 Description

The description string is used to describe the purpose and function of the code model. It is introduced by the "Description:" keyword followed by a C string literal.

3.4.4.2 The Port Table

The port table is introduced by the "Port_Table:" keyword. It defines the set of valid ports available to the code model. The following sections define the valid fields that may be specified in the port table.

3.4.4.2.1 Port Name

The port name is a valid SPICE identifier. It is introduced by the "Port_Name:" keyword followed by the name of the port. Note that this port name will be used to obtain and return input and output values within the model function. This will be discussed in more detail in the next section.

3.4.4.2.2 Description

The description string is used to describe the purpose and function of the code model. It is introduced by the "Description:" keyword followed by a C string literal.

3.4.4.2.3 Direction

The direction of a port specifies the dataflow direction through the port. A direction must be one of "in", "out", or "inout". It is introduced by the "Direction:" keyword followed by a valid direction value.

3.4.4.2.4 Default Type

The Default_Type field specifies the type of a port. These types are identical to those used to define the port types on a SPICE deck instance card (see Table 3.2), but without the percent sign (%) preceding them. Table 3.5 summarizes the allowable types.

3.4.4.2.5 Allowed Types

A port must specify the types it is allowed to assume. An allowed type value must be a list of type names (a blank or comma separated list of names delimited by square brackets, e.g. "[v vd i id]" or "[d]"). The type names must be taken from those listed in Table 3.5.

3.4.4.2.6 Vector

A port which is a vector can be thought of as a bus. The Vector field is introduced with the "Vector:" keyword followed by a boolean value: "YES", "TRUE", "NO" or "FALSE".

Default Types		
Type	Description	Valid Directions
d	digital	in or out
g	conductance (VCCS)	inout
gd	differential conductance (VCCS)	inout
h	resistance (CCVS)	inout
hd	differential resistance (CCVS)	inout
i	current	in or out
id	differential current	in or out
v	voltage	in or out
vd	differential voltage	in or out
<identifier>	user-defined type	in or out

Table 3.5 Port Types

The values "YES" and "TRUE" are equivalent and specify that this port is a vector. Likewise, "NO" and "FALSE" specify that the port is not a vector. Vector ports must have a corresponding vector bounds field that specifies valid sizes of the vector port.

3.4.4.2.7 Vector Bounds

If a port is a vector, limits on its size must be specified in the vector bounds field. The Vector Bounds field specifies the upper and lower bounds on the size of a vector. The Vector Bounds field is usually introduced by the "Vector_Bounds:" keyword followed by a range of integers (e.g. "[1 7]" or "[3, 20]"). The lower bound of the vector specifies the minimum number of elements in the vector; the upper bound specifies the maximum number of elements. If the range is unconstrained, or the associated port is not a vector, the vector bounds may be specified by a hyphen ("-"). Using the hyphen convention, partial constraints on the vector bound may be defined (e.g., "[2, -]" indicates that the least number of port elements allowed is two, but there is no maximum number).

3.4.4.2.8 Null Allowed

In some cases, it is desirable to permit a port to remain unconnected to any electrical node in a circuit. The Null_Allowed field specifies whether this constitutes an error for a particular port. The Null_Allowed field is introduced by the "Null_Allowed:" keyword and is followed by a boolean constant: "YES", "TRUE", "NO" or "FALSE". The values "YES" and "TRUE" are equivalent and specify that it is legal to leave this port unconnected. "NO" or "FALSE" specify that the port must be connected.

3.4.4.3 The Parameter Table

The parameter table is introduced by the "Parameter_Table:" keyword. It defines the set of valid parameters available to the code model. The following sections define the valid fields that may be specified in the parameter table.

3.4.4.3.1 Parameter Name

The parameter name is a valid SPICE identifier which will be used on SPICE .MODEL cards to refer to this parameter. It is introduced by the "Parameter_Name:" keyword followed by a valid SPICE identifier.

3.4.4.3.2 Description

The description string is used to describe the purpose and function of the parameter. It is introduced by the "Description:" keyword followed by a string literal.

3.4.4.3.3 Data Type

The parameter's data type is specified by the Data Type field. The Data Type field is introduced by the keyword "Data_Type:" and is followed by a valid data type. Valid data types include boolean, complex, int, real, and string.

3.4.4.3.4 Null Allowed

The Null_Allowed field is introduced by the "Null_Allowed:" keyword and is followed by a boolean literal. A value of "TRUE" or "YES" specify that it is valid for the corresponding SPICE .MODEL card to omit a value for this parameter. If the parameter is omitted, the default value is used. If there is no default value, an undefined value is passed to the code model, and the PARAM_NULL( ) macro will return a value of "TRUE" so that defaulting can be handled within the model itself. If the value of Null_Allowed is "FALSE" or "NO", then the simulator will flag an error if the SPICE .MODEL card omits a value for this parameter.

3.4.4.3.5 Default Value

If the Null_Allowed field specifies "TRUE" for this parameter, then a default value may be specified. This value is supplied for the parameter in the event that the SPICE .MODEL card does not supply a value for the parameter. The default value must be of the correct type. The Default Value field is introduced by the "Default_Value:" keyword and is followed by a numeric, boolean, complex, or string literal, as appropriate.

3.4.4.3.6 Limits

Integer and real parameters may be constrained to accept a limited range of values. The following range syntax is used whenever such a range of values is required. A range is specified by a square bracket followed by a value representing a lower bound separated by space from another value representing an upper bound and terminated with a closing square bracket (e.g."[0 10]"). The lower and upper bounds are inclusive. Either the lower or the upper bound may be replaced by a hyphen ("-") to indicate that the bound is unconstrained (e.g. "[10 -]" is read as "the range of values greater than or equal to 10"). For a totally unconstrained range, a single hyphen with no surrounding brackets may be used. The parameter value limit is introduced by the "Limits:" keyword and is followed by a range.

3.4.4.3.7 Vector

The Vector field is used to specify whether a parameter is a vector or a scalar. Like the PORT_TABLE Vector field, it is introduced by the "Vector:" keyword and followed by a boolean value. "TRUE" or "YES" specify that the parameter is a vector. "FALSE" or "NO" specify that it is a scalar.

3.4.4.3.8 Vector Bounds

The valid sizes for a vector parameter are specified in the same manner as are port sizes (see Section 3.4.4.2.7). However, in place of using a numeric range to specify valid vector bounds it is also possible to specify the name of a port. When a parameter's vector bounds are specified in this way, the size of the vector parameter must be the same as the associated vector port.

3.4.4.4 Static Variable Table

The Static Variable table is introduced by the "Static_Var_Table:" keyword. It defines the set of valid static variables available to the code model. These are variables whose values are retained between successive invocations of the code model by the simulator. The following sections define the valid fields that may be specified in the Static Variable Table.

3.4.4.4.1 Name

The Static variable name is a valid C identifier that will be used in the code model to refer to this static variable. It is introduced by the "Static_Var_Name:" keyword followed by a valid C identifier.

3.4.4.4.2 Description

The description string is used to describe the purpose and function of the static variable. It is introduced by the "Description:" keyword followed by a string literal.

3.4.4.4.3 Data Type

The static variable's data type is specified by the Data Type field. The Data Type field is introduced by the keyword "Data_Type:" and is followed by a valid data type. Valid data types include boolean, complex, int, real, string and pointer.

Note that pointer types are used to specify vector values; in such cases, the allocation of memory for vectors must be handled by the code model through the use of the malloc() or calloc() C function. Such allocation must only occur during the initialization cycle of the model (which is identified in the code model by testing the INIT macro for a value of TRUE). Otherwise, memory will be unnecessarily allocated each time the model is called.

Following is an example of the method used to allocate memory to be referenced by a static pointer variable "x" and subsequently use the allocated memory. The example assumes that the value of "size" is at least 2, else an error would result. The references to `STATIC_VAR(x)` that appear in the example illustrate how to set the value of, and then access, a static variable named "x". In order to use the variable "x" in this manner, it must be declared in the Static Variable Table of the code model's Interface Specification File.

```
/* Define local pointer variable */
double  *local_x;

/* Allocate storage to be referenced by the static variable x. */
/* Do this only if this is the initial call of the code model. */
if (INIT == TRUE) {
    STATIC_VAR(x) = calloc(size, sizeof(double));
}

/* Assign the value from the static pointer value to the local */
/* pointer variable. */
local_x = STATIC_VAR(x);

/* Assign values to first two members of the array */
local_x[0] = 1.234;
local_x[1] = 5.678;
```

3.4.5 Model Definition File

The Model Definition File is a C source code file that defines a code model's behavior given input values which are passed to it by the simulator. The file itself is always given the name "cfunc.mod". In order to allow for maximum flexibility, passing of input, output, and simulator-specific information is handled through accessor macros, which are described below. In addition, certain predefined library functions (e.g. smoothing interpolators, complex arithmetic routines) are included in the simulator in order to ease the burden of the code model programmer. These are also described below.

3.4.5.1 Macros

The use of the accessor macros is illustrated in the following example. Note that the argument to most accessor macros is the name of a parameter or port as defined in the Interface Specification File. Note also that all accessor macros except "ARGS" resolve to an lvalue (C language terminology for something that can be assigned a value). Accessor macros do not implement expressions or assignments.

```
void code_model(ARGS) /* private structure accessed by
                                            accessor macros */
{
/* The following code fragments are intended to show how         */
/* information in the argument list is accessed.  The reader     */
/* should not attempt to relate one fragment to another.         */
/* Consider each fragment as a separate example.                 */

    double p,  /* variable for use in the following code fragments */
           x,  /* variable for use in the following code fragments */
           y;  /* variable for use in the following code fragments */

    int    i;  /* indexing variable for use in the following */
           j;  /* indexing variable for use in the following */



    UDN_Example_Type *a_ptr, /* A pointer used to access a User-
                                Defined Node type */
                     *y_ptr; /* A pointer used to access a User-
                                Defined Node type */


    /* Initializing and outputting a User-Defined Node result */
    if(INIT) {
        OUTPUT(y) = malloc(sizeof(user_defined_struct));
        y_ptr = OUTPUT(y);
        y_ptr->component1 = 0.0;
        y_ptr->component2 = 0.0;
    }
```

```
else {
    y_ptr = OUTPUT(y);
    y_ptr->component1 = x1;
    y_ptr->component2 = x2;
}


/* Determining analysis type */
if(ANALYSIS == AC) {

    /* Perform AC analysis-dependent operations here */
}


/* Accessing a parameter value from the .model card */
p = PARAM(gain);


/* Accessing a vector parameter from the .model card */
for(i = 0; i < PARAM_SIZE(in_offset); i++)
    p = PARAM(in_offset[i]);


/* Accessing the value of a simple real-valued input */
x = INPUT(a);


/* Accessing a vector input and checking for null port */
if( ! PORT_NULL(a))
    for(i = 0; i < PORT_SIZE(a); i++)
        x = INPUT(a[i]);


/* Accessing a digital input */
x = INPUT(a);


/* Accessing the value of a User-Defined Node input...     */
/* This node type includes two elements in its definition. */
a_ptr = INPUT(a);
x = a_ptr->component1;
y = a_ptr->component2;


/* Outputting a simple real-valued result */
OUTPUT(out1) = 0.0;


/* Outputting a vector result and checking for null */
if( ! PORT_NULL(a))
    for(i = 0; i < PORT_SIZE(a); i++)
        OUTPUT(a[i]) = 0.0;
```

```
    /* Outputting the partial of output out1 w.r.t. input a */
    PARTIAL(out1,a) = PARAM(gain);


    /* Outputting the partial of output out2(i) w.r.t. input b(j) */
    for(i = 0; i < PORT_SIZE(out2); i++) {
        for(j = 0; j < PORT_SIZE(b); j++) {
            PARTIAL(out2[i],b[j]) = 0.0;
        }
    }

    /* Outputting gain from input c to output out3 in an AC analysis */
    complex_gain.real = 1.0;
    complex_gain.imag = 0.0;
    AC_GAIN(out3,c) = complex_gain;


    /* Outputting a digital result */
    OUTPUT_STATE(out4) = ONE;


    /* Outputting the delay for a digital or user-defined output */
    OUTPUT_DELAY(out5) = 1.0e-9;
}
```

3.4.5.1.1 Macro Definitions

The full set of accessor macros is listed below. Arguments shown in parenthesis are examples only. Explanations of the accessor macros are provided in the subsections below.

```
Circuit Data:
      ARGS
      CALL_TYPE
      INIT
      ANALYSIS
      FIRST_TIMEPOINT
      TIME
      T(n)
      RAD_FREQ
      TEMPERATURE

Parameter Data:
      PARAM(gain)
      PARAM_SIZE(gain)
      PARAM_NULL(gain)
```

```
Port Data:
     PORT_SIZE(a)
     PORT_NULL(a)
     LOAD(a)
     TOTAL_LOAD(a)

Input Data:
     INPUT(a)
     INPUT_STATE(a)
     INPUT_STRENGTH(a)

Output Data:
     OUTPUT(y)
     OUTPUT_CHANGED(a)
     OUTPUT_DELAY(y)
     OUTPUT_STATE(a)
     OUTPUT_STRENGTH(a)

Partial Derivatives:
     PARTIAL(y,a)

AC Gains:
     AC_GAIN(y,a)

Static Variable:
     STATIC_VAR(x)
```

3.4.5.1.2 Circuit Data

```
     ARGS
     CALL_TYPE
     INIT
     ANALYSIS
     FIRST_TIMEPOINT
     TIME
     T(n)
     RAD_FREQ
     TEMPERATURE
```

ARGS is a macro which is passed in the argument list of every code model. It is there to provide a way of referencing each model to all of the remaining macro values. It **must** be

present in the argument list of every code model; it must also be the **only** argument present in the argument list of every code model.

CALL_TYPE is a macro which returns one of two possible symbolic constants. These are EVENT and ANALOG. Testing may be performed by a model using CALL_TYPE to determine whether it is being called by the analog simulator or the event-driven simulator. This will, in general, only be of value to a hybrid model such as the adc_bridge or the dac_bridge.

INIT is an integer (int) that takes the value 1 or 0 depending on whether this is the first call to the code model instance or not, respectively.

ANALYSIS is an enumerated integer that takes values of DC, AC, or TRANSIENT.

FIRST_TIMEPOINT is an integer that takes the value 1 or 0 depending on whether this is the first call for this instance at the current analysis step (i.e., timepoint) or not, respectively.

TIME is a double representing the current analysis time in a transient analysis.

T(n) is a double vector giving the analysis time for a specified timepoint in a transient analysis, where n takes the value 0 or 1. T(0) is equal to TIME. T(1) is the last accepted timepoint. (T(0) - T(1)) is the timestep (i.e., the delta-time value) associated with the current time.

RAD_FREQ is a double representing the current analysis frequency in an AC analysis expressed in units of radians per second.

TEMPERATURE is a double representing the current analysis temperature.

3.4.5.1.3 Parameter Data

```
PARAM(gain)
PARAM_SIZE(gain)
PARAM_NULL(gain)
```

PARAM(gain) resolves to the value of the scalar (i.e., non-vector) parameter "gain" which was defined in the Interface Specification File tables. The type of "gain" is the type given in the ifspec.ifs file. The same accessor macro can be used regardless of type. If "gain" is a string, then PARAM(gain) would resolve to a pointer. PARAM(gain[n]) resolves to the value of the nth element of a vector parameter "gain".

PARAM_SIZE(gain) resolves to an integer (int) representing the size of the "gain" vector (which was dynamically determined when the SPICE deck was read). PARAM_SIZE(gain) is undefined if gain is a scalar.

PARAM_NULL(gain) resolves to an integer with value 0 or 1 depending on whether a value was specified for gain, or whether the value is defaulted, respectively.

3.4.5.1.4 Port Data

```
PORT_SIZE(a)
PORT_NULL(a)
LOAD(a)
TOTAL_LOAD(a)
```

PORT_SIZE(a) resolves to an integer (int) representing the size of the "a" port (which was dynamically determined when the SPICE deck was read). PORT_SIZE(a) is undefined if gain is a scalar.

PORT_NULL(a) resolves to an integer (int) with value 0 or 1 depending on whether the SPICE deck has a node specified for this port, or has specified that the port is null, respectively.

LOAD(a) is used in a digital model to post a capacitive load value to a particular input or output port during the INIT pass of the simulator. All values posted for a particular event-driven node using the LOAD() macro are summed, producing a total load value which

TOTAL_LOAD(a) returns a double value which represents the total capacitive load seen on a specified node to which a digital code model is connected. This information may be used after the INIT pass by the code model to modify the delays it posts with its output states and strengths. Note that this macro can also be used by non-digital event-driven code models (see LOAD(), above).

3.4.5.1.5 Input Data

```
INPUT(a)
INPUT_STATE(a)
INPUT_STRENGTH(a)
```

INPUT(a) resolves to the value of the scalar input "a" that was defined in the Interface Specification File tables ("a" can be either a scalar port or a port value from a vector; in the latter case, the notation used would be "a[i]", where "i" is the index value for the port). The type of "a" is the type given in the ifspec.ifs file. The same accessor macro can be used regardless of type.

INPUT_STATE(a) resolves to the state value defined for digital node types. These will be one of the symbolic constants ZERO, ONE, or UNKNOWN.

INPUT_STRENGTH(a) resolves to the strength with which a digital input node is being driven. This is determined by a resolution algorithm which looks at all outputs to a node and determines its final driven strength. This value in turn is passed to a code model when requested by this macro. Possible strength values are:

1. STRONG
2. RESISTIVE
3. HI_IMPEDANCE
4. UNDETERMINED

3.4.5.1.6 Output Data

```
OUTPUT(y)
OUTPUT_CHANGED(a)
OUTPUT_DELAY(y)
OUTPUT_STATE(a)
OUTPUT_STRENGTH(a)
```

OUTPUT(y) resolves to the value of the scalar output "y" that was defined in the Interface Specification File tables. The type of "y" is the type given in the ifspec.ifs file. The same accessor macro can be used regardless of type. If "y" is a vector, then OUTPUT(y) would resolve to a pointer.

OUTPUT_CHANGED(a) may be assigned one of two values for any particular output from a **digital** code model. If assigned the value TRUE (the default), then an output state, strength and delay must be posted by the model during the call. If, on the other hand, no change has occurred during that pass, the OUTPUT_CHANGED(a) value for an output can be set to FALSE. In this case, no state, strength or delay values need subsequently be posted by the model. Remember that this macro applies to a single output port. If a model has multiple outputs that have not changed, OUTPUT_CHANGED(a) must be set to FALSE for each of them.

OUTPUT_DELAY(y) may be assigned a double value representing a delay associated with a particular digital or User-Defined Node output port. Note that this macro must be set for each digital or User-Defined Node output from a model during each pass, unless the OUTPUT_CHANGED(a) macro is invoked (see above). **Note also that a non-zero value must be assigned to OUTPUT_DELAY()**. Assigning a value of zero (or a negative value) will cause an error.

OUTPUT_STATE(a) may be assigned a state value for a digital output node. Valid values are ZERO, ONE, and UNKNOWN. This is the normal way of posting an output state from a digital code model.

OUTPUT_STRENGTH(a) may be assigned a strength value for a digital output node. This is the normal way of posting an output strength from a digital code model. Valid values are:

1. STRONG
2. RESISTIVE
3. HI_IMPEDANCE
4. UNDETERMINED

3.4.5.1.7 Partial Derivatives

```
PARTIAL(y,a)
PARTIAL(y[n],a)
PARTIAL(y,a[m])
PARTIAL(y[n],a[m])
```

PARTIAL(y,a) resolves to the value of the partial derivative of scalar output "y" with respect to scalar input "a". The type is always double since partial derivatives are only defined for nodes with real valued quantities (i.e., analog nodes).

The remaining uses of PARTIAL are shown for the cases in which either the output, the input, or both are vectors.

Partial derivatives are required by the simulator to allow it to solve the non-linear equations that describe circuit behavior for analog nodes. Since coding of partial derivatives can become difficult and error-prone for complex analog models, you may wish to consider using the cm_analog_auto_partial() code model support function instead of using this macro.

3.4.5.1.8 AC Gains

```
AC_GAIN(y,a)
AC_GAIN(y[n],a)
AC_GAIN(y,a[m])
AC_GAIN(y[n],a[m])
```

AC_GAIN(y,a) resolves to the value of the AC analysis gain of scalar output "y" from scalar input "a". The type is always a structure ("Complex_t") defined in the standard code model header file:

```
typedef struct Complex_s {
   double  real; /* The real part of the complex number */
   double  imag; /* The imaginary part of the complex number */
} Complex_t;
```

The remaining uses of AC_GAIN are shown for the cases in which either the output, the input, or both are vectors.

3.4.5.1.9 Static Variables

```
STATIC_VAR(x)
```

STATIC_VAR(x) resolves to an lvalue or a pointer which is assigned the value of some scalar code model result or state defined in the Interface Spec File tables, or a pointer to a value or a vector of values. The type of "x" is the type given in the Interface Specification File. The same accessor macro can be used regardless of type since it simply resolves to an lvalue. If "x" is a vector, then STATIC_VAR(x) would resolve to a pointer. In this case, the code model is responsible for allocating storage for the vector and assigning the pointer to the allocated storage to STATIC_VAR(x).

3.4.5.1.10 Accessor Macros

Table 3.6 describes the accessor macros available to the Model Definition File programmer and their C types. The PARAM and STATIC_VAR macros, whose types are labeled CD (context dependent), return the type defined in the Interface Specification File. Arguments listed with "[i]" take an optional square bracket delimited index if the corresponding port or parameter is a vector. The index may be any C expression - possibly involving calls to other accessor macros (e.g., "`OUTPUT(out[PORT_SIZE(out)-1])`")

3.4.5.2 Function Library

3.4.5.2.1 Overview

Aside from the accessor macros, the simulator also provides a library of functions callable from within code models. The header file containing prototypes to these functions is automatically inserted into the Model Definition File for you. The complete list of available functions follows:

```
Smoothing Functions:
   void cm_smooth_corner
   void cm_smooth_discontinuity
   double cm_smooth_pwl

Model State Storage Functions:
   void *cm_analog_alloc
   void *cm_event_alloc
   void *cm_analog_get_ptr
   void *cm_event_get_ptr
```

Name	Type	Args	Description
AC_GAIN	Complex_t	y[i],x[i]	AC gain of output y with respect to input x
ANALYSIS	enum	<none>	Type of analysis: DC, AC, TRANSIENT
ARGS	Mif_Private_t	<none>	Standard argument to all code model functions
CALL_TYPE	enum	<none>	Type of model evaluation call: ANALOG or EVENT
INIT	Boolean_t	<none>	Is this the first call to the model?
INPUT	double or void *	name[i]	Value of analog input port, or value of structure pointer for User-Defined Node port.
INPUT_STATE	enum	name[i]	State of a digital input: ZERO, ONE, or UNKNOWN.
INPUT_STRENGTH	enum	name[i]	Strength of digital input: STRONG, RESISTIVE, HI_IMPEDANCE, or UNDETERMINED
INPUT_TYPE	char *	name[i]	The port type of the input
LOAD	double	name[i]	The digital load value placed on a port by this model.
MESSAGE	char *	name[i]	A message output by a model on an event-driven node.
OUTPUT	double or void *	name[i]	Value of the analog output port or value of structure pointer for User-Defined Node port.
OUTPUT_CHANGED	Boolean_t	name[i]	Has a new value been assigned to this event-driven output by the model?
OUTPUT_DELAY	double	name[i]	Delay in seconds for an event-driven output
OUTPUT_STATE	enum	name[i]	State of a digital output: ZERO, ONE, or UNKNOWN.
OUTPUT_STRENGTH	enum	name[i]	Strength of digital output: STRONG, RESISTIVE, HI_IMPEDANCE, or UNDETERMINED
OUTPUT_TYPE	char *	name[i]	The port type of the output
PARAM	CD	name[i]	Value of the parameter
PARAM_NULL	Boolean_t	name[i]	Was the parameter not included on the SPICE .model card?
PARAM_SIZE	int	name	Size of parameter vector
PARTIAL	double	y[i],x[i]	Partial derivative of output y with respect to input x
PORT_NULL	Mif_Boolean_t	name	Has this port been specified as unconnected?
PORT_SIZE	int	name	Size of port vector
RAD_FREQ	double	<none>	Current analysis frequency in radians per second
STATIC_VAR	CD	name	Value of an static variable
STATIC_VAR_SIZE	int	name	Size of static var vector (currently unused).
T(n)	int	index	Current and previous analysis times (T(0) = TIME = current analysis time, T(1) = previous analysis time)
TEMPERATURE	double	<none>	Current analysis temperature
TIME	double	<none>	Current analysis time (same as T(0))
TOTAL_LOAD	double	name[i]	The total of all loads on the node attached to this event-driven port.

Table 3.6 Accessor Macros

```
Integration and Convergence Functions:
   int cm_analog_integrate
   int cm_analog_converge
   void cm_analog_not_converged
   void cm_analog_auto_partial
   double cm_analog_ramp_factor

Message Handling Functions:
   char *cm_message_get_errmsg
   void cm_message_send

Breakpoint Handling Functions:
   int cm_analog_set_temp_bkpt
   int cm_analog_set_perm_bkpt
   int cm_event_queue

Special Purpose Functions:
   void cm_climit_fcn
   double cm_netlist_get_c
   double cm_netlist_get_l

Complex Math Functions:
   complex_t cm_complex_set
   complex_t cm_complex_add
   complex_t cm_complex_sub
   complex_t cm_complex_mult
   complex_t cm_complex_div
```

3.4.5.2.2 Smoothing Functions

```
void cm_smooth_corner(x_input, x_center, y_center, domain,
                      lower_slope, upper_slope, y_output, dy_dx)

     double x_input;       /* The value of the x input */
     double x_center;      /* The x intercept of the two slopes */
     double y_center;      /* The y intercept of the two slopes */
     double domain;        /* The smoothing domain */
     double lower_slope;   /* The lower slope */
     double upper_slope;   /* The upper slope */
     double *y_output;     /* The smoothed y output */
     double *dy_dx;        /* The partial of y wrt x */
```

```
void cm_smooth_discontinuity(x_input, x_lower, y_lower, x_upper, y_upper
                             y_output, dy_dx)

     double x_input;       /* The x value at which to compute y */
     double x_lower;       /* The x value of the lower corner */
     double y_lower;       /* The y value of the lower corner */
     double x_upper;       /* The x value of the upper corner */
     double y_upper;       /* The y value of the upper corner */
     double *y_output;     /* The computed smoothed y value */
     double *dy_dx;        /* The partial of y wrt x */

double cm_smooth_pwl(x_input, x, y, size, input_domain, dout_din)

     double x_input;         /* The x input value */
     double *x;              /* The vector of x values */
     double *y;              /* The vector of y values */
     int    size;            /* The size of the xy vectors */
     double input_domain;    /* The smoothing domain */
     double *dout_din;       /* The partial of the output wrt the input */
```

cm_smooth_corner() automates smoothing between two arbitrarily-sloped lines that meet at a single center point. You specify the center point (x_center, y_center), plus a domain (x-valued delta) above and below x_center. This defines a smoothing region about the center point. Then, the slopes of the meeting lines outside of this smoothing region are specified (lower_slope, upper_slope). The function then interpolates a smoothly-varying output (*y_output) and its derivative (*dy_dx) for the x_input value. This function helps to automate the smoothing of piecewise-linear functions, for example. Such smoothing aids the simulator in achieving convergence.

cm_smooth_discontinuity() allows you to obtain a smoothly-transitioning output (*y_output) that varies between two static values (y_lower, y_upper) as an independent variable (x_input) transitions between two values (x_lower, x_upper). This function is useful in interpolating between resistances or voltage levels that change abruptly between two values.

cm_smooth_pwl() duplicates much of the functionality of the predefined pwl code model. The cm_smooth_pwl() takes an input value plus x-coordinate and y-coordinate vector values along with the total number of coordinate points used to describe the piecewise linear transfer function and returns the interpolated or extrapolated value of the output based on that transfer function. More detail is available by looking at the description of the pwl code model. Note that the output value is the function's returned value.

3.4.5.2.3 Model State Storage Functions

```
void *cm_analog_alloc(tag, size)

     int tag;      /* The user-specified tag for this block of memory */
     int size;     /* The number of bytes to allocate */

void *cm_event_alloc(tag, size)

     int tag;         /* The user-specified tag for the memory block */
     int size;        /* The number of bytes to be allocated */

void *cm_analog_get_ptr(tag, timepoint)

     int tag;            /* The user-specified tag for this block of memory */
     int timepoint;      /* The timepoint of interest - 0=current 1=previous */

void *cm_event_get_ptr(tag, timepoint)

    int tag;            /* The user-specified tag for the memory block */
    int timepoint;      /* The timepoint - 0=current, 1=previous */
```

*cm_analog_alloc() and *cm_event_alloc() allow you to allocate storage space for analog and event-driven model state information. The storage space is not static, but rather, like the T(n) accessor macro information (see section 3.4.3.2), represents a storage vector of two values which rotate with each accepted simulator timepoint evaluation. This is explained more fully below. The "tag" parameter allows you to specify an integer tag when allocating space. This allows more than one rotational storage location per model to be allocated. The "size" parameter specifies the size in bytes of the storage (computed by the C language "sizeof()" operator). Both *cm_analog_alloc() and *cm_event_alloc() return a generic pointer to allocated space. The former should be used by an analog model; the latter should be used by an event-driven model.

*cm_analog_get_ptr() and *cm_event_get_ptr() retrieve the pointer location of previously-allocated rotational storage space. The functions take the integer "tag" used to allocate the space, and an integer from 0 to 1 which specifies the timepoint with which the desired state variable is associated (e.g. timepoint = 0 will retrieve the address of storage for the current timepoint; timepoint = 1 will retrieve the address of storage for the last accepted timepoint). **Note that once a model is exited, storage to the current timepoint state storage location (i.e., timepoint = 0) will, upon the next timepoint iteration, be rotated to the previous location (i.e., timepoint = 1).** When rotation is done, a copy of the old "timepoint = 0" storage value is placed in the new "timepoint = 0" storage location. Thus, if a value does not change for a particular iteration, specific writing to "timepoint = 0" storage is not required. These features allow a model coder to constantly know which piece of state information is being dealt with within the model function at each timepoint.

3.4.5.2.4 Integration and Convergence Functions

```
int  cm_analog_integrate(integrand, integral, partial)

     double integrand;    /* The integrand */
     double *integral;    /* The current and returned value of integral */
     double *partial;     /* The partial derivative of integral wrt integrand */

int  cm_analog_converge(state)

     double *state;    /* The state to be converged */

void cm_analog_not_converged()

void cm_analog_auto_partial()

double cm_ramp_factor()
```

cm_analog_integrate takes as input the integrand (the input to the integrator) and produces as output the integral value and the partial of the integral with respect to the integrand. The integration itself is with respect to time, and the pointer to the integral value must have been previously allocated using *cm_analog_alloc(). This is required because of the need for the integrate routine itself to have access to previously-computed values of the integral.

cm_analog_converge() takes as an input the address of a state variable that was previously allocated using *cm_analog_alloc(). The function itself serves to notify the simulator that for each timestep taken, that variable must be iterated upon until it converges.

cm_analog_not_converged() is a function that can and should be called by an analog model whenever it performs internal limiting of one or more of its inputs to aid in reaching convergence. This causes the simulator to call the model again at the current timepoint and continue solving the circuit matrix. A new timepoint will not be attempted until the code model returns without calling the cm_analog_not_converged() function. For circuits which have trouble reaching a converged state (often due to multiple inputs changing too quickly for the model to react in a reasonable fashion), the use of this function is virtually mandatory.

cm_analog_auto_partial() may be called at the end of a code model function in lieu of calculating the values of partial derivatives explicitly in the function. When this function is called, no values should be assigned to the PARTIAL macro since these values will be computed automatically by the simulator. The automatic calculation of partial derivatives can save considerable time in designing and coding a model, since manual computation of partial derivatives can become very complex and error-prone for some models. However, the automatic evaluation may also increase simulation run time significantly. Function cm_analog_auto_partial() causes the model to be called N additional times (for a model

with N inputs) with each input varied by a small amount (1e-6 for voltage inputs and 1e-12 for current inputs). The values of the partial derivatives of the outputs with respect to the inputs are then approximated by the simulator through divided difference calculations.

cm_analog_ramp_factor() will then return a value from 0.0 to 1.0, which indicates whether or not a ramp time value requested in the SPICE analysis deck (with the use of .option ramptime=<duration>) has elapsed. If the RAMPTIME option is used, then cm_analog_ramp_factor returns a 0.0 value during the DC operating point solution and a value which is between 0.0 and 1.0 during the ramp. A 1.0 value is returned after the ramp is over or if the RAMPTIME option is not used. This value is intended as a multiplication factor to be used with all model outputs which would ordinarily experience a "power-up" transition. Currently, all sources within the simulator are automatically ramped to the "final" time-zero value if a RAMPTIME option is specified.

3.4.5.2.5 Message Handling Functions

```
char *cm_message_get_errmsg()

int cm_message_send(char *msg)

    char *msg;        /* The message to output. */
```

*cm_message_get_errmsg() is a function designed to be used with other library functions to provide a way for models to handle error situations. More specifically, whenever a library function which returns type "int" is executed from a model, it will return an integer value, n. If this value is not equal to zero (0), then an error condition has occurred (likewise, functions which return pointers will return a NULL value if an error has occurred). At that point, the model can invoke *cm_message_get_errmsg to obtain a pointer to an error message. This can then in turn be displayed to the user or passed to the simulator interface through the cm_message_send() function. The C code required for this is as follows:

```
err = cm_analog_integrate(in, &out, &dout_din);
if (err) \{
     cm_message_send(cm_message_get_errmsg());
\}
else \{ ...
```

cm_message_send() sends messages to either the standard output screen or to the simulator interface, depending on which is in use.

3.4.5.2.6 Breakpoint Handling Functions

```
int cm_analog_set_perm_bkpt(time)

     double time;       /* The time of the breakpoint to be set */

int cm_analog_set_temp_bkpt(time)

     double time;       /* The time of the breakpoint to be set */

int  cm_event_queue(time)

     double time;    /* The time of the event to be queued */
```

cm_analog_set_perm_bkpt() takes as input a time value. This value is posted to the analog simulator algorithm and is used to force the simulator to choose that value as a breakpoint at some time in the future. The simulator may choose as the next timepoint a value less than the input, but not greater. Also, regardless of how many timepoints pass before the breakpoint is reached, it will not be removed from posting. Thus, a breakpoint is guaranteed at the passed time value. Note that a breakpoint may also be set for a time prior to the current time, but this will result in an error if the posted breakpoint is prior to the last accepted time (i.e., T(1)).

cm_analog_set_temp_bkpt() takes as input a time value. This value is posted to the simulator and is used to force the simulator, for the next timestep only, to not exceed the passed time value. The simulator may choose as the next timepoint a value less than the input, but not greater. In addition, once the next timestep is chosen, the posted value is removed regardless of whether it caused the break at the given timepoint. This function is useful in the event that a timepoint needs to be retracted after its first posting in order to recalculate a new breakpoint based on new input data (for controlled oscillators, controlled one-shots, etc), since temporary breakpoints automatically "go away" if not reposted each timestep. Note that a breakpoint may also be set for a time prior to the current time, but this will result in an error if the posted breakpoint is prior to the last accepted time (i.e., T(1)).

cm_event_queue() is similar to cm_analog_set_perm_bkpt(), but functions with event-driven models. When invoked, this function causes the model to be queued for calling at the specified time. All other details applicable to cm_analog_set_perm_bkpt() apply to this function as well.

3.4.5.2.7 Special Purpose Functions

```
void cm_climit_fcn(in, in_offset, cntl_upper, cntl_lower, lower_delta, upper_delta,
                   limit_range, gain, fraction, out_final, pout_pin_final,
                   pout_pcntl_lower_final, pout_pcntl_upper_final)

    double in,                      /* The input value */
    double in_offset,               /* The input offset */
    double cntl_upper,              /* The upper control input value */
    double cntl_lower,              /* The lower control input value */
    double lower_delta,             /* The delta from control to limit value */
    double upper_delta,             /* The delta from control to limit value */
    double limit_range,             /* The limiting range */
    double gain,                    /* The gain from input to output */
    int    percent,                 /* The fraction vs. absolute range flag */
    double *out_final,              /* The output value */
    double *pout_pin_final,         /* The partial of output wrt input */
    double *pout_pcntl_lower_final, /* The partial of output wrt lower control input */
    double *pout_pcntl_upper_final) /* The partial of output wrt upper control input */

double cm_netlist_get_c ()

double cm_netlist_get_l ()
```

cm_climit_fcn() is a very specific function that mimics the behavior of the climit code model (see the Predefined Models section). In brief, the cm_climit_fcn() takes as input an "in" value, an offset, and controlling upper and lower values. Parameter values include delta values for the controlling inputs, a smoothing range, gain, and fraction switch values. Outputs include the final value, plus the partial derivatives of the output with respect to signal input, and both control inputs. These all operate identically to the similarly-named inputs and parameters of the climit model.

The function performs a limit on the "in" value, holding it to within some delta of the controlling inputs, and handling smoothing, etc. The cm_climit_fcn() was originally used in the ilimit code model to handle much of the primary limiting in that model, and can be used by a code model developer to take care of limiting in larger models that require it. See the detailed description of the climit model for more in-depth description.

cm_netlist_get_c() and cm_netlist_get_l() functions search the analog circuitry to which their input is connected, and total the capacitance or inductance, respectively, found at that node. The functions, as they are currently written, assume they are called by a model which has only one single-ended analog input port.

3.4.5.2.8 Complex Math Functions

```
Complex_t cm_complex_set (real_part, imag_part)

    double real_part;    /* The real part of the complex number */
    double imag_part;    /* The imaginary part of the complex number */

Complex_t cm_complex_add (x, y)

    Complex_t x;    /* The first operand of x + y */
    Complex_t y;    /* The second operand of x + y */

Complex_t cm_complex_sub (x, y)

    Complex_t x;    /* The first operand of x - y */
    Complex_t y;    /* The second operand of x - y */

Complex_t cm_complex_mult (x, y)

    Complex_t x;    /* The first operand of x * y */
    Complex_t y;    /* The second operand of x * y */

Complex_t cm_complex_div (x, y)

    Complex_t x;    /* The first operand of x / y */
    Complex_t y;    /* The second operand of x / y */
```

cm_complex_set() takes as input two doubles, and converts these to a Complex_t. The first double is taken as the real part, and the second is taken as the imaginary part of the resulting complex value.

cm_complex_add(), cm_complex_sub(), cm_complex_mult(), and cm_complex_div() each take two complex values as inputs and return the result of a complex addition, subtraction, multiplication, or division, respectively.

3.4.6 User-Defined Node Definition File

The User-Defined Node Definition File (udnfunc.c) defines the C functions which implement basic operations on user-defined nodes such as data structure creation, initialization, copying, and comparison. Unlike the Model Definition File which uses the Code Model Preprocessor to translate Accessor Macros, the User-Defined Node Definition file is a pure C language file. This file uses macros to isolate you from data structure definitions, but the macros are defined in a standard header file (EVTudn.h), and translations are performed by the standard C Preprocessor.

When a directory is created for a new User-Defined Node with 'mkudndir', a structure of type 'Evt_Udn_Info_t' is placed at the bottom of the User-Defined Node Definition File.

This structure contains the type name for the node, a description string, and pointers to each of the functions that define the node. This structure is complete except for a text string that describes the node type. This string is stubbed out and may be edited by you if desired.

3.4.6.1 Macros

You must code the functions described in the following section using the macros appropriate for the particular function. You may elect whether not to provide the optional functions.

It is an error to use a macro not defined for a function. Note that a review of the sample directories for the "real" and "int" UDN types will make the function usage clearer.

The macros used in the User-Defined Node Definition File to access and assign data values are defined in Table 3.7. The translations of the macros and of macros used in the function argument lists are defined in the document Interface Design Document for the XSPICE Simulator of the Automatic Test Equipment Software Support Environment (ATESSE).

Name	Type	Description
MALLOCED_PTR	void *	Assign pointer to alloced structure to this macro
STRUCT_PTR	void *	A pointer to a structure of the defined type
STRUCT_PTR_1	void *	A pointer to a structure of the defined type
STRUCT_PTR_2	void *	A pointer to a structure of the defined type
EQUAL	Mif_Boolean_t	Assign TRUE or FALSE to this macro according to the results of structure comparison
INPUT_STRUCT_PTR	void *	A pointer to a structure of the defined type
OUTPUT_STRUCT_PTR	void *	A pointer to a structure of the defined type
INPUT_STRUCT_PTR_ARRAY	void **	An array of pointers to structures of the defined type
INPUT_STRUCT_PTR_ARRAY_SIZE	int	The size of the array
STRUCT_MEMBER_ID	char *	A string naming some part of the structure
PLOT_VAL	double	The value of the specified structure member for plotting purposes
PRINT_VAL	char *	The value of the specified structure member for printing purposes

Table 3.7 User-Defined Node Macros

3.4.6.2 Function Library

The functions (required and optional) that define a User-Defined Node are listed below. For optional functions, the function "stub" can be deleted from the udnfunc.c file template created by "mkudndir," and the pointer in the Evt_Udn_Info_t structure can be changed to NULL.

Required functions:

create	Allocate data structure used as inputs and outputs to code models.
initialize	Set structure to appropriate initial value for first use as model input.
copy	Make a copy of the contents into created but possibly uninitialized structure.
compare	Determine if two structures are equal in value.

Optional functions:

dismantle	Free allocations inside structure (but not structure itself).
invert	Invert logical value of structure.
resolve	Determine the resultant when multiple outputs are connected to a node.
plot_val	Output a real value for specified structure component for plotting purposes.
print_val	Output a string value for specified structure component for printing.
ipc_val	Output a binary representation of the structure suitable for sending over the IPC channel.

The required actions for each of these functions are described in the following subsections. In each function, "mkudndir" replaces the XXX with the node type name specified by you when mkudndir is invoked. The macros used in implementing the functions are described in a later section.

3.4.6.2.1 Function udn_XXX_create

Allocate space for the data structure defined for the User-Defined Node to pass data between models. Then assign pointer created by the storage allocator (e.g. malloc) to MALLOCED_PTR.

3.4.6.2.2 Function udn_XXX_initialize

Assign STRUCT_PTR to a pointer variable of defined type and then initialize the value of the structure.

3.4.6.2.3 Function udn_XXX_compare

Assign STRUCT_PTR_1 and STRUCT_PTR_2 to pointer variables of the defined type. Compare the two structures and assign either TRUE or FALSE to EQUAL.

3.4.6.2.4 Function udn_XXX_copy

Assign INPUT_STRUCT_PTR and OUTPUT_STRUCT_PTR to pointer variables of the defined type and then copy the elements of the input structure to the output structure.

3.4.6.2.5 Function udn_XXX_dismantle

Assign STRUCT_PTR to a pointer variable of defined type and then free any allocated substructures (but not the structure itself!). If there are no substructures, the body of this function may be left null.

3.4.6.2.6 Function udn_XXX_invert

Assign STRUCT_PTR to a pointer variable of the defined type, and then invert the logical value of the structure.

3.4.6.2.7 Function udn_XXX_resolve

Assign INPUT_STRUCT_PTR_ARRAY to a variable declared as an array of pointers of the defined type - e.g.:

```
<type> **struct_array;
struct_array = INPUT_STRUCT_PTR_ARRAY;
```

Then, the number of elements in the array may be determined from the integer valued INPUT_STRUCT_PTR_ARRAY_SIZE macro.

Assign OUTPUT_STRUCT_PTR to a pointer variable of the defined type.

Scan through the array of structures, compute the resolved value, and assign it into the output structure.

3.4.6.2.8 Function udn_XXX_plot_val

Assign STRUCT_PTR to a pointer variable of the defined type. Then, access the member of the structure specified by the string in STRUCT_MEMBER_ID and assign some real valued quantity for this member to PLOT_VALUE.

3.4.6.2.9 Function udn_XXX_print_val

Assign STRUCT_PTR to a pointer variable of the defined type. Then, access the member of the structure specified by the string in STRUCT_MEMBER_ID and assign some string valued quantity for this member to PRINT_VALUE.

If the string is not static, a new string should be allocated on each call. Do not free the allocated strings.

3.4.6.2.10 Function udn_XXX_ipc_val

Use STRUCT_PTR to access the value of the node data. Assign to IPC_VAL a binary representation of the data. Typically this can be accomplished by simply assigning STRUCT_PTR to IPC_VAL.

Assign to IPC_VAL_SIZE an integer representing the size of the binary data in bytes.

3.4.6.3 Example UDN Definition File

The following is an example UDN Definition File which is included with the XSPICE system. It illustrates the definition of the functions described above for a User-Defined Node type which is included with the XSPICE system: in this case, the "int" (for "integer") node type.

```
#include "EVTudn.h"

void *malloc(unsigned);

/* ------------------------------------------------------- */

void udn_int_create(CREATE_ARGS)
{
    /* Malloc space for an int */
    MALLOCED_PTR = malloc(sizeof(int));
}

/* ------------------------------------------------------- */
```

```
void udn_int_dismantle(DISMANTLE_ARGS)
{
    /* Do nothing.  There are no internally malloc'ed things to dismantle */
}

/* ------------------------------------------------------- */

void udn_int_initialize(INITIALIZE_ARGS)
{
    int  *int_struct = STRUCT_PTR;


    /* Initialize to zero */
    *int_struct = 0;
}

/* ------------------------------------------------------- */

void udn_int_invert(INVERT_ARGS)
{
    int      *int_struct = STRUCT_PTR;


    /* Invert the state */
    *int_struct = -(*int_struct);
}

/* ------------------------------------------------------- */

void udn_int_copy(COPY_ARGS)
{
    int  *int_from_struct = INPUT_STRUCT_PTR;
    int  *int_to_struct   = OUTPUT_STRUCT_PTR;

    /* Copy the structure */
    *int_to_struct = *int_from_struct;
}

/* ------------------------------------------------------- */

void udn_int_resolve(RESOLVE_ARGS)
{
    int **array    = INPUT_STRUCT_PTR_ARRAY;
    int *out       = OUTPUT_STRUCT_PTR;
    int num_struct = INPUT_STRUCT_PTR_ARRAY_SIZE;

    int         sum;
    int         i;

    /* Sum the values */
    for(i = 0, sum = 0; i < num_struct; i++)
        sum += *(array[i]);
```

```
    /* Assign the result */
    *out = sum;
}

/* ------------------------------------------------------- */

void udn_int_compare(COMPARE_ARGS)
{
    int  *int_struct1 = STRUCT_PTR_1;
    int  *int_struct2 = STRUCT_PTR_2;

    /* Compare the structures */
    if((*int_struct1) == (*int_struct2))
        EQUAL = TRUE;
    else
        EQUAL = FALSE;
}

/* ------------------------------------------------------- */

void udn_int_plot_val(PLOT_VAL_ARGS)
{
    int   *int_struct = STRUCT_PTR;

    /* Output a value for the int struct */
    PLOT_VAL = *int_struct;
}

/* ------------------------------------------------------- */

void udn_int_print_val(PRINT_VAL_ARGS)
{
    int   *int_struct = STRUCT_PTR;

    /* Allocate space for the printed value */
    PRINT_VAL = malloc(30);

    /* Print the value into the string */
    sprintf(PRINT_VAL, "%8d", *int_struct);
}

/* ------------------------------------------------------- */

void udn_int_ipc_val(IPC_VAL_ARGS)
{
    /* Simply return the structure and its size */
    IPC_VAL = STRUCT_PTR;
    IPC_VAL_SIZE = sizeof(int);
}

/* ------------------------------------------------------- */
```

```
Evt_Udn_Info_t udn_int_info = {

    "int",
    "integer valued data",

    udn_int_create,
    udn_int_dismantle,
    udn_int_initialize,
    udn_int_invert,
    udn_int_copy,
    udn_int_resolve,
    udn_int_compare,
    udn_int_plot_val,
    udn_int_print_val,
    udn_int_ipc_val

};
```

3.5 Predefined Code Models

3.5.1 Analog Models

The following analog models are supplied with XSPICE. The descriptions included consist of the model Interface Specification File and a description of the model's operation. This is followed by an example of a simulator-deck placement of the model, including the .MODEL card and the specification of all available parameters.

3.5.1.1 Gain

```
NAME_TABLE:
C_Function_Name:    cm_gain
Spice_Model_Name:   gain
Description:        "A simple gain block"

PORT_TABLE:
Port Name:          in              out
Description:        "input"         "output"
Direction:          in              out
Default_Type:       v               v
Allowed_Types:      [v,vd,i,id]     [v,vd,i,id]
Vector:             no              no
Vector_Bounds:      -               -
Null_Allowed:       no              no

PARAMETER_TABLE:
Parameter_Name:     in_offset           gain          out_offset
Description:        "input offset"      "gain"        "output offset"
Data_Type:          real                real          real
Default_Value:      0.0                 1.0           0.0
Limits:             -                   -             -
Vector:             no                  no            no
Vector_Bounds:      -                   -             -
Null_Allowed:       yes                 yes           yes
```

Description: This function is a simple gain block with optional offsets on the input and the output. The input offset is added to the input, the sum is then multiplied by the gain, and the result is produced by adding the output offset. This model will operate in DC, AC, and Transient analysis modes.

```
Example SPICE Usage:

a1 1 2 amp
.model amp gain(in_offset=0.1  gain=5.0  out_offset=-0.01)
```

3.5.1.2 Summer

```
NAME_TABLE:
C_Function_Name:    cm_summer
Spice_Model_Name:   summer
Description:        "A summer block"

PORT_TABLE:
Port Name:          in              out
Description:        "input vector"  "output"
Direction:          in              out
Default_Type:       v               v
Allowed_Types:      [v,vd,i,id]     [v,vd,i,id]
Vector:             yes             no
Vector_Bounds:      -               -
Null_Allowed:       no              no

PARAMETER_TABLE:
Parameter_Name:     in_offset               in_gain
Description:        "input offset vector"   "input gain vector"
Data_Type:          real                    real
Default_Value:      0.0                     1.0
Limits:             -                       -
Vector:             yes                     yes
Vector_Bounds:      in                      in
Null_Allowed:       yes                     yes

PARAMETER_TABLE:
Parameter_Name:     out_gain                out_offset
Description:        "output gain"           "output offset"
Data_Type:          real                    real
Default_Value:      1.0                     0.0
Limits:             -                       -
Vector:             no                      no
Vector_Bounds:      -                       -
Null_Allowed:       yes                     yes
```

Description: This function is a summer block with 2-to-N input ports. Individual gains and offsets can be applied to each input and to the output. Each input is added to its respective offset and then multiplied by its gain. The results are then summed, multiplied by the output gain and added to the output offset. This model will operate in DC, AC, and Transient analysis modes.

```
Example SPICE Usage:
a2 [1 2] 3 sum1
.model sum1 summer(in_offset=[0.1 -0.2]   in_gain=[2.0 1.0]
+                           out_gain=5.0  out_offset=-0.01)
```

3.5.1.3 Multiplier

```
NAME_TABLE:
C_Function_Name:    cm_mult
Spice_Model_Name:   mult
Description:        "multiplier block"

PORT_TABLE:
Port_Name:          in              out
Description:        "input vector"  "output"
Direction:          in              out
Default_Type:       v               v
Allowed_Types:      [v,vd,i,id]     [v,vd,i,id]
Vector:             yes             no
Vector_Bounds:      [2 -]           -
Null_Allowed:       no              no

PARAMETER_TABLE:
Parameter_Name:     in_offset               in_gain
Description:        "input offset vector"   "input gain vector"
Data_Type:          real                    real
Default_Value:      0.0                     1.0
Limits:             -                       -
Vector:             yes                     yes
Vector_Bounds:      in                      in
Null_Allowed:       yes                     yes

PARAMETER_TABLE:
Parameter_Name:     out_gain                out_offset
Description:        "output gain"           "output offset"
Data_Type:          real                    real
Default_Value:      1.0                     0.0
Limits:             -                       -
Vector:             no                      no
Vector_Bounds:      -                       -
Null_Allowed:       yes                     yes
```

Description: This function is a multiplier block with 2-to-N input ports. Individual gains and offsets can be applied to each input and to the output. Each input is added to its respective offset and then multiplied by its gain. The results are multiplied along with the output gain and are added to the output offset. This model will operate in DC, AC, and Transient analysis modes. However, in ac analysis it is important to remember that results

are invalid unless only ONE INPUT of the multiplier is connected to a node which bears an AC signal (this is exemplified by the use of a multiplier to perform a potentiometer function: one input is DC, the other carries the AC signal).

```
Example SPICE Usage:

a3 [1 2 3] 4 sigmult
.model sigmult mult(in_offset=[0.1 0.1 -0.1]  in_gain=[10.0 10.0
+                   10.0]   out_gain=5.0  out_offset=0.05)
```

3.5.1.4 Divider

```
NAME_TABLE:

C_Function_Name:        cm_divide
Spice_Model_Name:       divide
Description:            "divider block"


PORT_TABLE:

Port_Name:              num                 den                 out
Description:            "numerator"         "denominator"       "output"
Direction:              in                  in                  out
Default_Type:           v                   v                   v
Allowed_Types:          [v,vd,i,id,vnam]    [v,vd,i,id,vnam]    [v,vd,i,id]
Vector:                 no                  no                  no
Vector_Bounds:          -                   -                   -
Null_Allowed:           no                  no                  no


PARAMETER_TABLE:

Parameter_Name:     num_offset                  num_gain
Description:        "numerator offset"          "numerator gain"
Data_Type:          real                        real
Default_Value:      0.0                         1.0
Limits:             -                           -
Vector:             no                          no
Vector_Bounds:      -                           -
Null_Allowed:       yes                         yes


PARAMETER_TABLE:

Parameter_Name:     den_offset                  den_gain
Description:        "denominator offset"        "denominator gain"
Data_Type:          real                        real
Default_Value:      0.0                         1.0
Limits:             -                           -
Vector:             no                          no
```

```
Vector_Bounds:      -                           -
Null_Allowed:       yes                         yes


PARAMETER_TABLE:

Parameter_Name:     den_lower_limit
Description:        "denominator lower limit"
Data_Type:          real
Default_Value:      1.0e-10
Limits:             -
Vector:             no
Vector_Bounds:      -
Null_Allowed:       yes


PARAMETER_TABLE:

Parameter_Name:     den_domain
Description:        "denominator smoothing domain"
Data_Type:          real
Default_Value:      1.0e-10
Limits:             -
Vector:              no
Vector_Bounds:       -
Null_Allowed:       yes


PARAMETER_TABLE:

Parameter_Name:    fraction
Description:       "smoothing fraction/absolute value switch"
Data_Type:         boolean
Default_Value:     false
Limits:            -
Vector:            no
Vector_Bounds:     -
Null_Allowed:      yes
```

```
PARAMETER_TABLE:

Parameter_Name:     out_gain        out_offset
Description:        "output gain"   "output offset"
Data_Type:          real            real
Default_Value:      1.0             0.0
Limits:             -               -
Vector:             no              no
Vector_Bounds:      -               -
Null_Allowed:       yes             yes
```

Description: This function is a two-quadrant divider. It takes two inputs; num (numerator) and den (denominator). Divide offsets its inputs, multiplies them by their respective gains, divides the results, multiplies the quotient by the output gain, and offsets the result. The denominator is limited to a value above zero via a user specified lower limit. This limit is approached through a quadratic smoothing function, the domain of which may be specified as a fraction of the lower limit value (default), or as an absolute value. This model will operate in DC, AC and Transient analysis modes. However, in ac analysis it is important to remember that results are invalid unless only ONE INPUT of the divider is connected to a node which bears an AC signal (this is exemplified by the use of the divider to perform a potentiometer function: one input is DC, the other carries the AC signal).

```
Example SPICE Usage:

a4 1 2 4 divider
.model divider divide(num_offset=0.1  num_gain=2.5  den_offset=-0.1
+           den_gain=5.0  den_lower_limit=1e-5  den_domain=1e-6
+           fraction=FALSE  out_gain=1.0  out_offset=0.0)
```

3.5.1.5 Limiter

```
NAME_TABLE:
C_Function_Name:    cm_limit
Spice_Model_Name:   limit
Description:        "limit block"

PORT_TABLE:
Port Name:          in              out
Description:        "input"         "output"
Direction:          in              out
Default_Type:       v               v
Allowed_Types:      [v,vd,i,id]     [v,vd,i,id]
Vector:             no              no
Vector_Bounds:      -               -
Null_Allowed:       no              no

PARAMETER_TABLE:
Parameter_Name:     in_offset           gain
Description:        "input offset"      "gain"
Data_Type:          real                real
Default_Value:      0.0                 1.0
Limits:             -                   -
Vector:             no                  no
Vector_Bounds:      -                   -
Null_Allowed:       yes                 yes

PARAMETER_TABLE:
Parameter_Name:     out_lower_limit          out_upper_limit
Description:        "output lower limit"     "output upper limit"
Data_Type:          real                     real
Default_Value:      0.0                      1.0
Limits:             -                        -
Vector:             no                       no
Vector_Bounds:      -                        -
Null_Allowed:       yes                      yes

PARAMETER_TABLE:
Parameter_Name:     limit_range
Description:        "upper & lower smoothing range"
Data_Type:          real
```

```
Default_Value:      1.0e-6
Limits:             -
Vector:             no
Vector_Bounds:      -
Null_Allowed:       yes

PARAMETER_TABLE:
Parameter_Name:     fraction
Description:        "smoothing fraction/absolute value switch"
Data_Type:          boolean
Default_Value:      FALSE
Limits:             -
Vector:             no
Vector_Bounds:      -
Null_Allowed:       yes
```

Description: The Limiter is a single input, single output function similar to the Gain Block. However, the output of the Limiter function is restricted to the range specified by the output lower and upper limits. This model will operate in DC, AC and Transient analysis modes.

Note that the limit range is the value BELOW THE UPPER LIMIT AND ABOVE THE LOWER LIMIT at which smoothing of the output begins. For this model, then, the limit_range represents the delta WITH RESPECT TO THE OUTPUT LEVEL at which smoothing occurs. Thus, for an input gain of 2.0 and output limits of 1.0 and -1.0 volts, the output will begin to smooth out at ± 0.9 volts, which occurs when the input value is at ± 0.4.

```
Example SPICE Usage:

a5 1 2 limit5
.model limit5 limit(in_offset=0.1  gain=2.5  out_lower_limit=-5.0
+           out_upper_limit=5.0  limit_range=0.10  fraction=FALSE)
```

3.5.1.6 Controlled Limiter

```
NAME_TABLE:

C_Function_Name:        cm_climit
Spice_Model_Name:       climit
Description:            "controlled limiter block"


PORT_TABLE:

Port_Name:              in                  cntl_upper
Description:            "input"             "upper lim. control input"
Direction:              in                  in
Default_Type:           v                   v
Allowed_Types:          [v,vd,i,id,vnam]    [v,vd,i,id,vnam]
Vector:                 no                  no
Vector_Bounds:          -                   -
Null_Allowed:           no                  no


PORT_TABLE:

Port_Name:              cntl_lower                    out
Description:            "lower limit control input"   "output"
Direction:              in                            out
Default_Type:           v                             v
Allowed_Types:          [v,vd,i,id,vnam]              [v,vd,i,id]
Vector:                 no                            no
Vector_Bounds:          -                             -
Null_Allowed:           no                            no


PARAMETER_TABLE:

Parameter_Name:     in_offset                 gain
Description:        "input offset"            "gain"
Data_Type:          real                      real
Default_Value:      0.0                       1.0
Limits:             -                         -
Vector:             no                        no
Vector_Bounds:      -                         -
```

```
Null_Allowed:       yes                         yes


PARAMETER_TABLE:

Parameter_Name:     upper_delta                 lower_delta
Description:        "output upper delta"        "output lower delta"
Data_Type:          real                        real
Default_Value:      0.0                         0.0
Limits:             -                           -
Vector:             no                          no
Vector_Bounds:      -                           -
Null_Allowed:       yes                         yes


PARAMETER_TABLE:

Parameter_Name:     limit_range                 fraction
Description:        "upper & lower sm. range"   "smoothing %/abs switch"
Data_Type:          real                        boolean
Default_Value:      1.0e-6                      FALSE
Limits:             -                           -
Vector:             no                          no
Vector_Bounds:      -                           -
Null_Allowed:       yes                         yes
```

Description: The Controlled Limiter is a single input, single output function similar to the Gain Block. However, the output of the Limiter function is restricted to the range specified by the output lower and upper limits. This model will operate in DC, AC, and Transient analysis modes.

Note that the limit range is the value BELOW THE CNTL_UPPER LIMIT AND ABOVE THE CNTL_LOWER LIMIT at which smoothing of the output begins (minimum positive value of voltage must exist between the CNTL_UPPER input and the CNTL_LOWER input at all times). For this model, then, the limit_range represents the delta WITH RESPECT TO THE OUTPUT LEVEL at which smoothing occurs. Thus, for an input gain of 2.0 and output limits of 1.0 and -1.0 volts, the output will begin to smooth out at ± 0.9 volts, which occurs when the input value is at ± 0.4.

Note also that the Controlled Limiter code tests the input values of cntl_lower and cntl_upper to make sure that they are spaced far enough apart to guarantee the existence of a linear range between them. The range is calculated as the difference between (cntl_upper - up-

per_delta - limit_range) and (cntl_lower + lower_delta + limit_range) and must be greater than or equal to zero. Note that when the limit_range is specified as a fractional value, the limit_range used in the above is taken as the calculated fraction of the difference between cntl_upper and cntl_lower. Still, the potential exists for too great a limit_range value to be specified for proper operation, in which case the model will return an error message.

```
Example SPICE Usage:

a6 3 6 8 4 varlimit
.
.
.model varlimit climit(in_offset=0.1  gain=2.5  upper_delta=0.0
+                   lower_delta=0.0  limit_range=0.10 fraction=FALSE)
```

3.5.1.7 PWL Controlled Source

```
NAME_TABLE:

C_Function_Name:        cm_pwl
Spice_Model_Name:       pwl
Description:            "piecewise linear controlled source"


PORT_TABLE:

Port_Name:              in                      out
Description:            "input"                 "output"
Direction:              in                      out
Default_Type:           v                       v
Allowed_Types:          [v,vd,i,id,vnam]        [v,vd,i,id]
Vector:                 no                      no
Vector_Bounds:          -                       -
Null_Allowed:           no                      no


PARAMETER_TABLE:

Parameter_Name:     x_array                     y_array
Description:        "x-element array"           "y-element array"
Data_Type:          real                        real
Default_Value:      -                           -
Limits:             -                           -
Vector:             yes                         yes
Vector_Bounds:      [2 -]                       [2 -]
Null_Allowed:       no                          no


PARAMETER_TABLE:

Parameter_Name:     input_domain            fraction
Description:        "input sm. domain"      "smoothing %/abs switch"
Data_Type:          real                    boolean
Default_Value:      0.01                    TRUE
Limits:             [1e-12 0.5]              -
Vector:             no                      no
Vector_Bounds:      -                       -
```

```
Null_Allowed:       yes                     yes


STATIC_VAR_TABLE:

Static_Var_Name:    last_x_value
Data_Type:          pointer
Description:        "iteration holding variable for limiting"
```

Description: The Piece-Wise Linear Controlled Source is a single input, single output function similar to the Gain Block. However, the output of the PWL Source is not necessarily linear for all values of input. Instead, it follows an I/O relationship specified by you via the x_array and y_array coordinates. This is detailed below.

The x_array and y_array values represent vectors of coordinate points on the x and y axes, respectively. The x_array values are progressively increasing input coordinate points, and the associated y_array values represent the outputs at those points. There may be as few as two (x_array[n],y_array[n]) pairs specified, or as many as memory and simulation speed allow. This permits you to very finely approximate a non-linear function by capturing multiple input-output coordinate points.

Two aspects of the PWL Controlled Source warrant special attention. These are the handling of endpoints and the smoothing of the described transfer function near coordinate points.

In order to fully specify outputs for values of "in" outside of the bounds of the PWL function (i.e., less than x_array[0] or greater than x_array[n], where n is the largest user-specified coordinate index), the PWL Controlled Source model extends the slope found between the lowest two coordinate pairs and the highest two coordinate pairs. This has the effect of making the transfer function completely linear for "in" less than x_array[0] and "in" greater than x_array[n]. It also has the potentially subtle effect of unrealistically causing an output to reach a very large or small value for large inputs. You should thus keep in mind that the PWL Source does not inherently provide a limiting capability.

In order to diminish the potential for nonconvergence of simulations when using the PWL block, a form of smoothing around the x_array, y_array coordinate points is necessary. This is due to the iterative nature of the simulator and its reliance on smooth first derivatives of transfer functions in order to arrive at a matrix solution. Consequently, the "input_domain" and "fraction" parameters are included to allow you some control over the amount and nature of the smoothing performed.

"Fraction" is a switch that is either TRUE or FALSE. When TRUE (the default setting), the simulator assumes that the specified input_domain value is to be interpreted as a fractional figure. Otherwise, it is interpreted as an absolute value. Thus, if fraction=TRUE

and input_domain=0.10, The simulator assumes that the smoothing radius about each coordinate point is to be set equal to 10% of the length of either the x_array segment above each coordinate point, or the x_array segment below each coordinate point. The specific segment length chosen will be the smallest of these two for each coordinate point.

On the other hand, if fraction=FALSE and input=0.10, then the simulator will begin smoothing the transfer function at 0.10 volts (or amperes) below each x_array coordinate and will continue the smoothing process for another 0.10 volts (or amperes) above each x_array coordinate point. Since the overlap of smoothing domains is not allowed, checking is done by the model to ensure that the specified input_domain value is not excessive.

One subtle consequence of the use of the fraction=TRUE feature of the PWL Controlled Source is that, in certain cases, you may inadvertently create extreme smoothing of functions by choosing inappropriate coordinate value points. This can be demonstrated by considering a function described by three coordinate pairs, such as (-1,-1), (1,1), and (2,1). In this case, with a 10% input_domain value specified (fraction=TRUE, input_domain=0.10), you would expect to see rounding occur between in=0.9 and in=1.1, and nowhere else. On the other hand, if you were to specify the same function using the coordinate pairs (-100,-100), (1,1) and (201,1), you would find that rounding occurs between in=-19 and in=21. Clearly in the latter case the smoothing might cause an excessive divergence from the intended linearity above and below in=1.

```
Example SPICE Usage:

a7 2 4 xfer_cntl1
.
.
.model xfer_cntl1 pwl(x_array=[-2.0 -1.0 2.0 4.0 5.0]
+                     y_array=[-0.2 -0.2 0.1 2.0 10.0]
+                     input_domain=0.05  fraction=TRUE)
```

3.5.1.8 Analog Switch

```
NAME_TABLE:
C_Function_Name:    cm_aswitch
Spice_Model_Name:   aswitch
Description:        "analog switch"

PORT_TABLE:
Port Name:          cntl_in         out
Description:        "input"         "resistive output"
Direction:          in              out
Default_Type:       v               gd
Allowed_Types:      [v,vd,i,id]     [gd]
Vector:             no              no
Vector_Bounds:      -               -
Null_Allowed:       no              no

PARAMETER_TABLE:
Parameter_Name:     cntl_off                cntl_on
Description:        "control 'off' value"   "control 'on' value"
Data_Type:          real                    real
Default_Value:      0.0                     1.0
Limits:             -                       -
Vector:             no                      no
Vector_Bounds:      -                       -
Null_Allowed:       yes                     yes

PARAMETER_TABLE:
Parameter_Name:     r_off               log
Description:        "off resistance"    "log/linear switch"
Data_Type:          real                boolean
Default_Value:      1.0e12              TRUE
Limits:             -                   -
Vector:             no                  no
Vector_Bounds:      -                   -
Null_Allowed:       yes                 yes

PARAMETER_TABLE:
Parameter_Name:     r_on
Description:        "on resistance"
Data_Type:          real
```

```
Default_Value:      1.0
Limits:             -
Vector:             no
Vector_Bounds:      -
Null_Allowed:       yes
```

Description: The Analog Switch is a resistor that varies either logarithmically or linearly between specified values of a controlling input voltage or current. Note that the input is not internally limited. Therefore, if the controlling signal exceeds the specified OFF state or ON state value, the resistance may become excessively large or excessively small (in the case of logarithmic dependence), or may become negative (in the case of linear dependence). For the experienced user, these excursions may prove valuable for modeling certain devices, but in most cases you are advised to add limiting of the controlling input if the possibility of excessive control value variation exists.

```
Example SPICE Usage:

a8 3 (6 7) switch3
.
.
.model switch3 aswitch(cntl_off=0.0  cntl_on=5.0  r_off=1e6
+                      r_on=10.0  log=TRUE)
```

3.5.1.9 Zener Diode

```
NAME_TABLE:
C_Function_Name:    cm_zener
Spice_Model_Name:   zener
Description:        "zener diode"

PORT_TABLE:
Port Name:          z
Description:        "zener"
Direction:          inout
Default_Type:       gd
Allowed_Types:      [gd]
Vector:              no
Vector_Bounds:       -
Null_Allowed:       no

PARAMETER_TABLE:
Parameter_Name:     v_breakdown         i_breakdown
Description:        "breakdown voltage" "breakdown current"
Data_Type:          real                real
Default_Value:      -                   2.0e-2
Limits:             [1.0e-6 1.0e6]      [1.0e-9 -]
Vector:             no                  no
Vector_Bounds:      -                   -
Null_Allowed:       no                  yes

PARAMETER_TABLE:
Parameter_Name:     i_sat                    n_forward
Description:        "saturation current"    "forward emission
                                             coefficient"
Data_Type:          real                     real
Default_Value:      1.0e-12                  1.0
Limits:             [1.0e-15 -]              [0.1 10]
Vector:             no                       no
Vector_Bounds:      -                        -
Null_Allowed:       yes                      yes

PARAMETER_TABLE:
Parameter_Name:     limit_switch
Description:        "switch for on-board limiting (convergence aid)"
```

```
Data_Type:          boolean
Default_Value:      FALSE
Limits:             -
Vector:             no
Vector_Bounds:      -
Null_Allowed:       yes

STATIC_VAR_TABLE:
Static_Var_Name:    previous_voltage
Data_Type:          pointer
Description:        "iteration holding variable for limiting"
```

Description: The Zener Diode models the DC characteristics of most zeners. This model differs from the Diode/Rectifier by providing a user-defined dynamic resistance in the reverse breakdown region. The forward characteristic is defined by only a single point, since most data sheets for zener diodes do not give detailed characteristics in the forward region.

The first three parameters define the DC characteristics of the zener in the breakdown region and are usually explicitly given on the data sheet.

The saturation current refers to the relatively constant reverse current that is produced when the voltage across the zener is negative, but breakdown has not been reached. The reverse leakage current determines the slight increase in reverse current as the voltage across the zener becomes more negative. It is modeled as a resistance parallel to the zener with value v_breakdown / i_rev.

Note that the limit_switch parameter engages an internal limiting function for the zener. This can, in some cases, prevent the simulator from converging to an unrealistic solution if the voltage across or current into the device is excessive. If use of this feature fails to yield acceptable results, the convlimit option should be tried (add the following statement to the SPICE input deck: .options convlimit)

```
Example SPICE Usage:

a9 3 4 vref10
.
.
.model vref10 zener(v_breakdown=10.0  i_breakdown=0.02
+                   r_breakdown=1.0  i_rev=1e-6  i_sat=1e-12)
```

3.5.1.10 Current Limiter

```
NAME_TABLE:
C_Function_Name:    cm_ilimit
Spice_Model_Name:   ilimit
Description:        "current limiter block"

PORT_TABLE:
Port Name:          in              pos_pwr
Description:        "input"         "positive power supply"
Direction:          in              inout
Default_Type:       v               g
Allowed_Types:      [v,vd]          [g,gd]
Vector:             no              no
Vector_Bounds:      -               -
Null_Allowed:       no              yes

PORT_TABLE:
Port Name:          neg_pwr                     out
Description:        "negative power supply"     "output"
Direction:          inout                       inout
Default_Type:       g                           g
Allowed_Types:      [g,gd]                      [g,gd]
Vector:             no                          no
Vector_Bounds:      -                           -
Null_Allowed:       yes                         no

PARAMETER_TABLE:
Parameter_Name:     in_offset           gain
Description:        "input offset"      "gain"
Data_Type:          real                real
Default_Value:      0.0                 1.0
Limits:             -                   -
Vector:             no                  no
Vector_Bounds:      -                   -
Null_Allowed:       yes                 yes

PARAMETER_TABLE:
Parameter_Name:     r_out_source                r_out_sink
Description:        "sourcing resistance"       "sinking resistance"
Data_Type:          real                        real
```

```
Default_Value:      1.0                         1.0
Limits:             [1.0e-9 1.0e9]              [1.0e-9 1.0e9]
Vector:             no                          no
Vector_Bounds:      -                           -
Null_Allowed:       yes                         yes

PARAMETER_TABLE:
Parameter_Name:     i_limit_source
Description:        "current sourcing limit"
Data_Type:          real
Default_Value:      -
Limits:             [1.0e-12 -]
Vector:             no
Vector_Bounds:      -
Null_Allowed:       yes

PARAMETER_TABLE:
Parameter_Name:     i_limit_sink
Description:        "current sinking limit"
Data_Type:          real
Default_Value:      -
Limits:             [1.0e-12 -]
Vector:             no
Vector_Bounds:       -
Null_Allowed:       yes

PARAMETER_TABLE:
Parameter_Name:     v_pwr_range                 i_source_range
Description:        "upper & lower power      "sourcing current
                     supply smoothing range"    smoothing range"
Data_Type:          real                        real
Default_Value:      1.0e-6                      1.0e-9
Limits:             [1.0e-15 -]                 [1.0e-15 -]
Vector:             no                          no
Vector_Bounds:      -                           -
Null_Allowed:       yes                         yes

PARAMETER_TABLE:
Parameter_Name:     i_sink_range
Description:        "sinking current smoothing range"
Data_Type:          real
Default_Value:      1.0e-9
```

```
Limits:             [1.0e-15 -]
Vector:             no
Vector_Bounds:      -
Null_Allowed:       yes

PARAMETER_TABLE:
Parameter_Name:     r_out_domain
Description:        "internal/external voltage delta smoothing range"
Data_Type:          real
Default_Value:      1.0e-9
Limits:             [1.0e-15 -]
Vector:             no
Vector_Bounds:      -
Null_Allowed:       yes
```

Description: The Current Limiter models the behavior of an operational amplifier or comparator device at a high level of abstraction. All of its pins act as inputs; three of the four also act as outputs. The model takes as input a voltage value from the "in" connector. It then applies an offset and a gain, and derives from it an equivalent internal voltage (veq), which it limits to fall between pos_pwr and neg_pwr. If veq is greater than the output voltage seen on the "out" connector, a sourcing current will flow from the output pin. Conversely, if the voltage is less than vout, a sinking current will flow into the output pin.

Depending on the polarity of the current flow, either a sourcing or a sinking resistance value (r_out_source, r_out_sink) is applied to govern the vout/i_out relationship. The chosen resistance will continue to control the output current until it reaches a maximum value specified by either i_limit_source or i_limit_sink. The latter mimics the current limiting behavior of many operational amplifier output stages.

During all operation, the output current is reflected either in the pos_pwr connector current or the neg_pwr current, depending on the polarity of i_out. Thus, realistic power consumption as seen in the supply rails is included in the model.

The user-specified smoothing parameters relate to model operation as follows: v_pwr_range controls the voltage below vpos_pwr and above vneg_pwr inputs beyond which veq [= gain * (vin + voffset)] is smoothed; i_source_range specifies the current below i_limit_source at which smoothing begins, as well as specifying the current increment above i_out=0.0 at which i_pos_pwr begins to transition to zero; i_sink_range serves the same purpose with respect to i_limit_sink and i_neg_pwr that i_source range serves for i_limit_source & i_pos_pwr; r_out_domain specifies the incremental value above and below (veq-vout)=0.0 at which r_out will be set to r_out_source and r_out_sink, respectively. For values of (veq- vout) less than r_out_domain and greater than -r_out_domain, r_out is interpolated smoothly between r_out_source & r_out_sink.

```
Example SPICE Usage:

a10 3 10 20 4 amp3
.
.
.model amp3 ilimit(in_offset=0.0  gain=16.0  r_out_source=1.0
+                  r_out_sink=1.0  i_limit_source=1e-3
+                  i_limit_sink=10e-3  v_pwr_range=0.2
+                  i_source_range=1e-6  i_sink_range=1e-6
+                  r_out_domain=1e-6)
```

3.5.1.11 Hysteresis Block

```
NAME_TABLE:
C_Function_Name:    cm_hyst
Spice_Model_Name:   hyst
Description:        "hysteresis block"

PORT_TABLE:
Port Name:          in              out
Description:        "input"         "output"
Direction:          in              out
Default_Type:       v               v
Allowed_Types:      [v,vd,i,id]     [v,vd,i,id]
Vector:             no              no
Vector_Bounds:      -               -
Null_Allowed:       no              no

PARAMETER_TABLE:
Parameter_Name:     in_low              in_high
Description:        "input low value"   "input high value"
Data_Type:          real                real
Default_Value:      0.0                 1.0
Limits:             -                   -
Vector:             no                  no
Vector_Bounds:      -                   -
Null_Allowed:       yes                 yes

PARAMETER_TABLE:
Parameter_Name:     hyst                out_lower_limit
Description:        "hysteresis"        "output lower limit"
Data_Type:          real                real
Default_Value:      0.1                 0.0
Limits:             [0.0 -]             -
Vector:             no                  no
Vector_Bounds:      -                   -
Null_Allowed:       yes                 yes

PARAMETER_TABLE:
Parameter_Name:     out_upper_limit       input_domain
Description:        "output upper limit"  "input smoothing domain"
Data_Type:          real                  real
```

```
Default_Value:      1.0                 0.01
Limits:             -                   -
Vector:             no                  no
Vector_Bounds:      -                   -
Null_Allowed:       yes                 yes

PARAMETER_TABLE:
Parameter_Name:     fraction
Description:        "smoothing fraction/absolute value switch"
Data_Type:          boolean
Default_Value:      TRUE
Limits:             -
Vector:             no
Vector_Bounds:      -
Null_Allowed:       yes
```

Description: The Hysteresis block is a simple buffer stage that provides hysteresis of the output with respect to the input. The in_low and in_high parameter values specify the center voltage or current inputs about which the hysteresis effect operates. The output values are limited to out_lower_limit and out_upper_limit. The value of "hyst" is added to the in_low and in_high points in order to specify the points at which the slope of the hysteresis function would normally change abruptly as the input transitions from a low to a high value. Likewise, the value of "hyst" is subtracted from the in_high and in_low values in order to specify the points at which the slope of the hysteresis function would normally change abruptly as the input transitions from a high to a low value. In fact, the slope of the hysteresis function is never allowed to change abruptly but is smoothly varied whenever the input_domain smoothing parameter is set greater than zero.

```
Example SPICE Usage:

a11 1 2 schmitt1
.
.
.model schmitt1 hyst(in_low=0.7  in_high=2.4  hyst=0.5
+                    out_lower_limit=0.5  out_upper_limit=3.0
+                    input_domain=0.01  fraction=TRUE)
```

3.5.1.12 Differentiator

```
NAME_TABLE:
C_Function_Name:    cm_d_dt
Spice_Model_Name:   d_dt
Description:        "time-derivative block"

PORT_TABLE:
Port Name:          in              out
Description:        "input"         "output"
Direction:          in              out
Default_Type:       v               v
Allowed_Types:      [v,vd,i,id]     [v,vd,i,id]
Vector:             no              no
Vector_Bounds:      -               -
Null_Allowed:       no              no

PARAMETER_TABLE:
Parameter_Name:     gain                out_offset
Description:        "gain"              "output offset"
Data_Type:          real                real
Default_Value:      1.0                 0.0
Limits:             -                   -
Vector:             no                  no
Vector_Bounds:      -                   -
Null_Allowed:       yes                 yes

PARAMETER_TABLE:
Parameter_Name:     out_lower_limit       out_upper_limit
Description:        "output lower limit"  "output upper limit"
Data_Type:          real                  real
Default_Value:      -                     -
Limits:             -                     -
Vector:             no                    no
Vector_Bounds:      -                     -
Null_Allowed:       yes                   yes

PARAMETER_TABLE:
Parameter_Name:     limit_range
Description:        "upper & lower limit smoothing range"
Data_Type:          real
```

```
Default_Value:      1.0e-6
Limits:             -
Vector:             no
Vector_Bounds:      -
Null_Allowed:       yes
```

Description: The Differentiator block is a simple derivative stage that approximates the time derivative of an input signal by calculating the incremental slope of that signal since the previous timepoint. The block also includes gain and output offset parameters to allow for tailoring of the required signal, and output upper and lower limits to prevent convergence errors resulting from excessively large output values. The incremental value of output below the output_upper_limit and above the output_lower_limit at which smoothing begins is specified via the limit_range parameter. In AC analysis, the value returned is equal to the radian frequency of analysis multiplied by the gain.

Note that since truncation error checking is not included in the d_dt block, it is not recommended that the model be used to provide an integration function through the use of a feedback loop. Such an arrangement could produce erroneous results. Instead, you should make use of the "integrate" model, which does include truncation error checking for enhanced accuracy.

```
Example SPICE Usage:

a12 7 12 slope_gen
.
.
.model slope_gen d_dt(out_offset=0.0  gain=1.0
+                     out_lower_limit=1e-12  out_upper_limit=1e12
+                     limit_range=1e-9)
```

3.5.1.13 Integrator

```
NAME_TABLE:
C_Function_Name:    cm_int
Spice_Model_Name:   int
Description:        "time-integration block"

PORT_TABLE:
Port Name:          in              out
Description:        "input"         "output"
Direction:          in              out
Default_Type:       v               v
Allowed_Types:      [v,vd,i,id]     [v,vd,i,id]
Vector:             no              no
Vector_Bounds:      -               -
Null_Allowed:       no              no

PARAMETER_TABLE:
Parameter_Name:     in_offset           gain
Description:        "input offset"      "gain"
Data_Type:          real                real
Default_Value:      0.0                 1.0
Limits:             -                   -
Vector:             no                  no
Vector_Bounds:      -                   -
Null_Allowed:       yes                 yes

PARAMETER_TABLE:
Parameter_Name:     out_lower_limit          out_upper_limit
Description:        "output lower limit"     "output upper limit"
Data_Type:          real                     real
Default_Value:      -                        -
Limits:             -                        -
Vector:             no                       no
Vector_Bounds:      -                        -
Null_Allowed:       yes                      yes

PARAMETER_TABLE:
Parameter_Name:     limit_range
Description:        "upper & lower limit smoothing range"
Data_Type:          real
```

```
Default_Value:      1.0e-6
Limits:             -
Vector:             no
Vector_Bounds:      -
Null_Allowed:       yes

PARAMETER_TABLE:
Parameter_Name:     out_ic
Description:        "output initial condition"
Data_Type:          real
Default_Value:      0.0
Limits:             -
Vector:             no
Vector_Bounds:      -
Null_Allowed:       yes
```

Description: The Integrator block is a simple integration stage that approximates the integral with respect to time of an input signal. The block also includes gain and input offset parameters to allow for tailoring of the required signal, and output upper and lower limits to prevent convergence errors resulting from excessively large output values. Note that these limits specify integrator behavior similar to that found in an operational amplifier-based integration stage, in that once a limit is reached, additional storage does not occur. Thus, the input of a negative value to an integrator which is currently driving at the out_upper_limit level will immediately cause a drop in the output, regardless of how long the integrator was previously summing positive inputs. The incremental value of output below the output_upper_limit and above the output_lower_limit at which smoothing begins is specified via the limit_range parameter. In AC analysis, the value returned is equal to the gain divided by the radian frequency of analysis.

Note that truncation error checking is included in the "int" block. This should provide for a more accurate simulation of the time integration function, since the model will inherently request smaller time increments between simulation points if truncation errors would otherwise be excessive.

```
Example SPICE Usage:

a13 7 12 time_count
.
.
.model time_count int(in_offset=0.0  gain=1.0
+                     out_lower_limit=-1e12  out_upper_limit=1e12
+                     limit_range=1e-9  out_ic=0.0)
```

3.5.1.14 S-Domain Transfer Function

```
NAME_TABLE:
C_Function_Name:    cm_s_xfer
Spice_Model_Name:   s_xfer
Description:        "s-domain transfer function"

PORT_TABLE:
Port Name:          in              out
Description:        "input"         "output"
Direction:          in              out
Default_Type:       v               v
Allowed_Types:      [v,vd,i,id]     [v,vd,i,id]
Vector:             no              no
Vector_Bounds:      -               -
Null_Allowed:       no              no

PARAMETER_TABLE:
Parameter_Name:     in_offset           gain
Description:        "input offset"      "gain"
Data_Type:          real                real
Default_Value:      0.0                 1.0
Limits:             -                   -
Vector:             no                  no
Vector_Bounds:      -                   -
Null_Allowed:       yes                 yes

PARAMETER_TABLE:
Parameter_Name:     num_coeff
Description:        "numerator polynomial coefficients"
Data_Type:          real
Default_Value:      -
Limits:             -
Vector:             yes
Vector_Bounds:      [1 -]
Null_Allowed:       no

PARAMETER_TABLE:
Parameter_Name:     den_coeff
Description:        "denominator polynomial coefficients"
Data_Type:          real
```

```
Default_Value:      -
Limits:             -
Vector:             yes
Vector_Bounds:      [1 -]
Null_Allowed:       no

PARAMETER_TABLE:
Parameter_Name:     int_ic
Description:        "integrator stage initial conditions"
Data_Type:          real
Default_Value:      0.0
Limits:             -
Vector:             yes
Vector_Bounds:      den_coeff
Null_Allowed:       yes

PARAMETER_TABLE:


Parameter_Name:     denormalized_freq
Description:        "denorm. corner freq.(radians) for 1 rad/s coeffs"
Data_Type:          real
Default_Value:      1.0
Limits:             -
Vector:             no
Vector_Bounds:      -
Null_Allowed:       yes
```

Description: The s-domain transfer function is a single input, single output transfer function in the Laplace transform variable "s" that allows for flexible modulation of the frequency-domain characteristics of a signal. The code model may be configured to produce an arbitrary s-domain transfer function with the following restrictions:

```
1. The degree of the numerator polynomial cannot exceed that
of the denominator polynomial in the variable "s".

2. The coefficients for a polynomial must be stated
explicitly.  That is, if a coefficient is zero, it must be
included as an input to the num_coeff or den_coeff vector.
```

The order of the coefficient parameters is from that associated with the highest-powered term decreasing to that of the lowest. Thus, for the coefficient parameters specified below, the equation in "s" is shown:

```
.model filter s_xfer(gain=0.139713  num_coeff=[1.0 0.0 0.07464102]
+          den_coeff=[1.0 0.998942 0.01170077])
```

...specifies a transfer function of the form...

$$N(s) = 0.139713 \cdot \frac{s^2+0.07464102}{s^2+0.0998942s+0.001170077}$$

The s-domain transfer function includes gain and input offset parameters to allow for tailoring of the required signal. There are no limits on the internal signal values or on the output value of the s-domain transfer function, so you are cautioned to specify gain and coefficient values that will not cause the model to produce excessively large values. In AC analysis, the value returned is equal to the real and imaginary components of the total s-domain transfer function at each frequency of interest.

The denormalized_freq term allows you to specify coefficients for a normalized filter (i.e. one in which the frequency of interest is 1 rad/s). Once these coefficients are included, specifying the denormalized frequency value "shifts" the corner frequency to the actual one of interest. As an example, the following transfer function describes a Chebyshev lowpass filter with a corner (passband) frequency of 1 rad/s:

$$N(s) = \frac{1.0}{s^2+1.09773s+1.10251}$$

In order to define an s_xfer model for the above, but with the corner frequency equal to 1500 rad/s (9425 Hz), the following instance and model lines would be needed:

```
a12 cheby1
.model cheby1 s_xfer(num_coeff=[1] den_coeff=[1 1.09773 1.10251]
+                    denormalized_freq=1500)
```

In the above, you add the normalized coefficients and scales the filter through the use of the denormalized_freq parameter. Similar results could have been achieved by performing the denormalization prior to specification of the coefficients, and setting denormalized_freq to the value 1.0 (or not specifying the frequency, as the default is 1.0 rad/s) Note in the above that frequencies are ALWAYS SPECIFIED AS RADIANS/SECOND.

Truncation error checking is included in the s-domain transfer block. This should provide for more accurate simulations, since the model will inherently request smaller time increments between simulation points if truncation errors would otherwise be excessive.

```
Example SPICE Usage:

a14 9 22 cheby_LP_3KHz
.
.
.model cheby_LP_3KHz s_xfer(in_offset=0.0  gain=1.0  num_coeff=[1.0]
+                           den_coeff=[1.0 1.42562 1.51620])
```

3.5.1.15 Slew Rate Block

```
NAME_TABLE:
C_Function_Name:    cm_slew
Spice_Model_Name:   slew
Description:        "A simple slew rate follower block"

PORT_TABLE:
Port Name:          in              out
Description:        "input"         "output"
Direction:          in              out
Default_Type:       v               v
Allowed_Types:      [v,vd,i,id]     [v,vd,i,id]
Vector:             no              no
Vector_Bounds:      -               -
Null_Allowed:       no              no

PARAMETER_TABLE:
Parameter_Name:     rise_slope
Description:        "maximum rising slope value"
Data_Type:          real
Default_Value:      1.0e9
Limits:             -
Vector:             no
Vector_Bounds:      -
Null_Allowed:       yes

PARAMETER_TABLE:
Parameter_Name:     fall_slope
Description:        "maximum falling slope value"
Data_Type:          real
Default_Value:      1.0e9
Limits:             -
Vector:             no
Vector_Bounds:      -
Null_Allowed:       yes

PARAMETER_TABLE:
Parameter_Name:     range
Description:        "smoothing range"
Data_Type:          real
```

```
Default_Value:      0.1
Limits:             -
Vector:             no
Vector_Bounds:      -
Null_Allowed:       yes
```

Description: This function is a simple slew rate block that limits the absolute slope of the output with respect to time to some maximum or value. The actual slew rate effects of over-driving an amplifier circuit can thus be accurately modeled by cascading the amplifier with this model. The units used to describe the maximum rising and falling slope values are expressed in volts or amperes per second. Thus a desired slew rate of 0.5 V/μs will be expressed as 0.5e+6, etc.

The slew rate block will continue to raise or lower its output until the difference between the input and the output values is zero. Thereafter, it will resume following the input signal, unless the slope again exceeds its rise or fall slope limits. The range input specifies a smoothing region above or below the input value. Whenever the model is slewing and the output comes to within the input + or - the range value, the partial derivative of the output with respect to the input will begin to smoothly transition from 0.0 to 1.0. When the model is no longer slewing (output = input), dout/din will equal 1.0.

```
Example SPICE Usage:
a15 1 2 slew1
.model slew1 slew(rise_slope=0.5e6 fall_slope=0.5e6)
```

3.5.1.16 Inductive Coupling

```
NAME_TABLE:
C_Function_Name:        cm_lcouple
Spice_Model_Name:       lcouple
Description:            "inductive coupling (for use with 'core' model)"

PORT_TABLE:
Port_Name:              l               mmf_out
Description:            "inductor"      "mmf output (in ampere-turns)"
Direction:              inout           inout
Default_Type:           hd              hd
Allowed_Types:          [h,hd]          [hd]
Vector:                 no              no
Vector_Bounds:          -               -
Null_Allowed:           no              no

PARAMETER_TABLE:
Parameter_Name:     num_turns
Description:        "number of inductor turns"
Data_Type:          real
Default_Value:      1.0
Limits:             -
Vector:             no
Vector_Bounds:      -
Null_Allowed:       yes
```

Description: This function is a conceptual model which is used as a building block to create a wide variety of inductive and magnetic circuit models. This function is normally used in conjunction with the "core" model, but can also be used with resistors, hysteresis blocks, etc. to build up systems which mock the behavior of linear and nonlinear components.

The lcouple takes as an input (on the "l" port) a current. This current value is multiplied by the num_turns value, N, to produce an output value (a voltage value which appears on the mmf_out port). The mmf_out acts similar to a magnetomotive force in a magnetic circuit; when the lcouple is connected to the "core" model, or to some other resistive device, a current will flow. This current value (which is modulated by whatever the lcouple is connected to) is then used by the lcouple to calculate a voltage "seen" at the "l" port. The voltage is a function of the derivative with respect to time of the current value seen at mmf_out.

The most common use for lcouples will be as a building block in the construction of transformer models. To create a transformer with a single input and a single output, you would

require two lcouple models plus one "core" model. The process of building up such a transformer is described under the description of the "core" model, below.

```
Example SPICE Usage:
a150 (7 0) (9 10) lcouple1
.model lcouple1 lcouple(num_turns=10.0)
```

3.5.1.17 Magnetic Core

```
NAME_TABLE:
C_Function_Name:        cm_core
Spice_Model_Name:       core
Description:            "magnetic core"


PORT_TABLE:
Port_Name:              mc
Description:            "magnetic core"
Direction:              inout
Default_Type:           gd
Allowed_Types:          [g,gd]
Vector:                 no
Vector_Bounds:          -
Null_Allowed:           no

PARAMETER_TABLE:
Parameter_Name:     H_array                     B_array
Description:        "magnetic field array"      "flux density array"
Data_Type:          real                        real
Default_Value:      -                           -
Limits:             -                           -
Vector:             yes                         yes
Vector_Bounds:      [2 -]                       [2 -]
Null_Allowed:       no                          no


PARAMETER_TABLE:
Parameter_Name:     area                        length
Description:        "cross-sectional area"      "core length"
Data_Type:          real                        real
Default_Value:      -                           -
Limits:             -                           -
Vector:              no                          no
Vector_Bounds:       -                           -
Null_Allowed:       no                          no
```

```
PARAMETER_TABLE:
Parameter_Name:     input_domain
Description:        "input sm. domain"
Data_Type:          real
Default_Value:      0.01
Limits:             [1e-12 0.5]
Vector:             no
Vector_Bounds:      -
Null_Allowed:       yes


PARAMETER_TABLE:

Parameter_Name:     fraction
Description:        "smoothing fraction/abs switch"
Data_Type:          boolean
Default_Value:      TRUE
Limits:             -
Vector:             no
Vector_Bounds:      -
Null_Allowed:       yes


PARAMETER_TABLE:

Parameter_Name:     mode
Description:        "mode switch (1 = pwl, 2 = hyst)"
Data_Type:          int
Default_Value:      1
Limits:             [1 2]
Vector:             no
Vector_Bounds:      -
Null_Allowed:       yes


PARAMETER_TABLE:

Parameter_Name:     in_low                  in_high
Description:        "input low value"       "input high value"
Data_Type:          real                    real
Default_Value:      0.0                     1.0
```

```
Limits:             -                       -
Vector:             no                      no
Vector_Bounds:      -                       -
Null_Allowed:       yes                     yes


PARAMETER_TABLE:

Parameter_Name:     hyst                    out_lower_limit
Description:        "hysteresis"            "output lower limit"
Data_Type:          real                    real
Default_Value:      0.1                     0.0
Limits:             [0 -]                   -
Vector:             no                      no
Vector_Bounds:      -                       -
Null_Allowed:       yes                     yes


PARAMETER_TABLE:

Parameter_Name:     out_upper_limit
Description:        "output upper limit"
Data_Type:          real
Default_Value:      1.0
Limits:             -
Vector:             no
Vector_Bounds:      -
Null_Allowed:       yes
```

Description: This function is a conceptual model which is used as a building block to create a wide variety of inductive and magnetic circuit models. This function is almost always expected to be used in conjunction with the "lcouple" model to build up systems which mock the behavior of linear and nonlinear magnetic components. There are two fundamental modes of operation for the core model. These are the pwl mode (which is the default, and which is the most likely to be of use to you) and the hysteresis mode. These are detailed below.

PWL Mode (mode = 1)

The core model in PWL mode takes as input a voltage which it treats as a magnetomotive force (mmf) value. This value is divided by the total effective length of the core to produce a value for the Magnetic Field Intensity, H. This value of H is then used to find the

corresponding Flux Density, B, using the piecewise linear relationship described by you in the H_array / B_array coordinate pairs. B is then multiplied by the cross-sectional area of the core to find the Flux value, which is output as a current. The pertinent mathematical equations are listed below:

$$H = mmf/L, \text{ where L = Length}$$

Here H, the Magnetic Field Intensity, is expressed in ampere-turns/meter.

$$B = f(H)$$

The B value is derived from a piecewise linear transfer function described to the model via the (H_array[],B_array[]) parameter coordinate pairs. This transfer function does not include hysteretic effects; for that, you would need to substitute a HYST model for the core.

$$\phi = BA, \text{ where A = Area}$$

The final current allowed to flow through the core is equal to ϕ. This value in turn is used by the "lcouple" code model to obtain a value for the voltage reflected back across its terminals to the driving electrical circuit.

The following example code shows the use of two "lcouple" models and one core model to produce a simple primary/secondary transformer.

```
Example SPICE Usage:

a1 (2 0) (3 0) primary
.model primary lcouple (num_turns = 155)

a2 (3 4) iron_core
.model iron_core core (H_array = [-1000 -500 -375 -250 -188 -125 -63 0
+                           63 125 188 250 375 500 1000]
+                      B_array = [-3.13e-3 -2.63e-3 -2.33e-3 -1.93e-3
+                           -1.5e-3 -6.25e-4 -2.5e-4 0
+                            2.5e-4 6.25e-4 1.5e-3 1.93e-3 2.33e-3
+                            2.63e-3 3.13e-3]
+                      area = 0.01 length = 0.01)

a3 (5 0) (4 0) secondary
.model secondary lcouple (num_turns = 310)
```

HYSTERESIS Mode (mode = 2)

The core model in HYSTERESIS mode takes as input a voltage which it treats as a magnetomotive force (mmf) value. This value is used as input to the equivalent of a hysteresis code model block. The parameters defining the input low and high values, the output low and high values, and the amount of hysteresis are as in that model. The output from this mode, as in PWL mode, is a current value which is seen across the mc port. An example of the core model used in this fashion is shown below:

```
Example SPICE Usage:

a1 (2 0) (3 0) primary
.model primary lcouple (num_turns = 155)

a2 (3 4) iron_core
.model iron_core core (mode = 2 in_low=-7.0 in_high=7.0
+                      out_lower_limit=-2.5e-4 out_upper_limit=2.5e-4
+                      hyst = 2.3 )

a3 (5 0) (4 0) secondary
.model secondary lcouple (num_turns = 310)
```

One final note to be made about the two core model nodes is that certain parameters are available in one mode, but not in the other. In particular, the in_low, in_high, out_lower_limit, out_upper_limit, and hysteresis parameters are not available in PWL mode. Likewise, the H_array, B_array, area, and length values are unavailable in HYSTERESIS mode. The input_domain and fraction parameters are common to both modes (though their behavior is somewhat different; for explanation of the input_domain and fraction values for the HYSTERESIS mode, you should refer to the hysteresis code model discussion).

3.5.1.18 Controlled Sine Wave Oscillator

```
NAME_TABLE:
C_Function_Name:    cm_sine
Spice_Model_Name:   sine
Description:        "controlled sine wave oscillator"

PORT_TABLE:
Port Name:          cntl_in             out
Description:        "control input"     "output"
Direction:          in                  out
Default_Type:       v                   v
Allowed_Types:      [v,vd,i,id]         [v,vd,i,id]
Vector:             no                  no
Vector_Bounds:      -                   -
Null_Allowed:       no                  no

PARAMETER_TABLE:
Parameter_Name:     cntl_array          freq_array
Description:        "control array"     "frequency array"
Data_Type:          real                real
Default_Value:      0.0                 1.0e3
Limits:             -                   [0 -]
Vector:             yes                 yes
Vector_Bounds:      [2 -]               cntl_array
Null_Allowed:       no                  no

PARAMETER_TABLE:
Parameter_Name:     out_low                  out_high
Description:        "output peak low value"  "output peak high value"
Data_Type:          real                     real
Default_Value:      -1.0                     1.0
Limits:             -                        -
Vector:             no                       no
Vector_Bounds:      -                        -
Null_Allowed:       yes                      yes
```

Description: This function is a controlled sine wave oscillator with parameterizable values of low and high peak output. It takes an input voltage or current value. This value is used as the independent variable in the piecewise linear curve described by the coordinate points of the cntl_array and freq_array pairs. From the curve, a frequency value is determined, and the oscillator will output a sine wave at that frequency.

From the above, it is easy to see that array sizes of 2 for both the cntl_array and the freq_array will yield a linear variation of the frequency with respect to the control input. Any sizes greater than 2 will yield a piecewise linear transfer characteristic. For more detail, refer to the description of the piecewise linear controlled source, which uses a similar method to derive an output value given a control input.

```
Example SPICE Usage:

asine 1 2 in_sine
.model in_sine sine(cntl_array = [-1 0 5 6]
+                   freq_array=[10 10 1000 1000] out_low = -5.0
+                   out_high = 5.0)
```

3.5.1.19 Controlled Triangle Wave Oscillator

```
NAME_TABLE:
C_Function_Name:    cm_triangle
Spice_Model_Name:   triangle
Description:        "controlled triangle wave oscillator"

PORT_TABLE:
Port Name:          cntl_in             out
Description:        "control input"     "output"
Direction:          in                  out
Default_Type:       v                   v
Allowed_Types:      [v,vd,i,id]         [v,vd,i,id]
Vector:             no                  no
Vector_Bounds:      -                   -
Null_Allowed:       no                  no

PARAMETER_TABLE:
Parameter_Name:     cntl_array          freq_array
Description:        "control array"     "frequency array"
Data_Type:          real                real
Default_Value:      0.0                 1.0e3
Limits:             -                   [0 -]
Vector:             yes                 yes
Vector_Bounds:      [2 -]               cntl_array
Null_Allowed:       no                  no

PARAMETER_TABLE:
Parameter_Name:     out_low                   out_high
Description:        "output peak low value"   "output peak high value"
Data_Type:          real                      real
Default_Value:      -1.0                      1.0
Limits:             -                         -
Vector:             no                        no
Vector_Bounds:      -                         -
Null_Allowed:       yes                       yes

PARAMETER_TABLE:
Parameter_Name:     rise_duty
Description:        "rise time duty cycle"
Data_Type:          real
```

```
Default_Value:      0.5
Limits:             [1e-10 0.999999999]
Vector:             no
Vector_Bounds:      -
Null_Allowed:       yes
```

Description: This function is a controlled triangle/ramp wave oscillator with parameterizable values of low and high peak output and rise time duty cycle. It takes an input voltage or current value. This value is used as the independent variable in the piecewise linear curve described by the coordinate points of the cntl_array and freq_array pairs. From the curve, a frequency value is determined, and the oscillator will output a triangle wave at that frequency.

From the above, it is easy to see that array sizes of 2 for both the cntl_array and the freq_array will yield a linear variation of the frequency with respect to the control input. Any sizes greater than 2 will yield a piecewise linear transfer characteristic. For more detail, refer to the description of the piecewise linear controlled source, which uses a similar method to derive an output value given a control input.

```
Example SPICE Usage:

ain 1 2 ramp1
.model ramp1 triangle(cntl_array = [-1 0 5 6]
+                     freq_array=[10 10 1000 1000] out_low = -5.0
+                     out_high = 5.0 duty_cycle = 0.9)
```

3.5.1.20 Controlled Square Wave Oscillator

```
NAME_TABLE:
C_Function_Name:    cm_square
Spice_Model_Name:   square
Description:        "controlled square wave oscillator"

PORT_TABLE:
Port Name:          cntl_in             out
Description:        "control input"     "output"
Direction:          in                  out
Default_Type:       v                   v
Allowed_Types:      [v,vd,i,id]         [v,vd,i,id]
Vector:             no                  no
Vector_Bounds:      -                   -
Null_Allowed:       no                  no

PARAMETER_TABLE:
Parameter_Name:     cntl_array          freq_array
Description:        "control array"     "frequency array"
Data_Type:          real                real
Default_Value:      0.0                 1.0e3
Limits:             -                   [0 -]
Vector:             yes                 yes
Vector_Bounds:      [2 -]               cntl_array
Null_Allowed:       no                  no

PARAMETER_TABLE:
Parameter_Name:     out_low                   out_high
Description:        "output peak low value"   "output peak high value"
Data_Type:          real                      real
Default_Value:      -1.0                      1.0
Limits:             -                         -
Vector:             no                        no
Vector_Bounds:      -                         -
Null_Allowed:       yes                       yes

PARAMETER_TABLE:
Parameter_Name:     duty_cycle          rise_time
Description:        "duty cycle"        "output rise time"
Data_Type:          real                real
```

```
Default_Value:      0.5                 1.0e-9
Limits:             [1e-6 0.999999]     -
Vector:             no                  -
Vector_Bounds:      -                   -
Null_Allowed:       yes                 yes

PARAMETER_TABLE:
Parameter_Name:     fall_time
Description:        "output fall time"
Data_Type:          real
Default_Value:      1.0e-9
Limits:             -
Vector:             no
Vector_Bounds:      -
Null_Allowed:       yes
```

Description: This function is a controlled square wave oscillator with parameterizable values of low and high peak output, duty cycle, rise time, and fall time. It takes an input voltage or current value. This value is used as the independent variable in the piecewise linear curve described by the coordinate points of the cntl_array and freq_array pairs. From the curve, a frequency value is determined, and the oscillator will output a square wave at that frequency.

From the above, it is easy to see that array sizes of 2 for both the cntl_array and the freq_array will yield a linear variation of the frequency with respect to the control input. Any sizes greater than 2 will yield a piecewise linear transfer characteristic. For more detail, refer to the description of the piecewise linear controlled source, which uses a similar method to derive an output value given a control input.

```
Example SPICE Usage:

ain 1 2 pulse1
.model pulse1 square(cntl_array = [-1 0 5 6]
+                    freq_array=[10 10 1000 1000] out_low = 0.0
+                    out_high = 4.5 duty_cycle = 0.2
+                    rise_time = 1e-6 fall_time = 2e-6)
```

3.5.1.21 Controlled One-Shot

```
NAME_TABLE:
C_Function_Name:    cm_oneshot
Spice_Model_Name:   oneshot
Description:        "controlled one-shot"

PORT_TABLE:
Port Name:          cntl_in             clk
Description:        "control input"     "clock input"
Direction:          in                  out
Default_Type:       v                   v
Allowed_Types:      [v,vd,i,id]         [v,vd,i,id]
Vector:             no                  no
Vector_Bounds:      -                   -
Null_Allowed:       no                  no

PORT_TABLE:
Port Name:          out
Description:        "output"
Direction:          out
Default_Type:       v
Allowed_Types:      [v,vd,i,id]
Vector:             no
Vector_Bounds:      -
Null_Allowed:       no

PARAMETER_TABLE:
Parameter_Name:     clk_trig
Description:        "clock trigger value"
Data_Type:          real
Default_Value:      0.5
Limits:             -
Vector:             no
Vector_Bounds:      -
Null_Allowed:       no

PARAMETER_TABLE:
Parameter_Name:     pos_edge_trig
Description:        "positive/negative edge trigger switch"
Data_Type:          boolean
```

```
Default_Value:      TRUE
Limits:             -
Vector:             no
Vector_Bounds:      -
Null_Allowed:       no

PARAMETER_TABLE:
Parameter_Name:     cntl_array          pw_array
Description:        "control array"     "pulse width array"
Data_Type:          real                real
Default_Value:      0.0                 1.0e-6
Limits:             -                   [0.00 -]
Vector:             yes                 yes
Vector_Bounds:      -                   cntl_array
Null_Allowed:       yes                 yes

PARAMETER_TABLE:
Parameter_Name:     out_low             out_high
Description:        "output low value"  "output high value"
Data_Type:          real                real
Default_Value:      0.0                 1.0
Limits:             -                   -
Vector:             no                  no
Vector_Bounds:      -                   -
Null_Allowed:       yes                 yes

PARAMETER_TABLE:
Parameter_Name:     delay                       rise_time
Description:        "output delay from trig."   "output rise time"
Data_Type:          real                        real
Default_Value:      1.0e-9                      1.0e-9
Limits:             -                           -
Vector:             no                          no
Vector_Bounds:      -                           -
Null_Allowed:       yes                         yes

PARAMETER_TABLE:
Parameter_Name:     fall_time
Description:        "output fall time"
Data_Type:          real
Default_Value:      1.0e-9
Limits:             -
```

```
Vector:             -
Vector_Bounds:      -
Null_Allowed:       yes
```

Description: This function is a controlled oneshot with parameterizable values of low and high peak output, input trigger value level, delay, and output rise and fall times. It takes an input voltage or current value. This value is used as the independent variable in the piecewise linear curve described by the coordinate points of the cntl_array and pw_array pairs. From the curve, a pulse width value is determined, and the oscillator will output a pulse of that width, delayed by the delay value, and with specified rise and fall times.

From the above, it is easy to see that array sizes of 2 for both the cntl_array and the pw_array will yield a linear variation of the pulse width with respect to the control input. Any sizes greater than 2 will yield a piecewise linear transfer characteristic. For more detail, refer to the description of the piecewise linear controlled source, which uses a similar method to derive an output value given a control input.

```
Example SPICE Usage:

ain 1 2 3 4 pulse2
.model pulse1 oneshot(cntl_array = [-1 0 10 11]
+                     pw_array=[1e-6 1e-6 1e-4 1e-4]
+                     clk_trig = 0.9 pos_edge_trig = FALSE
+                     out_low = 0.0 out_high = 4.5 duty_cycle = 0.9
+                     rise_delay = 20.0-9 fall_delay = 35.0e-9)
```

3.5.1.22 Capacitance Meter

```
NAME_TABLE:
C_Function_Name:    cm_cmeter
Spice_Model_Name:   cmeter
Description:        "capacitance meter"

PORT_TABLE:
Port Name:          in              out
Description:        "input"         "output"
Direction:          in              out
Default_Type:       v               v
Allowed_Types:      [v,vd,i,id]     [v,vd,i,id]
Vector:             no              no
Vector_Bounds:      -               -
Null_Allowed:       no              no

PARAMETER_TABLE:
Parameter_Name:     gain
Description:        "gain"
Data_Type:          real
Default_Value:      1.0
Limits:             -
Vector:             no
Vector_Bounds:      -
Null_Allowed:       yes
```

Description: The capacitance meter is a sensing device which is attached to a circuit node and produces as an output a scaled value equal to the total capacitance seen on its input multiplied by the gain parameter. This model is primarily intended as a building block for other models which must sense a capacitance value and alter their behavior based upon it.

```
Example SPICE Usage:

atest1 1 2 ctest
.model ctest cmeter(gain=1.0e12)
```

3.5.1.23 Inductance Meter

```
NAME_TABLE:
C_Function_Name:    cm_lmeter
Spice_Model_Name:   lmeter
Description:        "inductance meter"

PORT_TABLE:
Port Name:          in              out
Description:        "input"         "output"
Direction:          in              out
Default_Type:       v               v
Allowed_Types:      [v,vd,i,id]     [v,vd,i,id]
Vector:             no              no
Vector_Bounds:      -               -
Null_Allowed:       no              no

PARAMETER_TABLE:
Parameter_Name:     gain
Description:        "gain"
Data_Type:          real
Default_Value:      1.0
Limits:             -
Vector:             no
Vector_Bounds:      -
Null_Allowed:       yes
```

Description: The inductance meter is a sensing device which is attached to a circuit node and produces as an output a scaled value equal to the total inductance seen on its input multiplied by the gain parameter. This model is primarily intended as a building block for other models which must sense an inductance value and alter their behavior based upon it.

```
Example SPICE Usage:

atest2 1 2 ltest
.model ltest lmeter(gain=1.0e6)
```

3.5.2 Hybrid Models

The following hybrid models are supplied with XSPICE. The descriptions included below consist of the model Interface Specification File and a description of the model's operation. This is followed by an example of a simulator-deck placement of the model, including the .MODEL card and the specification of all available parameters.

A note should be made with respect to the use of hybrid models for other than simple digital-to-analog and analog-to-digital translations. The hybrid models represented in this section address that specific need, but in the development of user-defined nodes you may find a need to translate not only betweem digital and analog nodes, but also between real and digital, real and int, etc. In most cases such translations will not need to be as involved or as detailed as shown in the following.

3.5.2.1 Digital-to-Analog Node Bridge

```
NAME_TABLE:
C_Function_Name:    cm_dac_bridge
Spice_Model_Name:   dac_bridge
Description:        "digital-to-analog node bridge"

PORT_TABLE:
Port Name:          in                          out
Description:        "input"                     "output"
Direction:          in                          out
Default_Type:       d                           v
Allowed_Types:      [d]                         [v,vd,i,id,d]
Vector:             yes                         yes
Vector_Bounds:      -                           -
Null_Allowed:       no                          no

PARAMETER_TABLE:    out_low
Parameter_Name:     "0-valued analog output"
Description:        real
Data_Type:          0.0
Default_Value:      -
Limits:             -
Vector:             no
Vector_Bounds:      -
Null_Allowed:       yes

PARAMETER_TABLE:    out_high
Parameter_Name:     "1-valued analog output"
Description:        real
Data_Type:          1.0
Default_Value:      -
Limits:             -
Vector:             no
Vector_Bounds:      -
Null_Allowed:       yes

PARAMETER_TABLE:    out_undef                   input_load
Parameter_Name:     "U-valued analog output"    "input load (F)"
Description:        real                        real
Data_Type:          0.5                         1.0e-12
```

```
Default_Value:      -                       -
Limits:             -                       -
Vector:             no                      no
Vector_Bounds:      -                       -
Null_Allowed:       yes                     yes

PARAMETER_TABLE:    t_rise              t_fall
Parameter_Name:     "rise time 0->1"     "fall time 1->0"
Description:        real                 real
Data_Type:          1.0e-9               1.0e-9
Default_Value:      -                    -
Limits:             -                    -
Vector:             no                   no
Vector_Bounds:      -                    -
Null_Allowed:       yes                  yes
```

Description: The dac_bridge is the first of two node bridge devices designed to allow for the ready transfer of digital information to analog values and back again. The second device is the adc_bridge (which takes an analog value and maps it to a digital one).

The dac_bridge takes as input a digital value from a digital node. This value by definition may take on only one of the values "0", "1" or "U". The dac_bridge then outputs the value "out_low", "out_high" or "out_undef", or ramps linearly toward one of these "final" values from its current analog output level. The speed at which this ramping occurs depends on the values of "t_rise" and "t_fall". These parameters are interpreted by the model such that the rise or fall slope generated is always constant.

Note that the dac_bridge includes test code in its cfunc.mod file for determining the presence of the out_undef parameter. If this parameter is not specified by you, and if out_high and out_low values are specified, then out_undef is assigned the value of the arithmetic mean of out_high and out_low. This simplifies coding of output buffers, where typically a logic family will include an out_low and out_high voltage, but not an out_undef value.

This model also posts an input load value (in farads) based on the parameter input_load.

```
Example SPICE Usage:

abridge1 7 2 dac1
.model dac1 dac_bridge(out_low = 0.7 out_high = 3.5 out_undef = 2.2
+                      input_load = 5.0e-12 t_rise = 50e-9
+                      f_fall = 20e-9)
```

3.5.2.2 Analog-to-Digital Node Bridge

```
NAME_TABLE:
C_Function_Name:    cm_adc_bridge
Spice_Model_Name:   adc_bridge
Description:        "analog-to-digital node bridge"

PORT_TABLE:
Port Name:          in                          out
Description:        "input"                     "output"
Direction:          in                          out
Default_Type:       v                           d
Allowed_Types:      [v,vd,i,id,d]               [d]
Vector:             yes                         yes
Vector_Bounds:      -                           -
Null_Allowed:       no                          no

PARAMETER_TABLE:
Parameter_Name:     in_low
Description:        "maximum 0-valued analog input"
Data_Type:          real
Default_Value:      1.0
Limits:             -
Vector:             no
Vector_Bounds:      -
Null_Allowed:       yes

PARAMETER_TABLE:
Parameter_Name:     in_high
Description:        "minimum 1-valued analog input"
Data_Type:          real
Default_Value:      2.0
Limits:             -
Vector:             no
Vector_Bounds:      -
Null_Allowed:       yes

PARAMETER_TABLE:
Parameter_Name:     rise_delay          fall_delay
Description:        "rise delay"        "fall delay"
Data_Type:          real                real
```

Default_Value:	1.0e-9	1.0e-9
Limits:	[1.0e-12 -]	[1.0e-12 -]
Vector:	no	no
Vector_Bounds:	-	-
Null_Allowed:	yes	yes

Description: The adc_bridge is one of two node bridge devices designed to allow for the ready transfer of analog information to digital values and back again. The second device is the dac_bridge (which takes a digital value and maps it to an analog one).

The adc_bridge takes as input an analog value from an analog node. This value by definition may be in the form of a voltage, or a current. If the input value is less than or equal to in_low, then a digital output value of "0" is generated. If the input is greater than or equal to in_high, a digital output value of "1" is generated. If neither of these is true, then a digital "UNKNOWN" value is output. Note that unlike the case of the dac_bridge, no ramping time or delay is associated with the adc_bridge. Rather, the continuous ramping of the input value provides for any associated delays in the digitized signal.

Example SPICE Usage:

```
abridge2 1 8 adc_buff
.model adc_buff adc_bridge(in_low = 0.3 in_high = 3.5)
```

3.5.2.3 Controlled Digital Oscillator

```
NAME_TABLE:
C_Function_Name:    cm_d_osc
Spice_Model_Name:   d_osc
Description:        "controlled digital oscillator"

PORT_TABLE:
Port Name:          cntl_in             out
Description:        "control input"     "output"
Direction:          in                  out
Default_Type:       v                   d
Allowed_Types:      [v,vd,i,id]         [d]
Vector:             no                  no
Vector_Bounds:      -                   -
Null_Allowed:       no                  no

PARAMETER_TABLE:
Parameter_Name:     cntl_array          freq_array
Description:        "control array"     "frequency array"
Data_Type:          real                real
Default_Value:      0.0                 1.0e6
Limits:             -                   [0 -]
Vector:             yes                 yes
Vector_Bounds:      [2 -]               cntl_array
Null_Allowed:       no                  no

PARAMETER_TABLE:
Parameter_Name:     duty_cycle          init_phase
Description:        "duty cycle"        "initial phase of output"
Data_Type:          real                real
Default_Value:      0.5                 0
Limits:             [1e-6 0.999999]     [-180.0 +360.0]
Vector:             no                  no
Vector_Bounds:      -                   -
Null_Allowed:       yes                 yes

PARAMETER_TABLE:
Parameter_Name:     rise_delay          fall_delay
Description:        "rise delay"        "fall delay"
Data_Type:          real                real
```

Default_Value:	1e-9	1e-9
Limits:	[0 -]	[0 -]
Vector:	no	no
Vector_Bounds:	-	-
Null_Allowed:	yes	yes

Description: The digital oscillator is a hybrid model which accepts as input a voltage or current. This input is compared to the voltage-to-frequency transfer characteristic specified by the cntl_array/freq_array coordinate pairs, and a frequency is obtained which represents a linear interpolation or extrapolation based on those pairs. A digital time-varying signal is then produced with this fundamental frequency.

The output waveform, which is the equivalent of a digital clock signal, has rise and fall delays which can be specified independently. In addition, the duty cycle and the phase of the waveform are also variable and can be set by you.

```
Example SPICE Usage:

a5 1 8 var_clock
.model var_clock d_osc(cntl_array = [-2 -1 1 2]
+                      freq_array = [1e3 1e3 10e3 10e3]
+                      duty_cycle = 0.4 init_phase = 180.0
+                      rise_delay = 10e-9 fall_delay=8e-9)
```

3.5.3 Digital Models

The following digital models are supplied with XSPICE. The descriptions included below consist of an example model Interface Specification File and a description of the model's operation. This is followed by an example of a simulator-deck placement of the model, including the .MODEL card and the specification of all available parameters. Note that these models have not been finalized at this time.

Some information common to all digital models and/or digital nodes is included here. The following are general rules which should make working with digital nodes and models more straightforward:

1. All digital nodes are initialized to ZERO at the start of a simulation (i.e., when INIT=TRUE). This means that a model need not post an explicit value to an output node upon initialization if its output would normally be a ZERO (although posting such would certainly cause no harm).

3.5.3.1 Buffer

```
NAME_TABLE:
C_Function_Name:    cm_d_buffer
Spice_Model_Name:   d_buffer
Description:        "digital one-bit-wide buffer"


PORT_TABLE:
Port Name:          in                  out
Description:        "input"             "output"
Direction:          in                  out
Default_Type:       d                   d
Allowed_Types:      [d]                 [d]
Vector:             no                  no
Vector_Bounds:      -                   -
Null_Allowed:       no                  no


PARAMETER_TABLE:
Parameter_Name:     rise_delay          fall_delay
Description:        "rise delay"        "fall delay"
Data_Type:          real                real
Default_Value:      1.0e-9              1.0e-9
Limits:             [1.0e-12 -]         [1.0e-12 -]
Vector:             no                  no
Vector_Bounds:      -                   -
Null_Allowed:       yes                 yes

PARAMETER_TABLE:
Parameter_Name:     input_load
Description:        "input load value (F)"
Data_Type:          real
Default_Value:      1.0e-12
Limits:             -
Vector:             no
Vector_Bounds:      -
Null_Allowed:       yes
```

Description: The buffer is a single-input, single-output digital buffer which produces as output a time-delayed copy of its input. The delays associated with an output rise and those associated with an output fall may be different. The model also posts an input load

value (in farads) based on the parameter input_load. The output of this model does NOT, however, respond to the total loading it sees on its output; it will always drive the output strongly with the specified delays.

Example SPICE Usage:

```
a6 1 8 buff1
.model buff1 d_buffer(rise_delay = 0.5e-9 fall_delay = 0.3e-9
+                     input_load = 0.5e-12)
```

3.5.3.2 Inverter

```
NAME_TABLE:
C_Function_Name:    cm_d_inverter
Spice_Model_Name:   d_inverter
Description:        "digital one-bit-wide inverter"


PORT_TABLE:
Port Name:          in                  out
Description:        "input"             "output"
Direction:          in                  out
Default_Type:       d                   d
Allowed_Types:      [d]                 [d]
Vector:             no                  no
Vector_Bounds:      -                   -
Null_Allowed:       no                  no


PARAMETER_TABLE:
Parameter_Name:     rise_delay          fall_delay
Description:        "rise delay"        "fall delay"
Data_Type:          real                real
Default_Value:      1.0e-9              1.0e-9
Limits:             [1.0e-12 -]         [1.0e-12 -]
Vector:             no                  no
Vector_Bounds:      -                   -
Null_Allowed:       yes                 yes

PARAMETER_TABLE:
Parameter_Name:     input_load
Description:        "input load value (F)"
Data_Type:          real
Default_Value:      1.0e-12
Limits:             -
Vector:             no
Vector_Bounds:      -
Null_Allowed:       yes
```

Description: The inverter is a single-input, single-output digital inverter which produces as output an inverted, time- delayed copy of its input. The delays associated with an output rise and those associated with an output fall may be specified independently. The model

also posts an input load value (in farads) based on the parameter input_load. The output of this model does NOT, however, respond to the total loading it sees on its output; it will always drive the output strongly with the specified delays.

```
Example SPICE Usage:

a6 1 8 inv1
.model inv1 d_inverter(rise_delay = 0.5e-9 fall_delay = 0.3e-9
+                      input_load = 0.5e-12)
```

3.5.3.3 And

```
NAME_TABLE:
C_Function_Name:    cm_d_and
Spice_Model_Name:   d_and
Description:        "digital 'and' gate"


PORT_TABLE:
Port Name:          in                  out
Description:        "input"             "output"
Direction:          in                  out
Default_Type:       d                   d
Allowed_Types:      [d]                 [d]
Vector:             yes                 no
Vector_Bounds:      [2 -]               -
Null_Allowed:       no                  no


PARAMETER_TABLE:
Parameter_Name:     rise_delay          fall_delay
Description:        "rise delay"        "fall delay"
Data_Type:          real                real
Default_Value:      1.0e-9              1.0e-9
Limits:             [1.0e-12 -]         [1.0e-12 -]
Vector:             no                  no
Vector_Bounds:      -                   -
Null_Allowed:       yes                 yes

PARAMETER_TABLE:
Parameter_Name:     input_load
Description:        "input load value (F)"
Data_Type:          real
Default_Value:      1.0e-12
Limits:             -
Vector:             no
Vector_Bounds:      -
Null_Allowed:       yes
```

Description: The digital 'and' gate is an n-input, single-output 'and' gate which produces an active "1" value if, and only if, all of its inputs are also "1" values. If ANY of the inputs is a "0", the output will also be a "0"; if neither of these conditions holds, the output will be

unknown. The delays associated with an output rise and those associated with an output fall may be specified independently. The model also posts an input load value (in farads) based on the parameter input_load. The output of this model does NOT, however, respond to the total loading it sees on its output; it will always drive the output strongly with the specified delays.

```
Example SPICE Usage:

a6 [1 2] 8 and1
.model and1 d_and(rise_delay = 0.5e-9 fall_delay = 0.3e-9
+                 input_load = 0.5e-12)
```

3.5.3.4 Nand

```
NAME_TABLE:
C_Function_Name:    cm_d_nand
Spice_Model_Name:   d_nand
Description:        "digital 'nand' gate"


PORT_TABLE:
Port Name:          in                  out
Description:        "input"             "output"
Direction:          in                  out
Default_Type:       d                   d
Allowed_Types:      [d]                 [d]
Vector:             yes                 no
Vector_Bounds:      [2 -]               -
Null_Allowed:       no                  no


parameter_TABLE:
Parameter_Name:     rise_delay          fall_delay
Description:        "rise delay"        "fall delay"
Data_Type:          real                real
Default_Value:      1.0e-9              1.0e-9
Limits:             [1.0e-12 -]         [1.0e-12 -]
Vector:             no                  no
Vector_Bounds:      -                   -
Null_Allowed:       yes                 yes

PARAMETER_TABLE:
Parameter_Name:     input_load
Description:        "input load value (F)"
Data_Type:          real
Default_Value:      1.0e-12
Limits:             -
Vector:             no
Vector_Bounds:      -
Null_Allowed:       yes
```

Description: The digital 'nand' gate is an n-input, single-output 'nand' gate which produces an active "0" value if and only if all of its inputs are "1" values. If ANY of the inputs is a "0", the output will be a "1"; if neither of these conditions holds, the output will be

unknown. The delays associated with an output rise and those associated with an output fall may be specified independently. The model also posts an input load value (in farads) based on the parameter input_load. The output of this model does NOT, however, respond to the total loading it sees on its output; it will always drive the output strongly with the specified delays.

```
Example SPICE Usage:

a6 [1 2 3] 8 nand1
.model nand1 d_nand(rise_delay = 0.5e-9 fall_delay = 0.3e-9
+                   input_load = 0.5e-12)
```

3.5.3.5 Or

```
NAME_TABLE:
C_Function_Name:    cm_d_or
Spice_Model_Name:   d_or
Description:        "digital 'or' gate"


PORT_TABLE:
Port Name:          in                  out
Description:        "input"             "output"
Direction:          in                  out
Default_Type:       d                   d
Allowed_Types:      [d]                 [d]
Vector:             yes                 no
Vector_Bounds:      [2 -]               -
Null_Allowed:       no                  no


PARAMETER_TABLE:
Parameter_Name:     rise_delay          fall_delay
Description:        "rise delay"        "fall delay"
Data_Type:          real                real
Default_Value:      1.0e-9              1.0e-9
Limits:             [1.0e-12 -]         [1.0e-12 -]
Vector:             no                  no
Vector_Bounds:      -                   -
Null_Allowed:       yes                 yes

PARAMETER_TABLE:
Parameter_Name:     input_load
Description:        "input load value (F)"
Data_Type:          real
Default_Value:      1.0e-12
Limits:             -
Vector:             no
Vector_Bounds:      -
Null_Allowed:       yes
```

Description: The digital 'or' gate is an n-input, single-output 'or' gate which produces an active "1" value if at least one of its inputs is a "1" value. The gate produces a "0" value if all inputs are "0"; if neither of these two conditions holds, the output is unknown. The

delays associated with an output rise and those associated with an output fall may be specified independently. The model also posts an input load value (in farads) based on the parameter input_load. The output of this model does NOT, however, respond to the total loading it sees on its output; it will always drive the output strongly with the specified delays.

```
Example SPICE Usage:

a6 [1 2 3] 8 or1
.model or1 d_or(rise_delay = 0.5e-9 fall_delay = 0.3e-9
+               input_load = 0.5e-12)
```

3.5.3.6 Nor

```
NAME_TABLE:
C_Function_Name:    cm_d_nor
Spice_Model_Name:   d_nor
Description:        "digital 'nor' gate"


PORT_TABLE:
Port Name:          in                  out
Description:        "input"             "output"
Direction:          in                  out
Default_Type:       d                   d
Allowed_Types:      [d]                 [d]
Vector:             yes                 no
Vector_Bounds:      [2 -]               -
Null_Allowed:       no                  no


PARAMETER_TABLE:
Parameter_Name:     rise_delay          fall_delay
Description:        "rise delay"        "fall delay"
Data_Type:          real                real
Default_Value:      1.0e-9              1.0e-9
Limits:             [1.0e-12 -]         [1.0e-12 -]
Vector:             no                  no
Vector_Bounds:      -                   -
Null_Allowed:       yes                 yes

PARAMETER_TABLE:
Parameter_Name:     input_load
Description:        "input load value (F)"
Data_Type:          real
Default_Value:      1.0e-12
Limits:             -
Vector:             no
Vector_Bounds       -
Null_Allowed:       yes
```

Description: The digital 'nor' gate is an n-input, single-output 'nor' gate which produces an active "0" value if at least one of its inputs is a "1" value. The gate produces a "0" value if all inputs are "0"; if neither of these two conditions holds, the output is unknown.

The delays associated with an output rise and those associated with an output fall may be specified independently. The model also posts an input load value (in farads) based on the parameter input_load. The output of this model does NOT, however, respond to the total loading it sees on its output; it will always drive the output strongly with the specified delays.

```
Example SPICE Usage:

anor12 [1 2 3 4] 8 nor12
.model nor12 d_or(rise_delay = 0.5e-9 fall_delay = 0.3e-9
+                 input_load = 0.5e-12)
```

3.5.3.7 Xor

```
NAME_TABLE:
C_Function_Name:    cm_d_xor
Spice_Model_Name:   d_xor
Description:        "digital exclusive-or gate"


PORT_TABLE:
Port Name:          in                  out
Description:        "input"             "output"
Direction:          in                  out
Default_Type:       d                   d
Allowed_Types:      [d]                 [d]
Vector:             yes                 no
Vector_Bounds:      [2 -]               -
Null_Allowed:       no                  no


PARAMETER_TABLE:
Parameter_Name:     rise_delay          fall_delay
Description:        "rise delay"        "fall delay"
Data_Type:          real                real
Default_Value:      1.0e-9              1.0e-9
Limits:             [1.0e-12 -]         [1.0e-12 -]
Vector:             no                  no
Vector_Bounds:      -                   -
Null_Allowed:       yes                 yes

PARAMETER_TABLE:
Parameter_Name:     input_load
Description:        "input load value (F)"
Data_Type:          real
Default_Value:      1.0e-12
Limits:             -
Vector:             no
Vector_Bounds:      -
Null_Allowed:       yes
```

Description: The digital 'xor' gate is an n-input, single-output 'xor' gate which produces an active "1" value if an odd number of its inputs are also "1" values. The delays associated with an output rise and those associated with an output fall may be specified independently.

The model also posts an input load value (in farads) based on the parameter input_load. The output of this model does NOT, however, respond to the total loading it sees on its output; it will always drive the output strongly with the specified delays. Note also that to maintain the technology-independence of the model, any UNKNOWN input, or any floating input causes the output to also go UNKNOWN.

```
Example SPICE Usage:

a9 [1 2] 8 xor3
.model xor3 d_xor(rise_delay = 0.5e-9 fall_delay = 0.3e-9
+                 input_load = 0.5e-12)
```

3.5.3.8 Xnor

```
NAME_TABLE:
C_Function_Name:    cm_d_xnor
Spice_Model_Name:   d_xnor
Description:        "digital exclusive-nor gate"


PORT_TABLE:
Port Name:          in                  out
Description:        "input"             "output"
Direction:          in                  out
Default_Type:       d                   d
Allowed_Types:      [d]                 [d]
Vector:             yes                 no
Vector_Bounds:      [2 -]               -
Null_Allowed:       no                  no


PARAMETER_TABLE:
Parameter_Name:     rise_delay          fall_delay
Description:        "rise delay"        "fall delay"
Data_Type:          real                real
Default_Value:      1.0e-9              1.0e-9
Limits:             [1.0e-12 -]         [1.0e-12 -]
Vector:             no                  no
Vector_Bounds:      -                   -
Null_Allowed:       yes                 yes

PARAMETER_TABLE:
Parameter_Name:     input_load
Description:        "input load value (F)"
Data_Type:          real
Default_Value:      1.0e-12
Limits:             -
Vector:             no
Vector_Bounds:      -
Null_Allowed:       yes
```

Description: The digital 'xnor' gate is an n-input, single-output 'xnor' gate which produces an active "0" value if an odd number of its inputs are also "1" values. It produces a "1" output when an even number of "1" values occurs on its inputs. The delays associated with

an output rise and those associated with an output fall may be specified independently. The model also posts an input load value (in farads) based on the parameter input_load. The output of this model does NOT, however, respond to the total loading it sees on its output; it will always drive the output strongly with the specified delays. Note also that to maintain the technology-independence of the model, any UNKNOWN input, or any floating input causes the output to also go UNKNOWN.

```
Example SPICE Usage:

a9 [1 2] 8 xnor3
.model xnor3 d_xnor(rise_delay = 0.5e-9 fall_delay = 0.3e-9
+                   input_load = 0.5e-12)
```

3.5.3.9 Tristate

```
NAME_TABLE:
C_Function_Name:    cm_d_tristate
Spice_Model_Name:   d_tristate
Description:        "digital tristate buffer"


PORT_TABLE:
Port Name:          in                  enable          out
Description:        "input"           "enable"       "output"
Direction:          in                  in              out
Default_Type:       d                   d               d
Allowed_Types:      [d]                 [d]             [d]
Vector:             no                  no              no
Vector_Bounds:      -                   -               -
Null_Allowed:       no                  no              no


PARAMETER_TABLE:
Parameter_Name:     delay
Description:        "delay"
Data_Type:          real
Default_Value:      1.0e-9
Limits:             [1.0e-12 -]
Vector:             no
Vector_Bounds:      -
Null_Allowed:       yes

PARAMETER_TABLE:
Parameter_Name:     input_load
Description:        "input load value (F)"
Data_Type:          real
Default_Value:      1.0e-12
Limits:             -
Vector:             no
Vector_Bounds:      -
Null_Allowed:       yes
```

```
PARAMETER_TABLE:
Parameter_Name:     enable_load
Description:        "enable load value (F)"
Data_Type:          real
Default_Value:      1.0e-12
Limits:             -
Vector:             no
Vector_Bounds:      -
Null_Allowed:       yes
```

Description: The digital tristate is a simple tristate gate which can be configured to allow for open-collector behavior, as well as standard tristate behavior. The state seen on the input line is reflected in the output. The state seen on the enable line determines the strength of the output. Thus, a ONE forces the output to its state with a STRONG strength. A ZERO forces the output to go to a HI_IMPEDANCE strength. The delays associated with an output state or strength change cannot be specified independently, nor may they be specified independently for rise or fall conditions; other gate models may be used to provide such delays if needed. The model posts input and enable load values (in farads) based on the parameters input_load and enable.The output of this model does NOT, however, respond to the total loading it sees on its output; it will always drive the output with the specified delay. Note also that to maintain the technology-independence of the model, any UNKNOWN input, or any floating input causes the output to also go UNKNOWN. Likewise, any UNKNOWN input on the enable line causes the output to go to an UNDETERMINED strength value.

```
Example SPICE Usage:

a9 1 2 8 tri7
.model tri7 d_tristate(delay = 0.5e-9 input_load = 0.5e-12
+                      enable_load = 0.5e-12)
```

3.5.3.10 Pullup

```
NAME_TABLE:
C_Function_Name:    cm_d_pullup
Spice_Model_Name:   d_pullup
Description:        "digital pullup resistor"


PORT_TABLE:
Port Name:          out
Description:        "output"
Direction:          out
Default_Type:       d
Allowed_Types:      [d]
Vector:             no
Vector_Bounds:      -
Null_Allowed:       no


PARAMETER_TABLE:
Parameter_Name:     load
Description:        "load value (F)"
Data_Type:          real
Default_Value:      1.0e-12
Limits:             -
Vector:             no
Vector_Bounds:      -
Null_Allowed:       yes
```

Description: The digital pullup resistor is a device which emulates the behavior of an analog resistance value tied to a high voltage level. The pullup may be used in conjunction with tristate buffers to provide open-collector wired "or" constructs, or any other logical constructs which rely on a resistive pullup common to many tristated output devices. The model posts an input load value (in farads) based on the parameters "load".

```
Example SPICE Usage:

a2 9 pullup1
.model pullup1 d_pullup(load = 20.0e-12)
```

3.5.3.11 Pulldown

```
NAME_TABLE:
C_Function_Name:    cm_d_pulldown
Spice_Model_Name:   d_pulldown
Description:        "digital pulldown resistor"


PORT_TABLE:
Port Name:          out
Description:        "output"
Direction:          out
Default_Type:       d
Allowed_Types:      [d]
Vector:             no
Vector_Bounds:      -
Null_Allowed:       no


PARAMETER_TABLE:
Parameter_Name:     load
Description:        "load value (F)"
Data_Type:          real
Default_Value:      1.0e-12
Limits:             -
Vector:             no
Vector_Bounds:      -
Null_Allowed:       yes
```

Description: The digital pulldown resistor is a device which emulates the behavior of an analog resistance value tied to a low voltage level. The pulldown may be used in conjunction with tristate buffers to provide open-collector wired "or" constructs, or any other logical constructs which rely on a resistive pulldown common to many tristated output devices. The model posts an input load value (in farads) based on the parameters "load".

```
Example SPICE Usage:

a4 9 pulldown1
.model pulldown1 d_pulldown(load = 20.0e-12)
```

3.5.3.12 D Flip Flop

```
NAME_TABLE:
C_Function_Name:    cm_d_dff
Spice_Model_Name:   d_dff
Description:        "digital d-type flip flop"

PORT_TABLE:
Port Name:          data                clk
Description:        "input data"        "clock"
Direction:          in                  in
Default_Type:       d                   d
Allowed_Types:      [d]                 [d]
Vector:             no                  no
Vector_Bounds:      -                   -
Null_Allowed:       no                  no

PORT_TABLE:
Port Name:          set                 reset
Description:        "asynch. set"       "asynch. reset"
Direction:          in                  in
Default_Type:       d                   d
Allowed_Types:      [d]                 [d]
Vector:             no                  no
Vector_Bounds:      -                   -
Null_Allowed:       yes                 yes

PORT_TABLE:
Port Name:          out                 Nout
Description:        "data output"       "inverted data output"
Direction:          out                 out
Default_Type:       d                   d
Allowed_Types:      [d]                 [d]
Vector:             no                  no
Vector_Bounds:      -                   -
Null_Allowed:       yes                 yes

PARAMETER_TABLE:
Parameter_Name:     clk_delay           set_delay
Description:        "delay from clk"    "delay from set"
Data_Type:          real                real
```

```
Default_Value:      1.0e-9              1.0e-9
Limits:             [1.0e-12 -]         [1.0e-12 -]
Vector:             no                  no
Vector_Bounds:      -                   -
Null_Allowed:       yes                 yes

PARAMETER_TABLE:
Parameter_Name:     reset_delay              ic
Description:        "delay from reset"       "output initial
                                              state"
Data_Type:          real                     int
Default_Value:      1.0                      0
Limits:             [1.0e-12 -]              [0 2]
Vector:             no                       no
Vector_Bounds:      -                        -
Null_Allowed:       yes                      yes

PARAMETER_TABLE:
Parameter_Name:     data_load                clk_load
Description:        "data load value (F)"    "clk load value (F)"
Data_Type:          real                     real
Default_Value:      1.0e-12                  1.0e-12
Limits:             -                        -
Vector:             no                       no
Vector_Bounds:      -                        -
Null_Allowed:       yes                      yes

PARAMETER_TABLE:
Parameter_Name:     set_load                 reset_load
Description:        "set load value (F)"     "reset load (F)"
Data_Type:          real                     real
Default_Value:      1.0e-12                  1.0e-12
Limits:             -                        -
Vector:             no                       no
Vector_Bounds:      -                        -
Null_Allowed:       yes                      yes

PARAMETER_TABLE:
Parameter_Name:     rise_delay          fall_delay
Description:        "rise delay"        "fall delay"
Data_Type:          real                real
Default_Value:      1.0e-9              1.0e-9
```

```
Limits:             [1.0e-12 -]         [1.0e-12 -]
Vector:             no                  no
Vector_Bounds:      -                   -
Null_Allowed:       yes                 yes
```

Description: The digital d-type flip flop is a one-bit, edge-triggered storage element which will store data whenever the clk input line transitions from low to high (ZERO to ONE). In addition, asynchronous set and reset signals exist, and each of the three methods of changing the stored output of the d_dff have separate load values and delays associated with them. Additionally, you may specify separate rise and fall delay values that are added to those specified for the input lines; these allow for more faithful reproduction of the output characteristics of different IC fabrication technologies.

Note that any UNKNOWN input on the set or reset lines immediately results in an UNKNOWN output.

```
Example SPICE Usage:

a7 1 2 3 4 5 6 flop1
.model flop1 d_dff(clk_delay = 13.0e-9 set_delay = 25.0e-9
+                  reset_delay = 27.0e-9 ic = 2 rise_delay = 10.0e-9
+                  fall_delay = 3e-9)
```

3.5.3.13 JK Flip Flop

```
NAME_TABLE:
C_Function_Name:    cm_d_jkff
Spice_Model_Name:   d_jkff
Description:        "digital jk-type flip flop"

PORT_TABLE:
Port Name:          j                   k
Description:        "j input"           "k input"
Direction:          in                  in
Default_Type:       d                   d
Allowed_Types:      [d]                 [d]
Vector:             yes                 yes
Vector_Bounds:      [2 -]               j
Null_Allowed:       no                  no

PORT_TABLE:
Port Name:          clk
Description:        "clock"
Direction:          in
Default_Type:       d
Allowed_Types:      [d]
Vector:             no
Vector_Bounds:      -
Null_Allowed:       no

PORT_TABLE:
Port Name:          set                 reset
Description:        "asynchronous set"  "asynchronous reset"
Direction:          in                  in
Default_Type:       d                   d
Allowed_Types:      [d]                 [d]
Vector:             no                  no
Vector_Bounds:      -                   -
Null_Allowed:       yes                 yes

PORT_TABLE:
Port Name:          out                 Nout
Description:        "data output"       "inverted data output"
Direction:          out                 out
```

```
Default_Type:       d                   d
Allowed_Types:      [d]                 [d]
Vector:             no                  no
Vector_Bounds       -                   -
Null_Allowed:       yes                 yes


PARAMETER_TABLE:
Parameter_Name:     clk_delay           set_delay
Description:        "delay from clk"    "delay from set"
Data_Type:          real                real
Default_Value:      1.0e-9              1.0e-9
Limits:             [1.0e-12 -]         [1.0e-12 -]
Vector:             no                  no
Vector_Bounds:      -                   -
Null_Allowed:       yes                 yes

PARAMETER_TABLE:
Parameter_Name:     reset_delay             ic
Description:        "delay from reset"      "output initial state"
Data_Type:          real                    int
Default_Value:      1.0                     0
Limits:             [1.0e-12 -]             [0 2]
Vector:             no                      no
Vector_Bounds:      -                       -
Null_Allowed:       yes                     yes

PARAMETER_TABLE:
Parameter_Name:     jk_load                 clk_load
Description:        "j,k load values (F)"   "clk load value (F)"
Data_Type:          real                    real
Default_Value:      1.0e-12                 1.0e-12
Limits:             -                       -
Vector:             no                      no
Vector_Bounds:      -                       -
Null_Allowed:       yes                     yes

PARAMETER_TABLE:
Parameter_Name:     set_load                reset_load
Description:        "set load value (F)"    "reset load (F)"
Data_Type:          real                    real
Default_Value:      1.0e-12                 1.0e-12
```

```
Limits:             -                       -
Vector:             no                      no
Vector_Bounds:      -                       -
Null_Allowed:       yes                     yes

PARAMETER_TABLE:
Parameter_Name:     rise_delay          fall_delay
Description:        "rise delay"        "fall delay"
Data_Type:          real                real
Default_Value:      1.0e-9              1.0e-9
Limits:             [1.0e-12 -]         [1.0e-12 -]
Vector:             no                  no
Vector_Bounds:      -                   -
Null_Allowed:       yes                 yes
```

Description: The digital jk-type flip flop is a one-bit, edge-triggered storage element which will store data whenever the clk input line transitions from low to high (ZERO to ONE). In addition, asynchronous set and reset signals exist, and each of the three methods of changing the stored output of the d_jkff have separate load values and delays associated with them. Additionally, you may specify separate rise and fall delay values that are added to those specified for the input lines; these allow for more faithful reproduction of the output characteristics of different IC fabrication technologies.

Note that any UNKNOWN inputs other than j or k cause the output to go UNKNOWN automatically.

```
Example SPICE Usage:

a8 1 2 3 4 5 6 7 flop2
.model flop2 d_jkff(clk_delay = 13.0e-9 set_delay = 25.0e-9
+                   reset_delay = 27.0e-9 ic = 2 rise_delay = 10.0e-9
+                   fall_delay = 3e-9)
```

3.5.3.14 Toggle Flip Flop

```
NAME_TABLE:
C_Function_Name:    cm_d_tff
Spice_Model_Name:   d_tff
Description:        "digital toggle flip flop"

PORT_TABLE:
Port Name:          t                   clk
Description:        "toggle input"      "clock"
Direction:          in                  in
Default_Type:       d                   d
Allowed_Types:      [d]                 [d]
Vector:             yes                 no
Vector_Bounds:      [2 -]               -
Null_Allowed:       no                  no

PORT_TABLE:
Port Name:          set                 reset
Description:        "set"               "reset"
Direction:          in                  in
Default_Type:       d                   d
Allowed_Types:      [d]                 [d]
Vector:             no                  no
Vector_Bounds:      -                   -
Null_Allowed:       yes                 yes

PORT_TABLE:
Port Name:          out                 Nout
Description:        "data output"       "inverted data output"
Direction:          out                 out
Default_Type:       d                   d
Allowed_Types:      [d]                 [d]
Vector:             no                  no
Vector_Bounds:      -                   -
Null_Allowed:       yes                 yes


PARAMETER_TABLE:
Parameter_Name:     clk_delay           set_delay
Description:        "delay from clk"    "delay from set"
```

```
Data_Type:          real                real
Default_Value:      1.0e-9              1.0e-9
Limits:             [1.0e-12 -]         [1.0e-12 -]
Vector:             no                  no
Vector_Bounds       -                   -
Null_Allowed:       yes                 yes

PARAMETER_TABLE:
Parameter_Name:     reset_delay             ic
Description:        "delay from reset"      "output initial state"
Data_Type:          real                    int
Default_Value:      1.0                     0
Limits:             [1.0e-12 -]             [0 2]
Vector:             no                      no
Vector_Bounds:      -                       -
Null_Allowed:       yes                     yes

PARAMETER_TABLE:
Parameter_Name:     t_load                  clk_load
Description:        "toggle load value (F)" "clk load value (F)"
Data_Type:          real                    real
Default_Value:      1.0e-12                 1.0e-12
Limits:             -                       -
Vector:             no                      no
Vector_Bounds:      -                       -
Null_Allowed:       yes                     yes

PARAMETER_TABLE:
Parameter_Name:     set_load                reset_load
Description:        "set load value (F)"    "reset load (F)"
Data_Type:          real                    real
Default_Value:      1.0e-12                 1.0e-12
Limits:             -                       -
Vector:             no                      no
Vector_Bounds:      -                       -
Null_Allowed:       yes                     yes

PARAMETER_TABLE:
Parameter_Name:     rise_delay          fall_delay
Description:        "rise delay"        "fall delay"
Data_Type:          real                real
Default_Value:      1.0e-9              1.0e-9
```

```
Limits:             [1.0e-12 -]         [1.0e-12 -]
Vector:             no                  no
Vector_Bounds:      -                   -
Null_Allowed:       yes                 yes
```

Description: The digital toggle-type flip flop is a one-bit, edge-triggered storage element which will toggle its current state whenever the clk input line transitions from low to high (ZERO to ONE). In addition, asynchronous set and reset signals exist, and each of the three methods of changing the stored output of the d_tff have separate load values and delays associated with them. Additionally, you may specify separate rise and fall delay values that are added to those specified for the input lines; these allow for more faithful reproduction of the output characteristics of different IC fabrication technologies.

Note that any UNKNOWN inputs other than t immediately cause the output to go UNKNOWN.

```
Example SPICE Usage:

a8 2 12 4 5 6 3 flop3
.model flop3 d_tff(clk_delay = 13.0e-9 set_delay = 25.0e-9
+                  reset_delay = 27.0e-9 ic = 2 rise_delay = 10.0e-9
+                  fall_delay = 3e-9 t_load = 0.2e-12)
```

3.5.3.15 Set-Reset Flip Flop

```
NAME_TABLE:
C_Function_Name:    cm_d_srff
Spice_Model_Name:   d_srff
Description:        "digital set-reset flip flop"

PORT_TABLE:
Port Name:          s                   r
Description:        "set input"         "reset input"
Direction:          in                  in
Default_Type:       d                   d
Allowed_Types:      [d]                 [d]
Vector:             no                  no
Vector_Bounds:      -                   -
Null_Allowed:       no                  no

PORT_TABLE:
Port Name:          clk
Description:        "clock"
Direction:          in
Default_Type:       d
Allowed_Types:      [d]
Vector:             no
Vector_Bounds:      -
Null_Allowed:       no

PORT_TABLE:
Port Name:          set                 reset
Description:        "asynchronous set"  "asynchronous reset"
Direction:          in                  in
Default_Type:       d                   d
Allowed_Types:      [d]                 [d]
Vector:             no                  no
Vector_Bounds:      -                   -
Null_Allowed:       yes                 yes

PORT_TABLE:
Port Name:          out                 Nout
Description:        "data output"       "inverted data output"
Direction:          out                 out
```

```
Default_Type:       d                   d
Allowed_Types:      [d]                 [d]
Vector:             no                  no
Vector_Bounds:      -                   -
Null_Allowed:       yes                 yes


PARAMETER_TABLE:
Parameter_Name:     clk_delay           set_delay
Description:        "delay from clk"    "delay from set"
Data_Type:          real                real
Default_Value:      1.0e-9              1.0e-9
Limits:             [1.0e-12 -]         [1.0e-12 -]
Vector:             no                  no
Vector_Bounds:      -                   -
Null_Allowed:       yes                 yes

PARAMETER_TABLE:
Parameter_Name:     reset_delay             ic
Description:        "delay from reset"      "output initial state"
Data_Type:          real                    int
Default_Value:      1.0e-9                  0
Limits:             [1.0e-12 -]             [0 2]
Vector:             no                      no
Vector_Bounds:      -                       -
Null_Allowed:       yes                     yes

PARAMETER_TABLE:
Parameter_Name:     sr_load                 clk_load
Description:        "set/reset loads (F)"   "clk load value (F)"
Data_Type:          real                    real
Default_Value:      1.0e-12                 1.0e-12
Limits:             -                       -
Vector:             no                      no
Vector_Bounds:      -                       -
Null_Allowed:       yes                     yes

PARAMETER_TABLE:
Parameter_Name:     set_load                reset_load
Description:        "set load value (F)"    "reset load (F)"
Data_Type:          real                    real
Default_Value:      1.0e-12                 1.0e-12
```

```
Limits:             -                       -
Vector:             no                      no
Vector_Bounds:      -                       -
Null_Allowed:       yes                     yes

PARAMETER_TABLE:
Parameter_Name:     rise_delay          fall_delay
Description:        "rise delay"        "fall delay"
Data_Type:          real                real
Default_Value:      1.0e-9              1.0e-9
Limits:             [1.0e-12 -]         [1.0e-12 -]
Vector:             no                  no
Vector_Bounds:      -                   -
Null_Allowed:       yes                 yes
```

Description: The digital sr-type flip flop is a one-bit, edge-triggered storage element which will store data whenever the clk input line transitions from low to high (ZERO to ONE). The value stored (i.e., the "out" value) will depend on the s and r input pin values, and will be:

```
out=ONE              if s=ONE  and r=ZERO;
out=ZERO             if s=ZERO and r=ONE;
out=previous value   if s=ZERO and r=ZERO;
out=UNKNOWN          if s=ONE  and r=ONE;
```

In addition, asynchronous set and reset signals exist, and each of the three methods of changing the stored output of the d_srff have separate load values and delays associated with them. You may also specify separate rise and fall delay values that are added to those specified for the input lines; these allow for more faithful reproduction of the output characteristics of different IC fabrication technologies.

Note that any UNKNOWN inputs other than s and r immediately cause the output to go UNKNOWN.

```
Example SPICE Usage:

a8 2 12 4 5 6 3 14 flop7
.model flop7 d_srff(clk_delay = 13.0e-9 set_delay = 25.0e-9
+                   reset_delay = 27.0e-9 ic = 2 rise_delay = 10.0e-9
+                   fall_delay = 3e-9)
```

3.5.3.16 D Latch

```
NAME_TABLE:
C_Function_Name:    cm_d_dlatch
Spice_Model_Name:   d_dlatch
Description:        "digital d-type latch"

PORT_TABLE:
Port Name:          data                enable
Description:        "input data"        "enable input"
Direction:          in                  in
Default_Type:       d                   d
Allowed_Types:      [d]                 [d]
Vector:             no                  no
Vector_Bounds:      -                   -
Null_Allowed:       no                  no

PORT_TABLE:
Port Name:          set                 reset
Description:        "set"               "reset"
Direction:          in                  in
Default_Type:       d                   d
Allowed_Types:      [d]                 [d]
Vector:             no                  no
Vector_Bounds:      -                   -
Null_Allowed:       yes                 yes

PORT_TABLE:
Port Name:          out                 Nout
Description:        "data output"       "inverter data output"
Direction:          out                 out
Default_Type:       d                   d
Allowed_Types:      [d]                 [d]
Vector:             no                  no
Vector_Bounds:      -                   -
Null_Allowed:       no                  no

PARAMETER_TABLE:
Parameter_Name:     data_delay
Description:        "delay from data"
Data_Type:          real
```

```
Default_Value:      1.0e-9
Limits:             [1.0e-12 -]
Vector:             no
Vector_Bounds:      -
Null_Allowed:       yes

PARAMETER_TABLE:
Parameter_Name:     enable_delay        set_delay
Description:        "delay from enable" "delay from SET"
Data_Type:          real                real
Default_Value:      1.0e-9              1.0e-9
Limits:             [1.0e-12 -]         [1.0e-12 -]
Vector:             no                  no
Vector_Bounds:      -                   -
Null_Allowed:       yes                 yes

PARAMETER_TABLE:
Parameter_Name:     reset_delay             ic
Description:        "delay from RESET"      "output initial state"
Data_Type:          real                    boolean
Default_Value:      1.0e-9                  0
Limits:             [1.0e-12 -]             -
Vector:             no                      no
Vector_Bounds:      -                       -
Null_Allowed:       yes                     yes

PARAMETER_TABLE:
Parameter_Name:     data_load               enable_load
Description:        "data load (F)"         "enable load value (F)"
Data_Type:          real                    real
Default_Value:      1.0e-12                 1.0e-12
Limits:             -                       -
Vector:             no                      no
Vector_Bounds:      -                       -
Null_Allowed:       yes                     yes

PARAMETER_TABLE:
Parameter_Name:     set_load                reset_load
Description:        "set load value (F)"    "reset load (F)"
Data_Type:          real                    real
Default_Value:      1.0e-12                 1.0e-12
Limits:             -                       -
```

```
Vector:             no                      no
Vector_Bounds:      -                       -
Null_Allowed:       yes                     yes

PARAMETER_TABLE:
Parameter_Name:     rise_delay          fall_delay
Description:        "rise delay"        "fall delay"
Data_Type:          real                real
Default_Value:      1.0e-9              1.0e-9
Limits:             [1.0e-12 -]         [1.0e-12 -]
Vector:             no                  no
Vector_Bounds:      -                   -
Null_Allowed:       yes                 yes
```

Description: The digital d-type latch is a one-bit, level-sensitive storage element which will output the value on the data line whenever the enable input line is high (ONE). The value on the data line is stored (i.e., held on the out line) whenever the enable line is low (ZERO).

In addition, asynchronous set and reset signals exist, and each of the four methods of changing the stored output of the d_dlatch (i.e., data changing with enable=ONE, enable changing to ONE from ZERO with a new value on data, raising set and raising reset) have separate delays associated with them. You may also specify separate rise and fall delay values that are added to those specified for the input lines; these allow for more faithful reproduction of the output characteristics of different IC fabrication technologies.

Note that any UNKNOWN inputs other than on the data line when enable=ZERO immediately cause the output to go UNKNOWN.

```
Example SPICE Usage:

a4 12 4 5 6 3 14 latch1
.model latch1 d_dlatch(data_delay = 13.0e-9 enable_delay = 22.0e-9
+                      set_delay = 25.0e-9
+                      reset_delay = 27.0e-9 ic = 2
+                      rise_delay = 10.0e-9 fall_delay = 3e-9)
```

3.5.3.17 Set-Reset Latch

```
NAME_TABLE:
C_Function_Name:    cm_d_srlatch
Spice_Model_Name:   d_srlatch
Description:        "digital sr-type latch"

PORT_TABLE:
Port Name:          s                   r
Description:        "set"               "reset"
Direction:          in                  in
Default_Type:       d                   d
Allowed_Types:      [d]                 [d]
Vector:             yes                 yes
Vector_Bounds:      [2 -]               r
Null_Allowed:       no                  no

PORT_TABLE:
Port Name:          enable
Description:        "enable"
Direction:          in
Default_Type:       d
Allowed_Types:      [d]
Vector:             no
Vector_Bounds:      -
Null_Allowed:       no

PORT_TABLE:
Port Name:          set                 reset
Description:        "set"               "reset"
Direction:          in                  in
Default_Type:       d                   d
Allowed_Types:      [d]                 [d]
Vector:             no                  no
Vector_Bounds:      -                   -
Null_Allowed:       yes                 yes

PORT_TABLE:
Port Name:          out                 Nout
Description:        "data output"       "inverted data output"
Direction:          out                 out
```

```
Default_Type:       d                   d
Allowed_Types:      [d]                 [d]
Vector:             no                  no
Vector_Bounds:      -                   -
Null_Allowed:       no                  no


PARAMETER_TABLE:
Parameter_Name:     sr_delay
Description:        "delay from s or r input change"
Data_Type:          real
Default_Value:      1.0e-9
Limits:             [1.0e-12 -]
Vector:             no
Vector_Bounds:      -
Null_Allowed:       yes

PARAMETER_TABLE:
Parameter_Name:     enable_delay        set_delay
Description:        "delay from enable" "delay from SET"
Data_Type:          real               real
Default_Value:      1.0e-9              1.0e-9
Limits:             [1.0e-12 -]         [1.0e-12 -]
Vector:             no                  no
Vector_Bounds:      -                   -
Null_Allowed:       yes                 yes

PARAMETER_TABLE:
Parameter_Name:     reset_delay             ic
Description:        "delay from RESET"      "output initial state"
Data_Type:          real                    boolean
Default_Value:      1.0e-9                  0
Limits:             [1.0e-12 -]             -
Vector:             no                      no
Vector_Bounds:      -                       -
Null_Allowed:       yes                     yes

PARAMETER_TABLE:
Parameter_Name:     sr_load                 enable_load
Description:        "s & r input loads (F)" "enable load value (F)"
Data_Type:          real                    real
Default_Value:      1.0e-12                 1.0e-12
```

```
Limits:             -                       -
Vector:             no                      no
Vector_Bounds:      -                       -
Null_Allowed:       yes                     yes

PARAMETER_TABLE:
Parameter_Name:     set_load                reset_load
Description:        "set load value (F)"    "reset load (F)"
Data_Type:          real                    real
Default_Value:      1.0e-12                 1.0e-12
Limits:             -                       -
Vector:             no                      no
Vector_Bounds:      -                       -
Null_Allowed:       yes                     yes

PARAMETER_TABLE:
Parameter_Name:     rise_delay          fall_delay
Description:        "rise delay"        "fall delay"
Data_Type:          real                real
Default_Value:      1.0e-9              1.0e-9
Limits:             [1.0e-12 -]         [1.0e-12 -]
Vector:             no                  no
Vector_Bounds:      -                   -
Null_Allowed:       yes                 yes
```

Description: The digital sr-type latch is a one-bit, level-sensitive storage element which will output the value dictated by the state of the s and r pins whenever the enable input line is high (ONE). This value is stored (i.e., held on the out line) whenever the enable line is low (ZERO). The particular value chosen is as shown below:

```
s=ZERO, r=ZERO => out=current value (i.e., not change in output)
s=ZERO, r=ONE  => out=ZERO
s=ONE,  r=ZERO => out=ONE
s=ONE,  r=ONE  => out=UNKNOWN
```

Asynchronous set and reset signals exist, and each of the four methods of changing the stored output of the d_srlatch (i.e., s/r combination changing with enable=ONE, enable changing to ONE from ZERO with an output-changing combination of s and r, raising set and raising reset) have separate delays associated with them. You may also specify separate rise and fall delay values that are added to those specified for the input lines; these allow for more faithful reproduction of the output characteristics of different IC fabrication technologies.

Note that any UNKNOWN inputs other than on the s and r lines when enable=ZERO immediately cause the output to go UNKNOWN.

```
Example SPICE Usage:

a4 12 4 5 6 3 14 16 latch2
.model latch2 d_srlatch(sr_delay = 13.0e-9 enable_delay = 22.0e-9
+                       set_delay = 25.0e-9
+                       reset_delay = 27.0e-9 ic = 2
+                       rise_delay = 10.0e-9 fall_delay = 3e-9)
```

3.5.3.18 State Machine

```
NAME_TABLE:
C_Function_Name:    cm_d_state
Spice_Model_Name:   d_state
Description:        "digital state machine"

PORT_TABLE:
Port Name:          in                  clk
Description:        "input"             "clock"
Direction:          in                  in
Default_Type:       d                   d
Allowed_Types:      [d]                 [d]
Vector:             yes                 no
Vector_Bounds:      [1 -]               -
Null_Allowed:       yes                 no

PORT_TABLE:
Port Name:          reset out
Description:        "reset"         "output"
Direction:          in              out
Default_Type:       d               d
Allowed_Types:      [d]             [d]
Vector:             no              yes
Vector_Bounds:      -               [1 -]
Null_Allowed:       yes             no

PARAMETER_TABLE:
Parameter_Name:     clk_delay           reset_delay
Description:        "delay from CLK"    "delay from RESET"
Data_Type:          real                real
Default_Value:      1.0e-9              1.0e-9
Limits:             [1.0e-12 -]         [1.0e-12 -]
Vector:             no                  no
Vector_Bounds:      -                   -
Null_Allowed:       yes                 yes

PARAMETER_TABLE:
Parameter_Name:     state_file
Description:        "state transition specification file name"
Data_Type:          string
```

```
Default_Value:      "state.txt"
Limits:             -
Vector:             no
Vector_Bounds:      -
Null_Allowed:       no

PARAMETER_TABLE:
Parameter_Name:     reset_state
Description:        "default state on RESET & at DC"
Data_Type:          int
Default_Value:      0
Limits:             -
Vector:             no
Vector_Bounds:      -
Null_Allowed:       no

PARAMETER_TABLE:
Parameter_Name:     input_load
Description:        "input loading capacitance (F)"
Data_Type:          real
Default_Value:      1.0e-12
Limits:             -
Vector:             no
Vector_Bounds:      -
Null_Allowed:       yes

PARAMETER_TABLE:
Parameter_Name:     clk_load
Description:        "clock loading capacitance (F)"
Data_Type:          real
Default_Value:      1.0e-12
Limits:             -
Vector:             no
Vector_Bounds:      -
Null_Allowed:       yes

PARAMETER_TABLE:
Parameter_Name:     reset_load
Description:        "reset loading capacitance (F)"
Data_Type:          real
Default_Value:      1.0e-12
Limits:             -
```

```
Vector:             no
Vector_Bounds:      -
Null_Allowed:       yes
```

Description: The digital state machine provides for straightforward descriptions of clocked combinational logic blocks with a variable number of inputs and outputs and with an unlimited number of possible states. The model can be configured to behave as virtually any type of counter or clocked combinational logic block and can be used to replace very large digital circuit schematics with an identically functional but faster representation.

The d_state model is configured through the use of a state definition file (state.in) which resides in a directory of your choosing. The file defines all states to be understood by the model, plus input bit combinations which trigger changes in state. An example state.in file is shown below:

```
---------- begin file -------------

* This is an example state.in file. This file
* defines a simple 2-bit counter with one input. The
* value of this input determines whether the counter counts
* up (in = 1) or down (in = 0).

0  0s 0s  0 -> 3
          1 -> 1

1  0s 1z  0 -> 0
          1 -> 2

2  1z 0s  0 -> 1
          1 -> 3

3  1z 1z  0 -> 2
3  1z 1z  1 -> 0
------------------ end file ---------------
```

Several attributes of the above file structure should be noted. First, ALL LINES IN THE FILE MUST BE ONE OF FOUR TYPES. These are:

1. A comment, beginning with a "*" in the first column.
2. A header line, which is a complete description of the current state, the outputs corresponding to that state, an input value, and the state that the model will

assume should that input be encountered. The first line of a state definition must ALWAYS be a header line.

3. A continuation line, which is a partial description of a state, consisting of an input value and the state that the model will assume should that input be encountered. Note that continuation lines may only be used after the initial header line definition for a state.

4. A line containing nothing but whitespace (space, formfeed, newline, carriage return, tab, vertical tab).

A line which is not one of the above will cause a file-loading error.

Note that in the example shown, whitespace (any combination of blanks, tabs, commas) is used to separate values, and that the character "->" is used to underline the state transition implied by the input preceding it. This particular character is not critical in of itself, and can be replaced with any other character or non-broken combination of characters that you prefer (e.g. "==>", ">>", ":", "resolves_to", etc.)

The order of the output and input bits in the file is important; the first column is always interpreted to refer to the "zeroth" bit of input and output. Thus, in the file above, the output from state 1 sets out[0] to "0s", and out[1] to "1z".

The state numbers need not be in any particular order, but a state definition (which consists of the sum total of all lines which define the state, its outputs, and all methods by which a state can be exited) must be made on contiguous line numbers; a state definition cannot be broken into sub-blocks and distributed randomly throughout the file. On the other hand, the state definition can be broken up by as many comment lines as you desire.

Header files may be used throughout the state.in file, and continuation lines can be discarded completely if you so choose: continuation lines are primarily provided as a convenience.

```
Example SPICE Usage:

a4 [2 3 4 5] 1 12 [22 23 24 25 26 27 28 29] state1
.model state1 d_state(clk_delay = 13.0e-9 reset_delay = 27.0e-9
                      state_file = newstate.txt reset_state = 2)
```

3.5.3.19 Frequency Divider

```
NAME_TABLE:
C_Function_Name:    cm_d_fdiv
Spice_Model_Name:   d_fdiv
Description:        "digital frequency divider"

PORT_TABLE:
Port Name:          freq_in             freq_out
Description:        "frequency input"   "frequency output"
Direction:          in                  out
Default_Type:       d                   d
Allowed_Types:      [d]                 [d]
Vector:             no                  no
Vector_Bounds:      -                   -
Null_Allowed:       no                  no

PARAMETER_TABLE:
Parameter_Name:     div_factor          high_cycles
Description:        "divide factor"     "# of cycles for high out"
Data_Type:          int                 int
Default_Value:      2                   1
Limits:             [1 -]               [1 div_factor-1]
Vector:             no                  no
Vector_Bounds:      -                   -
Null_Allowed:       yes                 yes

PARAMETER_TABLE:
Parameter_Name:     i_count
Description:        "divider initial count value"
Data_Type:          int
Default_Value:      0
Limits:             -
Vector:             no
Vector_Bounds:      -
Null_Allowed:       yes

PARAMETER_TABLE:
Parameter_Name:     rise_delay          fall_delay
Description:        "rise delay"        "fall delay"
Data_Type:          real                real
```

```
Default_Value:      1.0e-9              1.0e-9
Limits:             [1.0e-12 -]         [1.0e-12 -]
Vector:             yes                 yes
Vector_Bounds:      in                  in
Null_Allowed:       yes                 yes

PARAMETER_TABLE:
Parameter_Name:     freq_in_load
Description:        "freq_in load value (F)"
Data_Type:          real
Default_Value:      1.0e-12
Limits:             -
Vector:             no
Vector_Bounds:      -
Null_Allowed:       yes
```

Description: The digital frequency divider is a programmable step-down divider which accepts an arbitrary divisor (div_factor), a duty-cycle term (high_cycles), and an initial count value (i_count). The generated output is synchronized to the rising edges of the input signal. Rise delay and fall delay on the outputs may also be specified independently.

```
Example SPICE Usage:

a4 3 7 divider
.model divider d_fdiv(div_factor = 5 high_cycles = 3
                      i_count = 4 rise_delay = 23e-9
+                     fall_delay = 9e-9)
```

3.5.3.20 RAM

```
NAME_TABLE:
C_Function_Name:    cm_d_ram
Spice_Model_Name:   d_ram
Description:        "digital random-access memory"

PORT_TABLE:
Port Name:          data_in                      data_out
Description:        "data input line(s)"         "data output line(s)"
Direction:          in                           out
Default_Type:       d                            d
Allowed_Types:      [d]                          [d]
Vector:             yes                          yes
Vector_Bounds:      [1 -]                        data_in
Null_Allowed:       no                           no

PORT_TABLE:
Port Name:          address                   write_en
Description:        "address input line(s)"   "write enable line"
Direction:          in                        in
Default_Type:       d                         d
Allowed_Types:      [d]                       [d]
Vector:             yes                       no
Vector_Bounds:      [1 -]                     -
Null_Allowed:       no                        no

PORT_TABLE:
Port Name:          select
Description:        "chip select line(s)"
Direction:          in
Default_Type:       d
Allowed_Types:      [d]
Vector:             yes
Vector_Bounds:      [1 16]
Null_Allowed:       no


PARAMETER_TABLE:
Parameter_Name:     select_value
Description:        "decimal active value for select line comparison"
```

```
Data_Type:          int
Default_Value:      1
Limits:             [0 32767]
Vector:             no
Vector_Bounds:      -
Null_Allowed:       yes


PARAMETER_TABLE:
Parameter_Name:     ic
Description:        "initial bit state @ dc"
Data_Type:          int
Default_Value:      2
Limits:             [0 2]
Vector:             no
Vector_Bounds:      -
Null_Allowed:       yes

PARAMETER_TABLE:
Parameter_Name:     read_delay
Description:        "read delay from address/select/write_en active"
Data_Type:          real
Default_Value:      100.0e-9
Limits:             [1.0e-12 -]
Vector:             no
Vector_Bounds:      -
Null_Allowed:       yes


PARAMETER_TABLE:
Parameter_Name:     data_load                   address_load
Description:        "data_in load value (F)"    "addr. load value (F)"
Data_Type:          real                        real
Default_Value:      1.0e-12                     1.0e-12
Limits:             -                           -
Vector:             no                          no
Vector_Bounds:      -                           -
Null_Allowed:       yes                         yes

PARAMETER_TABLE:
Parameter_Name:     select_load
Description:        "select load value (F)"
```

```
Data_Type:          real
Default_Value:      1.0e-12
Limits:             -
Vector:             no
Vector_Bounds:      -
Null_Allowed:       yes

PARAMETER_TABLE:
Parameter_Name:     enable_load
Description:        "enable line load value (F)"
Data_Type:          real
Default_Value:      1.0e-12
Limits:             -
Vector:             no
Vector_Bounds:      -
Null_Allowed:       yes
```

Description: The digital RAM is an M-wide, N-deep random access memory element with programmable select lines, tristated data_out lines, and a single write/r̃ead line. The width of the RAM words (M) is set through the use of the word_width parameter. The depth of the RAM (N) is set by the number of address lines input to the device. The value of N is related to the number of address input lines (P) by the following equation:

$$2^p = N$$

There is no reset line into the device. However, an initial value for all bits may be specified by setting the ic parameter to either 0 or 1. In reading a word from the ram, the read_delay value is invoked, and output will not appear until that delay has been satisfied. Separate rise and fall delays are not supported for this device.

Note that UNKNOWN inputs on the address lines are not allowed during a write. In the event that an address line does indeed go unknown during a write, THE ENTIRE CONTENTS OF THE RAM WILL BE SET TO UNKNOWN. This is in contrast to the data_in lines being set to unknown during a write; in that case, only the selected word will be corrupted, and this is corrected once the data lines settle back to a known value. Note that protection is added to the write_en line such that extended UNKNOWN values on that line are interpreted as ZERO values. This is the equivalent of a read operation and will not corrupt the contents of the RAM. A similar mechanism exists for the select lines. If they are unknown, then it is assumed that the chip is not selected.

Detailed timing-checking routines are not provided in this model, other than for the enable_delay and select_delay restrictions on read operations. You are advised, therefore, to

carefully check the timing into and out of the RAM for correct read and write cycle times, setup and hold times, etc. for the particular device they are attempting to model.

Example SPICE Usage:

```
a4 [3 4 5 6] [3 4 5 6] [12 13 14 15 16 17 18 19] 30 [22 23 24] ram2
.model ram2 d_ram(select_value = 2 ic = 2 read_delay = 80e-9)
```

3.5.3.21 Digital Source

```
NAME_TABLE:
C_Function_Name:    cm_d_source
Spice_Model_Name:   d_source
Description:        "digital signal source"

PORT_TABLE:
Port Name:          out
Description:        "output"
Direction:          out
Default_Type:       d
Allowed_Types:      [d]
Vector:             yes
Vector_Bounds:      -
Null_Allowed:       no

PARAMETER_TABLE:
Parameter_Name:     input_file
Description:        "digital input vector filename"
Data_Type:          string
Default_Value:      "source.txt"
Limits:             -
Vector:             no
Vector_Bounds:      -
Null_Allowed:       no

PARAMETER_TABLE:

Parameter_Name:     input_load
Description:        "input loading capacitance (F)"
Data_Type:          real
Default_Value:      1.0e-12
Limits:             -
Vector:              no
Vector_Bounds:       -
Null_Allowed:       no
```

Description: The digital source provides for straightforward descriptions of digital signal vectors in a tabular format. The model reads input from the input file and, at the times specified in the file, generates the inputs along with the strengths listed.

The format of the input file is as shown below. Note that comment lines are delineated through the use of a single "*" character in the first column of a line. This is similar to the way the SPICE program handles comments.

```
* T        c    n    n    n    . . .
* i        l    o    o    o    . . .
* m        o    d    d    d    . . .
* e        c    e    e    e    . . .
*          k    a    b    c    . . .

0.0000     Uu   Uu   Us   Uu    . . .
1.234e-9   0s   1s   1s   0z    . . .
1.376e-9   0s   0s   1s   0z    . . .
2.5e-7     1s   0s   1s   0z    . . .
2.5006e-7  1s   1s   1s   0z    . . .
5.0e-7     0s   1s   1s   0z    . . .
```

Note that in the example shown, whitespace (any combination of blanks, tabs, commas) is used to separate the time and strength/state tokens. The order of the input columns is important; the first column is always interpreted to mean "time". The second through the N'th columns map to the out[0] through out[N-2] output nodes. A non-commented line which does not contain enough tokens to completely define all outputs for the digital_source will cause an error. Also, time values must increase monotonically or an error will result in reading the source file.

Errors will also occur if a line exists in source.txt which is neither a comment nor vector line. The only exception to this is in the case of a line that is completely blank; this is treated as a comment (note that such lines often occur at the end of text within a file; ignoring these in particular prevents nuisance errors on the part of the simulator).

```
Example SPICE Usage:

a3 [2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17] input_vector
.model input_vector d_source(input_file = source.simple.text)
```

3.6 Predefined Node Types

The following prewritten node types are included with the XSPICE simulator. These, along with the digital node type built into the simulator, should provide you not only with valuable event-driven modeling capabilities, but also with examples to use for guidance in creating new UDN types.

3.6.1 Real Node Type

The "real" node type provides for event-driven simulation with double-precision floating point data. This type is useful for evaluating sampled-data filters and systems. The type implements all optional functions for User-Defined Nodes, including inversion and node resolution. For inversion, the sign of the value is reversed. For node resolution, the resultant value at a node is the sum of all values output to that node.

3.6.2 Int Node Type

The "int" node type provides for event-driven simulation with integer data. This type is useful for evaluating roundoff error effects in sampled-data systems. The type implements all optional functions for User-Defined Nodes, including inversion and node resolution. For inversion, the sign of the integer value is reversed. For node resolution, the resultant value at a node is the sum of all values output to that node.

4 Error Messages

Error messages may be subdivided into three categories. These are:

1. Error messages generated during the development of a code model (Preprocessor Error Messages).
2. Error messages generated by the simulator during a simulation run (Simulator Error Messages).
3. Error messages generated by individual code models (Code Model Error Messages).

These messages will be explained in detail in the following subsections.

4.1 Preprocessor Error Messages

The following is a list of error messages that may be encountered when invoking the directory-creation and code modeling preprocessor tools. These are listed individually, and explanations follow the name/listing.

`Usage: cmpp [-ifs] [-mod [<filename>]] [-lst]`

The Code Model Preprocessor (cmpp) command was invoked incorrectly.

```
ERROR - Too few arguments
```

The Code Model Preprocessor (cmpp) command was invoked with too few arguments.

```
ERROR - Too many arguments
```

The Code Model Preprocessor (cmpp) command was invoked with too many arguments.

```
ERROR - Unrecognized argument
```

The Code Model Preprocessor (cmpp) command was invoked with an invalid argument.

```
ERROR - File not found: <filename>
```

The specified file was not found, or could not be opened for read access.

```
ERROR - Line <line number> of <filename> exceeds XX characters
```

The specified line was too long.

```
ERROR - Pathname on line <line number> of <filename> exceeds
        XX characters
```

The specified line was too long.

```
ERROR - No pathnames found in file: <filename>
```

The indicated modpath.lst file does not have pathnames properly listed.

```
ERROR - Problems reading ifspec.ifs in directory <pathname>
```

The Interface Specification File (ifspec.ifs) for the code model could not be read.

```
ERROR - Model name <model name> is same as internal SPICE model name
```

A model has been given the same name as an intrinsic SPICE device.

```
ERROR - Model name '<model name>' in directory: <pathname>
        is same as
        model name '<model name>' in directory: <pathname>
```

Two models in different directories have the same name.

```
ERROR - C function name '<function name>' in directory: <pathname>",
        is same as
        C function name '<function name>' in directory: <pathname>
```

Two C language functions in separate model directories have the same names; these would cause a collision when linking the final executable.

```
ERROR - Problems opening CMextrn.h for write
```

The temporary file CMextern.h used in building the XSPICE simulator executable could not be created or opened. Check permissions on directory.

```
ERROR - Problems opening CMinfo.h for write
```

The temporary file CMinfo.h used in building the XSPICE simulator executable could not be created or opened. Check permissions on directory.

```
ERROR - Problems opening objects.inc file for write
```

The temporary file objects.inc used in building the XSPICE simulator executable could not be created or opened. Check permissions on directory.

```
ERROR - Could not open input .mod file: <filename>
```

The Model Definition File that contains the definition of the Code Model's behavior (usually cfunc.mod) was not found or could not be read.

`ERROR - Could not open output .c: <filename>`

The indicated C language file that the preprocessor creates could not be created or opened. Check permissions on directory.

`Error parsing .mod file: <filename>`

Problems were encountered by the preprocessor in interpreting the indicated Model Definition File.

`ERROR - File not found: <filename>`

The indicated file was not found or could not be opened.

`Error parsing interface specification file`

Problems were encountered by the preprocessor in interpreting the indicated Interface Specification File.

`ERROR - Can't create file: <filename>`

The indicated file could not be created or opened. Check permissions on directory.

`ERROR - write_port_info() - Number of allowed types cannot be zero`

There must be at least one port type specified in the list of allowed types.

`illegal quoted character in string (expected ``\'' or ``\\'')`

A string was found with an illegal quoted character in it.

`unterminated string literal`

A string was found that was not terminated.

`Unterminated comment`

A comment was found that was not terminated.

`Port '<port name>' not found`

The indicated port name was not found in the Interface Specification File.

`Port type 'vnam' is only valid for 'in' ports`

The port type 'vnam' was used for a port with direction 'out' or 'inout'. This type is only allowed on 'in' ports.

`Port types 'g', 'gd', 'h', 'hd' are only valid for 'inout' ports`

Port type 'g', 'gd', 'h', or 'hd' was used for a port with direction 'out' or 'in'. These types are only allowed on 'inout' ports.

`Invalid parameter type - POINTER type valid only for STATIC_VARs`

The type POINTER was used in a section of the Interface Specification file other than the STATIC_VAR section.

`Port default type is not an allowed type`

A default type was specified that is not one of the allowed types for the port.

`Incompatible port types in 'allowed_types' clause`

Port types listed under 'Allowed_Types' in the Interface Specification File must all have the same underlying data type. It is illegal to mix analog and event-driven types in a list of allowed types.

`Invalid parameter type (saw <parameter type 1> - expected <parameter type 2>)`

A parameter value was not compatible with the specified type for the parameter.

```
Named range not allowed for limits
```

A name was found where numeric limits were expected.

```
Direction of port '<port number>' in <port name>()
 is not <IN or OUT> or INOUT
```

A problem exists with the direction of one of the elements of a port vector.

```
Port  '<port name>' is an array - subscript required
```

A port was referenced that is specified as an array (vector) in the Interface Specification File. A subscript is required (e.g. myport[i])

```
Parameter '<parameter name>' is an array - subscript required
```

A parameter was referenced that is specified as an array (vector) in the Interface Specification File. A subscript is required (e.g. myparam[i])

```
Port  '<port name>' is not an array - subscript prohibited
```

A port was referenced that is not specified as an array (vector) in the Interface Specification File. A subscript is not allowed.

```
Parameter '<parameter name>' is not an array - subscript prohibited
```

A parameter was referenced that is not specified as an array (vector) in the Interface Specification File. A subscript is not allowed.

```
Static variable '<static variable name>' is not an array - subscript prohibited
```

Array static variables are not supported. Use a POINTER type for the static variable.

```
Buffer overflow - try reducing the complexity of CM-macro array subscripts
```

The argument to a code model accessor macro was too long.

```
Unmatched )
```

An open (was found with no corresponding closing).

`Unmatched ]`

An open [was found with no corresponding closing].

4.2 Simulator Error Messages

The following is a list of error messages that may be encountered while attempting to run a simulation (with the exception of those error messages generated by individual code models). Most of these errors are generated by the simulator while attempting to parse a SPICE deck. These are listed individually, and explanations follow the name/listing.

`ERROR - Scalar port expected, [ found`

A scalar connection was expected for a particular port on the code model, but the symbol [which is used to begin a vector connection list was found.

`ERROR - Unexpected ]`

A] was found where not expected. Most likely caused by a missing [.

`ERROR - Unexpected [ - Arrays of arrays not allowed`

A [character was found within an array list already begun with another [character.

`ERROR - Tilde not allowed on analog nodes`

The tilde character ~was found on an analog connection. This symbol, which performs state inversion, is only allowed on digital nodes and on User-Defined Nodes only if the node type definition allows it.

`ERROR - Not enough ports`

An insufficient number of node connections was supplied on the instance line. Check the Interface Specification File for the model to determine the required connections and their types.

`ERROR - Expected node/instance identifier`

A special token (e.g. `[ ] < >` ...) was found when not expected.

`ERROR - Expected node identifier`

A special token (e.g. `[ ] < >` ...) was found when not expected.

`ERROR - unable to find definition of model <name>`

A .model line for the referenced model was not found.

`ERROR - model: %s - Array parameter expected - No array delimiter found`

An array (vector) parameter was expected on the .model card, but enclosing [ ] characters were not found to delimit its values.

`ERROR - model: %s - Unexpected end of model card`

The end of the indicated .model line was reached before all required information was supplied.

`ERROR - model: %s - Array parameter must have at least one value`

An array parameter was encountered that had no values.

`ERROR - model: %s - Bad boolean value`

A bad values was supplied for a Boolean. Value used must be TRUE, FALSE, T, or F.

`ERROR - model: %s - Bad integer, octal, or hex value`

A badly formed integer value was found.

`ERROR - model: %s - Bad real value`

A badly formed real value was found.

`ERROR - model: %s - Bad complex value`

A badly formed complex number was found. Complex numbers must be enclosed in < > delimiters.

4.3 Code Model Error Messages

The following is a list of error messages that may be encountered while attempting to run a simulation with certain code models. These are listed alphabetically based on the name of the code model, and explanations follow the name and listing.

4.3.1 Code Model aswitch

```
cntl_error:
    *****ERROR*****
    ASWITCH: CONTROL voltage delta less than 1.0e-12
```

This message occurs as a result of the cntl_off and cntl_on values being less than 1.0e-12 volts/amperes apart.

4.3.2 Code Model climit

```
climit_range_error:
    **** ERROR ****
    * CLIMIT function linear range less than zero. *
```

This message occurs whenever the difference between the upper and lower control input values are close enough that there is no effective room for proper limiting to occur; this indicates an error in the control input values.

4.3.3 Code Model core

```
allocation_error:
    ***ERROR***
    CORE: Allocation calloc failed!
```

This message is a generic message related to allocating sufficient storage for the H and B array values.

```
limit_error:
    ***ERROR***
    CORE: Violation of 50% rule in breakpoints!
```

This message occurs whenever the input_domain value is an absolute value and the H coordinate points are spaced too closely together (overlap of the smoothing regions will occur unless the H values are redefined).

4.3.4 Code Model d_osc

```
d_osc_allocation_error:
    **** Error ****
    D_OSC: Error allocating VCO block storage
```

Generic block storage allocation error.

```
d_osc_array_error:
    **** Error ****
    D_OSC: Size of control array different than frequency array
```

Error occurs when there is a different number of control array members than frequency array members.

```
d_osc_negative_freq_error:
    **** Error ****
    D_OSC: The extrapolated value for frequency
    has been found to be negative...
    Lower frequency level has been clamped to 0.0 Hz.
```

Occurs whenever a control voltage is input to a model which would ordinarily (given the specified control/freq coordinate points) cause that model to attempt to generate an output oscillating at zero frequency. In this case, the output will be clamped to some DC value until the control voltage returns to a more reasonable value.

4.3.5 Code Model d_source

```
loading_error:
    ***ERROR***
    D_SOURCE: source.txt file was not read successfully.
```

This message occurs whenever the d_source model has experienced any difficulty in loading the source.txt (or user-specified) file. This will occur with any of the following problems:

- Width of a vector line of the source file is incorrect.
- A Timepoint value is duplicated or is otherwise not monotonically increasing.
- One of the output values was not a valid 12-State value (0s, 1s, Us, 0r, 1r, Ur, 0z, 1z, Uz, 0u, 1u, Uu).

4.3.6 Code Model d_state

```
loading_error:
    ***ERROR***
    D_STATE: state.in file was not read successfully.
    The most common cause of this problem is a
    trailing blank line in the state.in file
```

This error occurs when the state.in file (or user-named state machine input file) has not been read successfully. This is due to one of the following:

- The counted number of tokens in one of the file's input lines does not equal that required to define either a state header or a continuation line (Note that all comment lines are ignored, so these will never cause the error to occur).
- An output state value was defined using a symbol which was invalid (i.e., it was not one of the following: 0s, 1s, Us, 0r, 1r, Ur, 0z, 1z, Uz, 0u, 1u, Uu).
- An input value was defined using a symbol which was invalid (i.e., it was not one of the following: 0, 1, X, or x).

```
index_error:
    ***ERROR***
    D_STATE: An error exists in the ordering of states values
    in the states->state[] array. This is usually caused
    by non-contiguous state definitions in the state.in file
```

This error is caused by the different state definitions in the input file being non-contiguous. In general, it will refer to the different states not being defined uniquely, or being "broken up" in some fashion within the state.in file.

4.3.7 Code Model oneshot

```
oneshot_allocation_error:
    **** Error ****
    ONESHOT: Error allocating oneshot block storage
```

Generic storage allocation error.

```
oneshot_array_error:
    **** Error ****
    ONESHOT: Size of control array different than pulse-width array
```

This error indicates that the control array and pulse-width arrays are of different sizes.

```
oneshot_pw_clamp:
    **** Warning ****
    ONESHOT: Extrapolated Pulse-Width Limited to zero
```

This error indicates that for the current control input, a pulse-width of less than zero is indicated. The model will consequently limit the pulse width to zero until the control input returns to a more reasonable value.

4.3.8 Code Model pwl

```
allocation_error:
    ***ERROR***
    PWL: Allocation calloc failed!
```

Generic storage allocation error.

```
limit_error:
    ***ERROR***
    PWL: Violation of 50% rule in breakpoints!
```

This error message indicates that the pwl model has an absolute value for its input_domain, and that the x_array coordinates are so close together that the required smoothing regions would overlap. To fix the problem, you can either spread the x_array coordinates out or make the input_domain value smaller.

4.3.9 Code Model s_xfer

```
num_size_error:
    ***ERROR***
    S_XFER: Numerator coefficient array size greater than
    denominator coefficient array size.
```

This error message indicates that the order of the numerator polynomial specified is greater than that of the denominator. For the s_xfer model, the orders of numerator and denominator polynomials must be equal, or the order of the denominator polynomial must be greater than that or the numerator.

4.3.10 Code Model sine

```
allocation_error:
    **** Error ****
    SINE: Error allocating sine block storage
```

Generic storage allocation error.

```
sine_freq_clamp:
    **** Warning ****
    SINE: Extrapolated frequency limited to 1e-16 Hz
```

This error occurs whenever the controlling input value is such that the output frequency ordinarily would be set to a negative value. Consequently, the output frequency has been clamped to a near-zero value.

```
array_error:
    **** Error ****
    SINE: Size of control array different than frequency array
```

This error message normally occurs whenever the controlling input array and the frequency array are different sizes.

4.3.11 Code Model square

```
square_allocation_error:
    **** Error ****
    SQUARE: Error allocating square block storage
```

Generic storage allocation error.

```
square_freq_clamp:
    **** WARNING  ****
    SQUARE: Frequency extrapolation limited to 1e-16
```

This error occurs whenever the controlling input value is such that the output frequency ordinarily would be set to a negative value. Consequently, the output frequency has been clamped to a near-zero value.

```
square_array_error:
    **** Error ****
    SQUARE: Size of control array different than frequency array
```

This error message normally occurs whenever the controlling input array and the frequency array are different sizes.

4.3.12 Code Model triangle

```
triangle_allocation_error:
    **** Error ****
    TRIANGLE: Error allocating triangle block storage
```

Generic storage allocation error.

```
triangle_freq_clamp:
    **** Warning ****
    TRIANGLE: Extrapolated Minimum Frequency Set to 1e-16 Hz
```

This error occurs whenever the controlling input value is such that the output frequency ordinarily would be set to a negative value. Consequently, the output frequency has been clamped to a near-zero value.

```
triangle_array_error:
    **** Error ****
    TRIANGLE: Size of control array different than frequency array
```

This error message normally occurs whenever the controlling input array and the frequency array are different sizes.

5 Notes

5.1 Glossary

card — A logical SPICE input line. A card may be extended through the use of the "+" sign in SPICE, thereby allowing it to take up multiple lines in a SPICE deck.

code model — A model of a device, function, component, etc. which is based solely on a C programming language-based function. In addition to the code models included with the XSPICE simulator, you can use code models that you develop for circuit modeling.

deck — A collection of SPICE cards which together specify all input information required in order to perform an analysis. A "deck" of "cards" will in fact be contained within a file on the host computer system.

element card — A single, logical line in an XSPICE circuit description deck which describes a circuit element. Circuit elements are connected to each other to form circuits (e.g., a logical card which describes a resistor, such as `R1 2 0 10K`, is an element card).

instance — A unique occurrence of a circuit element. See "element card", in which the instance "R1" is specified as a unique element (instance) in a hypothetical circuit description.

macro — A macro, in the context of this document, refers to a C-language macro which supports the construction of user-defined models by simplifying input/output and parameter-passing operations within the Model Definition File.

.mod — Refers to the Model Definition File. The file suffix reflects the filename of the model definition file: cfunc.mod.

.model — Refers to a model card associated with an element card in XSPICE. A model card allows for data defining an instance to be conveniently located in the XSPICE deck such that the general layout of the elements is more readable.

Nutmeg The SPICE3C1 default post-processor. This provides a simple stand-alone simulator interface which can be used with the ATESSE simulator (see referenced documents section for additional information on Nutmeg).

subcircuit A "device" within an XSPICE deck which is defined in terms of a group of element cards and which can be referenced in other parts of the XSPICE deck through element cards.

5.2 Acronyms and Abbreviations

ATE	Automatic Test Equipment
ATESSE	Automatic Test Equipment Software Support Environment
CAE	Computer-Aided Engineering
CCCS	Current Controlled Current Source. In some cases, this is abbreviated ICIS.
CCVS	Current Controlled Voltage Source. Also abbreviated as ICVS.
CSCI	Computer Software Configuration Item
FET	Field Effect Transistor
IDD	Interface Design Document
IFS	Refers to the Interface Specification File. The abbreviation reflects the filename of the Interface Specification File: ifspec.ifs.
MNA	Modified Nodal Analysis
MOSFET	Metal Oxide Semiconductor Field Effect Transistor
PWL	Piece-Wise Linear
RAM	Random Access Memory
ROM	Read Only Memory
SDD	Software Design Document
SI	Simulator Interface
SIM	The ATESSE Version 2.0 Simulator

SPICE Simulation Program with Integrated Circuit Emphasis. This program was developed at the University of California at Berkeley.

SPICE3 Version 3 of SPICE.

SRS Software Requirements Specification

SUM Software User's Manual

UCB University of California at Berkeley

UDN User-Defined Node(s)

VCCS Voltage Controlled Current Source. This is also sometimes abbreviated as VCIS.

VCIS Voltage Controlled Current Source.

VCVS Voltage Controlled Voltage Source

XSPICE Extended SPICE; synonymous with the ATESSE Version 2.0 Simulator.

APPENDICES

A XSPICE System Requirements

This appendix lists hardware and software requirements for installing and running XSPICE at your site. It does not cover the actual installation process. Refer to the Release Notes provided with your XSPICE distribution tape for installation instructions. Also, please refer to the Release Notes for any addendums to the requirements specified here since later releases may differ in some respects.

XSPICE was developed on HP/Apollo DN series and 400 series workstations under BSD 4.3 UNIX and X11R3. The software has also been successfully compiled on Sun workstations that satisfied the following requirements:

- 32 bit processor.
- Approximately 50 Mbytes of free disk space.
- 4 Mbytes of random access memory.
- BSD 4.3 compatible UNIX operating system with Bourne shell and "sed" stream editor installed.
- X Window System release 11, revision 3 or 4 with Athena Widgets.
- ANSI compatible C compiler.

A 32 bit processor is required to support 4 byte integers and pointer arithmetic. To date, the code has been compiled only for Motorola 68000 based machines but should work with other processors as well.

The distribution software is approximately 15 Mbytes in size before compilation. Allowing for 50 Mbytes of free space should provide sufficient room for compiled object files and

linked executables. More disk space will be required if you intend to run large simulations that generate large volumes of results data.

On HP/Apollo workstations running Domain OS 10.3, the linked XSPICE executable is approximately 2 Mbytes in size with all predefined code models and node types included and with no debugging information included ("-g" compile option not used). Allowing for 4 Mbytes of random access memory should provide sufficient room for the executable to run and to allocate dynamic memory for results data without excessive paging. More memory may be required on some machines depending on the space required by the machine's operating system.

Some of the header files used in the simulator code are BSD UNIX specific. Therefore, you will need to compile the software in a BSD 4.3 compatible environment. Bourne shell compatible scripts and the UNIX "sed" editor are used in parts of the Code Model Toolkit.

The Nutmeg user interface requires X11 X-Windows support for certain files/commands including the help system and plotting of results data. The code has been successfully compiled under X11R3 and X11R4. The optional Athena Widget Set must be installed on your machine. This may not be provided by default on some systems (e.g. Sun workstations).

Much of the code in XSPICE uses the type-checking features and generic (void) pointers of ANSI C. Therefore, you must use an ANSI compatible C compiler to compile the code. Certain workstations (e.g. Sun machines) do not provide an ANSI C compiler by default. If you do not have an ANSI C compatible compiler, you may wish to acquire a copy of the GNU C compiler ("gcc"). XSPICE has been successfully compiled with gcc version 1.59 and above. Earlier versions of gcc may not work due to problems with copying of structures in assignment statements.

B Code Model Data Type Definitions

There are three data types which you can incorporate into a model and which have already been used extensively in the code model library included with the simulator. These are detailed below:

Boolean_t

The Boolean type is an enumerated type which can take on values of FALSE (integer value 0) or TRUE (integer value 1). Alternative names for these enumerations are MIF_FALSE and MIF_TRUE, respectively.

Complex_t

The Complex type is a structure composed of two double values. The first of these is the .real type, and the second is the .imag type. Typically these values are accessed as shown:

For complex value "data", the real portion is "data.real", and the imaginary portion is "data.imag".

Digital_State_t

The Digital State type is an enumerated value which can be either ZERO (integer value 0), ONE (integer value 1), or UNKNOWN (integer value 2).

Digital_Strength_t

The Digital Strength type is an enumerated value which can be either STRONG (integer value 0), RESISTIVE (integer value 1), HI_IMPEDANCE (integer value 2) or UNDETERMINED (integer value 3).

Digital_t

The Digital type is a composite of the Digital_State_t and Digital_Strength_t enumerated datatypes. The actual variable names within the Digital type are .state and .strength and are accessed as shown below:

For complex value "data", the state portion is "data.state", and the strength portion is "data.strength".

C XSPICE/Nutmeg Simulation Examples

This section is designed to walk you through the basic features of XSPICE using the Nutmeg user interface. The operation of XSPICE and Nutmeg will be illustrated through three examples.

The first example uses the simple one-transistor amplifier circuit illustrated in Figure C.1. This circuit is constructed entirely with SPICE3 compatible devices and is used to introduce basic concepts, including:

- Invoking the simulator
- Running simulations in different analysis modes
- Printing and plotting analog results
- Examining status, including executation time and memory usage
- Exiting the simulator

The second example circuit, shown in Figure C.2, models the circuit of Figure C.1 using the XSPICE gain block code model to substitute for the more complex and computationally expensive SPICE3 transistor model. This example illustrates one way in which XSPICE code models can be used to raise the level of abstraction in circuit modeling to improve simulation speed.

The third and final example, shown in Figure C.3, illustrates many of the more advanced features offered by XSPICE. This circuit is a mixed-mode design incorporating digital data, analog data, and User-Defined Node data together in the same simulation. Some of the important features illustrated include:

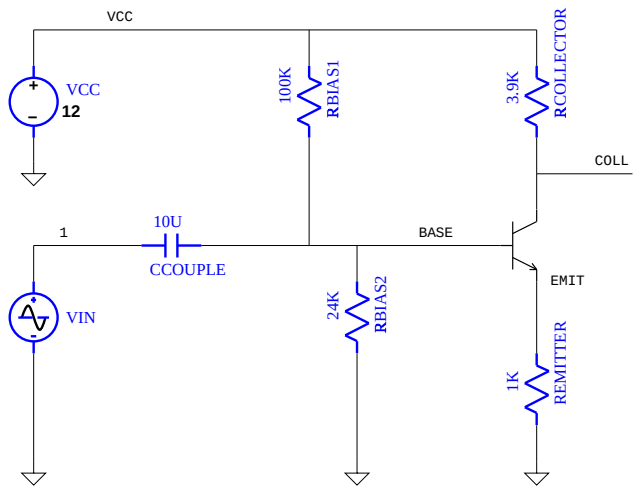

Figure C.1 Transistor Amplifier Simulation Example

- Creating and compiling Code Models
- Creating an XSPICE executable that incorporates these new models
- The use of "node bridge" models to translate data between the data types in the simulation
- Plotting analog and event-driven (digital and User-Defined Node) data
- Using the "eprint" command to print event-driven data

Throughout these examples, we assume that XSPICE has already been installed on your system and that your user account has been set up with the proper search path and environment variable data. If you experience problems, please see your system administrator for help.

The examples also assume that you are running under UNIX and will use standard UNIX commands such as "cp" for copying files, etc. If you are using a different set up, with different operating system command names, you should be able to translate the commands shown into those suitable for your installation.

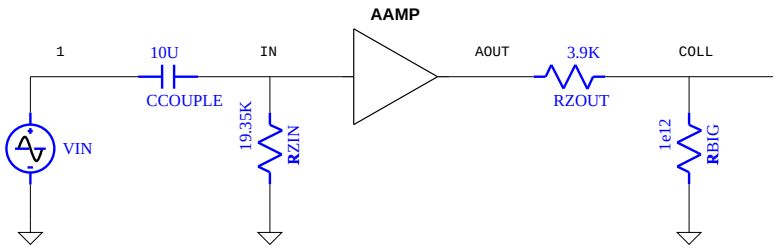

Figure C.2 Code Model Simulation Example

Finally, file system pathnames given in the examples assume that XSPICE has been installed on your system in directory "/usr/local/xspice-1-0". If your installation is different, you should substitute the appropriate root pathname where appropriate.

C.1 Simulation Example 1

The circuit shown in Figure C.1 is a simple one-transistor amplifier. The input signal is amplified with a gain of approximately `-(Rc/Re) = -(3.9K/1K) = -3.9`. The circuit description file for this example is shown below.

```
A Berkeley SPICE3 compatible circuit
*
* This circuit contains only Berkeley SPICE3 components.
*
* The circuit is an AC coupled transistor amplifier with
* a sinewave input at node "1", a gain of approximately -3.9,
* and output on node "coll".
*
.tran 1e-5 2e-3
*
vcc  vcc 0  12.0
vin  1 0  0.0 ac 1.0 sin(0 1 1k)
*
ccouple  1 base  10uF
*
rbias1  vcc base  100k
```

```
rbias2  base 0  24k
*
q1  coll base emit  generic
.model generic npn
*
rcollector  vcc coll  3.9k
remitter  emit 0  1k
*
.end
```

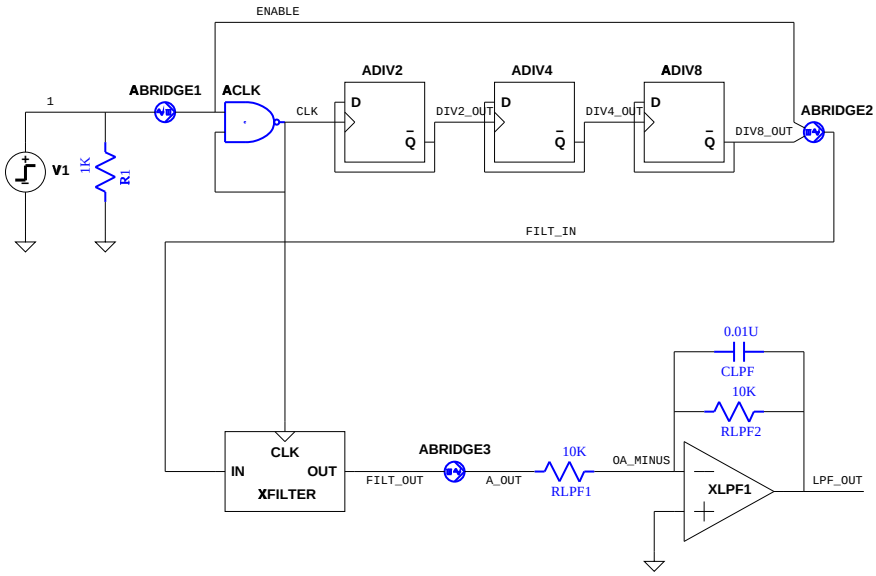

Figure C.3 Mixed-Mode Simulation Example

To simulate this circuit, move into a directory under your user account and copy the file `spice3.deck` from directory `/usr/local/xspice-1-0/lib/sim/examples`.

```
$ cp /usr/local/xspice-1-0/lib/sim/examples/spice3.deck spice3.deck
```

Now invoke the simulator on this circuit as follows:

```
$ xspice spice3.deck
```

After a few moments, you should see the XSPICE prompt:

```
XSPICE 1 ->
```

At this point, the XSPICE simulator has read-in the circuit description and checked it for errors. If any errors had been encountered, messages describing them would have been output to your terminal. Since no messages were printed for this circuit, the syntax of the circuit description was correct.

To see the circuit description read by the simulator you can issue the following command:

```
XSPICE 1 -> listing
```

The simulator shows you the circuit description currently in memory:

```
A Berkeley SPICE3 compatible circuit

 1 : A BERKELEY SPICE3 COMPATIBLE CIRCUIT
 9 : .TRAN 1E-5 2E-3
11 : VCC  VCC 0  12.0
12 : VIN  1 0  0.0 AC 1.0 SIN(0 1 1K)
14 : CCOUPLE  1 BASE  10UF
16 : RBIAS1  VCC BASE  100K
17 : RBIAS2  BASE 0  24K
19 : Q1  COLL BASE EMIT  GENERIC
20 : .MODEL GENERIC NPN
22 : RCOLLECTOR  VCC COLL  3.9K
23 : REMITTER  EMIT 0  1K
26 : .end
```

The title of this circuit is "A Berkeley SPICE3 compatible circuit". The circuit description contains a transient analysis control command `.TRAN 1E-5 2E-3` requesting a total simulated time of 2ms with a maximum timestep of 10us. The remainder of the lines in the circuit description describe the circuit of Figure C.1.

Before running this simulation, let's issue the "rusage" command to check the CPU time and memory used so far:

```
XSPICE 2 -> rusage

Total run time: 1.300 seconds.

Current data size = 237504,
Data limits: hard = 2147483647, soft = 2147483647.

Time since last call: 0.000 seconds.
```

From this output we notice that the simulator used 1.3 seconds while reading in and parsing the circuit description and has used 237504 bytes of dynamically allocated memory so far (numbers may be somewhat different on your system).

Now, execute the simulation by entering the "run" command:

```
XSPICE 3 -> run
```

The simulator will run the simulation and when execution is completed, will return with the XSPICE prompt. When the prompt returns, issue the rusage command again to see how much time and memory has been used now.

```
XSPICE 4 -> rusage

Total run time: 6.467 seconds.

Current data size = 270272,
Data limits: hard = 2147483647, soft = 2147483647.

Time since last call: 0.033 seconds.
```

From this information, we can compute that the total run time for this analysis was approximately `(6.5 - 1.3) = 4.2` seconds and that (270272 - 237504) = 32768 additional bytes of dynamically allocated memory have been used.

To examine the results of this transient analysis, we can use the "plot" command. First we will plot the nodes labeled "1" and "base".

```
XSPICE 5 -> plot v(1) base
```

The simulator responds by displaying an X Window System plot similar to that shown in Figure C.4.

Notice that we have named one of the nodes in the *circuit description* with a number ("1"), while the others are words ("base"). This was done to illustrate SPICE3's special requirements for plotting nodes labeled with numbers. Numeric labels are allowed in SPICE3 for backwards compatibility with SPICE2. However, they require special treatment in some commands such as "plot". The "plot" command is designed to allow expressions in its argument list in addition to names of results data to be plotted. For example, the expression `plot (base - 1)` would plot the result of subtracting 1 from the value of node "base".

If we had desired to plot the difference between the voltage at node "base" and node "1", we would need to enclose the node name "1" in the construction `v( )` producing a command such as `plot (base - v(1))`.

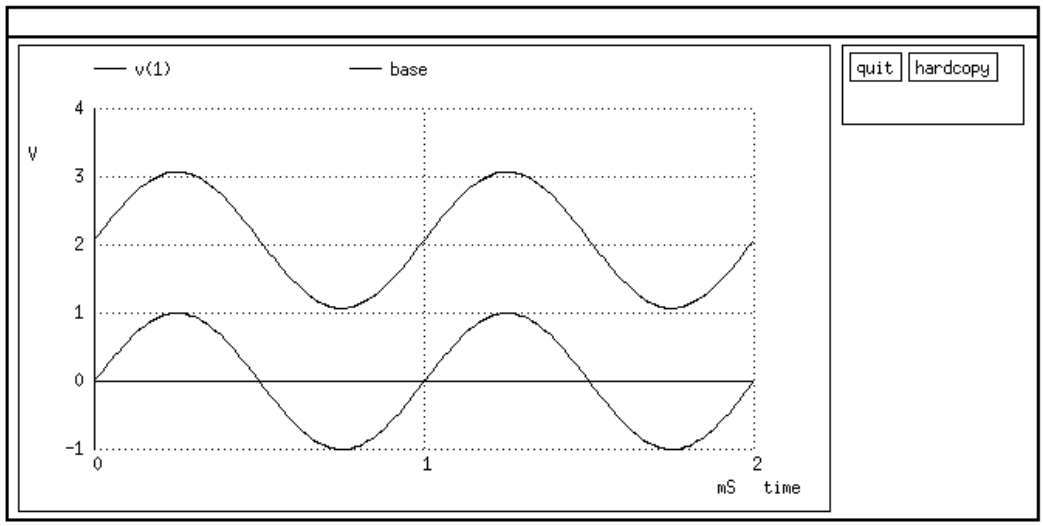

Figure C.4 Nutmeg Plot of Input and Base Voltages

Now, issue the following command to examine the voltages on two of the internal nodes of the transistor amplifier circuit:

```
XSPICE 6 -> plot vcc coll
```

The plot shown in Figure C.5 should appear.

Notice in the circuit description that the power supply voltage source and the node it is connected to both have the name "vcc". The plot command above has plotted the *node voltage* "vcc". However, it is also possible to plot *branch currents* through voltage sources in a circuit. SPICE3 always adds the special suffix "#branch" to voltage source names. Hence, to plot the current into the voltage source named "vcc", we would use a command such as `plot vcc#branch`.

Now let's run a simple DC simulation of this circuit and examine the bias voltages with the "print" command. One way to do this is to quit the simulator using the "quit" command, edit the input file to change the ".tran" line to ".op" (for 'operating point analysis'), re-invoke the simulator, and then issue the "run" command. However, Nutmeg allows analysis mode changes directly from the XSPICE prompt. All that is required is to enter the control

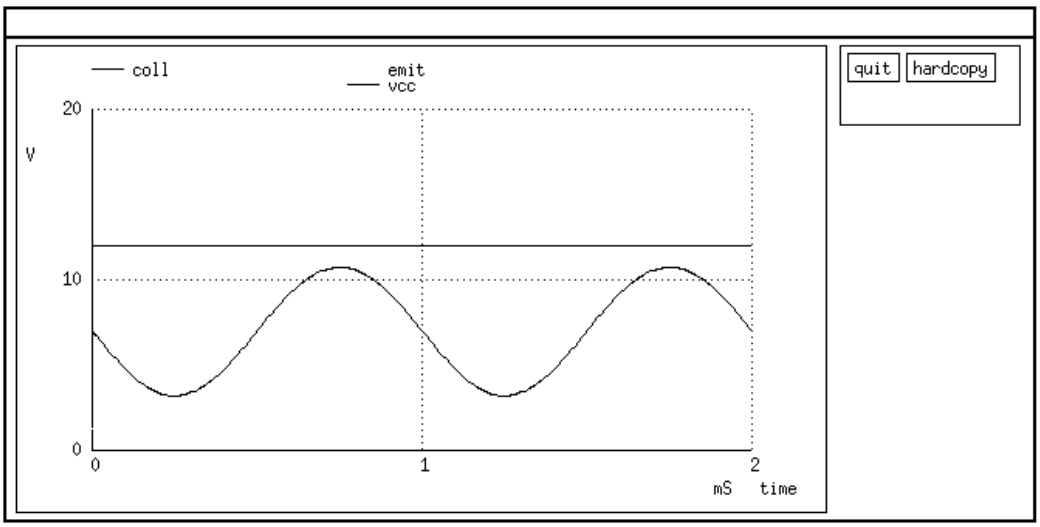

Figure C.5 Nutmeg Plot of VCC, Collector, and Emitter Voltages

line (without the leading "."). XSPICE will interpret the information on the line and start the new analysis run immediately, without the need to enter a new "run" command.

To run the DC simulation of the transistor amplifier, issue the following command:

```
XSPICE 7 -> op
```

After a moment the XSPICE prompt returns. Now issue the "print" command to examine the emitter, base, and collector DC bias voltages.

```
XSPICE 8 -> print emit base coll
```

XSPICE responds with:

```
emit = 1.293993e+00
base = 2.074610e+00
coll = 7.003393e+00
```

To run an AC analysis, enter the following command:

```
XSPICE 9 -> ac dec 10 0.01 100
```

This command runs a small-signal swept AC analysis of the circuit to compute the magnitude and phase responses. In this example, the sweep is logarithmic with "decade" scaling, 10 points per decade, and lower and upper frequencies of 0.01 Hz and 100 Hz. Since the command sweeps through a range of frequencies, the results are vectors of values and are examined with the plot command. Issue to the following command to plot the response curve at node "coll":

```
XSPICE 10 -> plot coll
```

This plot shows the AC gain from input to the collector. (Note that our input source in the circuit description "vin" contained parameters of the form "AC 1.0" designating that a unit-amplitude AC signal was applied at this point.)

To produce a more traditional "Bode" gain phase plot with logarithmic scaling on the frequency axis, we use the expression capability of the "plot" command and the built-in Nutmeg functions `db( )`, `log( )`, and `ph( )` together with the `vs` keyword:

```
XSPICE 11 -> plot db(coll) ph(coll) vs log(frequency)
```

The last analysis supported by XSPICE is a swept DC analysis. To perform this analysis, issue the following command:

```
XSPICE 12 -> dc vcc 0 15 0.1
```

This command sweeps the supply voltage "vcc" from 0 to 15 volts in 0.1 volt increments. To plot the results, issue the command:

```
XSPICE 13 -> plot emit base coll
```

Finally, to exit the simulator, use the "quit" command, and you will be returned to the operating system prompt.

```
XSPICE 14 -> quit

So long.
$
```

C.2 Simulation Example 2

The circuit shown in Figure C.2 is an abstract model of the circuit shown in Figure C.1, constructed using the XSPICE code model "gain". The XSPICE circuit description for this circuit is shown below.

```
A simple XSPICE amplifier circuit
*
* This uses an XSPICE "gain" code model to substitute for
* the transistor amplifier circuit in spice3.deck.
*
.tran 1e-5 2e-3
*
vin  1 0  0.0 ac 1.0 sin(0 1 1k)
*
ccouple  1 in  10uF
*
*
rzin  in 0  19.35k
*
aamp  in aout  gain_block
.model gain_block gain (gain = -3.9  out_offset = 7.003)
*
rzout aout coll 3.9k
rbig  coll 0   1e12
*
*
.end
```

Notice the component "aamp". This is an XSPICE code model device. All XSPICE code model devices begin with the letter "a" to distinguish them from other SPICE3 devices. The actual code model used is referenced through a user-defined identifier at the end of the line - in this case"gain_block". The type of code model used and its parameters appear on the associated .model card. In this example, the gain has been specified as -3.9 to approximate the gain of the transistor amplifier, and the output offset (out_offset) has been set to 7.003 according to the DC bias point information obtained from the DC analysis in Example 1.

Notice also that input and output impedances of the one-transistor amplifier circuit are modeled with the resistors "rzin" and "rzout", since the "gain" code model defaults to an ideal voltage-input, voltage-output device with infinite input impedance and zero output impedance.

Lastly, note that a special resistor "rbig" with value "1e12" has been included at the opposite side of the output impedance resistor "rzout". This resistor is required by SPICE3's matrix solution formula. Without it, the resistor "rzout" would have only one connection to the

circuit, and an ill-formed matrix could result. One way to avoid such problems without adding resistors explicitly is to use the XSPICE "rshunt" option described in this document under XSPICE Syntax Extensions/General Enhancements.

To simulate this circuit, copy the file `xspice.deck` from the directory `/usr/local/xspice-1-0/lib/sim/examples` into a directory in your account.

```
$ cp /usr/local/xspice-1-0/lib/sim/examples/xspice.deck xspice.deck
```

Invoke the simulator on this circuit:

```
$ xspice xspice.deck
```

After a few moments, you should see the XSPICE prompt:

```
XSPICE 1 ->
```

Now issue the "run" command and when the prompt returns, issue the "plot" command to examine the voltage at the node "coll".

```
XSPICE 1 -> run
XSPICE 2 -> plot coll
```

The resulting waveform closely matches that from the original transistor amplifier circuit simulated in Example 1.

When you are done, enter the "quit" command to leave the simulator and return to the command line.

```
XSPICE 3 -> quit

So long.
$
```

Using the "rusage" command, you can verify that this abstract model of the transistor amplfier runs somewhat faster than the full circuit of Example 1. This is because the code model is less complex computationally. This demonstrates one important use of XSPICE code models - to reduce run time by modeling circuits at a higher level of abstraction. Speed improvements vary and are most pronounced when a large amount of low-level circuitry can be replaced by a small number of code models and additional components.

An equally important use of code models is in creating models for circuits and systems that do not easily lend themselves to synthesis using standard SPICE3 primitives (resistors, capacitors, diodes, transistors, etc.). This occurs often when trying to create models of ICs for use in simulating board-level designs. Creating models of operational amplfiers such as an LM741 or timer ICs such as an LM555 is greatly simplified through the use of XSPICE code models. Another example of code model use is shown in the next example where a complete sampled-data system is simulated using XSPICE analog, digital, and User-Defined Node types simultaneously.

C.3 Simulation Example 3

The circuit shown in Figure C.3 is designed to demonstrate several of the more advanced features of XSPICE. In this example, you will be introduced to the process of creating code models and linking them into a new version of the XSPICE simulator. You will also learn how to print and plot the results of event-driven analysis data. The XSPICE circuit description for this example is shown below.

```
Mixed IO types
*
* This circuit contains a mixture of IO types, including
* analog, digital, user-defined (real), and 'null'.
*
* The circuit demonstrates the use of the digital and
* user-defined node capability to model system-level designs
* such as sampled-data filters.  The simulated circuit
* contains a digital oscillator enabled after 100us.  The
* square wave oscillator output is divided by 8 with a
* ripple counter.  The result is passed through a digital
* filter to convert it to a sine wave.
*
.tran 1e-5 1e-3
*
v1 1 0 0.0 pulse(0 1 1e-4 1e-6)
r1 1 0 1k
*
abridge1 [1] [enable] atod
.model atod adc_bridge
*
aclk [enable clk] clk nand
.model nand d_nand (rise_delay=1e-5 fall_delay=1e-5)
*
adiv2 div2_out clk NULL NULL NULL div2_out dff
adiv4 div4_out div2_out NULL NULL NULL div4_out dff
```

```
adiv8 div8_out div4_out NULL NULL NULL div8_out dff
.model dff d_dff
*
abridge2 div8_out enable filt_in node_bridge2
.model node_bridge2 d_to_real (zero=-1 one=1)
*
xfilter  filt_in clk filt_out  dig_filter
*
abridge3 filt_out a_out node_bridge3
.model node_bridge3 real_to_v
*
rlpf1 a_out oa_minus 10k
*
xlpf  0 oa_minus lpf_out  opamp
*
rlpf2 oa_minus lpf_out 10k
clpf  lpf_out oa_minus 0.01uF
*
*
.subckt  dig_filter  filt_in clk filt_out
*
.model n0 real_gain (gain=1.0)
.model n1 real_gain (gain=2.0)
.model n2 real_gain (gain=1.0)
.model g1 real_gain (gain=0.125)
.model zm1 real_delay
.model d0a real_gain (gain=-0.75)
.model d1a real_gain (gain=0.5625)
.model d0b real_gain (gain=-0.3438)
.model d1b real_gain (gain=1.0)
*
an0a filt_in x0a n0
an1a filt_in x1a n1
an2a filt_in x2a n2
*
az0a x0a clk x1a zm1
az1a x1a clk x2a zm1
*
ad0a x2a x0a d0a
ad1a x2a x1a d1a
*
az2a x2a filt1_out g1
az3a filt1_out clk filt2_in zm1
*
an0b filt2_in x0b n0
an1b filt2_in x1b n1
an2b filt2_in x2b n2
*
```

```
az0b x0b clk x1b zm1
az1b x1b clk x2b zm1
*
ad0 x2b x0b d0b
ad1 x2b x1b d1b
*
az2b x2b clk filt_out zm1
*
.ends dig_filter
*
*
.subckt  opamp  plus minus out
*
r1 plus minus 300k
a1 %vd (plus minus) outint lim
.model lim limit (out_lower_limit = -12 out_upper_limit = 12
+             fraction = true  limit_range = 0.2  gain=300e3)
r3 outint out 50.0
r2 out 0 1e12
*
.ends opamp
*
*
.end
```

This circuit is a high-level design of a sampled-data filter. An analog step waveform (created from a SPICE3 "pulse" waveform) is introduced as "v1" and converted to digital by code model instance "abridge". This digital data is used to enable a Nand-Gate oscillator ("aclk") after a short delay. The Nand-Gate oscillator generates a squarewave clock signal with a period of approximately two times the gate delay, which is specified as 1e-5 seconds. This 50 KHz clock is divided by a series of D Flip Flops ("adiv2", "adiv4", "adiv8") to produce a squarewave at approximately 6.25 KHz. Note particularly the use of the reserved word "NULL" for certain nodes on the D Flip Flops. This tells the code model that there is no node connected to these ports of the flip flop.

The divide-by-8 and enable waveforms are converted by the instance "abridge2" to the format required by the User-Defined Node type "real", which expected real-valued data. The output of this instance on node "filt_in" is a real valued square wave which oscillates between values of -1 and 1. Note that the associated code model "d_to_real" is not part of the standard XSPICE code model library but will be built later in this example.

This signal is then passed through subcircuit "xfilter" which contains a digital lowpass filter clocked by node "clk". The result of passing this squarewave through the digital lowpass filter is the production of a sampled sine wave (the filter passes only the fundamental of the squarewave input) on node "filt_out". This signal is then converted back to SPICE analog data on node "a_out" by node bridge instance "abridge3".

The resulting analog waveform is then passed through an opamp-based lowpass analog filter constructed around subcircuit "xlpf" to produce the final output at analog node "lpf_out".

Before this circuit can be simulated, we need to construct four code models used in it:

- d_to_real
- real_to_v
- real_gain
- real_delay

To construct these models, we will use the XSPICE Code Model Toolkit. However, to avoid typing in all of the model code, we will be copying files from the "examples" directory once the model directories have been created.

First, create the code model "d_to_real". To do so, move into a directory under your user account and invoke the Code Model Toolkit's "mkmoddir" command:

```
$ mkmoddir d_to_real

SPICE model name [d_to_real]:
C function name [ucm_d_to_real]:

Model Directory "d_to_real" created.

Edit files "ifspec.ifs" and "cfunc.mod"
to define your model. Then run "make" to
preprocess and compile it.
```

and press RETURN to accept the defaults when prompted for data. This creates a model directory named "d_to_real" and installs three files in it - a "Makefile", an Interface Specification File, and a Model Definition File.

Now move into this new directory:

```
$ cd d_to_real
```

and examine the Interface Specification and Model Definition files (ifspec.ifs and cfunc.mod). As explained in Chapter 3 of this document, these files are used to specify the model's inputs, output, and parameters, and to code the models behavior in the C programming language with help from the Code Model Toolkit's "accessor macros" and function library.

To save time in this example, we will copy these files from the directory `/usr/local/xspice-1-0/lib/sim/examples/d_to_real`.

```
$ cp /usr/local/xspice-1-0/lib/sim/examples/d_to_real/ifspec.ifs ifspec.ifs
$ cp /usr/local/xspice-1-0/lib/sim/examples/d_to_real/cfunc.mod  cfunc.mod
```

You may wish to examine these files once you have copied them. When you are done, issue the UNIX "make" command to send them through the XSPICE Code Model Preprocessor utility ("cmpp") and then through the C compiler to create the necessary object files to be bound in with the simulator.

```
$ make

/usr/local/xspice-1-0/bin/cmpp -ifs
cc -g -I. -I/usr/local/xspice-1-0/include/sim -c ifspec.c
/usr/local/xspice-1-0/bin/cmpp -mod cfunc.mod
cc -g -I. -I/usr/local/xspice-1-0/include/sim -c cfunc.c
```

Now move back up to the parent directory:

```
$ cd ..
```

and repeat this process for the three remaining code models required for the simulation (real_to_v, real_gain, and real_delay). Note that the process of compiling code models is automated by the XSPICE Code Model Toolkit and the associated UNIX "make" command. Once the interface specifications and model definitions for the models are developed, the process of compiling the models is reduced to three easy-to-remember steps:

- Create the model with the "mkmoddir" command.
- Edit the template files "ifspec.ifs" and "cfunc.mod" to specify the model's interface and it's behavior.
- Build the model by typing "make".

The body of this document tells you how to go about designing and coding a model's interface and behavior. In addition, numerous examples can be found in the form of the models in the XSPICE code model library.

Now that our models are ready, the remaining step is to link them with the XSPICE "core" to create a new XSPICE executable. For this, we use the Code Model Toolkit's "mksimdir" command. Move to the directory in which all the models were created if you are not already there. Then enter the command:

```
$ mksimdir mysim

Simulator directory "mysim" created.

Edit files "modpath.lst" and "udnpath.lst" to
specify desired models and node types respectively.
Then run "make" to build the simulator executable.
```

When the operating system prompt returns, move into this new directory, and edit the file "modpath.lst". This file holds the pathnames to model directories containing models to be included in the simulator. The file initially includes all the models in the XSPICE Code Model Library. You may add and/or delete files from the list according to your anticipated needs. For this example, we will simply add the four models we have just compiled at the bottom of the file, with one pathname per line.

Add the following lines at the bottom of the file:

```
../d_to_real
../real_to_v
../real_gain
../real_delay
```

Save this edited file, return to the operating system prompt, and enter the UNIX "make" command (Note: Making the simulator may take a couple of minutes depending on the number of models included).

```
$ make

Running preprocessor on modpath.lst and udnpath.lst ...
/usr/local/xspice-1-0/bin/cmpp -lst

Compiling list of models and node types...
cc  -o temp.o -g -I. -I/usr/local/xspice-1-0/include/sim ...

Linking XSPICE simulator ...
cc -o xspice temp.o \
/usr/local/xspice-1-0/lib/sim/object/core.o -L/usr/X11/lib ...
/usr/local/xspice-1-0/src/cml/aswitch/*.o \
/usr/local/xspice-1-0/src/cml/climit/*.o \
/usr/local/xspice-1-0/src/cml/d_dt/*.o \
/usr/local/xspice-1-0/src/cml/divide/*.o \
/usr/local/xspice-1-0/src/cml/gain/*.o \
/usr/local/xspice-1-0/src/cml/hyst/*.o \
/usr/local/xspice-1-0/src/cml/ilimit/*.o \
```

```
...

/usr/local/xspice-1-0/src/cml/d_dff/*.o \
/usr/local/xspice-1-0/src/cml/d_jkff/*.o \
/usr/local/xspice-1-0/src/cml/d_tff/*.o \
/usr/local/xspice-1-0/src/cml/d_srff/*.o \
/usr/local/xspice-1-0/src/cml/d_dlatch/*.o \
/usr/local/xspice-1-0/src/cml/d_srlatch/*.o \
/usr/local/xspice-1-0/src/cml/d_state/*.o \
/usr/local/xspice-1-0/src/cml/d_osc/*.o \
../d_to_real/*.o \
../real_to_v/*.o \
../real_gain/*.o \
../real_delay/*.o \
/usr/local/xspice-1-0/src/udnl/real/*.o \
/usr/local/xspice-1-0/src/udnl/int/*.o

Deleting temporary files ...


XSPICE simulator created.

Type: "xspice <input deck>" to run.
```

Now copy the file "mixed_mode.deck" from directory `/usr/local/xspice-1-0/lib/sim/examples` into the "mysim" directory:

```
$ cp /usr/local/xspice-1-0/lib/sim/examples/mixed_mode.deck mixed_mode.deck
```

and invoke the new simulator executable as you did in the previous examples.

```
$ xspice mixed_mode.deck
```

Execute the simulation with the "run" command.

```
XSPICE 1 -> run
```

After several seconds, the XSPICE prompt should return.

Results of this simulation are examined in the manner illustrated in the previous two examples. You can use the "plot" command to plot either analog nodes, event-driven nodes, or both. For example, you can plot the values of the sampled-data filter input node and the analog lowpass filter output node as follows:

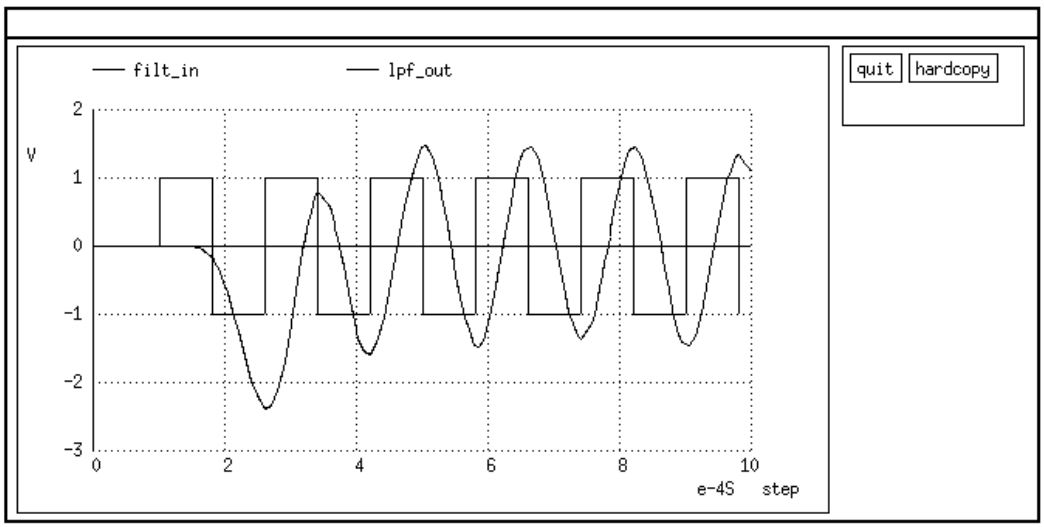

Figure C.6 Nutmeg Plot of Filter Input and Output

```
XSPICE 2 -> plot filt_in lpf_out
```

The plot shown in Figure C.6 should appear.

You can also plot data from nodes inside a subcircuit. For example, to plot the data on node "x1a" in subcircuit "xfilter", create a pathname to this node in reverse order with a colon separator.

```
XSPICE 3 -> plot x1a:xfilter
```

The output from this command is shown in Figure C.7. Note that the waveform contains vertical segments. These segments are caused by the non-zero delays in the "real_gain" models used within the subcircuit. Each vertical segment is actually a step with a width equal to the model delay (1e-9 seconds).

Plotting nodes internal to subcircuits works for both analog and event-driven nodes. The reverse order format arises because instance names are expanded similarly when processing subcircuits and SPICE uses the first character of a name to determine the device type.

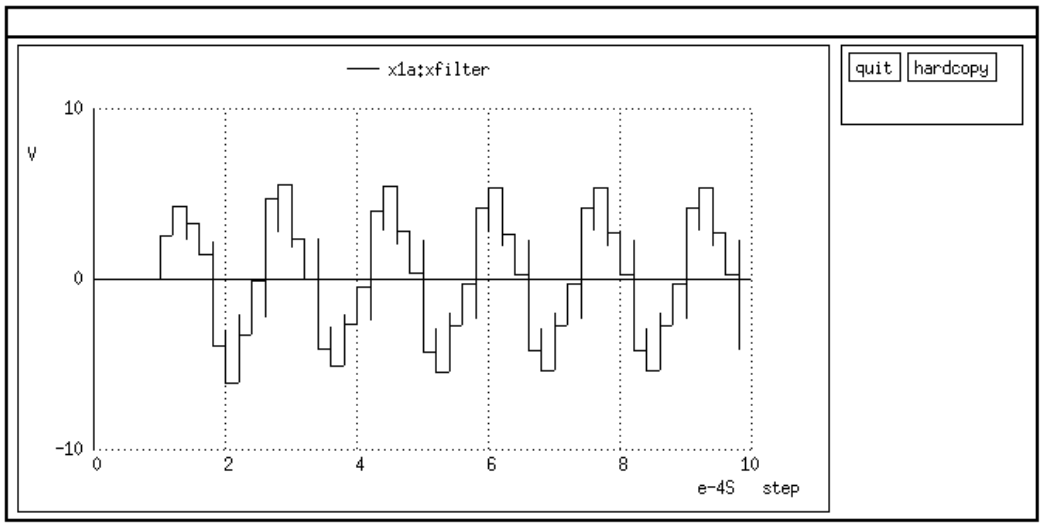

Figure C.7 Nutmeg Plot of Subcircuit Internal Node

Hence, by building the name in reverse order, the first character of the instance in the subcircuit is unchanged in the expanded name.

To examine data such as the closely spaced events inside the subcircuit at node "x1a:xfilter", it is often convenient to use the "eprint" command to produce a tabular listing of events. Try this by entering the following command:

```
XSPICE 4 -> eprint x1a:xfilter


**** Results Data ****

Time or Step
x1a:xfilter


0.000000000e+00        0.000000e+00
1.010030000e-04        2.000000e+00
1.010040000e-04        2.562500e+00
1.210020000e-04        2.812500e+00
1.210030000e-04        4.253906e+00
```

```
1.410020000e-04        2.332031e+00
1.410030000e-04        3.283447e+00
1.610020000e-04        2.014893e+00
1.610030000e-04        1.469009e+00
1.810020000e-04        2.196854e+00
1.810030000e-04        1.176232e+00
1.810090000e-04       -2.823768e+00
1.810100000e-04       -3.948768e+00
2.010020000e-04       -3.087939e+00
2.010030000e-04       -6.135439e+00
2.210020000e-04       -2.072106e+00
2.210030000e-04       -3.302109e+00

...

9.010090000e-04        3.049473e+00
9.010100000e-04        4.174473e+00
9.210020000e-04        2.867375e+00
9.210030000e-04        5.380142e+00
9.410020000e-04        2.029786e+00
9.410030000e-04        2.707975e+00
9.610020000e-04        1.803723e+00
9.610030000e-04        3.006294e-01
9.810020000e-04        2.304755e+00
9.810030000e-04        9.506230e-01
9.810090000e-04       -3.049377e+00
9.810100000e-04       -4.174377e+00



**** Messages ****


**** Statistics ****

Operating point analog/event alternations:  1
Operating point load calls:                 37
Operating point event passes:               2
Transient analysis load calls:              4299
Transient analysis timestep backups:        87
```

This command produces a tabular listing of event-times in the first column and node values in the second column. The 1 ns delays can be clearly seen in the fifth decimal place of the event times.

Note that the eprint command also gives statistics from the event-driven algorithm portion of XSPICE. For this example, the simulator alternated between the analog solution algo-

rithm and the event-driven algorithm one time while performing the initial DC operating point solution prior to the start of the transient analysis. During this operating point analysis, 37 total calls were made to event-driven code model functions, and two separate event passes or iterations were required before the event nodes obtained stable values. Once the transient analysis commenced, there were 4299 total calls to event-driven code model functions. Lastly, the analog simulation algorithm performed 87 timestep backups that forced the event-driven simulator to backup its state data and its event queues.

A similar output is obtained when printing the values of digital nodes. For example, print the values of the node "div8_out" as follows:

```
XSPICE 5 -> eprint div8_out


**** Results Data ****

Time or Step
div8_out


0.000000000e+00     1s
1.810070000e-04     0s
2.610070000e-04     1s
3.410070000e-04     0s
4.210070000e-04     1s
5.010070000e-04     0s
5.810070000e-04     1s
6.610070000e-04     0s
7.410070000e-04     1s
8.210070000e-04     0s
9.010070000e-04     1s
9.810070000e-04     0s



**** Messages ****


**** Statistics ****

Operating point analog/event alternations:  1
Operating point load calls:                 37
Operating point event passes:               2
Transient analysis load calls:              4299
Transient analysis timestep backups:        87
```

From this printout, we see that digital node values are composed of a two character string. The first character (0, 1, or U) gives the state of the node (logic zero, logic one, or unknown logic state). The second character (s, r, z, u) gives the "strength" of the logic state (strong, resistive, hi-impedance, or undetermined).

If you wish, examine other nodes in this circuit with either the plot or eprint commands. When you are done, enter the "quit" command to exit the simulator and return to the operating system prompt:

```
XSPICE 6 -> quit

So long.
$
```

Index