Personal Computer
Circuit Design
Tools

CODE MODEL SOFTWARE DEVELOPMENT KIT

© copyright ***intusoft*** 1995

P.O.Box 710
San Pedro, Ca. 90733-0710
Tel. (310) 833-0710
Fax (310) 833-9658

intusoft provides this manual “as is" without warranty of any kind, either expressed or implied, including but not limited to the implied warranties of merchantability and fitness for a particular purpose.

This publication could contain technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of this publication.

Copyright
intusoft, 1995. All Rights Reserved. No part of this publication may be reproduced, transmitted, transcribed, stored in a retrieval system, or translated into any language in any form by any means without written permission of Intusoft.

IsSpice4 is based on Berkeley SPICE 3F.2 which was developed by the Department of Electrical Engineering and Computer Sciences, University of California, Berkeley CA and XSPICE, which was developed by Georgia Tech Research Corp., Georgia Institute of Technology, Atlanta Georgia, 30332-0800

is a trademark of ***intusoft***

Intusoft, the Intusoft logo, IsSpice, IsSpice3, IsSpice4, SpiceNet, IntuScope, PreSpice, IsEd, and CMSDK are trademarks of Intusoft, Inc. All company/product names are trademarks/registered trademarks of their respective owners.

All company/product names are trademarks/registered trademarks of their respective owners. Microsoft Visual C++, Windows, Windows95 and Windows NT are registered trademarks of Microsoft Corporation.

Printed in the U.S.A.

Table Of Contents

Chapter 1 Table Of Contents

7 Introduction
7 What's In the CMSDK Package
7 Hardware & Software Requirements
7 Installing The CMSDK
8 Hardware Protection Key Installation
9 Definitions

Chapter 2 Simulation Algorithms

11 IsSpice4 System Overview
15 Analog Simulation
17 Digital Simulation
18 Node Bridge Models
18 User-Defined Nodes

Chapter 3 Code Model Development

21 Introduction
22 The Code Modeling SDK
25 Adding Tools To Visual C++
27 Creating Code Models
28 Creating A Project Directory
28 Editing Support Files
29 Opening The Project File
29 Creating The Interface Specification File
37 Creating The Model Definition File
43 Building The Code Model DLL
44 Building A DEBUG DLL
44 Setting Up The Debug Environment
45 Accessing A Code Model in IsSpice4
46 User-Defined Nodes
47 UDN Directory
47 Editing UDNpath.Lst
48 UDN Project File
48 User-Defined Node Definition File
50 Building A UDN

Chapter 4 API Calls

51 What Are API Calls?
53 API Calls
57 AC_GAIN
58 ANALYSIS
58 ARGS
59 CALL_TYPE
59 CALLOC
59 cktABSTOL
60 cktNOMTEMP
60 cktRELTOL
60 cktTEMP
61 cktVOLT_TOL
61 cm_analog_auto_partial
62 cm_analog_converge
62 cm_analog_not_converged
62 cm_ramp_factor
63 cm_analog_set_perm_bkpt
64 cm_analog_set_temp_bkpt
64 cm_climit_fcn
66 cm_complex_add
66 cm_complex_div
66 cm_complex_mult
67 cm_complex_set
67 cm_complex_sub
68 cm_event_alloc
68 cm_event_get_ptr
69 cm_event_queue
69 cm_message_get_errmsg
70 cm_message_send
71 cm_netlist_get_c
71 cm_netlist_get_l
72 cm_smooth_corner
73 cm_smooth_discontinuity
74 cm_smooth_pwl
74 deltaTemp
75 EQUAL
75 FREE
76 getVar
77 getVarPtr
77 gMIN
77 imagFreq
78 INIT
78 INPUT

79 INPUT_STATE
80 INPUT_STRENGTH
80 INPUT_STRUCT_PTR
81 INPUT_STRUCT_PTR_ARRAY
82 INPUT_STRUCT_PTR_ARRAY_SIZE
82 isBYPASS
82 isINIT
83 isMODEAC
83 isMODEINITFIX
83 isMODEINITJCT
84 isMODEINITPRED
84 isMODEINITSMSIG
84 isMODEINITTRAN
85 isMODETRAN
85 isMODETRANOP
86 isMODEUIC
86 lastSTATE
86 lastSTATEptr
87 LOAD
88 MALLOCED_PTR
88 MALLOC
88 newState
89 newVar
90 OUTPUT
90 OUTPUT_CHANGED
91 OUTPUT_DELAY
92 OUTPUT_STATE
92 OUTPUT_STRENGTH
93 OUTPUT_STRUCT_PTR
94 PARAM
94 PARAM_SIZE
95 PARAM_NULL
96 PARTIAL
97 PORT_SIZE
97 PORT_NULL
98 postQuit
98 RAD_FREQ
98 realFreq
99 REALLOC
99 stateIntegrate
101 STATIC_VAR
102 STRUCT_MEMBER_ID
102 STRUCT_PTR
103 STRUCT_PTR_1
103 STRUCT_PTR_2

104 thisSTATE
104 thisSTATEptr
105 T( )
106 TEMPERATURE
106 TIME
106 TOTAL_LOAD
107 udn_XXX_compare
107 udn_XXX_create
108 udn_XXX_dismantle
108 udn_XXX_copy
109 udn_XXX_initialize
110 udn_XXX_invert
110 udn_XXX_ipc_val
111 udn_XXX_plot_val
111 udn_XXX_print_val
111 udn_XXX_resolve

Chapter 5 Examples

113 A Simple Gain Block
119 A Capacitor Model
130 A Digital OR Gate
135 User-Defined Node
140 Node Bridges (Hybrid Models)

Appendices

145 Appendix A: Translation Of SPICE 3 Data Structures
148 Appendix B: The SPICE 3 CKTcircuit Data Structure
149 Appendix C: Project Settings

Introduction

Welcome to Intusoft's XDL model development environment for Windows NT or Windows95. This manual provides complete installation instructions and reference material.

We would like to thank you for purchasing Intusoft software. As you already know, circuit simulation is an important part of the overall circuit design task. The ICAPS simulation system and the CMSDK will provide you with capabilities that will save you money, improve the understanding of your circuit designs, and shorten your design cycle.

What's In the CMSDK Package

Your package should include:

CMSDK disks
Code Model SDK Manual

Hardware & Software Requirements

- Windows NT or Windows95/98
- 486 or Pentium
- Hard disk (approximately 10 megs of disk space)
- Microsoft Visual C++ version 4.x or 5.x

Installing The CMSDK

- Insert the CMSDK Disk 1 into the floppy drive.

- From Windows, run **A:Setup** if the disk is in the A drive or **B:Setup** if the disk is in the B drive.

Back up your distribution diskettes!

Follow the directions. When complete, all software will be installed in the SPICE4 directory.

Hardware Protection Key Installation

Included in your package is a hardware protection key that must be installed before the software can be run. The key has a 25-pin connector at each end and should be plugged directly into your parallel port. The key should be located as close to the port on the back of the computer as possible. If you already have an output device connected to the parallel port install the key between the device and the port. The protection key is transparent to other users of the parallel port and does not interfere with data sent in either direction on the port.

DO NOT LOSE THIS KEY!

The CMSDK software can not be run without it.

- Once the software and key are in place, the installation is complete. You may proceed to view the readme files or the tutorial sections.

Troubleshooting Hardware Protection Key Operation
Although every effort has been made to ensure the highest levels of quality and compatibility, the protection key, like any other PC peripheral, might not run on certain PC configurations.

If you have a problem with the key it can usually be overcome by checking the following:

1) Are the key and printer card inside the computer well-connected and are their screws properly fastened? Reseat them if they are not.

2) Is a printer connected to the same port as the key? If so, does it work properly? (Please bear in mind that although this is a good sign, it does not necessarily mean that the port or the printer are 100% sound). If there is a printing problem try using another printer cable and if possible, another printer.

3) If possible, try installing a second parallel port for the protection key, and thus circumventing any problem caused by the printer. If this is not possible try replacing the original parallel port.

4) Try using the ICAP/4 system on another PC.

5) If you are using another key(s), try moving the Intusoft supplied key closer towards the computer's parallel port. Many protection keys use the power from the printer port to operate. If a number of keys are placed in-line, some printer ports may not be able to supply enough power and the power drop along the string of keys will cause the keys at the end to fail.

Definitions

The following terms will be used throughout the documentation:

accessor macros - An API call used processed by CMPP.

AHDL - Analog Hardware Description Language. A high level language used to describe analog and mixed signal functions.

API calls - Functions, macros, and accessor macros that are used to interface a code model to the IsSpice4 simulator.

behavioral modeling - The process of describing the behavior and structure of a continuous time function using the set of primitive elements included in the IsSpice program.

CML.DLL - The Windows DLL that contains all of the Intusoft XDL code models that are shipped with IsSpice4. CML.DLL contains over 40 analog, digital and hybrid code models.

CMSDK - Code Model Software Development Kit. The set of software tools used in conjunction with MS VC++ to convert IFS and MOD files into a Windows DLL that IsSpice4 can access.

code model - A model developed using XDL. A code model consists of an IFS file and a MOD file.

DLL - Windows Dynamic Linked Library.

hybrid - An XDL code model which has ports that are analog and event driven (digital, real, integer, etc.).

IFS file - A file that contains a description of a code model's ports and model parameters.

macromodel - A combination of primitive IsSpice elements grouped together to emulate a particular function.

MOD file - A C language file that describes the behavior of a code model.

primitives - The most basic building blocks included in the IsSpice4 program. Blocks include electrical (resistors, capacitors, BJTs), mathematical (equations, Tables, polynomials), digital (boolean expressions), and procedural (If-Then-Else) elements.

state variable - A variable for which a time history is stored.

VHDL - VHSIC (Very High Speed Integrated Circuit) Hardware Description Language. A high level language used to model and develop digital circuitry.

Visual C++ - Refers to Microsoft Visual C++ version 1.1 or 2.x

XDL - XSPICE eXtended Description Language. The combination of the IFS and MOD files make up a model's XDL description.

XSPICE - a derivative of Berkeley SPICE 3C.1 produced by the Georgia Tech Research Corporation, Georgia Institute of Technology, a unit of the University System of Georgia.

Simulation Algorithms

IsSpice4 System Overview

A top-level diagram of the IsSpice4 simulator is outlined in the following paragraphs.

Transient Simulation of Analog Circuits
The IsSpice4 simulator solves a set of nonlinear differential equations describing the behavior of an electrical circuit. The equations are formulated using Kirchoff's current law, KCL; that is, the summation of currents at each circuit node equals a constant. The following matrix algebra illustrates the KCL solution:

[Y][V] = [J]
where: Y is the admittance matrix
V are the node voltages
J is the node current excitation

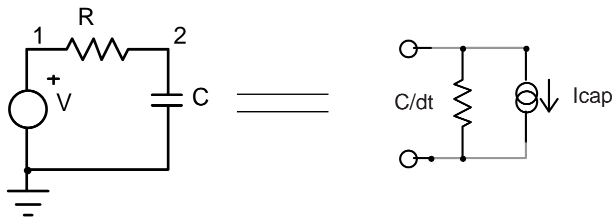
This formulation does not handle voltage defined branches or current controlled branches. In order to handle theses cases without having to use a different topology for Transient and DC analysis, a modified nodal analysis (MNA) formulation is used [1].

$$\begin{bmatrix} Y & B \\ C & D \end{bmatrix} \begin{bmatrix} Vn \\ Ib \end{bmatrix} = \begin{bmatrix} J \\ E \end{bmatrix}$$

where:
Y,V and J are defined above and there are n nodes and b branches
I is the vector of voltage defined branch currents
B is the coupling matrix for current controlled, current defined branches
C, D and E define the relations for the voltage defined branches.

The code model XDL hides these details, filling in the matrix entries depending on how you set up the interface specification file. There is, however, one aspect of the solution you must understand. Your model needs to include the state variables and provide the partial derivatives needed for the Newton-Raphson iterations. The following example illustrates the matrix entries for a simple r-c network using a (CAPG) capacitor code model.

$$\begin{bmatrix} \frac{1}{R} & \frac{-1}{R} & 1 \\ 1 & \frac{1}{R}+C\frac{d}{dt} & 0 \\ 1 & 0 & 0 \end{bmatrix} \begin{bmatrix} V1 \\ V2 \\ Ivsrc \end{bmatrix} = \begin{bmatrix} 0 \\ Icap \\ Esrc \end{bmatrix}$$



To account for nonlinearities, the equations are iterated using the Newton-Raphson technique until convergence is determined. The parameters passed to you for input are the solution vectors from the last iteration. The values you get for the first iteration have been predicted forward based on a time history of solution vectors. Be careful not to alter these input parameters or to use them to hold values that are iterated since they will be discarded and replaced by the next solution vector.

See the stateIntegrate API call for more information.

When doing a transient simulation, the circuit's dynamic states are represented by their large signal values along with the partial derivatives used for estimating solution changes in the Newton-Raphson iteration. The circuit states are saved in an array of doubles. In the code model XDL you reserve as many states as you need using the newSTATE function. States carry with them a time history that depends on the order of the integration (ie. gear 3). For most applications only the current and previous states are important. For each state variable, you

See the Capacitor example in the Examples chapter for more information.

will need these 2 entries in order to have the data required for integration. These states are used for the integrand and integral in a call to stateIntegrate in order to perform the integration during a transient analysis. This corresponds to current and charge in a capacitor code model.

If you use instance variables you can reserve space for them using the newVar function. Instance variables are values that are computed and vary from one instance of a model to another. A transistor's transconductance or base-emitter voltage would be examples of instance variables. Storage for states and instance variables will be deleted by IsSpice4 during re-initialization, as shown in the flow diagram on the next page.

If you have experience in writing SPICE 3 models, Appendix A illustrates the differences between the way the XDL and SPICE 3 reference data. The code model XDL makes detailed simulator knowledge less important than was the case in SPICE 3. There is no need to be concerned with these differences unless you are porting existing models.

The ".MOD" file is parsed into a ".C" file by the code model compiler, CMPP. This process gives you simple access to otherwise private parts of the XDL data structures. In addition, several pointers and values are initialized in your main code model function. This allows direct access to the IsSpice4 CKTcircuit data structure. Appendix B shows how this was done so you can add additional accessor macros.

IsSpice4 separates the temperature dependent calculations from the other load operations in order to reduce the computational overhead of the model. This is not done in the code model XDL. If you have computationally intensive calculations that only need to be performed once, they should be placed in the INIT section of your code model.

Steps in the simulation process are as follows:

Load each circuit “line”

Expand SPICE 2 polynomial controlled sources into ISSPICE4 arbitrary source syntax (Convert E, F, G, and H elements to the B element)

Flatten subcircuit hierarchies

Load model parameters

For any requested simulation

Initialize simulation constants to default or optional values

Unsetup[1], release storage from last simulation

Setup[1], allocate storage and make “fast” pointers to model data

Calculate Temperature[1] dependent coefficients for models

Do the Transient simulation **Interactive Entry Point for Transient Analysis**
Similar entry points exist for the other analysis modes.

Transient specific initialization

DCOP[1], Compute operating point if no UIC is specified on the .TRAN line
if OP fails, try gmin stepping
if OP fails, try source stepping

If UIC[1] is present, find initial operating point
if initial OP fails, do DCOP, and apply UIC states at first transient iteration

Set Mode, INIT,...Tran

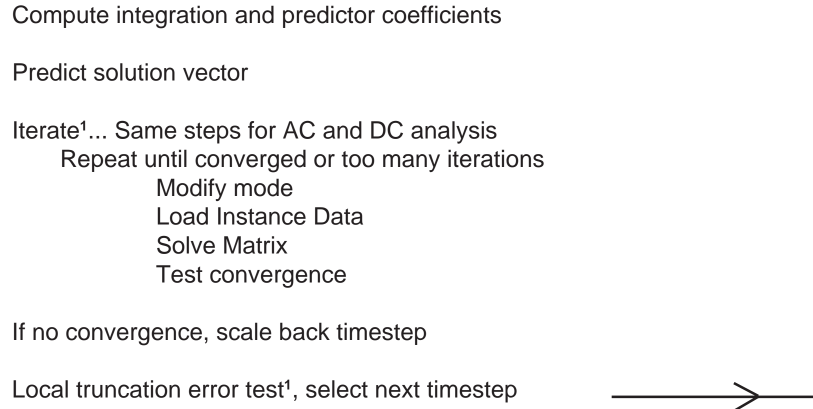
Load timestep

Set breakpoints[1]

Delete old breakpoints

Housekeeping (adjust storage, output, pause, abort, etc.)

If breakpoint, set the timestep



Note 1: These steps iterate through all models and device instances.

Analog Simulation

See the IsSpice4 User's guide for more information

This section provides descriptions for each analysis that is supported by the CMSDK for analog code models. These descriptions will focus on the elements of the code model you will be responsible for in order to implement an XDL code model. They are not meant to be a definitive discussion of the analysis types.

DC Operating Point and DC Sweep

DC and swept DC analyses are steady-state forms of system analysis. 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 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 viewed.

If your code model must handle this type of analysis it will be necessary to provide code that correctly produces the voltages, currents, and partial derivatives for all ports defined for your device. You will also be responsible for evaluating any necessary initial conditions that may apply to the code model.

This code will be called and evaluated for every iteration in a DC type of analysis. This includes the transient DC operating point, the small signal operating point and the DC sweep analyses.

AC - Small Signal Frequency Analysis
The AC analysis is a small signal, linear, frequency analysis. No nonlinearities are taken into account during this analysis. The analysis begins by performing a DC operating point. Inside IsSPICE4 these DC voltages are used to determined the small signal equivalent circuit for all active devices.

Inside a code model the DC operating point, as calculated by the code, is inserted for the DC analysis, and can be used to determine the characteristics of the AC response of the code model. You are required to provide the AC gain, output with-respect-to-input, of every port combination in your code model. Special macros, as described in later chapters, are used to accomplish this. To be compatible with future Pole-Zero analysis capabilities, you must provide these calculations using a complex frequency axis. By default, IsSPICE4 provides the complex component of frequency, $j\omega$, for AC frequency calculations. Additional macros are provided to reduce the complexity of these calculations.

Transient - Nonlinear Time Domain Analysis
A transient analysis begins by obtaining a DC solution to provide an initial starting point for simulating time-varying behavior. Once the DC solution is obtained, the time-dependent aspects of the system are reintroduced, and the event-driven and analog algorithms incrementally solve for the time-varying behavior of the entire system. Inconsistencies in node values are resolved by the two simulation algorithms such that the time-dependent waveforms created by the analysis are consistent across the entire simulated time interval. The result-

ing time-varying descriptions of node behavior for the specified time interval are all accessible.

It is your responsibility to provide the state variables, perform the proper arithmetic, and provide an output, and partial derivative, for each port assigned to the code model for each time point. The partial derivatives should be continuous through the second derivative to provide proper convergence.

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.

Support is included for digital nodes that are simulated by a general purpose 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 a simulation of the same circuit using analog code models and nodes.

When creating a digital, event-driven, code model you are responsible for; providing the proper DC operating points for all outputs, and a state, strength, or output for each event. If an event-driven code model is not going to post an event the OUTPUT_CHANGED API call should be set to FALSE. Any time the OUTPUT_CHANGED API call is set TRUE an output value must be posted. More information can be found in the OR gate example in the Examples chapter.

Node Bridge Models

ISSPICE4'S Code Model support allows you to develop models that work under the analog simulation algorithm, the event-driven simulation algorithm, or both. When event driven and analog models are mixed in the same simulation, some method must be provided for translating data between the different simulation algorithms. Node bridges are code models developed for the express purpose of translating between the different algorithms or between different User-Defined Node types.

When making a code model that process both analog and event-driven algorithms you will have to provide the proper mechanisms for translating the event-driven data into continuous analog data or vise versa if the situation applies.

User-Defined Nodes

In addition to the Code Model features of ISSPICE4 that support traditional analog and digital modeling, ISSPICE4 supports creation of "User-Defined Node" types. AlthoughISSPICE4 provides built-in support for a 12 state digital simulation, the event-based simulation capability is not limited to digital simulation. It is available for all code models. This combination of the user-defined signal and the event-based simulation allows ISSPICE4 to support mixed-level, as well as mixed-signal simulation and indeed, to support application domains well beyond the electrical domain.

User-Defined Node types allow you to specify nodes that propagate arbitrary data structures and data other than voltages, currents, and digital states. 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. This example is covered in more detail in the Examples chapter. More complex applications may define types that involve complex data such as digital data vectors or even nonelectronic data.

ISSPICE4's digital simulation is 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 level and a strength value. The real and integer node types supplied with ISSPICE4 are also more simplified examples of User-Defined Node types.

When constructing a User-Defined Node you are responsible for providing the functions to allocate, initialize, copy and compare the data structures that make up your User-Defined Node. Other functions are available to allow greater control over the characteristics of the User-Defined Node. A mode detailed description of a User-Defined Node can be found in the Examples chapter.

[1] L. W. Nagel, "SPICE2: A computer Program to Simulate Semiconductor Circuits", ERL-M520, U.C. Berkeley, 1975, pg. 65

Code Model Development

Introduction

Traditionally, SPICE models are constructed from a set of predefined primitive electrical and mathematical elements provided with the simulator. The term “macromodeling” or “behavioral modeling” is used to reference this style of model development. As IsSpice matured beyond SPICE 2G.6 additional built-in elements were provided to allow greater modeling flexibility. For example, IsSpice3, introduced If-Then-Else statements and in-line math equations. If-Then-Else statements allow models to perform procedural decisions while math equations greatly improve upon the limitations of SPICE 2 polynomials.

However, more complex models and system level blocks can tax the efficiency and complexity bounds of even these powerful elements. In addition, interfacing SPICE with other simulation software is extremely difficult.

With Code Modeling Software Development Kit (CMSDK); Intusoft lowers all of the aforementioned barriers and provides the flexibility and power to add tremendous functionality to IsSpice4.

Throughout this chapter we will introduce the concept of Code Modeling. Code Modeling is the process of creating new primitive elements and functions using XDL (eXtended Description Language).

The Code Modeling SDK

The CMSDK consists of a set of tools that provide easy access to the appropriate data structures and simulation routines for the analog and event driven algorithms available in ISSPICE4.

The CMSDK requires Windows NT or Windows95 and Microsoft Visual C++ version 1.1 or 2.x.

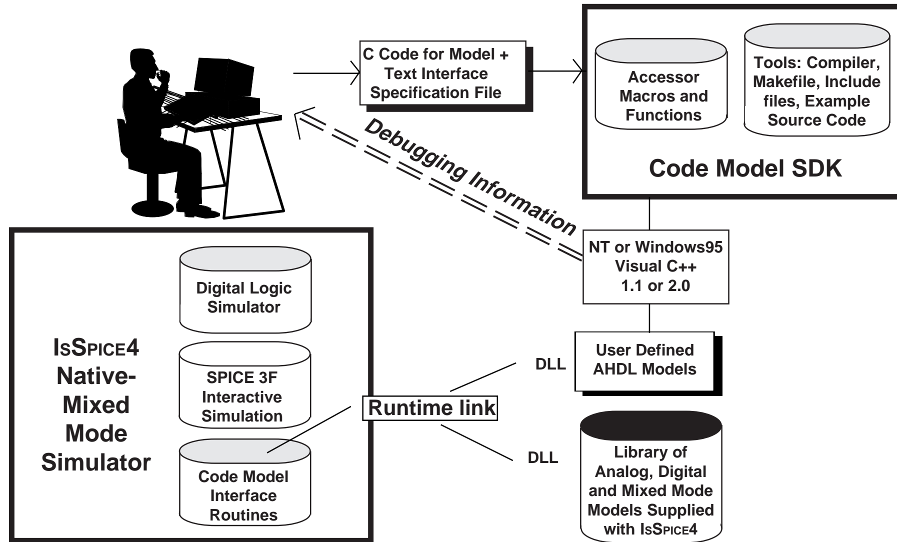
The benefits of developing models using C code have not been widely recognized. This is mainly due to the limited programming and simulator support and extensive knowledge of SPICE that is required. The CMSDK alleviates these difficulties. The CMSDK works with additions to the ISSPICE4 core to tell the simulator how to parse the code model netlist and how to call C code that defines the model's behavior (Figure on next page). Code models should be seen as an extension to the set of primitive devices offered in ISSPICE4. Support is provided for the AC, DC, and Transient analyses for the analog models and DC and Transient analyses for event driven models.

An XDL model is composed of two files. An interface specification file, which describes the model's connections and parameters, and a C file that describes the model's behavior. The CMSDK provides a comprehensive set of API calls to simplify the model development process and to insulate you from the underlying SPICE data structures.

The code modeling architecture is based on the XSPICE program developed by the Georgia Tech Research Corp., a unit of the Georgia Institute of Technology.

A makefile is used to compile and link the code model. Compiler and linker diagnostics aid in the debugging process as they would for any application being developed under Microsoft Visual C++. The result of a successful build is a code model library (DLL) that can be accessed by ISSPICE4. Visual C++ then provides the ability to examine variables, step through the execution, and otherwise support debugging.

Once the model is performing correctly, the code model DLL can be copied for use on other machines running ISSPICE4. These steps are easily learned and only require a few minutes to perform once the model description is written.



The following directory structure is set up by the ICAP/4Windows software and the CMSDK. The CMSDK is placed below the ICAPS directory (default: SPICE4) at the same level as the IS directory holding the IsSpice4 executable as shown below.

Spice4
- **Circuits**
- **CMSDK**
  - **bin** - utilities
  - **include** - header files
  - **src** - batch files, routines to construct DLLs
    - **cmcommon** - source code for DLL (dll_main.c)
      - **\obj** - Lib files for common DLL routines
    - **real** - example of real code models and a real UDN
    - **sample** - misc. code model examples
    - **simple** - simple analog gain code model
      - **\gain** - gain.mod, ifspec.ifs, gain.c, gain!.c
    - ...
    - *all other DLL directories are placed at this level*
    - DLL\ - DLL containing new models (MOD1, MOD2 ... MODn)
      - \MOD1 - Model Definition, IFS, and C files
      - \MOD2
      - \MODn
  - **test** test circuits
- **IS** - ICAP/4 IsSpice4 subdirectory
- **IN** - ICAP/4 IntuScope subdirectory
- **PR** - ICAP/4 model library subdirectory
- **SN** - ICAP/4 SpiceNet subdirectory

The contents of the directories below the CMSDK directory are;

bin
Contains the CMPP preprocessing utility and the DLL makefile (MakeDLL.Mak). CMPP is used to expand the code model accessor macros and IFS file into the proper C code. MakeDLL.Mak is used to run CMPP from within an nmake compatible makefile.

doc
Miscellaneous documents describing various topics involved in code model development.

test
Contains test circuits for the code models distributed with the CMSDK.

include
Contains all of the necessary header files to build code model DLLs.

src
Contains the makefiles and batch files that control the compilation of code model DLLs. A batch file called Cleanxx.Bat (xx represents the version of Visual C++ you are using) will remove all compiler generated files for the currently active project. It should be run from within Visual C++ from the TOOLS menu. The remaining files; CMLtgt.mak, makefile.cml, mod_to_c.cml, and mod_to_c.mak are used to run CMPP and construct the C source from the model definition file.

Directories below SRC are used as working directories for new DLLs. The models within a DLL are stored in directories below the DLL directory. Each DLL directory will contain the Visual C++ project file for the DLL and Visual C++ generated files, as well as three support files, Ident.h, Modpath.Lst and/or Udnpath.Lst file(s). These support files will be covered in the Editing Support Files section later in this chapter.

Adding Tools To Visual C++

Before we proceed to create code models there are two tools that should be added to the Tools menu of your Visual C++ package. For instructions about adding tools to your Visual C++ work space please consult the documentation that accompanies your version of Visual C++.

Clean DLL
This tool is not necessary for creating a code model DLL. It is used to delete all extraneous files from the DLL directory you are working on. If you wish to edit this file to provide additional clean up please feel free to do so. The following settings should be used if you wish to add this tool to the Tools menu.

Visual C++ version 1.1

Command Line - path to (including) Clean11.Bat
Menu Text - Clean DLL
Arguments - $ProjDir
Check the Redirect to Output Window check box.

Visual C++ version 2.0

Menu Text - Clean DLL
Command - path to (including) Cleanout.Bat
Arguments - $ProjDir
Check the Redirect to Output Window check box.

The batch file for the command lines described above can be found in the SRC directory of the CMSDK.

Convert Mod to C

This tool is required in order to compile a code model DLL. It calls the nmake utility provided with Visual C++ and provides, as an argument, the makefile containing the rules for converting the model definition files (.MOD) into standard C source files (.C). The makefile uses the CMPP utility provided as part of the CMSDK. The following settings should be used when adding the tool to the Tools menu.

Visual C++ version 1.1

Command Line - path to (including) Nmake.Exe
Menu Text - Convert MOD to C
Arguments - /f ..\MOD_to_C.MAK
Initial Directory - $ProjDir
Check the Redirect to Output Window check box.

Visual C++ version 2.0

Menu Text - Convert MOD to C
Command - path to (including) Nmake.Exe
Arguments - /f ..\MOD_to_C.MAK
Initial Directory - $ProjDir
Check the Redirect to Output Window check box.

Nmake is located in the BIN directory of your Visual C++ installation.

Creating Code Models

Code Models are created using a combination of CMSDK preprocessing tools and Visual C++. A code model project will consist of a Visual C++ project, an external makefile (mod_to_c.mak), and a set of definition files. The external makefile, mod_to_c.mak, is run by nmake prior to compiling a code model DLL. This makefile translates the definition files into C code which the Visual C++ compiler uses to construct the final code model DLL. This makefile appears in your Visual C++ Tools menu as “Convert MOD to C”. The basic steps required to create, compile the DLL containing Code Models or User-Defined Nodes are;

- Create the project DLL directory and any model directories
- Edit support files
- Opening the project file
- Create the Interface Specification file (IFS)
- Create the model definition file (MOD)
- Run “Convert Mod To C” to construct the proper C files
- Build the Windows DLL containing the new code model

The explanations that follow will assume that you have a working knowledge of Microsoft Visual C++, version 1.1 or 2.x, and are familiar with the Windows 95 or Windows NT operating system.

Creating A Project Directory

The first step in creating a code model, or User-Defined Node, is to create the DLL directory. This directory must be placed below the SRC subdirectory within the CMSDK directory structure. The easiest way to create the DLL directory is to;

- Use the File Manager to duplicate the contents of the Simple subdirectory provided with the CMSDK.
- Change the name of the Visual C++ project files in the duplicated directory.

Editing Support Files

The next step is to open the Modpath.Lst file and enter the names of all model directories that will be associated with the new DLL. These directories must be located below your new DLL directory. The format of the file is simple and consists of the name of each model directory on a new line with a "\" after each, except the last directory named in the file.

If the DLL is set up to contain user defined nodes, then a Udnpath.Lst file must also be present in the DLL directory. The format of this file is the same as the Modpath.Lst file except that the actual filename in the UDN directory is the name on each line as well.

The remaining support file is ident.h. This file contains reserved definitions for licensing and an ID string. The licensing definitions are reserved for future use and should not be altered. The ID string can contain any text string up to 255 characters. This string will be embedded into the DLL.

Opening The Project File

Start Visual C++ and open the project file located in the DLL directory for the DLL you wish to work on. If you copied an existing project most of the defaults for the project will be set with the following exceptions;

Debug Information
The debug settings described in the Setting Up The Debug Environment section will have to be changed depending on the model and DLL you are working on. All test circuits are located in the TEST directory. This directory is located at the same directory level as the SRC directory.

Output Generation
When using Visual C++ 2.0 the output of the build should be directed to the IS directory located under the SPICE4 directory. This will facilitate debugging the new DLL. For those using Visual C++1.1, a copy of SPICE4.EXE will have to be placed in the project's DLL directory.

Creating The Interface Specification File

The Interface Specification File is a text file that describes, in a tabular format, all the information needed to interface the code model to ISSPICE4. This information includes all nodal connections (ports), all model parameters, their default values, and the name of the model itself. This file is read by the CMPP utility and translated into C code for implementing the interface between the code model and ISSPICE4. In order for the interface to work correctly the format of the Interface Specification File template must be followed.

An example IFS file is given next. The example is followed by detailed descriptions of each of the entries.

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	num_coeff
Description:	“input offset”	“gain”	“numerator coeff”
Data_Type:	real	real	real
Default_Value:	0.0	1.0	-
Limits:	-	-	-
Vector:	no	no	yes
Vector_Bounds:	-	-	[1 -]
Null_Allowed:	yes	yes	no

Parameter Table

Parameter_Name:	den_coeff	out_ic
Description:	“denominator coeff”	“output initial value”
Data_Type:	real	real
Default_Value:	-	-
Limits:	-	-
Vector:	yes	no
Vector_Bounds:	[1 -]	-
Null_Allowed:	no	yes

Parameter Table

Parameter_Name:	denorm_freq
Description:	“denormalized corner freq.(radians)”
Data_Type:	real
Default_Value:	1.0
Limits:	-
Vector:	no
Vector_Bounds:	-
Null_Allowed:	yes

Example; Device Call Line and Model
A12 1 2 Cheby3K
.Model Cheby3K s_xfer(in_offset=0.0 gain=1.0 num_coeff=[1.0]
+ den_coeff=[1.0 1.42562 1.51620])

Please refer to the Design Entry & Data Analysis manual for details about interfacing your new code model to SpiceNet.

The Name Table

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

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 preceded 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.

The name used in the Spice_Model_Name field, must be entered in lower case.

SPICE Model Name

This parameter is the model TYPE parameter that is required on a .MODEL line. It may or may not be the same as the C function name. It is preceded by the "Spice_Model_Name:" keyword. For the IFS file given the .MODEL line would be

```
.Model mytrans xfer(param1=#, ..)
```

Description

The description (C string literal) is used to describe the purpose of the code model. It is preceded by the "Description:" keyword.

The Port Table

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

Port Name

The port name is preceded by the “Port_Name:” keyword and followed by the name of the port. This port name is used to obtain and return input and output values within the model function. This will be discussed in more detail in the next section.

Description

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

Direction

The direction of a port specifies the intended data flow direction of the port. The direction must be either “in”, “out”, or “inout”. It is preceded by the “Direction:” keyword.

Default Type

The Default_Type specifies the expected signal type. The following table summarizes the allowable port 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
vnam	voltage source name	in
<identifier>	user-defined type	in or out

Allowed Types

A port must specify the types it is allowed to assume. The type names must be taken from those listed in Port Types table. The list of type names are separated by a blank or comma and delimited by square brackets, e.g. "[v vd i id]" or "[d]"). See the IsSpice4 documentation for information on how to override the default port type on the

Vector

A port, which is a vector, can be thought of as a bus. The Vector field is preceded with the "Vector:" keyword and followed by a boolean value: "YES", "TRUE", "NO" or "FALSE". The values "YES" and "TRUE" are equivalent and specify that this port is a vector (may have 1 or more connections). Likewise, "NO" and "FALSE" specify that the port is not a vector. Vector ports must have a corresponding vector bounds field that specifies the minimum and maximum size of the vector port.

Vector Bounds

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

Null Allowed

In some cases, it is desirable to permit a port to remain unconnected. The Null_Allowed field specifies whether this constitutes an error condition. The Null_Allowed field is preceded by the "Null_Allowed:" keyword and 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. If the Null_Allowed is TRUE, and the port is unconnected, the keyword "NULL" must be inserted on the code model call line, for example, "A1 1 2 NULL 3 4 Mygain".

The Parameter Table

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

Parameter Name

The parameter name is the name of the model parameter used on SPICE .MODEL line. It is preceded by the "Parameter_Name:" keyword.

Description

The description string is used to describe the purpose of the parameter. It is preceded by the "Description:" keyword.

Data Type

The Data_Type field determines the type of value the model parameter will accept. It is preceded by the keyword "Data_Type:" and is followed by one of the following: boolean, complex, int, real, and string.

Default Value

If the Null_Allowed field is "TRUE" for this parameter, then a default value may be specified. This value is supplied in the event that the .MODEL line does not supply a value. The default value must be of the correct type. The Default Value field is preceded by the "Default_Value:" and is followed by a numeric, boolean, complex, or string literal (usually a filename).

Limits

Integer and real parameters may be constrained to accept a limited range of values. A range is specified by a square bracket followed by a value representing a lower bound separated by a 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 preceded by the “Limits:” keyword.

Vector

The Vector field is used to specify whether a model parameter is a vector (several values for one model parameter) or a scalar. Like the Port_Table vector field, it is preceded by the “Vector:” keyword and followed by a boolean value. “TRUE” or “YES” means that the parameter is a vector. “FALSE” or “NO” means that it is a scalar.

Vector Bounds

The valid sizes for a vector model parameter are specified in the same manner as are port sizes (see Vector Bounds section).

Null Allowed

The Null_Allowed field is preceded by the “Null_Allowed:” keyword and followed by a boolean literal. A value of “TRUE” or “YES” means that it is valid for the corresponding model parameter to be omitted. 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 model parameter value is omitted.

Static Variable Table

The static variable table is preceded 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.

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 preceded by the "Static_Var_Name:" keyword.

Description

The description string is used to describe the purpose of the static variable. It is preceded by the "Description:" keyword.

Data Type

The static variable's data type is specified by the Data_Type field. The Data_Type field is preceded by the keyword "Data_Type:" and is followed by a valid data type: 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 API functions. 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.

The newVar() API call allows you to allocate memory for static "instance" variables without referencing the IFS file.

Shown next is an example of allocating memory to be referenced by a static pointer variable "x". 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.

```
/* 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;
```

A more flexible method of allocating and using memory for these types of variables is to use the newVar API call. The newVar API call allows you to allocate and use static “instance” variables without the need to specify static variables in the IFS file. Please refer to the Capacitor example in the Examples chapter for more details.

Creating The Model Definition File

The Model Definition file is a C source file that defines a code model’s behavior. Within this file input, output, model parametersn and simulator-specific information is handled through a set of API calls. These API calls, in conjunction with standard C source code, define the code model’s behavior.

A complete list of the available API calls is available in the API Calls chapter. In general, you will use the following API calls most frequently.

PARAM(arg)	PARAM_SIZE(arg)	PARAM_NULL(arg)
INPUT(arg)	INPUT_SIZE(arg)	OUTPUT(arg)
OUTPUT_SIZE(arg)	AC_GAIN(arg,arg)	PARTIAL(arg,arg)

The arguments, arg, are fields specified in the IFS file.

All accessor macros are resolved by the CMPP preprocessor into members of a private data structure called ARGS. While you can dereference this data using the debugger, it is not likely to make much sense and it is not a supported part of the code model XDL. You should limit your debugging to the items available within your model definition file.

The following pseudo code illustrates the basic format that an analog code model definition file will have.

```
if(INIT) {
      // do setup, then
      // calculate temperature and other
      // coefficients that do not change
      // usually those dependent on ".options"
      }
    // initialize "fast" pointers
    switch(ANALYSIS) {
      case MIF_AC:
        // do AC analysis, use both real and imaginary
        // frequency for future compatibility
        break;
      case MIF_DC:
        // do DC analysis
        break;
      case MIF_TRAN:
        if(isMODEINITTRAN) {
          // the first iteration of the first transient
          // time point
          }
        if(isMODEINITPRED) {
          // the first iteration of each time point
          // good for debugging
          }
        // do general transient analysis processing
        // pass through here for every iteration
        break;
      }
```

INIT

You may wish to calculate some values for use in subsequent iterations. To reserve storage for this "instance" data, use the newVar() function.

The initialization of all code models is performed in the IsSPICE4 setup routine. The INIT macro is used to detect the initialization call. This is the location were one time operations should be performed (i.e. circuit temperature dependence, internal coefficient calculation, etc.) Also, during initialization you must allocate memory for state variables using the newState() API call. These states are provided as a pointer to the data accepted by IsSPICE4, in other words double *state[]. The dimension of *state[] is 1 greater than the order of integration. Hence, for Trapezoidal integration, which has an order of 1, the dimension would be 2. This pointer is marched, in time, automatically. This means that the first position in the pointer is always the current state, the second position is the next state and so on. This allows you to quickly access the current and last state pointers. The argument to newState() is the number of pointers you need. For instance, newState(2) will create a 2 pointers.

Initializing FAST pointers

This array of pointers is freed by IsSPICE4 during its unsetup routine. There is no need for you to free this memory.

After allocating the state pointers, the next step is to initialize them. Assuming you are interested in a current and charge for a particular code model this would be done as;

```
double *current, *charge, *currentlast;

    current = thisSTATEptr(0);
    charge = thisSTATEptr(1);
    currentlast = lastSTATEptr(0);
```

State variables are used during integration.

These two macros, thisSTATEptr and lastSTATEptr, are provided for access to the elements of the state pointers allocated by newState(). The first macro, thisSTATEptr, returns a pointer to the array of current states. The second macro, lastSTATEptr, returns a pointer to the array of previous states.

These states will be marched in time, automatically, as the simulation progresses. Hence, thisSTATEptr always points to the current state value and lastSTATEptr always points to the previous state value.

Analysis

Once allocation and initialization have taken place for state variables, the code model must process the different analysis modes. In the pseudo-code provided this is done with a switch-case arrangement. A chain of if-elseif-else statements could have been used as well.

DC Analysis (MIF_DC)

The DC analysis is detected by comparing the return of the ANALYSIS macro with the MIF_DC definition. In this section you must calculate each output as a function of the inputs. For example;

```
OUTPUT(out) = INPUT(in) * r_shunt;
```

You must also calculate the partial derivative of each output with respect to all inputs. For example;

```
PARTIAL(out,in) = r_shunt.
```

ISSPICE4 will make the appropriate entries in the matrix and solution vector depending on the port types specified in the IFS file.

This section of the code (MIF_DC) will be called repeatedly during a DC analysis until the outputs converge. See MIF_TRAN for more information.

AC Analysis (MIF_AC)

The AC analysis is detected by comparing the return of the ANALYSIS macro with the MIF_AC definition. In this section all AC analysis operations must be performed. Calculations depending on operating point can be obtained using the same macros as in the DC analysis (See MIF_DC).

By default, the frequency will be imaginary with no real part; however, if you want your model to work for the pole zero analysis you must handle complex frequency. This is done using the realFreq and imagFreq API calls.

If you have no frequency dependent parts, you must output the small signal gain which is the same as the PARTIAL calculated in the DC analysis, for example:

```
    result.real = r_shunt;
    result.imag = 0;
    AC_GAIN(out,in) = result;
```

Transient Analysis (MIF_TRAN)

The Transient analysis is detected by comparing the return of the ANALYSIS macro with the MIF_TRAN definition. In this section you should process all transient analysis related behavior.

The initial time point for the transient analysis can be detected by using the isMODEINITTRAN API call. This will allow you to set initial conditions when necessary. For example;

```
switch(ANALYSIS){
  ....
case(MIF_TRAN):
  if(!PARAM(ic) && isMODEINITTRAN)
    OUTPUT(cap) = PARAM(ic);
else{
  // process necessary info if no ic
}
...
```

This code fragment demonstrates the use of the isMODEINITTRAN macro. The return value, and the presence of an “ic” parameter in the model’s .Model statement, will establish the output initial conditions. The else if clause allows proper initialization for the case when no ic is present.

During the remainder of the transient analysis you must provide an output and its partial derivatives for the current time and time step, based on the operating point for the current time and the history of inputs. **The convergence of your code model will depend on the ability to create continuous partial derivatives.** Convergence is determined by the simulator for each value that is integrated.

For states that must be integrated, or differentiated, you must call stateIntegrate(), passing the integrand, integral, and a storage pointer to receive the partial derivative.

Integration takes place when the integral is allowed to vary and the integrand is kept constant. For example,

$Q = \int i\,d$

```
cur = *current = INPUT(cap);

stateIntegrate(current,charge,&partial,currentlast);

*current = cur;
```

ISSPICE4 will iterate to a solution until *current is equal to cur producing charge for each iteration.

Differentiation takes place when the integrand is allowed to vary and the integral is kept constant. For example,

$i = \frac{dQ}{dt}$

```
chrg = *charge = vcap * capvalue;

stateIntegrate(current,charge,&partial,currentlast);

*charge = chrg;
```

The integration and differentiation techniques are covered in the Capacitor example in the Examples chapter.

ISSPICE4 will iterate to a solution until *charge is equal to chrg producing current for each iteration.

The matrix being solved during these iterations is a set of small signal sensitivities superimposed on a large signal operating point. The partial derivative provides the small signal model while the outputs are the exact solution. **You will get the best results; that is, the fewest number of iterations to converge, if you make sure the partial derivatives are continuous through the second derivative and you limit very large changes in output to sensible values.** For instance, avoid hard limiters that have no partial derivatives, and don't allow outputs to be off the wall, like millions of amps in a general purpose diode.

It is possible for the simulator to arrive at a correct solution with an incorrect partial derivative. This is one of the more difficult problems to debug. You might want to keep track of the number of iterations your model needs to converge as part of your debugging code. Most problems converge in 2 to 6 iterations;

more than that indicates problems in computing the partial derivatives. By using the isMODEINITPRED macro, which detects the first iteration of each timepoint, you can establish special code (iteration counter), as well as set break points to investigate the convergence of your code model. See the s_xfer (Laplace) code model example.

Memory Allocation
Memory allocated with the newSTATE and newVar API calls, will be freed by ISSPICE4 during the unsetup routine at the end of the simulation. Therefore, there is no need for you to free memory if you use these API calls to allocate the memory.

Building The Code Model DLL

See the Adding Tools section for more details.

Building a code model DLL requires two steps. First the CMPP preprocessing utility is run to convert the .MOD file to a .C file. Then, Visual C++ is used to compile the resulting .C files.

If the project has been copied from an example, then run CMPP by simply selecting the “Convert MOD to C” tool from the Visual C++ Tools menu. Any time a .MOD file has been modified it must be saved and “Convert Mod to C” must be run. This tool is assigned the following options;

executable - nmake.exe
menu text - Convert MOD to C
command line - /f ..\mod_to_c.mak
working dir - $ProjDir

If you have created your own project this should be added to the Tools menu.

Once compiled, you can execute the new DLL in one of two ways.

- First, use Alt+Tab to switch to the Program Manager and run ICAPS. Select the test circuit from the TEST directory located under the CMSDK directory structure. Run a simulation as you normally do.

- The second method is to execute Spice4.Exe directly from within the Visual C++ environment. To do this you will have to set the proper options for the version of Visual C++ you are using. Please refer to the Setting-Up The Debug Environment section for more information.

Building A DEBUG DLL

Building the debug version of the DLL is as simple as building the release version. The first step is to select the DEBUG build option for the version of Visual C++ you are using. Then compile the project as you would for the release version.

Setting Up The Debug Environment

After building the debug version of the code model DLL you must set several options to establish a debugging environment within Visual C++. If you are using, or have copied, one of the examples provided with the CMSDK, these steps are not necessary. With one exception, if you are using Visual C++ 1.1 you must perform the first step listed below.

Visual C++ 1.1
There are two steps to perform in order to create a debugging environment under Visual C++ 1.1.

- Copy SPICE4.Exe and CML.DLL into the DLL directory of the code model you are debugging.

- Select Debug... from the Options menu. In this dialog file in the name of the executable and the path and name of the file you will use to debug the code model DLL.

Visual C++ 2.x
To establish a debug environment for verison 2.x;

- Select Settings... from the Project menu.

- Select both the Debug and Release items from the Settings for: field.

- Select the Debug tab and fill in the following information;

Executable for Debug Session: path to, and name of, the ISSPICE4 executable that you will use for debugging. Typically this will be the version that was installed into the SPICE4\IS directory when the CMSDK was installed.

Program Arguments: path to, and name of, the circuit file to use for debugging the code model DLL.

- Select the Link tab and fill in the following information;

Output file Name: Enter the path to the location of the ISSPICE4 executable and the name of the DLL. By default this would be \SPICE4\IS*dllname*.

Accessing A Code Model in ISSPICE4

The syntax for calling a code model from an ISSPICE4 netlist is as follows:

Format: A*name N1 N2... value*

Examples: A1 1 2 Mygain
.model Mygain gain(...)

- All code models use the reference designation letter "A".
- All code models require a .model statement

Note: once the proper ICAP/4 model library entries and SPICENET symbol are created, a new schematic database can be compiled. You will then be able to access your new code model directly from the SPICENET schematic entry program. The following chapters can be explored for more information:

- Code Model syntax: ISSPICE4 User's Guide, chapter 9

- Adding a Code Model to a model library: Design Entry and Data Analysis manual, chapter 5
- Relationship between the code model call line (i.e. `A1 1 2 Mygain`) and the .Model line (i.e. `.Model Mygain gain(...)`): ISSPICE4 User's Guide, chapter 8

User-Defined Nodes

User-Defined nodes (UDN) provide a method of processing a data type other than analog or digital within the ISSPICE4 simulator. All User-Defined Nodes operate using the event-driven algorithms of ISSPICE4. Currently, the digital and real node types provided with ISSPICE4 are UDNs. In order to interface a node that uses a UDN data type to a node that uses the analog algorithm, a node bridge must be constructed. Node bridges are special code models that translate different data types, for example from digital event-driven data into continuous time analog data. For more information please refer to the Examples chapter.

User-Defined Nodes are created in a manner similar to the code models discussed previously. The basic steps are listed below;

- Create a directory within the desired DLL for the UDN
- Edit any project specific support files
- Open the project file
- Create the UDN definition file
- Build the Windows DLL containing the new code model

UDN Directory

A directory for the UDN definition file should be created in the DLL directory for which the UDN applies. For instance, if you are creating an integer type UDN to work with a set of integer type code models, the UDN directory should be placed with the integer code models. For clarity, the UDNs should be separated from the models within the DLL. For example;

```
SRC
  \INTMODS
    \MULT
    \GAIN
    \UDNINT
```

In this example, a DLL called INTMODS is constructed with code models called MULT and GAIN. The UDNINT directory is where the UDN definition file for the integer code models in INTMODS is placed.

Editing UDNpath.Lst

After creating the directory to hold the UDN definition file the directory name and definitions filename must be added to the DLL's UDNpath.Lst file. This file is located in the DLL directory. Each UDN directory and definition file filename should be stated on a separate line. In general, there will be only one type of node. However, if you are going to group multiple types together in one DLL, separate lines must be used. At the end of every line, except the last, a "\" should be inserted. This is used as a continuation symbol during preprocessing. The following example demonstrates the contents of UDNpath.Lst for integer and real user-defined node types.

```
UND\UDNINT.C \
UND\UNDREAL.C
```

This file can be found in the Samples directory of the CMSDK.

UDN Project File

UDNs are generally part of an existing project. Therefore, you will seldom have a project that contains only a UDN. Aside from editing the UDNpath.Lst file, adding a UDN definition file to a project is straight forward and no more complicated than adding a standard C file to a Visual C++ project. Please consult the on-line help for your version of Visual C++ for more details.

User-Defined Node Definition File

Unlike the Model Definition File which uses CMPP to translate accessor macros, the User-Defined Node Definition file is a pure C language file. The User-Defined Node Definition File (name.c) defines the C functions which implement operations on user-defined nodes. This file uses macros to isolate you from data structure definitions. The macros are defined in a standard header file (EVTudn.h), and translations are performed by the standard C preprocessor.

A complete list of the available macros and functions for UDNs can be found in the API Calls chapter. The following pseudo code illustrates a typical UDN. A complete example is given in the Examples Chapter.

```
void  udn_int_create(CREAT_ARGS)
{
     // Malloc space for an UDN data structure
}
void  udn_int_dismantle(DISMANTLE_ARGS)
{
     // free internally malloc'ed variables
}
```

```
void  udn_int_initialize(INITIALIZE_ARGS)
}
     // Initialize UDN data structure
}
void  udn_int_invert(INVERT_ARGS)
{
     // Invert the UDN node value
}
void  udn_int_copy(COPY_ARGS)
{
     // Copy the structure
}
void  udn_int_resolve(RESOLVE_ARGS)
}
     // Assign the result for node connections
}
void  udn_int_compare(COMPARE_ARGS)
}
     // Compare the structures
}
void  udn_int_plot_val(PLOT_VAL_ARGS)
}
     // Output a value for the UDN data structure
}
void  udn_int_print_val(PRINT_VAL_ARGS)
}
     // Allocate space for the printed value
     // Print the value into the string
}

void  udn_int_ipc_val(IPC_VAL_ARGS)
}
     // send data over an established ipc channel
}
```

The arguments to these functions are macros that are expanded into the correct argument list. At the bottom of the UDN definition file is the data structure that is used to access these functions. The Evt_Udn_Info_t data structure contains the

name of the node type, a description of the node type, and pointers to the functions defining the node type. An example of this structure for an integer type node is given below.

```
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
};
```

Building A UDN

Building the DLL containing the UDN is no different than building any other Visual C++ project. Unlike the code model, the UDN does not need special preprocessing. Therefore, there is no need to run an external processing program such as CMPP.

API Calls

What Are API Calls?

The CMSDK includes several different types of API Calls:

- Accessor Macros (capitalized API calls with the exception of macros used within user-defined nodes) preprocessed by a CMSDK utility
- API calls beginning with
  cm_ - code model functions
  is - macros used to translate SPICE3 models
  ckt - macros that return circuit related data
  udn - functions used within user-defined nodes

Accessor macros are similar to traditional C macros in their use, however, they are not defined in the same way as traditional C macros. An accessor macro, used within a Code Model definition file, is expanded into the correct C code by the CMPP compiler provided with the CMSDK. For example, to obtain a parameter value from the .Model line of a code model you simply use the following syntax;

```
param = PARAM(gain)
```

Here, param is the variable used within the code model definition file and gain is the name of the parameter (as defined in the IFS file) for which a value is to be obtained. The accessor macro

PARAM retrieves the correct value and sets param equal to it. If this accessor macro was not available the following syntax would be necessary;

```
param = private->param[0]->element[1]->rvalue;
```

You can easily see that accessor macros are much easier to use. Accessor macros, and their automated expansion into correct C code, provide maximum flexibility when using input, output, and simulator specific information in the code model definition file.

Arguments to most of the accessor macros are parameter names or port names as defined in Interface Specification Files. If the corresponding port or parameter is a vector type parameter the argument is a bracket delimited index, port[i]. It is also possible for the argument to be an expression involving other accessor macros. For instance;

```
OUTPUT(out[PORT_SIZE(out)]-1])
```

All accessor macros, with the exception of ARGS, resolve to lvalues. That is, they can be assigned a value. For example;

```
OUTPUT(out1) = 0.0
```

In this example, the accessor macro, OUTPUT, accepts an argument, out1, creating access to output port out1. The real value, 0.0, is then assigned to the output port.

The remaining API calls are defined as starndard macros and functions. The header file containing prototypes to these functions is automatically inserted into the Model Definition File.

The following tables list all of the available API calls, by functionality. Following these tables is an alphabetic list of all API functions with a complete description.

API Calls

Circuit

Name	Type	Args	Description
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?
ANALYSIS	enum	<none>	Type of analysis: DC, AC, TRANSIENT
TIME	double	<none>	Current analysis time (same as T(0))
T(n)	double	index	Current and previous analysis times (T(0) = TIME = current analysis time,T(1) = previous analysis time)
RAD_FREQ	double	<none>	Current analysis frequency in radians per second
TEMPERATURE	double	<none>	Current analysis temperature

Parameter

Name	Type	Args	Description
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

Port Data

Name	Type	Args	Description
PORT_NULL	Mif_Boolean_t	name	Has this port been specified as unconnected?
PORT_SIZE	int	name	Size of port vector
LOAD	double	name[i]	The digital load value placed on a port by this model.
TOTAL_LOAD	double	name[i]	The total of all loads on the node attached to this event-driven port.

Input

Name	Type	Args	Description
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, HIIMPEDANCE, or UNDETERMINED
INPUT_TYPE	char *	name[i]	The port type of the input

Output

Name	Type	Args	Description
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, HIIMPEDANCE, or UNDETERMINED
OUTPUT_TYPE	char *	name[i]	The port type of the output

Miscellaneous

Name	Type	Args	Description
AC_GAIN	Complex_t	y[i],x[i]	AC gain of output y with respect to input x
deltaTemp	double	void	returns TEMP-TNOM
gMIN	double	void	returns the value of gmin
imagFreq	double	void	returns the imaginary value of the frequency axis
MESSAGE	char *	name[i]	A message output by a model on an event-driven node.
PARTIAL	double	y[i],x[i]	Partial derivative of output y with respect to input x
postQuit	void	void	request IsSpice4 to Quit
realFreq	double	void	returns the real value of the frequency axis
STATIC_VAR	CD	name	Value of a static variable
STATIC_VAR_SIZE	int	name	Size of static var (currently unused).

User Defined Nodes

Name	Type	Description
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
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
MALLOCED_PTR	void *	Assign pointer to alloced structure to this macro
OUTPUT_STRUCT_PTR	void *	A pointer to a structure of the defined type
STRUCT_MEMBER_ID	char *	A string naming some part of the structure
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

Memory (State and Static)

Name	Type	Args	Description
CALLOC	void*	size_t,size_t	improved calloc function
cm_event_alloc		void*	allocates memory for events and return a temporary pointer
cm_event_get_ptr		void*	returns a pointer to memory allocated by cm_event_alloc
FREE	void	&void	improved free function
getVar	double	int	returns a value stored in memory
getVarPtr	void*	int	returns a pointer to a value stored in memory
lastSTATE	double	int	returns the previous state value
lastSTATEptr	void*	int	returns a pointer to the previous state value
MALLOC	void*	size_t	improved malloc function
newState	int	int	allocates memory for state variables
newVar	int	int	allocates memory for static variables
REALLOC	void*	void*,size_t	improved realloc function
thisSTATE	double	int	returns the current state value
thisSTATEptr	void*	int	returns a pointer to the current state value

SPICE3 Conversion

Name	Type	Description
isBYPASS	int	detects the bypass option
isINIT	int	detects initialization
isMODEAC	int	detects the AC analysis
isMODEINITFIX	int	detects processing of OFF keyword
isMODEINITJCT	int	detects processing of junction voltages
isMODEINITPRED	int	detects the first iteration of each timepoint
isMODEINITSMSIG	int	detects small signal intilization
isMODEINITTRAN	int	detects transient intitialization
isMODETRAN	int	detects a transient analysis
isMODETRANOP	int	detects transient OP
isMODEUIC	int	detects UIC processing

Smoothing

Name	Type	Description
cm_smooth_corner	void	smooths the transition between two slopes into a quadratic (parabolic) curve
cm_smooth_discontinuity	void	smooths the transition between two values using an x^2 function
cm_smooth_pwl	double	smooths a pwl curve described by x and y input arrays

Integration, Differentiation and Convergence

Name	Type	Description
cm_analog_converge	int	performs a convergence test on the argument
cm_analog_not_converged	void	restricts final iteration
cm_analog_auto_partial	void	allows IsSpice4 to determine the partial
cm_ramp_factor	double	returns the current rampfactor value
stateIntegrate	double*	performs integration, or differentiation, depending on its implementation.

Message Handling

Name	Type	Description
cm_message_get_errmsg	char *	returns an error message from a function
cm_message_send	int	prints a message

Breakpoint Handling

Name	Type	Description
cm_analog_set_perm_bkpt	int	sets a perminant breakpoint
cm_analog_set_temp_bkpt	int	sets a temporary breakpoint
cm_event_queue	int	queues an event

Special Purpose

Name	Type	Description
cm_climit_fcn	void	a controlled limiting function
cm_netlist_get_c	double	the capacitance connected to a port
cm_netlist_get_l	double	the inductance connected to a port

Circuit Related

Name	Type	Description
cktABSTOL	double	returns ABSTOL
cktNOMTEMP	double	returns TNOM
cktRELTOL	double	returns RELTOL
cktTEMP	double	returns TEMP
cktVOLT_TOL	double	returns VNTOL

Complex Math

Name	Type	Description
cm_complex_add	Complex_t	adds complex numbers
cm_complex_div	Complex_t	divides complex numbers
cm_complex_mult	Complex_t	multiplies complex numbers
cm_complex_set	Complex_t	creates a Complex_t structure from two doubles
cm_complex_sub	Complex_t	subtracts complex numbers

User-Defined Node

Name	Type	Description
udn_XXX_create	void	creates the UDN data struct
udn_XXX_initialize	void	initializes the UDN data
udn_XXX_copy	void	copies UDNs
udn_XXX_compare	void	compares UDNs
udn_XXX_dismantle	void	used to free internally allocated memory
udn_XXX_invert	void	process the inversion key “~”
udn_XXX_resolve	void	deterines the proper output from a node with more than 1 connection
udn_XXX_plot_val	void	plots output
udn_XXX_print_val	void	prints output
udn_XXX_ipc_val	void	sends data to the ipc channel

The XXX in the UDN function name is replaced with the name of the UDN. See the description of UDNs in the Code Model Development chapter.

AC_GAIN

```
Complex_t AC_GAIN(output port, input port)
Complex_t AC_GAIN(output port[i], input port)
Complex_t AC_GAIN(output port, input port[j])
Complex_t AC_GAIN(output port[i], input port[j])
```

```
input port  - represents the name of a non-vector input
              port
input port[i]- represents the i'th port in a vector
              input port
output port - represents the name of the non-vector
              output port
output port[i] - represents the j'th port in a vector
              output port
```

This macro resolves to an lvalue that accepts the complex AC gain output to input. The type is always a structure (“Complex_t”) defined in the standard code model header file:

```
typedef struct Complex_t{
     double real; // Real part of a complex number
     double imag; // Imaginary part of a complex number
}Complex_t;
```

Return Value
Returns the ac gain as a Complex_t data structure. The value returned will reflect the ac gain set for the current pass through the code model.

Example
```
/* 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;
```

ANALYSIS

`int ANALYSIS`

ANALYSIS is used to determine the analysis mode. The value returned by this macro will reflect the analysis type being run by IsSpice4.

Return Type
The return value is an enumerated integer that takes values of DC, AC, or TRANSIENT.

Example
```
/* Determining analysis type */
     if(ANALYSIS == MIF_AC)
     // Perform AC analysis-dependent operations here
```

ARGS

`Mif_Private_t ARGS`

ARGS() is passed in the argument list of every code model to provide a way of referencing each model to all of the remaining macro values. It must be present, and the only argument, in the argument list of every code model.

Return Type
This returns a Mif_Private_t structure. This is a private structure. The information is not of general interest and should not be used for debugging.

CALL_TYPE

```
int CALL_TYPE
```

CALL_TYPE() is used to determine whether the analog simulator or the event-driven simulator is being called. This will, in general, only be of value to a hybrid model such as the adc_bridge or the dac_bridge.

Return Type
The return value is an enumerated value, EVENT or ANALOG, specifying the simulator type.

CALLOC

```
void* CALLOC(size_t num,size_t size)
```

This macro provides improved performance over the standard calloc function. The syntax described by the standard calloc function applies to this macro. Always use this macro instead of the calloc function.

cktABSTOL

```
double cktABSTOL
```

This macro is used to retrieve the ABSTOL .OPTIONS parameter for the current simulation. See the IsSpice4 User's Guide for mode details about the ABSTOL parameter.

Return Value
A double equal to the circuit ABSTOL.

cktNOMTEMP

```
double cktNOMTEMP
```

This macro is used to retrieve the nominal operating temperature for the circuit being simulated. This is equivalent to the .OPTIONS parameter TNOM. See the IsSpice4 User's Guide for on the TNOM parameter.

Return Value
A double representing the TNOM .OPTIONS parameter.

cktRELTOL

```
double cktRELTOL
```

This macro is used to retrieve the .OPTIONS parameter RELTOL for the current simulation. See the IsSpice4 User's Guide for mode details about the RELTOL parameter.

Return Value
A double representing the RELTOL .OPTIONS parameter.

cktTEMP

```
double cktTEMP
```

This macro is used to retrieve the .OPTIONS parameter TEMP for the current simulation. See the IsSpice4 User's Guide for mode details about the TEMP parameter.

Return Value
A double representing the TEMP .OPTIONS parameter.

cktVOLT_TOL

```
double cktVOLT_TOL
```

This macro is used to retrieve the .OPTIONS parameter VNTOL for the current simulation. See the ISSPICE4 User's Guide for mode details about the VNTOL parameter.

Return Value
A double representing the VNTOL .OPTIONS parameter.

cm_analog_auto_partial

```
void cm_analog_auto_partial()
```

This function can be called at the end of a code model in lieu of calculating the values of partial derivatives explicitly. When using this function, no values should be assigned to the PARTIAL macro. Automatic calculation of partial derivatives can save considerable time designing and coding a model since manual computation of partial derivatives can become very complex and error-prone. However, automatic evaluation will increase simulation run time significantly. Automatic calculation of partials causes the model to be called n additional times (for a model with n-inputs). Each input is varied by a small amount (1e-6 for voltage inputs and 1e-12 for current inputs) and the values of the partial derivatives are approximated by the simulator through divided difference calculations.

cm_analog_converge

```
int   cm_analog_converge(state)

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

This function accepts the address of a state variable that was previously allocated using newState(). The function itself serves to notify the simulator that for each timestep taken, the variable specified by the argument must be iterated upon until it converges.

Return Value
Returns 1 if the function fails. Otherwise this function returns 0.

cm_analog_not_converged

```
void  cm_analog_not_converged()
```

This function should be called by an analog model whenever it performs internal limiting of one or more of its inputs to aid in reaching convergence. It 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 this function. This function should be used for code models with multiple inputs expected to change abruptly.

cm_ramp_factor

```
double  cm_ramp_factor()
```

This function indicates whether a ramp time value, requested in the IsSpice4 netlist (with the use of .OPTION RAMPTIME= <duration>), has elapsed. If the RAMPTIME option is used, the function returns a 0.0 during the DC operating point solution and a value which is between 0.0 and 1.0 during the ramp. A value of 1.0 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.

Return Value
A double representing the multiplication factor for ramping. The return value is 1.0 if the ramping is finished or not used, between 0.0 and 1.0 while ramping, and 0.0 for the DC analysis.

cm_analog_set_perm_bkpt

```
int  cm_analog_set_perm_bkpt(time)

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

This function is used to post time points in the analog simulator algorithm. It forces the simulator to choose the argument, a time value, as a breakpoint. The simulator may choose the argument as the next timepoint or a value less than the argument, but not greater, regardless of how many timepoints pass before the breakpoint is reached. The timepoint set with this function will not be removed from the simulator breakpoint list. 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)).

Return Value
Returns 1 if the function fails. Otherwise this function returns 0.

cm_analog_set_temp_bkpt

```
int  cm_analog_set_temp_bkpt(time)

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

This function is used in a similar manner as the cm_analog_set_perm_brkpt, except the time set by this function is not permenently added to the internal breakpoint list. The breakpoint established by this function is removed as soon as a new timepoint is accepted. 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 requested at 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)).

Return Value
Returns 1 if the function fails. Otherwise this function returns 0.

cm_climit_fcn

```
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, //Delta from control to limit value
double upper_delta, //Delta from control to limit value
double limit_range, // The limiting range
double gain,        // The gain from input to output
int percent,       // TRUE = absolute FALSE = fractional
double *out_final,   // The output value
double *pout_pin_final,  // partial of output wrt input
```

```
double  *pout_pcntl_lower_final,// The partial of
             // the output wrt lower control input
double  *pout_pcntl_upper_final)// The partial of
             // the output wrt upper control input
```

This is a controlled limiter function. A single input, single output function similar to a gain block. However, the output of this function is restricted to the range specified by the cntl_lower and cntl_upper limits.

The limit range is the value BELOW THE CNTLUPPER LIMIT AND ABOVE THE CNTLLOWER LIMIT at which smoothing of the output begins (minimum positive value difference must exist between the CNTLUPPER input and the CNTLLOWER input at all times). The limit_range represents the delta WITH RESPECT TO THE OUTPUT LEVEL at which smoothing occurs. Thus, for an gain of 2.0 and 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 cntl_lower and cntl_upper are checked to make sure 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 - upper_delta - limit_range) and (cntl_lower + lower_delta + limit_range) and must be greater than zero. When limit_range is specified as a fractional value, it is calculated as a fraction of the difference between cntl_upper and cntl_lower. Still, the potential exists for too great a limitrange value to be specified for proper operation, in which case an error message is generated.

Return Value

The output as a pointer to a double and the partials with respect to the lower and upper limits.

cm_complex_add

```
Complex_t cm_complex_add(x, y)

Complex_t x; //   (x.real+jx.imag)
Complex_t y; // The second operand of x + y
```

Takes two complex values, Complex_t structures, as inputs and adds them. See cm_complex_set for a description of the Complex_t data structure.

Return Value
A Complex_t structure containing the results of the addition.

cm_complex_div

```
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
```

Takes two complex values, Complex_t structures, and divides them. See cm_complex_set for a description of the Complex_t data structure.

Return Value
A Complex_t structure containing the results of the division.

cm_complex_mult

```
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
```

Takes two complex values, Complex_t structures, as inputs and multiplies them. See cm_complex_set for a description of the Complex_t data structure.

Return Value
A Complex_t structure containing the results of the multiplication.

cm_complex_set

```
Complex_t cm_complex_set(real_part, imag_part)

double real_part; // Real part of a complex number
double imag_part; // Imaginary part of a complex number
```

Takes two doubles and converts them to a Complex_t data structure. The first double is taken as the real part and the second is taken as the imaginary part of the resulting complex value.

Return Value
A Complex_t data structure defined as;

```
typedef Mif_Complex_t Complex_t

struct{
     double real;
     double imag;
}Mif_Complex_t
```

cm_complex_sub

```
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
```

Takes two complex values, Complex_t structures, as inputs and subtracts them. See cm_complex_set for a description of the Complex_t data structure.

Return Value
A Complex_t structure containing the results of the subtraction.

cm_event_alloc

```
void *cm_event_alloc(tag, size)

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

This function allocates storage space for event-driven state information. The storage space is not static. Like the T(n) API call, the information in this storage represents a vector of two values which rotate with each accepted IsSpice4 timepoint evaluation. The first location is the current state value. The second location is the previous state value. 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 and easily managed. The “size” parameter specifies the size in bytes of the storage (computed by the C language “sizeof()” operator).

Return Value
A NULL pointer is returned if this function fails. Otherwise this function returns a temporary void pointer that can be cast to the appropriate type through an assignment line in the code model definition file. The pointer should not be used if another call to cm_event_alloc is used. The cm_event_get_ptr API call should be used to retrieve a pointer to the allocated memory.

cm_event_get_ptr

```
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
```

This function retrieves the pointer to previously allocated rotational storage space allocated using cm_event_alloc. The functions take the integer “tag” used to allocate the space, and an integer from 0 to 1 representing the timepoint for the desired state variable. For example, an argument of 0 will retrieve the address of storage for the current timepoint. An argument of 1

will retrieve the address of storage for the last accepted timepoint. 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 storage value is placed in the new storage location. These features allow you to know which piece of state information is being used within the model function at each timepoint.

Return Value
This function returns a void pointer that should be cast to the appropriate type. If the function fails NULL is returned.

cm_event_queue

```
int   cm_event_queue(time)

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

This function is similar to cm_analog_set_perm_bkpt(), but functions with event-driven models. This function queues an event at the specified time. All other details applicable to cm_analog_set_perm_bkpt() apply to this function as well.

Return Value
Returns 1 if the function fails. Otherwise this function returns 0.

cm_message_get_errmsg

```
char  *cm_message_get_errmsg()
```

This function is used with other functions to provide error handling. More specifically, whenever a library function which returns type “int” is executed from a model, this function will return an integer value, n. If this value is not equal to zero (0), an error condition has occurred. Functions which return pointers will return a NULL value if an error has occurred. This

function is used to obtain a pointer to an error message. This is passed to the simulator interface through the cm_message_send() function.

Return Value
A pointer to a character string containing the proper error message for a function.

Example
```
err = cm_analog_set_perm_bkpt(TIME);
if (err) \{
     cm_message_send(cm_message_get_errmsg());
\}
else  \{...
```

cm_message_send

```
int cm_message_send(char *msg)

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

This function sends messages to the .ERR file. If the message includes the string “Error:” the simulation will terminate if sent on the initial pass through the code model. A message will be sent every time this function is executed. You will have to provide a condition that limits the number of times this function is executed within your code model. The easiest way to accomplish this is to use newVar() to allocate a variable and use it as a flag to conditionally execute cm_message_send. If the message is sent more than once a “\n” must be placed at the end of the string.

Return Value
Returns 1 if the function fails. Otherwise this function returns 0.

Example

```
...
     if(error)
       cm_message_send("Error: Parameter capvalue create
   divide by zero\n");
...
```

cm_netlist_get_c

double cm_netlist_get_c()

This function searchs the analog circuitry to which the input is connected, and totals the capacitance found at that node. The function, as currently written, assumes tthe model has only one single-ended analog input port.

Return Value
This function returns 0 if an error is encountered. Otherwise a double is returned representing the total capacitance connected to the code models' port.

cm_netlist_get_l

double cm_netlist_get_l()

This function searchs the analog circuitry to which the input is connected, and totals the inductance found at that node. The function, as currently written, assumes the model has only one single-ended analog input port.

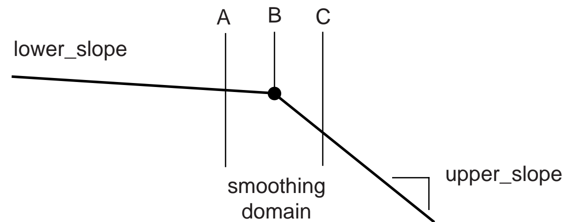
Return Value
This function returns 0 if an error is encountered. Otherwise a double is returned representing the total inductance connected to the code models' port.

cm_smooth_corner

```
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 smoother Y output
double *dy_dx     // The partial of Y wrt X
```

This function automates smoothing between two arbitrarily-sloped lines that meet at a point as shown below.



You specify the center point B (x_center, y_center), plus a smoothing domain (x-valued delta) above and below x_center that defines a smoothing region about the center point. The slopes of the meeting lines outside of this smoothing region are also 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 troughout the smoothing domain (from points A to C). This function helps to automate the smoothing of piecewise-linear functions which, in turn, aids in convergence.

Return Value

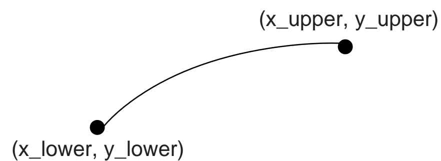
This function returns the smoothed Y output (*y_output) as a pointer to a double along with the partial (*dy_dx) as a pointer to a double.

cm_smooth_discontinuity

```
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
```

This function allows you to obtain a smoothly-transitioning output (*y_output) that varies between two independent points (x_lower, y_lower) to (x_upper, y_upper) as shown below.



The function accepts an x value and produces the y and dy/dx fofr an X^2 function between the points specified.

Return Value

This function returns the smoothed Y output (y_output) as a pointer to a double along with the partial (*dy_dx) as a pointer to a double.

cm_smooth_pwl

```
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 out wrt the in
```

This function duplicates much of the functionality of the predefined PWL (Table Model) 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 (see the IsSpice4 User's Guide). Note that the output value is the function's returned value.

Return Value
This function returns a double representing the Y output interpolated, or extrapolated, from the x input given the PWL tansfer function determined by the arrays *y and *x..

deltaTemp

```
double deltaTemp
```

This function returns a double representing the difference between TNOM and TEMP.

EQUAL

Mif_Boolean_tEQUAL

This is a UDN macro. An assignment should be made to this macro according to the results of structure comparison within the compare function defined in the UDN definition file.

Example

```
void  udn_real_compare(COMPARE_ARGS)
{
double *real_struct1 = STRUCT_PTR_1
double *real_struct2 = STRUCT_PTR_2

// Compare structures
if((*real_struct1) == (*real_struct2))
     EQUAL=TRUE;
else
     EQUAL = FALSE;
}
```

FREE

void FREE(&void)

This macro provides improved performance over the standard free function. The syntax described by the standard free function applies to this macro. Always use this macro instead of the free function.

getVar

```
double getVar(index)

int index - an integer value that is the index into the
            allocated storage.
```

This command is used to retrieve the value of a variable located at the index specified by the argument.

Return Value
This macro returns the contents of the variable allocated by newVar().

Example

```
// index to VAR's
#define DENCOEFFICIENT 0
#define NUMCOEFFICIENT den_size + 1
#define GAIN den_size+num_size + 2
#define NUMVARS den_size+num_size + 3

    double gainPtr;        // pointer to the gain
    double denCoefficient; // pointer to array
    double numCoefficient; // pointer to array
    double *integrator; // outputs of the integrators
    double *integrator_in; //previous integrator outputs
    double *lastintegrator_in;

    if (INIT==1) { //First pass...allocate storage ...
      newVar(NUMVARS); // initialize memory
      newState(2 * den_size+1); // initialize memory
    }
    // set pointers to alloced states and vars
    denCoefficient = getVar(DENCOEFFICIENT);
    numCoefficient = getVar(NUMCOEFFICIENT);
    gainPtr = getVar(GAIN);

    integrator = thisSTATEptr(0);
    integrator_in = thisSTATEptr(den_size);
    lastintegrator_in = lastSTATEptr(den_size);
```

getVarPtr

```
double* getVar(index)

int index - an integer value that is the index into the
            allocated storage.
```

This command is used to retrieve a pointer to the value of a variable located at the index specified by the argument.

Example
See the example for getVar.

gMIN

```
double gMIN
```

This macro is used to retrieve the .OPTIONS parameter GMIN for the current simulation. See the ISSPICE4 User's Guide for mode details about the GMIN parameter.

Return Value
A double representing the GMIN .OPTIONS parameter.

imagFreq

```
double imagFreq
```

This retrieves the imaginary component of the frequency axis for the AC analysis. It represents $j\omega / 2\pi$. In order for your code model to function with the PZ analysis it must process both the real and imaginary frequency components. The capacitor code model example demonstrates the use of this function.

Return Value
A double representing the imaginary frequency component.

INIT

```
Boolean_t INIT
```

INIT is used to determine if the code model is being initialized. The first time IsSpice4 access a code model this function returns TRUE, 1. This is quite useful for initializing memory storage.

Return Type
This macro returns TRUE, 1, if IsSpice4 is passing through the code model for the first time, otherwise this value is FALSE, 0.

Example
```
/* 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;
```

INPUT

```
double INPUT(input port)
double INPUT(input port[i])
void* INPUT(input port)

input port  - represents the name of the input port
input port[i]- represents the i'th port in the vector
               input port
```

This macro is used to obtain the current input value for a code model from the port specified in the argument. The values returned are used to process the outputs for the code model.

Return Type
The return value depends on the type of node being referened by the argument to this macro. The proer return type is determined by the entries in the IFS file.

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

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

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

Example
```
// 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;
```

INPUT_STATE

```
int INPUT_STATE(input port)
int INPUT_STATE(input port[i])

input port  - represents the name of the non-vector
              input port
input port[i]- represents the i'th port in the vector
               input port
```

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

INPUT_STRENGTH

```
int INPUT_STRENGTH(input port)
int INPUT_STRENGTH(input port[i])

input port  - represents the name of the non-vector
              input port
input port[i]- represents the i'th port in the vector
               input port
```

This function is used to determine the strength with which a digital input node, specified by the argument, is being driven. The strength value is determined by a resolution algorithm which looks at all outputs connected to a node and determines a final strength.

Return Value
Returns an integer represented by the Digital_Strength_t enumeration;

```
typedef enum{
     STRONG,
     RESISTIVE,
     HI_IMPEDANCE,
     UNDETERMINED,
} Digital_Strength_t;
```

INPUT_STRUCT_PTR

```
void* INPUT_STRUCT_PTR
```

This is a UDN macro used to retrieve a pointer to the input structure of the defined type.

Return Value
A pointer to the input structure of the defined type.

Example

```
void  udn_real_copy(COPY_ARGS)
{
double *real_from_struct = INPUT_STRUCT_PTR;
double *real_to_struct = OUTPUT_STRUCT_PTR;

// Copy the structure
*real_to_struct = *real_from_struct;

}
```

INPUT_STRUCT_PTR_ARRAY

`void**  INPUT_STRUCT_PTR_ARRAY`

This is a UDN macro used to retreive an array of pointers to the inputs of a node. This would represent all connections to a node of the defined type.

Return Value
An array of pointers.

Example

```
void  udn_real_resolve(RESOLVE_ARGS)
{
double **array = (double**) INPUT_STRUCT_PTR_ARRAY;
double *out = OUTPUT_STRUCT_ARRAY;
int struct_size = INPUT_STRUCT_PTR_ARRAY_SIZE;

double sum;
int i;

// Sum the input values
for(i=o, sum=0;i < struct_size;i++)
     sum += *(array[i]);

// Assign the result
*out = sum;

}
```

INPUT_STRUCT_PTR_ARRAY_SIZE

```
int  INPUT_STRUCT_PTR_ARRAY_SIZE
```

This is a UDN macro used to retrieve the size of the INPUT_STRUCT_PTR_ARRAY. This represents the number of connections at a particular node.

Return Value
This macro returns an integer representing the size of the INPUT_STRUCT_PTR_ARRAY.

Example
See the example for INPUT_STRUCT_PTR_ARRAY.

isBYPASS

```
int isBYPASS
```

This function detects the point at which IsSpice4 processes the BYPASS option.

Return Value
An integer value of 1 is returned if the simulation is in the BYPASS, mode. Otherwise, a 0 is returned.

isINIT

```
int isINIT
```

This macro is used to determine the mode of the current simulation. This function is equivalent to the INIT macro. The INIT macro should be used instead of this macro.

Return Value
An integer value of 1 is returned if the simulation is in the initialization, INIT, mode. Otherwise, a 0 is returned.

isMODEAC

```
int isMODEAC
```

See the Laplace (s_xfer) example.

This macro is used to determine the mode of the current simulation. This function is equivalent to comparing the return of the ANALYSIS macro with MIF_AC. The later should be used instead of this macro.

Return Value
An integer value of 1 is returned if the simulation is in the AC, MIF_AC, mode. Otherwise, a 0 is returned.

isMODEINITFIX

```
int isMODEINITFIX
```

This macro detects when the OFF keyword is being processed within ISSPICE4.

Return Value
An integer value of 1 is returned if the simulation is in the INITFIX mode. Otherwise, a 0 is returned.

isMODEINITJCT

```
int isMODEINITJCT
```

This macro detects the first iteration of the circuit. This is when the junction voltages for all devices are set to something reasonable. When there is no advance knowledge, such initial conditions, the best results have been obtained by initializing the junctions to Vto or its equivalent. This allows the device junction voltage to move in either direction in a single iteration.

Return Value
An integer value of 1 is returned if the simulation is in the INITJCT mode. Otherwise, a 0 is returned.

isMODEINITPRED

```
int isMODEINITPRED
```

An if statement using this macro is a good location for breakpoints.

This macro detects the first iteration at every timepoint. This is the point at which terminal voltages are predicted. An if statement using this macro is a good location for breakpoints. This will allow a code model to be debugged without having to step through all iterations. This is also a good place to keep track of iteration counts. A static integer can be used to count the iterations and set to 0 inside an if statement using this macro. This will allow convergence problems to be pin-pointed.

Return Value
An integer value of 1 is returned if the simulation is in the INITPRED mode. Otherwise, a 0 is returned.

isMODEINITSMSIG

```
int isMODEINITSMSIG
```

This macro detects when the intialization for the small signal AC analysis is processed. This mode can be used to store special values needed for the small signal analysis. Only those values not already available should be computed at this time.

Return Value
An integer value of 1 is returned if the simulation is in the INITSMSIG mode. Otherwise, a 0 is returned.

isMODEINITTRAN

```
int isMODEINITTRAN
```

This macro detects the first iteration of the first timepoint after the DC solution. Historically, this mode would be used to set the previous state variable for the preditor-corrector algorithm to

predict correctly. However, the predictor algorithm generates data to fill ***non-existent time, previous, timepoints***. This is always equivalent to isMODEINITPRED. (see isMODEINITPRED).

Return Value
An integer value of 1 is returned if the simulation is in the INITJCT mode. Otherwise, a 0 is returned.

isMODETRAN

```
int isMODETRAN
```

This macro is used to determine the mode of the current simulation. This function is equivalent to comparing the return of the ANALYSIS macro with MIF_TRAN. The later should be used instead of this macro.

Return Value
An integer value of 1 is returnedif the simulation is in the transient analysis mode. Otherwise, a 0 is returned.

isMODETRANOP

```
int isMODETRANOP
```

This macro is used to determine if the simulation is operating in the transient analysis operating point mode.

Return Value
An integer value of 1 is returned if the simulation is in the transient analysis mode. Otherwise, a 0 is returned.

isMODEUIC

```
int isMODEUIC
```

This macro can be used to determine if the .TRAN UIC keyword has been set. If so, the code model should process all necessary initial conditions.

Return Value
An integer value of 1 is returned if the UIC keyword has been used for the simulation. Otherwise, a 0 is returned.

lastSTATE

```
double lastSTATE(index)

int index - an integer value that corresponds to the
            state variable allocated by newState()
```

This macro is used to retreive the state value for the last **(previous) time point**. The index refers to the array location that was allocated by the newState() API call.

Return Value
The state value for the previous timepoint. IsSpice4 will automatically mark the state variables in time so that the previous state is always returned.

lastSTATEptr

```
double* lastSTATEptr(index)

int index - an integer value that corresponds to the
            state variable allocated by newState()
```

This macro is used to retreive a pointer to the state value for the last **(previous) time point**. The index refers to the array location that was allocated by the newState() API call.

Return Value
A pointer to the state value for the previous timepoint. IsSpice4 will automatically mark the state variables in time so that a pointer to the previous state is always returned.

Example

```
...
#define CURRENT 0
#define CHARGE 1
....
double *i_in, *i_last, *charge;

if(INIT)
     newState(2)

     i_in = this STATEptr(CURRENT);
     charge = thisSTATEptr(CHARGE);
     i_last = lastSTATEptr(CURRENT);
...
```

A 2 element array of pointers was allocated using newState. Element 0 was assigned to current (i_in and i_last) and element 1 was assigned to CHARGE.

LOAD

```
double LOAD(input port)
double LOAD(input port[i])

input port  - represents the name of the non-vector
              input port
input port[i]- represents the i'th port in the vector
               input port
```

This macro is used in a code 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.

Return Value
LOAD returns a double representing the capacitive load for the node specified.

MALLOCED_PTR

```
void* MALLOCED_PTR
```

This is a UDN macro used to allocate storage for the UDN's data structure. This is typically done once in the udn_xxx_create function.

Return Value
Returns a void pointer.

Example

```
void  udn_real_create(CREAT_ARGS)
{
     MALLOCED_PTR    =  MALLOC(sizeof(double));
}
```

MALLOC

```
void MALLOC(size_t size)
```

This macro provides improved performance over the standard malloc function. The syntax described by the standard malloc function applies to this macro. Always use this macro instead of the malloc function.

newState

```
int newState(count)

int count - the number of states to allocate
```

This macro is used to allocate an array of pointers that will be used for state variables. Each element of the array is a pointer

to two elements. The first element of the pointer is the state value for the current time point and the second element is the state at the previous time point. These values are marched in time by IsSpice4 so that they are always pointing to the current and previous state values. The thisSTATEptr and lastSTATEptr API calls are used to retrieve the pointers.

Return Value
Returns the number of states allocated.

Example

```
#define CURRENT 0
#define CHARGE 1
....
double *i_in, *i_last, *charge;

if(INIT)
     newState(2)

     i_in = this STATEptr(CURRENT);
     charge = thisSTATEptr(CHARGE);
     i_last = lastSTATEptr(CURRENT);
...
```

A 2 element array of pointers was allocated using newState. Element 0 was assigned to current (i_in and i_last) and element 1 was assigned to CHARGE.

newVar

```
void newVar(count)

count - The number of elements to allocate.
```

This macro allocates storage for parameter values. The count argument specifies the number of elements to allocate. The getVarPtr macro is used to assign the pointer. This command is used during initialization, INIT, of the code model to allocate one time storage.

Return Value
This function does not return a value.

Example
See the example for getVar.

OUTPUT

```
void* OUTOUT(output port)
void* OUTOUT(output port[i])
void* OUTOUT(output port)

output port - represents the name of the output port
output port[i] - represents the i'th port in the vector
                 output port
```

This macro is used to assign an output value to the named port. If the port is described as a vector port in the IFS file then the assignment will resolve to a pointer (The third description listed above). The type that this macro resolves to is detemined by the port type description in the IFS file.

Return Value
A void pointer that can be used to assign values to OUTPUT.

Example

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

OUTPUT_CHANGED

```
Boolean_t OUPUT_CHANGED(output port)
Boolean_t OUPUT_CHANGED(output port[i])

output port - represents the name of the non-vector
             output port
output port[i] - represents the i'th port in the vector
                 output port
```

This macro resolves to an lvalue and is used to inform the event driven simulation that the code model will post new outputs. If the macro is set TRUE (default value) an output state, strength and delay must be posted by the model during the call. If no output is generated by the pass the macro should be set to FALSE and no output state, strength or delay need be posted. This macro applies to a single output only. Therefore, if the model has a vector output port, the macro must be issued for each port in the vector port.

OUTPUT_DELAY

```
double OUTPUT_DELAY(output port)
double OUTPUT_DELAY(output port[i])

output port - represents the name of the non-vector
              output port
output port[i] - represents the i'th port in the vector
                 output port
```

This macro resolves to an lvalue that accepts a double representing the delay assigned to the event driven port named as the argument. 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 set to FALSE. A non-zero value must be assigned to OUTPUT_DELAY(). Assigning a value of zero (or a negative value) will cause an error.

Example

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

OUTPUT_STATE

```
int OUTPUT_STATE(output port)
int OUTPUT_STATE(output port[i])

output port - represents the name of the non-vector
              output port
output port[i] - represents the i'th port in the vector
                 output port
```

This macro resolves to an lvalue that accepts the state to be placed at the output port named as the argument. Valid values are defined by the Digital_State_t enumeration.

```
typedef enum{
     ZERO,
     ONE,
     UNKNOWN,
} Digital_State_t;
```

This is the normal way of posting an output state from a digital code model.

Example

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

OUTPUT_STRENGTH

```
int OUTPUT_STRENGTH(output port)
int OUTPUT_STRENGTH(output port[i])

output port - represents the name of the non-vector
              output port
output port[i] - represents the i'th port in the vector
                 output port
```

The macro resolves to an lvalue that accepts the strength to be assigned to the output port named as the argument. This macro does not accept the name of a vector port. Each element must

be assigned a strength individually. Possible values are any of the enumerated types listed below.

```
typedef enum{
     STRONG,
     RESISTIVE,
     HI_IMPEDANCE,
     UNDETERMINED,
} Digital_Strength_t;
```

OUTPUT_STRUCT_PTR

`void* OUTPUT_STRUCT_PTR`

This is a UDN macro that provides a pointer to the defined node structure for the output that results from resolving input structures. It is used with the udn_XXX_resolve function.

Return Value
A void pointer to the defined UDN data structure.

Example

```
void udn_real_resolve(RESOLVE_ARGS)
{
double **array = (double**) INPUT_STRUCT_PTR_ARRAY;
double *out = OUTPUT_STRUCT_ARRAY;
int struct_size = INPUT_STRUCT_PTR_ARRAY_SIZE;

double sum;
int i;

// Sum the input values
for(i=o, sum=0;i < struct_size;i++)
     sum += *(array[i]);

// Assign the result
*out = sum;

}
```

PARAM

```
CD  PARAM(parameter name)
CD  PARAM(parameter name[i])

parameter - The name of the model parameter as defined
            in the IFS file.
```

PARAM is used to obtain a model parameter value from a instance's .Model line. The argument to this macro is the name, as defined in the Interface Specification file, of the model parameter you wish to obtain.

Return Value
In the first form, shown above, the return value would be the value of the .Model parameter name. The type of value returned would depend on the definition of the model parameter in the IFS file. If the model parameter gain was defined as a real value then PARAM(gain) would return the real value asssigned to gain in the .Model line. In the second form, the return value would be the i'th element of the vector parameter name.

PARAM_SIZE

```
int  PARAM_SIZE(parameter)

parameter - The name of the model parameter as defined
            in the IFS file.
```

This macro is used to determine the number of values assigned to a vector parameter. Since the number of values for a vector parameter will vary, this macro should be used to determine a limit when looping through the values of a particular vector parameter.

This macro is undefined if the argument is a scalar parameter.

Return Value
The macro returns the number elements assigned to the parameter for the given instance.

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

PARAM_NULL

```
int  PARAM_NULL(parameter)

parameter - The name of the model parameter as defined
            in the IFS file.
```

This macro is used to determine if a parameter has been assigned a value.

Return Value
The macro returns TRUE, 1, if the parameter was not given a value. FLASE, 0, is returned if the parameter was given a value.

Example
```
if(ANALYSIS == DC)
     if(!PARAM_NULL(ic)  && isMODETRANOP){
       OUPUT(cap) = PARAM(ic);
     }
     PARTIAL(cap,cap) = 0.0;
}
```

PARTIAL

```
double PARTIAL(output port, input port)
double PARTIAL(output port[i], input port)
double PARTIAL(output port, input port[j])
double PARTIAL(output port[i], input port[j])

input port     - represents the name of a non-vector
                 input port
input port[i]  - represents the i'th port in a vector
                 input port
output port    - represents the name of the non-vector
                 output port
output port[i] - represents the j'th port in a vector
                 output port
```

This macro resolves to an lvalue that accepts the partial of the output with respect to the input. It is the responsibility of the code model to generate the partial of every output/input combination. The type is always double since partial derivatives are only defined for nodes with real valued quantities (i.e., analog nodes).

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

Example

```
//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;
```

PORT_SIZE

```
int PORT_SIZE(port name)

port name - represents the vector port, port name
```

This macro is used to determine the number of ports assigned to a vector port. Since the number of ports for a vector port will vary, this macro should be used to determine a limit when looping through the ports of a particular vector port.

This macro is undefined if the argument is a scalar parameter.

Return Value
This macro returns an integer representing the number of ports used for the named vector port.

Example

```
/* 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;
```

PORT_NULL

```
int PORT_NULL(input port)
int PORT_NULL(input port[i])

input port     - represents the name of the non-vector
                 input port
input port[i]  - represents the i'th port in the vector
                 input port
```

This macro is used to determine if a port has been connected in the ISSPICE4 netlist. A null, or unconnected, port is defined by placing the NULL keyword as the node name. This macro should be used to control the execution of node processing if NULL ports are allowed by a code model.

Return Value
The return value is TRUE, 1, if the port has been assigned NULL. The return is FALSE, 0, if the port has been assigned a connection.

Example
```
// 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;
```

postQuit

`void postQuit`

This macro is used to request that the simulation be terminated. This macro should be used in conjunction with error detection inside your code model.

RAD_FREQ

`double RAD_FREQ`

This macro is used to obtain the current AC analysis frequency in radians/second.

Return Type
Returns a double representing the current analysis frequency expressed in units of radians per second.

realFreq

`double realFreq`

This macro is used to process results for the real portion of the frequency axis of the AC analysis. This is a requirement if your code model is to operate correctly with the PZ analysis.

Return Value
Returns a double representing the real portion of the frequency axis for the AC analysis.

REALLOC

```
void REALLOC(void* block, size_t size)
```

This macro provides improved performance over the standard realloc function. The syntax described by the standard realloc function applies to this macro. Always use this macro instead of the realloc function.

stateIntegrate

```
void stateIntegrate(integrand,integal,partial,lastintegrand)

     double *integrand     - a pointer to the state variable
                             used as the integrand
     double *integral      - a pointer to the current and
                             returned value of the integral
     double *partial       - a pointer to the returned partial
     double *lastintegrand - a pointer to the last value of
                             the state variable representing
                             the integral
```

Performs an implicit integration of the form:

$$X_{n+1} = X_n + h_n \dot{X}_{n+1} - \Delta$$

that yields a relation between X_{n+1} and $\dot{X}_{n+1}$ at each time point. The last term, Δ, represents the local truncation error. This relation is combined with the circuit equations to produce a system of algebraic equations for each timepoint.

The simplified system of equations is solved iteratively for each timepoint, t_{n+1}, like the DC analysis. This essentially reduces the simulation to a series of N repetitive “quasi-dc” analyses where N is the number of timepoints.

The solution to the intergral is obtained by functional iteration. First, the solution x(n+1) is predicited by an explicict predictor,

$$x_{n+1}^{[0]} = x_n + h_n \dot{x}_n$$

and then corrected iteratively as,

$$x_{n+1}^{[k]} = x_n + h_n \dot{x}_{n+1}^{[k+1]}$$

where k is the iteration number.

The implicit integration method allows this function to perform integration or differentiation depending on the variable held constatnt.

Integration is performed when the integral is allowed to vary and the integrand is kept constant.

Example

```
...
double *i_in, *i_last, *charge, partial;
...
// initialize fast pointers
     i_in = thisSTATEptr(CURRENT);
     charge = thisSTATEptr(CHARGE);
     i_last  = lastSTATEptr(CURRENT);
...
     cur = *i_in = INPUT(cap);//initialize inp. current

     stateIntegrate(i_in,charge,&partial,i_last);

     *i_in = cur;                // reset to iterate

     OUTPUT(cap) = *charge * one_over_c;
                                 // output V = Q/C
     PARTIAL(cap,cap) = partial * one_over_c; // 1/C
...
```

Differentiation is performed when the integrand is allowed to vary and the integral is kept constant.

Example

```
...
double *current, *currentlast, *charge, dt;
...
// initialize fast pointers
     current = thisSTATEptr(CURRENT);
     charge = thisSTATEptr(CHARGE);
     currentlast  = lastSTATEptr(CURRENT);
...
     chrg = *charge = vcap * capvalue;

stateIntegrate(current,charge,&dt,currentlast);

     *charge = chrg; // reset to iterate

     OUTPUT(cap) = *current;
     PARTIAL(cap,cap) = (capvalue / dt); // C/dt
...
```

Return Value

This function returns pointers to the integral, integrand, and partial.

STATIC_VAR

```
typeof variable STATIC_VAR(variable)

typeof variable - The type given in the code models's
                  IFS file in the STATIC_VAR_TABLE
variable- The variable name given in the code model's
          IFS file in the STATIC_VAR_TABLE
```

This macro resolves to an lvalue that accepts the type of data specified in the Data_Type field of the code model's STATIC_VAR_TABLE. The argument is the variable name given in the Static_Var_Name field. This macro is used to establish static storage for variables that will be needed by the code model.

If variable is a defined as a pointer STATIC_VAR(variable) 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(*variable*).

Return Value
The returned value is defined by the code model's IFS file.

Example
```
/* assume freq is the Static_Var_Name and pointer is the
   Data_Type*/
STATIC_VAR(freq)  = MALLOC(NUM_NOTES*sizeof(double));
     freq = STATIC_VAR(freq);
```

STRUCT_MEMBER_ID

`char*  STRUCT_MEMBER_ID`

This is a UDN macro. It returns the member id for the data structure.

STRUCT_PTR

`void*  STRUCT_PTR`

This is a UDN macro used to retrieve a pointer to the structure of the defined data type.

Return Value
A void pointer the the defined data type.

Example
```
void  udn_real_initialize(INITIALIZE_ARGS)
{
double *real_struct = STRUCT_PTR

     *real_struct = 0.0;
}
```

STRUCT_PTR_1

void* STRUCT_PTR_1

This is a UDN macro used to retrieve a pointer to the structure of the defined node data type.

Return Value
A void pointer to the defined data type.

Example
```
void  udn_real_compare(COMPARE_ARGS)
{
double *real_struct1 = STRUCT_PTR_1
double *real_struct2 = STRUCT_PTR_2

// Compare structures
if((*real_struct1) == (*real_struct2))
     EQUAL=TRUE;
else
     EQUAL = FALSE;
}
```

STRUCT_PTR_2

void* STRUCT_PTR_2

This is a UDN macro used to retrieve a pointer to the structure of the defined node data type.

Return Value
A void pointer the the defined data type.

Example
```
void  udn_real_compare(COMPARE_ARGS)
{
double *real_struct1 = STRUCT_PTR_1
double *real_struct2 = STRUCT_PTR_2
```

```
// Compare structures
if((*real_struct1) == (*real_struct2))
     EQUAL=TRUE;
else
     EQUAL = FALSE;
}
```

thisSTATE

```
double  thisSTATE(index)

int index - an integer value that corresponds to the
            array index of the state variable allocation
            you wish to use
```

This macro is used to retreive the current state variable given by the argument index. The index refers to the array location that is associated with the desired state variable.

Return Value
The returned value is always the state value for the current timepoint. IsSpice4 will automatically mark the state variables in time so that the state for the current timepoint is always returned.

thisSTATEptr

```
void*  thisSTATEptr(index)

int index - an integer value that corresponds to the
            array index of the state variable allocation
            you wish to use
```

This macro is used to retreive a pointer to the current state variable given by the argument index. The index refers to the array location that is associated with the desired state variable.

Return Value
The returned value is always a pointer to the state value for the current timepoint. IsSpice4 will automatically mark the state variables in time so that the state at the current timepoint is always returned.

Example

```
#define CURRENT 0
#define CHARGE 1
....
double *i_in, *i_last, *charge;

if(INIT)
     newState(2)

     i_in = this STATEptr(CURRENT);
     charge = thisSTATEptr(CHARGE);
     i_last = lastSTATEptr(CURRENT);
...
```

A 2 element array of pointers was allocated using newState. Element 0 was assigned to current (i_in and i_last) and element 1 was assigned to CHARGE.

T( )

```
double T(int index)

index - An integer variable
```

This macro is used to obtain the time value for the index passed. An index of 0, T(0), represents the current time. An index of 1, T(1), represents the previous time point. Therefore, (T(0) - T(1)) represents the last timestep.

Return Type
A double representing the time value for the specified index.

TEMPERATURE

```
double TEMPERATURE
```

The macro is used to obtain the operating temperature set for the current sinulation.

Return Type
A double representing the current analysis temperature.

TIME

```
double TIME
```

Time is used to retrieve the current time point in a transient analysis.

Return Type
Returns a double representing the current time point.

TOTAL_LOAD

```
double TOTAL_LOAD(input port)
double TOTAL_LOAD(input port[i])

input port   - represents the name of the non-vector
               input port
input port[i]- represents the i'th port in the vector
               input port
```

The information returned by this function can be used by a code model, after the INIT pass, 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).

Return Value
This macro returns a double value representing the total capacitive load seen on the specified node.

udn_XXX_compare

```
void  udn_XXX_compare(COMPARE_ARGS)

COMPARE_ARGS - this argument resolves to the following;
     void *evt_struct_ptr_1
     void *evt_struct_ptr_2
     Mif_Boolean_t  *evt_equal
```

XXX is replaced with the UDN name. See Code Model Development for more details.

Use the STRUCT_PTR_1 and STRUCT_PTR_2 to retrieve pointer variables. These macros resolve to *evt_struct_ptr_1 and *evt_struct_ptr_2 respectively. Compare the two structures and assign either TRUE or FALSE to EQUAL. The equal macro resolves to a Mif_Boolean_t value. This is a Required Function.

Example

```
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;
}
```

udn_XXX_create

```
void  udn_XXX_create(CREATE_ARGS)

CREATE_ARGS - this argument resolves to the following;
     void **evnt_struct_ptr
```

Allocate space for the data structure defined for the User-Defined Node to pass data between models. Then assign the pointer created by the storage to MALLOCED_PTR.

MALLOCED_PTR resolves to a pointer to evt_struct_ptr. This is a Required Function.

Example

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

udn_XXX_dismantle

```
void  udn_XXX_dismantle(DISMANTLE_ARGS)

DISMANTLE_ARGS - this argument resolves to the following;
     void  *evt_struct_ptr
```

Use the STRUCT_PTR, which resolves to *evt_struct_ptr, to a assign 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 should be left null.

Example

```
void  udn_int_dismantle(DISMANTLE_ARGS)
{
    // free internally malloc'ed items
}
```

udn_XXX_copy

```
void  udn_XXX_copy(COPY_ARGS)

COPY_ARGS   - this argument resolves to the following;
     void  *evt_input_struct_ptr
     void  *evt_output_struct_ptr
```

Use the INPUT_STRUCT_PTR and OUTPUT_STRUCT_PTR macros, which resolve to *evt_input_struct_ptr and

*evt_output_struct_ptr, to assign pointer variables of the defined type and then copy the elements of the input structure to the output structure. This is a required function.

Example

```
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;
}
```

udn_XXX_initialize

```
void  udn_XXX_initialize(INITIALIZE_ARGS)

INITIALIZE_ARGS   - this argument resolves to;
      void *evt_struct_ptr
```

Use the STRUCT_PTR macro, which resolves to *evt_struct_ptr, to assign a pointer variable of defined type and then initialize the value of the structure. This is a required function.

Example

```
void  udn_int_initialize(INITIALIZE_ARGS)
{
     int  *int_struct = STRUCT_PTR;
     // Initialize to zero
     *int_struct = 0;
}
```

XXX is replaced with the UDN name. See Code Model Development for more details.

udn_XXX_invert

```
void  udn_XXX_invert(INVERT_ARGS)

INVERT_ARGS  - this argument resolves to;
     void *evt_struct_ptr
```

Use the STRUCT_PTR macro, which resolves to *evt_struct_ptr, to assign a pointer variable of the defined type, and then invert the logical value of the structure. This is an optional function. If you do not plan to support the inversion netlist key, “~”, then the function should be left empty. See the udn_XXX_dismantle function for an example of a blank function.

Example

```
void  udn_int_invert(INVERT_ARGS)
{
    int       *int_struct = STRUCT_PTR;
    // Invert the state
    *int_struct = -(*int_struct);
}
```

udn_XXX_ipc_val

This is a UDN Macro used to provide output for UDNs. This function is not currently supported. Node bridges should be constructed to translate the UDN data to analog. Output can then be obtained from standard IsSpice4 output statements. An empty entry should be placed in the UDN definition file.

Example

```
void  udn_real_ipc_val(IPC_VAL_ARGS)
{
}
```

udn_XXX_plot_val

This is a UDN Macro used to provide output for UDNs. This function is not currently supported. Node bridges should be constructed to translate the UDN data to analog. Output can then be obtained from standard IsSpice4 output statements. An empty entry should be placed in the UDN definition file.

Example

```
void  udn_real_plot_val(PLOT_VAL_ARGS)
{
}
```

udn_XXX_print_val

XXX is replaced with the UDN name. See Code Model Development for more details.

This is a UDN Macro used to provide output for UDNs. This function is not currently supported. Node bridges should be constructed to translate the UDN data to analog. Output can then be obtained from standard IsSpice4 output statements. An empty entry should be placed in the UDN definition file.

Example

```
void  udn_real_print_val(PRINT_VAL_ARGS)
{
}
```

udn_XXX_resolve

```
void  udn_XXX_resolve(RESOLVE_ARGS)

INVERT_ARGS  - this argument resolves to;
     int  evt_input_struct_ptr_array_size
     void  **evt_input_struct_ptr_array
     void  *evt_output_struct_ptr
```

Use the INPUT_STRUCT_PTR_ARRAY macro to assign 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 which resolves to evt_input_struct_ptr_array_size.

Use the OUTPUT_STRUCT_PTR macro, which resolves to *evt_output_struct_ptr, to assign a pointer variable of the defined type.

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

Example

```
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;
}
```

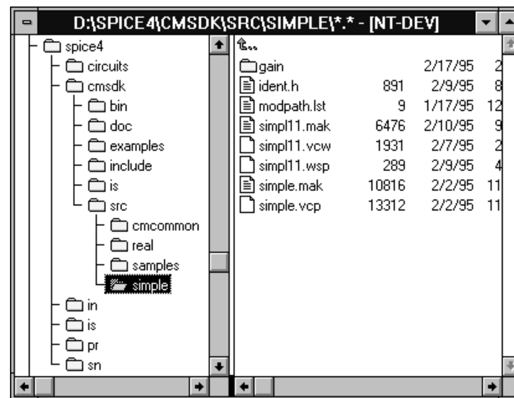
Examples

A Simple Gain Block

This example will use the files located in the Simple DLL directory found under the SRC directory of the CMSDK. This DLL project consists of a simple analog code model named Gain. To familiarize you with the process of creating a code model the following items will be covered;

- Examine DLL project file and support files.
- Examine the IFS and MOD files necessary to develop a code model.
- Compile a release version of the code model.
- Compile a debug version of the code model.
- Set a break point and examine a variable.

The DLL Project
The project file for this example is called Simple.Mak. The file is located in the Simple subdirectory under SRC in your CMSDK directory structure.



The Simple subdirectory is termed the DLL directory. All files necessary to construct a code model DLL will be found in this directory or subdirectories below this directory.

DLL directories must have an ident.h file and a modpath.lst and/or a udnpath.lst file, as well as the Visual C++ project files. Subdirectories below the DLL directory are termed model, or UDN, directories. A model directory is where the .MOD and .IFS files describing a particular code model will be located. UDN directories will contain a standard C file describing the UDN.

- **Open the Simple project by locating the project file, Simple.Mak for Visual C++ 2.0 and Simp11.Mak for Visual C++ 1.1, in the File Manager and double clicking on it.**

- **Locate Ident.h in the project and open the file.**

This file contains information that sets license levels and an ID string in the DLL. Any string, up to 256 characters, can be used as an ID. This string will be used as part of the return value when the DLL is loaded.

- **Open the Modpath.lst file.**

This file contains a list of Model subdirectories that will be part of the DLL. Each Model directory should be placed on a separate line. The "\" character should be placed after each line except the last. It is used as a continuation line delimiter during preprocessing. This file is used by the preprocessing makefiles that convert the .Mod files into .C files to be used with Visual C++.

- **Close the Modpath.lst <u>without saving any changes</u>.**

Interface Specification File (.IFS)
The IFS file for the Simple project is found in the GAIN directory and is called ifspec.ifs.

- **Open the ifspec.ifs file located in the GAIN model directory.**

This file is quite simple and describes all the input and output ports and model parameters for the code model. The Name Table provides the name of the model and the C function name used to represent the model within ISSPICE4. The Port Table describes all the ports of the code model. The Parameter Table describes all the characteristics of the model parameters that can be accessed from the code model's .Model line.

A complete description of the IFS file can be found in the Code Model Developement Chapter.

This IFS file is preprocessed by the CMPP utility into a standard C file and two header files. The C file will be given the same name as the .MOD file with an "!" appended to the end. This file contains all the data structures necessary for interfacing the new code model. The header files are used by the main source file for the DLL (DLL_MAIN.C in the CMCOMMON directory). You will never have to alter or manipulate the header files or the main DLL source file. Please take a moment to explore this file and the description in the Code Model Development Chapter. When you are finished;

- **Close the IFS file <u>without saving any changes</u>.**

Model definition File (.MOD)
The model definition file for the gain model is located in the GAIN model directory under the Simple DLL directory.

- **Open GAIN.MOD.**

```
void cm_gain(ARGS)
{
Complex_t ac_gain;
double mygain;

     mygain = PARAM(gain);

     if(ANALYSIS != MIF_AC) {
       OUTPUT(out) = PARAM(out_offset) + mygain *
               (INPUT(in) + PARAM(in_offset));
       PARTIAL(out,in) = PARAM(gain);
     }
     else {
       ac_gain.real = PARAM(gain);
       ac_gain.imag= 0.0;
       AC_GAIN(out,in) = ac_gain;
     }
}
```

The .MOD file describes the behavior of the code model. The function name, cm_gain, matches the name used in the IFS file. The ARGS argument expands into a private structure used by ISSPICE4. It will be visible through the debugger but the contents will not make much sense and will not be supported. All access to the simulator from the code model should be done through the API calls provided.

The code demonstrates the basic format of a code model. First all variables are declared. Next, the API call PARAM is used to get the "gain" .Model parameter which is used to initialize mygain. At this point, the analysis types are processed. The if-else block detects the transient and DC analysis and provides the output and partial derivative as required. If the ANALYSIS API call returns MIF_AC then the AC analysis response is processed. The AC_GAIN macro is used to provide ISSPICE4 with the correct gain output.

Compiling a Debug Version
Once the mode definition file has been created we can build a debug version of the DLL to test the new gain model.

- **Select the debug build option for your version of Visual C++.**

- **Select "Convert MOD to C" from the Visual C++ Tools menu.**

This runs the CMPP utility provided with the CMSDK. Once the preprocessing is complete;

- **Select Build from the Project menu of Visual C++.**

Important Note: The debug version of the DLL will be placed in the location chosen for output. For those using Visual C++ 2.0 the output DLL will be called Simple.DLL and located in the \IS directory of your ICAP/4Windows installation. For those using Visual C++ 1.1 the DLL will be named Simp11.DLL and placed in the Simple DLL directory.

Testing The DLL
To test the new DLL simply select Go from the Visual C++'s Debug menu. The default settings for this project will cause it to use the ISSPICE4 executable shipped with the kit and the DLL you just built. To view or change the defaults please refer to Appendix C or explore the options of this example project.

Setting A Breakpoint
To set a breakpoint;

- **Open Gain.Mod if it is not already open.**

- **Place the text cursor on the following line:**

```
mygain = PARAM(gain);
```

- **Select the breakpoint button from the Toolbar or Select Breakpoint from the Debug menu and select the Add button in the resulting dialog.**

- **After the breakpoint has been set select Go from the Debug menu.**

If you are running Visual C++ 1.1 the Find Source dialog requesting the location of the .MOD file will appear the first time a breakpoint is set. Append the GAIN directory to the path provided and select the OK button.

Examining Data
When the breakpoint is reached we can examine the data and determine the status of the code model. Any variable can be viewed using the QuickWatch command from the Debug menu.

- **Select the mygain variable.**

- **Select QuickWatch from the Debug menu or select the Watch button from the toolbar.**

The current value of mygain will be displayed. This value is the value of the model parameter as entered on the .Model line in the netlist. The PARAM API call does not evaluate to a value that can be watched. Neither does the argument to the PARAM API call. The argument, gain, is a the name of the parameter whose value is to be returned. It is not a variable.

Compiling a Release Version
To build a release version of the DLL;

- **Simply select the release option from the project settings and select Build from the Project menu.**

A Capacitor Model

This example will use the files located in the Samples DLL directory found under the SRC directory of the CMSDK. This DLL project consists of a several different code models. The two that will be investigated in this example are CAPH and CAPG. These code models use two different approaches to implement a capacitor. Both code models accept a linear temperature coefficient. The default for this coefficient is 0. Hence, by default the code models will produce results equivalent to the default capacitor found in ISSPICE4. These two code models will be used to cover the following topics;

- Input and Output Specifications
- The structure of an analog code model
- State variables and static instance variables.
- Integration and differentiation
- Calculating one time parameters

Input and Output Specifications
The differences in the two approaches to the capacitor start with the specification of the port type in the IFS file. CAPH uses a differential resistance, specified as "hd" in its IFS file, which makes it a current controlled voltage source. CAPG uses a differential conductance, specified as "gd" in its IFS file, which makes it a voltage controlled current source. These definitions directly effect the method by which the code model can produce the correct voltage-current relationships. The definitions also control the methods used to implement initial conditions. Since CAPH is a voltage controlled current source the only way to implement an initial voltage is through an iterative solution. CAPH provides an initial voltage directly.

Since CAPH has only a current as input the only way to generate the correct V-I characteristics is to integrate the current to obtain charge. The resulting charge is then divided by the calculated capacitance value to obtain the output voltage.

See the IsSPICE4 User's Guide for more details.

Since CAPG has only a voltage to start with the only way to generate the correct V-I characteristics is to multiply the voltage and the calculated capacitance value to obtain charge. Then differentiate the charge to obtain the output current.

Because the ports were defined as differential ports the input and output ports are implicitly the same. Throughout the code for the capacitor models you will see API calls like INPUT(cap) and OUTPUT(cap). The name of the port, for both CAPH and CAPG, is cap and it represents their input and output.

Analog Code Model Structure

The following explanations will use the CAPH code model as a foundation. When appropriate the CAPG code model will be explained to highlight significant differences. The structure of the two code models are essentially the same. You should print the .MOD and IFS files for each of the code models (CAPH and CAPG) so you can follow along.

The name of the code model procedure used in the code model definition file must match the name specified in the IFS file. For this example;

```
void cm_caph (ARGS)
```

All code models pass the parameter ARGS. As discussed in earlier chapters this is a private data structure. All information required within the code model can be obtained from the API calls provided. There will be no need to access the data passed in ARGS directly.

The next section of the code model declares all local variables. For this model we use a Complex_t data structure and various doubles and pointers to doubles.

```
Complex_t ac_gain; // structure to store the ac gain
double    partial, // partial returned from stateIntegrate
    dtmp,  // intermediate variable for ac gain calculation
    cur,   // stores current for transient iteration
    v_out, // used to calculate the input voltage
```

```
    *capvalue,   //stores the calculated capacitance value
    *one_over_c, // stores the reciprocal of the capacitance
    *r_shunt,    // shunt resistance for voltage calculation
    *charge,     // stores the charge state information
    *i_in, // stores the current state information
    *i_last;     // stores the last, previous, state
enum Vars{R_SHUNT, CAPVALUE, ONE_OVER_C};
    // enum for local Var pointers
```

The Complex_t structure is defined in the definition of AC_GAIN found in the API Calls chapter.

The Complex_t structure stores the real and imaginary portions of a complex number. This will be used to pass the calculated gain to the output of the capacitor for the AC analysis.

All doubles are used for intermediate calculations. These variables are not per-instance variables, they do not keep their value from iteration to iteration, and will have to be reset upon every iteration. The pointers to doubles are used to store per-instance variables and state variables. These variables are allocated during IsSpice4's INIT routine as shown below.

```
if(INIT) {
// allocate storage for state and local variables
     newState(2); // use 2 states - current and charge
     newVar(ONE_OVER_C+1);

// retrieve pointers to allocated local variables
     r_shunt = getVarPtr(R_SHUNT);
     capvalue = getVarPtr(CAPVALUE);
     one_over_c = getVarPtr(ONE_OVER_C);

// assign values to local variables
     *r_shunt = 1/gMIN;
     *capvalue =PARAM(c) * (1.0 + PARAM(TC) * deltaTemp);
     *one_over_c = 1 / *capvalue;
}
else{
// initialize local variable pointers
     r_shunt = getVarPtr(R_SHUNT);
     capvalue = getVarPtr(CAPVALUE);
     one_over_c = getVarPtr(ONE_OVER_C);
}
```

The newState API call is used to allocate pointers to storage for state variables. These state variables are used to store the states for every time point accepted by IsSpice4. The newVar API call establishes a per-instance static variables. Each of these API calls are documented in the API Calls chapter.

In order to simplify access to the per-instance variables an enumeration was declared (enum Vars). The enumerated values are use to reference the proper memory locations when calls to getVarPtr are made.

After allocating storage for the per-instance pointers the getVarPtr API call is used to assign the pointer to a local pointer to a double. These pointers to doubles are used to store the results of one-time parameter calculations. This method prevents the parameters from being calculated every iteration as would be the case if local variables were used. (See the capacitance calculation in CPAG.MOD for the latter.)

The r_shunt variable is used in CAPH to calculate a voltage output for the DC operating point when no ic is given. (V = IR)

The else clause of the INIT section is used to fetch the pointers to the per-instance static variables previously allocated, and calculated, during the initialization pass.

The next step is to assign state variables to the pointers allocated with newState. This is done with the following code.

```
// initialize fast pointers
     i_in = thisSTATEptr(CURRENT);
     charge = thisSTATEptr(CHARGE);
     i_last = lastSTATEptr(CURRENT);
```

The API call thisSTATEptr returns a pointer to the current state value, which is assigned to the appropriate pointers to doubles. The lastSTATEptr API call returns a pointer to the last, previous state, which is also assigned to a pointer to a double.

As with the enumeration used to reference the memory locations allocated with newVar, the arguments to the functions retrieving the state variable pointers are defined prior to the procedure call as:

#define CHARGE 1
#define CURRENT 0

This simply provides an easy way to reference the pointers when calling thisSTATEptr or lastSTATEptr. Either a definition of enumeration can be used.

Now that all parameters have been calculated and all state and per-instance variables have been initialized, the analysis types can be processed. This can be done through a switch-case arrangement or with if-else if-else clauses. The latter was used in this example.

DC Analysis

The DC analysis for the CAPH code model is detected by the following code.

```
if(ANALYSIS == MIF_DC){
// Calculate the DC analysis
     if(!PARAM_NULL(ic) && isMODETRANOP){
       OUTPUT(cap) = v_out = PARAM(ic);
       PARTIAL(cap,cap) = 0.0;
       *i_in = v_out/r_shunt;
     }
     else{
       *i_in = INPUT(cap);
       v_out = *i_in * *r_shunt
       OUTPUT(cap) = v_out;
       PARTIAL(cap,cap) = *r_shunt;
     }
     *charge = v_out * *capvalue;
     lastSTATE(CURRENT) = thisSTATE(CURRENT);
}
```

The ANALYSIS API call is used to indicate the analysis mode. The if statement uses MIF_DC and the return value of ANALYSIS to detect the DC analysis.

The IFS file for CAPH, and CAPG, include a model parameter, “ic”, to provide an initial condition. To ensure than the initial condition is set correctly, the PARAM_NULL API call is used to determine if the parameter was entered. The PARAM_NULL API call is combined with the isMODETRANOP API call in CAPH to apply the initial condition only during the transient operating point. When either of these conditions fail the input current is used to calculate the output and assigned to the output port using the OUTPUT API call. The equivalent DC analysis code for CAPG is given below;

```
if(!PARAM_NULL(ic) && shouldInit)
     vcap = PARAM(ic);
else
     vcap = INPUT(cap);

if(ANALYSIS == MIF_DC){
// Calculate the DC analysis
     *charge = vcap * capvalue;
     lastSTATE(CHARGE) = thisSTATE(CHARGE);
     OUTPUT(cap) = 0;
     PARTIAL(cap,cap) = 0.0;
}
```

Notice the additional code before the DC analysis. This is used to establish the initial condition, if requested, and to iterate to the correct DC voltage. Unlike CAPH, CAPG will only set the initial condition if the UIC option is present on the .TRAN line. This is equivalent to the standard SPICE capacitor. Also, because CAPG is essentially a current source the only way to set the initial voltage is through iteration.

As mentioned earlier, the port types for both CAPH and CAPG are differential. Therefore, the same port name will be used to reference the input and output port. This can be seen in the call to the, INPUT, OUTPUT, and PARTIAL API calls.

Common to both implementations is the code that calculates charge and the code that establishes the current and last state variables for the states used for to calculate the respective outputs. CAPH uses the current, i_in, to calculate the output. CAPG uses charge to calculate the output. In order to ensure that the DC value is carried to the transient analysis the thisSTATE and lastSTATE API calls are used.

AC Analysis

The AC analysis is detected the same way as the DC, but using MIF_AC. The AC analysis code for CAPH is given below;

```
else if(ANALYSIS == MIF_AC){
// Calculate the AC analysis
// gain = 1/Cs
dtmp = 1 /(*capvalue * (realFreq * realFreq + imagFreq *
imagFreq));
       ac_gain.real = realFreq * dtmp;
       ac_gain.imag = (-imagFreq) * dtmp;
       AC_GAIN(cap,cap) = ac_gain;
}
```

In this section the real and imaginary components of the AC gain are calculated and assigned to the output using the AC_GAIN API call. By default SPICE provides the imaginary component, jω, and all calculations are relative to this. In order for new code models to be compatible with future additions of the ISSPICE4 pole-zero analysis, it is recommended that calculation include both the real and imaginary components of the frequency axis. Once the AC gain is calculated it is passed to the AC_GAIN API call.

As a comparison the AC analysis code for CAPG is provided.

```
else if(ANALYSIS == MIF_AC){
// Calculate the AC analysis
// gain = Cs
       ac_gain.real = realFreq * capvalue;
       ac_gain.imag = (-imagFreq) * capvalue;
       AC_GAIN(cap,cap) = ac_gain;
}
```

Transient Analysis
As was the case for the DC and AC analyses the ANALYSIS API call is used in a comparison against MIF_TRAN to detect the transient analysis. The code for the CAPH transient analysis is given below.

```
else if(ANALYSIS == MIF_TRAN){
// Calculate the Transient analysis
     if(isMODEINITTRAN){
// process initial conditions for the model
     lastSTATE(CURRENT) = thisSTATE(CURRENT);
     }
// these calculations are processed for every iteration
// initialize cur for iteration
     cur = *i_in = INPUT(cap);

     stateIntegrate(i_in,charge,&partial,i_last);

// reset to iterate
     *i_in = cur;

// output Q=CV or V = Q/C
     OUTPUT(cap) = *charge * *one_over_c;
     PARTIAL(cap,cap) =partial * *one_over_c; // 1/C
     }
}
```

For the initial pass we use the isMODEINITTRAN API call to detect the first timepoint and, in conjunction with PARAM_NULL, set initial conditions as we did in the DC analysis. If “ic” is not present the transient analysis proceeds to calculate the correct output.

First the input current is read and assigned to the local variable cur and the state variable *i_in. Next we integrate current to obtain charge. The charge is then used to calculate the output voltage.

$$Q = CV \text{ or } V=Q/C$$

The implicit integration method employed by IsSpice4 allows the stateIntegrate API call to integrate, or differentiate, depending on the which argument is held constant. In this case The local variable cur was used to keep *i_in from changing. This forces IsSpice4 to iterate, performing integration, until a solution is determined.

In the case of CAPG, shown below, the local variable qvin is used to hold the charge constant. This forces IsSpice4 to iterate, performing differentiation, until a solution is determined.

```
else if(ANALYSIS == MIF_TRAN){
// Calculate the Transient analysis
// these calculations are processed for every iteration
// initialize qvin for iteration
     *charge = qvin = vcap * capvalue;

     if(isMODEINITTRAN)
       lastSTATE(CHARGE) = thisSTATE(CHARGE);

     stateIntegrate(current,charge,&dt,currentlast);

     if(isMODEINITTRAN)
       *currentlast = *current;
     else
// reset to iterate
       *charge = qvin; // iterate to make this true

// output Q=CV or V = Q/C
     {
     double diout_dvin = capvalue / dt;
     OUTPUT(cap) = *current;
     PARTIAL(cap,cap) = diout_dvin;
     }
     }
}
```

In order for "Convert MOD to C" to run correctly you must remember to save your changes when editing the .MOD files.

Building a Debug DLL
Building a code model DLL is a two step process;

- Run "Convert MOD to C" from the Tools menu.

- Select Build from the Project menu.

The first step is necessary to process some of the API calls. The settings for the "Convert Mod to C" tool is explained in the Adding Tools To Visual C++ section. Essentially it calls a makefile that runs the CMPP preprocessor distributed which the CMSDK. CMPP reads the files associated with the code model DLL, (modepath.lst, udnpath.lst, ifspec.ifs etc.) and creates various header and C files in the DLL and MOD directories. Once these files are created the DLL is built as a normal Visual C++ project.

To build a Debug version;

- **Select the Debug build option for your version of Visual C++.**

- **Build the code model DLL by selecting Build from the Project menu.**

Visual C++ 1.1 does not make as good a connection as Visual C++ 2.x between the errors displayed in the output window and the contents of the .MOD file. Therefore, if you are using Visual C++ 1.1 you will not be able to double click on compiler errors in the output window and have the line selected. However, the line number and filename are given so you can manually open the file and locate any errors.

Before setting a breakpoint the proper debug environment must be established. This is covered in the Setting Up The Debug Environment section of the Code Model Development chapter. For this example the debugging environment is set except for coping SPICE4.EXE into the Samples directory.

- **Copy SPICE4.EXE into the \SAMPLES directory.**

Setting A Breakpoint

After the build has completed the project should will act like any Visual C++ project. All of the functions under the Debug menu, Step Over, Step Into, etc., will work as expected. Note, however, there is no source code, or debug information, provided for the API calls so you will not be able to use the debug functions to access any of the API calls.

The Quickwatch function can be used to view any variable within the code except the API calls. To view the result of an API call you should use a temporary variable. For example, consider the code shown below.

```
// assign values to local variables
     *r_shunt = 1/gMIN;
     *capvalue =PARAM(c)* (1.0 + PARAM(TC) * deltaTemp);
     *one_over_c = 1 / *capvalue;
}
```

The API call PARAM , or the argument c, cannot be viewed using the Quickwatch function. PARAM cannot be viewed because this API call is translated into the proper IsSpice4 call according to the argument passed. The argument, c, cannot be viewed because it is not a variable. It is a keyword defined in the IFS file to reference the capacitance model parameter. In order to view the value returned by PARAM(c) you should assign the result to a variable, such as myparam, and then view and use myparam in the equation.

```
// assign values to local variables
     *r_shunt = 1/gMIN;
     myparam = PARAM(c);
     *capvalue = myparam* (1.0 + PARAM(TC) * deltaTemp);
     *one_over_c = 1 / *capvalue;
}
```

Building a Release DLL

Building a release version of a DLL is as simple as selecting the release option for Visual C++ and then selecting Build from the Project menu.

A Digital OR Gate

This example will use one of he files associated with the Samples DLL called tut_or.mod. This can be found in the tut_or subdirectory under the Samples DLL directory. This example will cover;

- The body of an event driven digital code model.
- State variables and per-instance local static variables.
- Processing event-driven output.

The format for an event-driven code model is much the same as an analog code model. The main exception is that there is no AC analysis support for event driven simulations. Therefore, the only analyses that need to be detected are DC and Transient. The last example demonstrated how to detect these analyses using the ANALYSIS API call and the MIF_DC and MIF_TRAN definitions. The event driven code model uses these same techniques.

As with the analog code model, all code models start with the function declaration passing the ARGS API structure.

```
void cm_tut_or(ARGS)
```

After this, the internal variables are declared using standard C notation.

```
int    i, // generic loop counter
       *size, // number of input ports
       new_out;// storage for new output value

Digital_State_t  *out_old, // previous output
                 input; // storage for input
```

The pointer, *size, is used to store the number of input ports for the code model instance. The new_out variable is used to store the new output for the current event. This value is compared to

the previous output and a new output is generated if the two are different. The Digital_State_t typedef is used to facilitate digital state values. The definition for this is a typedef enum described as:

```
typedef enum{
     ZERO,
     ONE,
     UNKNOWN,
} Digital_State_t;
```

The pointer, *old_output, is used as a state variable to hold the previous value of the code model. The use of this pointer and the new_output integer will be covered shortly.

The first block of code is called only when the code model instance is initialized. This is detected by the INIT API call. This section of code is used for the same purposes as in the analog code model, to initialize memory storage and compute one-time parameter computations.

```
if(INIT) {

/* allocate storage for the outputs */
out_old = (Digital_State_t *) cm_event_alloc(0,sizeof(Digital_State_t));
/** allocate and retrieve size **/
size = (int *) cm_event_alloc(1,sizeof(int));
*size = PORT_SIZE(in);

for (i=0; i< *size; i++)      LOAD(in[i]) = PARAM(input_load);

}
```

The first step is to allocate memory. This is done using the cm_event_alloc API call. The arguments to this call are an integer tag for identification and the size of memory to be allocated. The return is a void pointer that is cast to the appropriate type. The Digital_State_t pointer, *old_output, is assigned a 0 tag for identification. The sizeof function is used to pass the required amount of memory.

The next step is the allocation of memory for the integer pointer, *size, to hold the number of input ports for the code model instance. The PORT_SIZE API call is used to return the number of ports for this instance of the code model.

At this point we also set the capacitive load of the input ports. This is done by looping through the input ports and using the PARAM API call to retrieve the load parameter from the .Model line and set these values using the LOAD API call.

```
for (i=0; i< *size; i++)      LOAD(in[i]) = PARAM(input_load);
```

Performing this in the INIT block reduces the number of times this loop is called. Remember the model is called for every iteration. Try to place one-time code in the initialization block.

The else clause of the INIT block is performed on all subsequent calls to the code model.

```
else {

/* retrieve storage for the outputs */
out_old = (Digital_State_t *) cm_event_get_ptr(0,0);
size = (int *) cm_event_get_ptr(1,0);

}
```

In this section we use cm_event_get_ptr to retrieve the pointers allocated during initialization. The integer tag assigned by cm_event_alloc is passed in order to retrieve the correct pointer. The second parameter is the timepoint of interest. For more information please consult the API Calls chapter.

Now that initialization is complete we start processing the information necessary to compute an output.

```
for (i=0; i< *size; i++) {
/* make sure this input isn't floating... */
   if ( FALSE == PORT_NULL(in) ) {
```

```
/* if a 1, set *out high */
       if ( ONE == (input = INPUT_STATE(in[i])) ) {
          new_out = ONE;
          break;
       }
/* if an unknown input, set *out to unknown */
       else if ( UNKNOWN == input ) {
          new_out = UNKNOWN;
       }
       else
          new_out = ZERO;
       }
   else {
/* at least one port is floating. output is unknown */
       new_out = UNKNOWN;
       break;
   }
}
```

This section computes the new output, new_output, depending on the current inputs. It does this by looping through all inputs, checking to verify that the input is connected (PORT_NULL API call) and then comparing the input against the known available state values, ONE, UNKNOWN, and ZERO. The value of new_output is determined by the first ONE value encountered.

Once the new_output has been calculated we can determine if the code model should post a new output. This is important because the simulation algorithm is event driven. Each change in an output posts an event in the queue to be processed. Therefore, the speed of the event driven simulation is determined by the number of events that are posted. So if an event-driven code model does not need to post an event it should not do so. It should set the API call OUTPUT_CHANGED to FALSE. This tells the event algorithm that the output of this code model will not generate an event and further processing is not required. The following section of code describes this sequence.

```
if ( new_out != *out_old ) { // output value is changing
   if (ANALYSIS == DC) {  // DC output w/o delays
      OUTPUT_STATE(out) = new_out;
   }
   else { // Transient Analysis
      switch ( new_out ) {
// fall to zero value
      case 0: OUTPUT_STATE(out) = new_out;
            OUTPUT_DELAY(out) = PARAM(fall_delay);
            break;
// rise to one value
      case 1: OUTPUT_STATE(out) = new_out;
            OUTPUT_DELAY(out) = PARAM(rise_delay);
            break;
// unknown output
      default:
            OUTPUT_STATE(out) = new_out;
// based on old value, add rise or fall delay
         if (0 == *out_old) {   // add rising delay
            OUTPUT_DELAY(out) = PARAM(rise_delay);
         }
         else { // add falling delay
            OUTPUT_DELAY(out) = PARAM(fall_delay);
         }
      break;
      }
}
   OUTPUT_STRENGTH(out) = STRONG;
   *out_old = new_out;
   }
   else{
      OUTPUT_CHANGED(out) = FALSE;
   }
```

Notice that the first comparison is used to determine if an output needs to be posted. If the comparison fails the OPUTPUT_CHANGED API call is used. Otherwise the desired output is posted using the OUTPUT_STATE API call. Also, notice that a delay is assigned each of these outputs. Notice also that all outputs are assigned a strength using the OUTPUT_STRENGTH API call. This is used by the digital node type to resolve competing driving values.

User-Defined Node

In this example we will cover XDL's ability to construct and simulate User-Defined Node and signal types. The files associated with this example can be found under the Real DLL directory. This example will cover:

- What a User-Defined Node is.
- An overview of the algorithms involved with using UDNs.
- Creating a UDN definition file.

User-Defined Nodes
User defined nodes are data structures that store a component's data and the functions that operate on that data. As an example, consider the Digital Node. This node type stores two values, a logic level and a logic strength. This combination produces the state of a digital node. Also defined within the digital node are functions that allow ISSPICE4 to perform operations on the data such as, allocate storage, compare node values, resolve discrepancies, and copy node values. These functions define the behavior of the data stored within the node type.

Algorithms using the UDN
All UDNs operate using the event-driven simulator of ISSPICE4. This restricts the UDN to operating in the DC and Transient analyses only. The general operation of UDNs is as follows;

- Before any a DC and/or Transient analysis begins, the UDN's udn_XXX_create and udn_XXX_initialize functions are called for every UDN port.
- For the DC operating point, the UDN compares the values of all connected ports using the udn_XXX_compare function. This is done for every iteration until a solution is obtained.

- If the simulation successfully computes the DC operating point, new data elements are created using udn_XXX_create to store the value calculated. The new values are copied into the new data elements using udn_XXX_copy.

- If a transient analysis is being performed the first event is extracted from the event queue. The previous calculations are repeated until a solution is found. Then another event is extracted from the event queue. This process continues until all events are processed.

A UDN definition file

The definition file for a UDN consists of two sections; functions describing the data and the behavior of the data, and a structure used to interface the UDN to the event-driven algorithm. We will start by discussing the latter.

At the bottom of every UDN definition file is the data structure that defines the interface between the algorithm and the models that use the node. The Evt_Udn_Info_t structure contains the name of the node to be used, “real2”, a brief comment about the node type, and a list of functions used to describe the node’s behavior.

```
Evt_Udn_Info_t udn_real2_info = {

     “real2”,
     “example real node”,

     udn_r2_create,
     udn_r2_initialize,
     udn_r2_copy,
     udn_r2_compare,
     udn_r2_invert,
     udn_r2_dismantle,
     udn_r2_resolve,
     udn_r2_plot_val,
     udn_r2_print_val,
     udn_r2_ipc_val

};
```

The following functions are required to describe the correct behavior of a UDN;

```
udn_r2_create
udn_r2_initialize
udn_r2_copy
udn_r2_compare
```

The remaining functions are optional. This does not mean they can be left out of the definition file. ***A function must exist in the definition file even if it is a do nothing function.*** A complete list of the functions can be found in the API Calls chapter.

The name of the node used in this structure must match the name entered in a code model's IFS file if the code model is going to process the UDN's data type. You can see this by examining the rgain.mod file in the rgain Model directory of under the Real DLL directory.

udn_r2_create
Any time new data is needed for a this node type (real2), IsSpice4 will call the udn_r2_create function of the udn_real2_info data structure.

```
void  udn_r2_create(CREATE_ARGS)
{
     /* allocate space for a real struct */
     MALLOCED_PTR = MALLOC(sizeof(double));
}
```

The purpose of this function is to allocate storage for the data elements. In the case of real data, a simple double value.

udn_r2_dismantle
This function is used to free any memory used during a processing step. Since this function is not required, and no memory is allocated within this node type, this function is left empty.

```
void  udn_r2_dismantle(DISMANTLE_ARGS)
{ // There is nothing to dismantle
}
```

udn_r2_initialize
This function is called any time the create function is called and a copy is not being performed. It is used to provide a stable initial value for the data elements within the UDN. For this example, the double allocated is set to 0.0.

```
void  udn_r2_initialize(INITIALIZE_ARGS)
{
     double *real_struct = STRUCT_PTR;

     /* Initialize to zero */
     *real_struct = 0.0;
}
```

udn_r2_invert
This function is used to process the leading "~" that can be placed in front of a node in a netlist. The code should correctly process the inversion for the data type of the UDN. In the case of real data, the value is multiplied by minus one.

```
void  udn_r2_invert(INVERT_ARGS)
{
     double *real_struct = STRUCT_PTR;


     /* Invert the state */
     *real_struct = -(*real_struct);
}
```

und_r2_copy
This function is used to copy the contents of one data structure into another structure of the same type. This function is most often called after an event has caused a new value to be added to the event queue. In this case, the new structure is created, using udn_r2_create, and the new contents are copied into the data structure.

```
void  udn_r2_copy(COPY_ARGS)
{
     double *real_from_struct = INPUT_STRUCT_PTR;
     double *real_to_struct = OUTPUT_STRUCT_PTR;
```

```
     /* Copy the structure */
     *real_to_struct = *real_from_struct;
}
```

und_r2_resolve
This function is called whenever two or more ports are connected to a node. IsSPICE4 will use this function to determine the correct value for the node. In the case or real data, the resolve function is used to provide a summing function.

```
void  udn_r2_resolve(RESOLVE_ARGS)
{
     double **array = (double**) INPUT_STRUCT_PTR_ARRAY;
     double *out = OUTPUT_STRUCT_PTR;
     int num_struct = INPUT_STRUCT_PTR_ARRAY_SIZE;
     double sum;
     int i;
     /* Sum the values */
     for(i = 0, sum = 0.0; i < num_struct; i++)
     sum += *(array[i]);
     /* Assign the result */
     *out = sum;
}
```

udn_r2_compare
This function is used to define the comparison operation for the UDN data type.

```
void  udn_r2_compare(COMPARE_ARGS)
{
     double *real_struct1 = STRUCT_PTR_1;
     double *real_struct2 = STRUCT_PTR_2;
     /* Compare the structures */
     if((*real_struct1) == (*real_struct2))
     EQUAL = TRUE;
     else
     EQUAL = FALSE;
}
```

udn_r2_plot_val, udn_r2_print_val, udn_r2_ipc_val
These functions are not currently supported. They should appear as empty, do nothing function, like udn_XXX_dismantle.

Node Bridges (Hybrid Models)

A node bridge, also referred to as a hybrid model, is a code model that uses multiple node types. The node bridge is a special case of the hybrid model. Its sole purpose is to translate a signal from one data type to another. This example will cover;

- Creating the node bridge IFS file.
- The structure of a node bridge model definition file.

IFS File

As with all code models an IFS file must be created. The only distinction between this code model's IFS file and other IFS files is that the PORT_TABLE section will describe two different node types.

```
PORT_TABLE:

Port_Name:       in        out
Description:     "input"   "output"
Direction:       in        out
Default_Type:    real2     v
Allowed_Types:   real2     [v, vd, i, id]
Vector:          no        no
Vector_Bounds:   -         -
Null_Allowed:    no        no
```

The ports, in and out, describe two different node types. Notice that "in" requires that the input connection be to a node type of "real2". If any other node type is connected to this port ISSPICE4 will generate an error. The name specified for the Allowed or Default types should reflect the name given to the node type in the UDN definition file for the desired node type. The remaining output port, "out", can be any analog voltage or current.

Node Bridge Definition File

As with the other code models the definition file starts by calling the function as named in the IFS file with the ARGS argument.

```
void ucm_r2_to_a (ARGS)
```

After the initial function call all variables are declared.

```
double *t, *v, *g;
double *in;
```

Before we proceed to the initialization code we use the pointer, *in, to retrieve the input to the code model.

```
in = INPUT(in);
```

This line will be executed for ever iteration through the code model. Next, the INIT section is called during the initialization of the code model.

```
if(INIT) {
// allocate memory
     cm_event_alloc(TS, 2 * sizeof(double));
     cm_event_alloc(VS, 2 * sizeof(double));
     cm_event_alloc(GAIN,  sizeof(double));
 // retrieve pointers
     t = (double *) cm_event_get_ptr(TS, 0);
     v = (double *) cm_event_get_ptr(VS, 0);
     g = (double *) cm_event_get_ptr(GAIN, 0);
// initialize pointers
     t[0] = -2.0;
     t[1] = -1.0;
     v[0] = *in;
     v[1] = *in;
     *g = PARAM(gain);
}
```

As with other code models the INIT section is used to perform one-time allocation and initialization. The cm_event_alloc API call allocates memory for event pointers. The pointers are retrieved using cm_event_get_ptr. These API calls use integer

tags, TS, VS, and GAIN, that uniquely identify the block of memory. The easiest way to provide these tags is through enumeration or definition. In this case, we defined the tags as:

```
#define TS 0
#define VS 1
#define GAIN 2
```

The else clause of the INIT section retrieves the pointers for all subsequent iterations through the code model.

```
else {
// retrieve pointers
      t = (double *) cm_event_get_ptr(TS, 0);
      v = (double *) cm_event_get_ptr(VS, 0);
      g = (double *) cm_event_get_ptr(GAIN, 0);
}
```

Now that the input has been read and all memory has been allocated and initialized we can process the input and create an output. The significant feature of the node bridge's definition file is that it contains calls to both the analog and event-driven simulation algorithms. For this code model we use a case-switch statement and the CALL_TYPE API call to detect the algorithm that is calling the bridge code model.

```
// detect simulation algorithm
switch(CALL_TYPE) {
```

The case statements use the ANALOG or EVENT definitions to detect the algorithm being used.

```
case ANALOG:
      if(TIME == 0.0) {
        OUTPUT(out) = *in * *g;
        v[0] = *in;
        v[1] = *in;
      }
      else {
        if(TIME <= t[0])
          OUTPUT(out) = v[0] * *g;
        else if(TIME >= t[1])
```

```
        OUTPUT(out) = v[1] * *g;
      else {
          OUTPUT(out) = (v[0] + (v[1] - v[0]) *
              (TIME - t[0]) / (t[1] - t[0])) * *g;
      }
    }
    break;

case EVENT:
      if(TIME == 0.0)
        return;
      if(TIME >= t[1]) {
        v[0] = v[1];
        v[1] = *in;
        t[0] = TIME;
        t[1] = TIME + PARAM(transition_time);
      }
      else {
        v[0] = v[0] + (v[1] - v[0]) *
            (TIME - t[0]) / (t[1] - t[0]);
        v[1] = *in;
        t[0] = TIME;
        t[1] = TIME + PARAM(transition_time);
      }
    break;
}
```

The ANALOG section is used to process the analog outputs based on the information calculated in the EVENT section.

Appendices

Appendix A: Translation Of SPICE 3 Data Structures

The following table illustrates commonly encountered SPICE 3 data members and indexes the appropriate paragraph that explains a method of using the code model XDL to make simplified access.

SPICE 3	Code Model XDL	Notes to See
ckt->CKTmode & MODEXXX	isMODEXXX	"codedef.h"
here->instanceVar	newVar	instance variables
model->modelParam	PARAM(modelParam)	model parameters
*(ckt->CKTstate0 + here->CAPqcap)	*charge	State variables
*(ckt->CKTrhs + here->myNode)	*output	Outputs

Defining Model Parameters:
The code model XDL accesses model parameters using the PARAM accessor. You simply replace:

```
      model->DIOjunctionPot
with  PARAM(DIOjunctionPot)
```

where `DIOjunctionPot` is defined in your ".IFS" file.

Note: you cannot use XDL accessors in header files because the code model compiler does not expand header files.

Instance Variables:
In SPICE 3, there are 2 kinds of instance variables
1) Parameters passed into the model
2) Computed values

In the code model XDL, there is no provision for the first type; each model must describe all parameters. You can either convert these to model parameters as shown above or use instance variables and initialize them in the code as shown.

Replace SPICE 3 references like `here->DIOarea` with:

Initialization:
```
newVar(n); /* reserves space for n instance
              variables (all are doubles) */
```
Access:
```
double * here = getVarPtr(0);
/* gets a pointer to an array of variables for
this instance */
```
Setup:
```
DIOarea = 1.0 // set default if non zero
```

Header file:
```
#define IdxDIOarea 12  // the 12th element
#define DIOarea  (here[IdxDIOarea]) // lvalue
```

Defining State Variables:
In SPICE 3, state variables are accessed using macros; for example, `*(ckt->CKTstate0 + here->CAPqcap)` is expanded to `*(ckt->CKTstate[0] + here->CAPqcap)`, where `here->CAPqcap` is an integer offset into the array. The equivalent in the code model XDL is:

`thisStatePtr[here->CAPqcap]` or
`*(thisStatePtr() +here->CAPqcap)`.

We recommend that the SPICE 3 code be altered as follows:

Change the SPICE 3 state references from:
`*(ckt->CKTstate0 + here->CAPqcap)`
to: `CKTstate0[CAPqcap]`

Initialize:

```
newState(CAPsize); // reserve CAPsize states
// get a fast pointer to current states
double * CKTstate0 = thisStatePtr(0);
// get a fast pointer to previous states
double * CKTstate1 = lastStatePtr(0);
```

Place the following in a header to be included:

```
typedef enum {
    ...
     CAPqcap,
     CAPccap,
     CAPsize;
  } CAPstates;
```

Inputs:
In SPICE 3, inputs are found in the solution vector and are of the form:

```
*(ckt->CKTrhs + here->myNode)
```

In the code model XDL, the inputs are specified by name in the ".IFS" file and are referenced in your code using the INPUT accessor so that the above translates to INPUT(myNode).

Outputs:
In SPICE 3, outputs are place in the MNA matrix and its excitation Vector using rules based on their type. In the code model XDL, the matrix loading is done for you so all you need to do is send the output to the XDL using the OUTPUT and PARTIAL accessors. For example the following shows how a diode could be models:

SPICE 3	Code Model XDL
`*(ckt->CKTrhs + here->DIOnegNode) += cdeq;`	`OUTPUT(diode) = cdeq;`
`*(ckt->CKTrhs + here->DIOposNode) -= cdeq;`	
`*(here->DIOposPosPtr) += gd;`	`PARTIAL(diode,diode)= gd;`
`*(here->DIOnegNegPtr) -= gd;`	

General Considerations:
SPICE 3 allows models to add internal nodes; this is usually done to add series resistance. There is currently no provision for this in the code model XDL. You must either solve the equations inside the code model or add the extra parts using the subcircuit capabilities of IsSPICE4.

Code Models do not pass parameters to the models. In general, these parameters should be placed in the ".IFS" file as model parameters. Be sure to assign defaults. There will be a different model for each instance with different model parameters.

SPICE 3 models will iterate through all models and instances; this is unnecessary in the code model. The XDL takes care of resolving objects down to the instance level.

SPICE 3 models have separate files for temperature, setup, AC load, DC load and Tran load. These are all done within the ".MOD" file as shown in the Code Model Development chapter.

Appendix B: The SPICE 3 CKTcircuit Data Structure

In IsSPICE4, all models are passed a pointer to the current circuit, CKTcircuit *, data. In the code model XDL this data is found within the private data structure by including the header "codeinit.h" just after the code model cm_model(ARGS) function declaration in the ".mod" file. The include is automatically added to the ".c" file by the code model compiler.

This header establishes 3 stack variables which are:

```
Mif_Info_t * _mif_info_ptr_; // code model data
int _state_index_;    /* the index of the first state
   used by this instance */
CKTcircuit * _ckt_; // the current circuit data
```

These variables are then used in the "codedefs.h" header to expand the various macros. Not all possible CKTcircuit struct members have been resolved by these macros. You can access other members directly using the _ckt_ pointer or by

adding your own header file. Again, the _mif_info_ptr_ and the private pointer should be used only for debugging; they are not described and they may change in future releases.

Appendix C: Project Settings

The following is a list of project settings for developing a code model DLL using the CMSDK. These settings are not the only way to arrange you project. They are meant as a starting point. Please adjust your environment to suit your needs.

- The project must be set to a Dynamic Link Library not using MFC. It should include all .C files that will be constructed from .MOD files and dll_main.C located in the CMCOMMON directory. If non exist at the time the project is created, the project will only include dll_main.c. After running the "Convert MOD to C" tool add the resulting C files to the project.

- The INCLUDE directory of the CMSDK and the DLL directory must be added to the Include Files search path. This can be done by adding ".\..\..\INCLUDE" to the Visual C++ Include path option.

- The OBJ directory under the CMCOMMON directory must be added to the Library search path. This is done by adding "..\CMCOMMON\OBJ" to the Visual C++ Library path.

- The _X86_ definition should be removed from the project.

The remaining options are set depending on the way you want to run and debug the DLL. The best way to determine the settings is to examine on of the example projects for you version of Visual C++.

Visual C++ 1.1

- Select Debug... from the Options menu and enter the following information;

spice4.exe *path to test directory**testfile*.cir

- Copy SPICE4.EXE into the DLL directory when it is time to run the DLL.

Visual C++ 2.0

- Select Settings... from the Project menu.

- Select both the Release and Debug from the Setting For: field.

- Select the Debug tab and enter the path and name of the IsSpice4 executable located in the ICAP/4Windows directory structure.

- Enter the name of the test file, and the path to the test file, in the Program Arguments field.

- Select the Link tab.

- Enter the path to the IsSpice4 executable located in the ICAP/4Windows directory structure and the name of the DLL you are building into the Output file Name: field.

Index

.C 43
.H 24
.MOD 37, 43
.Model 31, 45
\ 47
~ 138

A

AC analysis 16, 121, 125, 130
AC_GAIN 37, 41, 54, 57, 116, 125
accessor macros 9, 51
adding tools 25
AHDL 9
allocate memory 131
Allowed Types 33
Allowed_Types 30
ANALOG 142
analog code model
    example 113
ANALYSIS 38, 40, 53, 58, 116
API 51
API call 122
API calls 9
arbitrary source 14
ARGS 38, 52, 53, 58, 120

B

B element 14
behavioral modeling 9, 21
Berkeley SPICE 3C.1 10
bin 24
breakpoint 117
breakpoints 14
building the DLL 128

C

C Function Name 31
C_Function_Name 30
CALL_TYPE 53, 59, 142
CALLOC 55, 59
capacitive load 132
capacitor example 13
CAPG 12
cktABSTOL 59
CKTcircuit data structure 13
cktNOMTEMP 60
cktRELTOL 60
cktTEMP 60
cktVOLT_TOL 61
Clean DLL 25
Clean11.Bat 25
Cleanout.Bat 25
cm_analog_auto_partial 56, 61
cm_analog_converge 56, 62
cm_analog_not_converged 56, 62
cm_analog_set_perm_bkpt 56, 63
cm_analog_set_temp_bkpt 56, 64
cm_climit_fcn 56, 64
cm_complex_add 56, 66
cm_complex_div 56, 66
cm_complex_mult 56, 66
cm_complex_set 56, 67
cm_complex_sub 56, 67
cm_event_alloc 55, 68, 131, 141
cm_event_get_ptr 55, 68, 132, 142
cm_event_queue 56, 69
cm_gain 116
cm_message_get_errmsg 56, 69
cm_message_send 56, 70
cm_netlist_get_c 56, 71

cm_netlist_get_l 56, 71
cm_ramp_factor 56, 62
cm_smooth_corner 55, 72
cm_smooth_discontinuity 55, 73
cm_smooth_pwl 55, 74
CMCOMMON directory 115
CML.DLL 9
CMLtgt.mak 24
CMPP 13, 24, 26, 38, 43, 51, 128
CMSDK 9
  installing 7
code model 9
  building 43
  example 113
  opening project 29
code model compiler 13
code model requirements 15
code models
  creating 27
  project directory 28
COMPARE_ARGS 49
Compiling 118
Complex_t 67, 120
convergence 12, 41, 42
Convert MOD to C 43, 149
  example 117
Convert Mod To C 27
Convert Mod to C 26
COPY_ARGS 49
CPAG.MOD 122
CREAT_ARGS 48

D

d, port type 32
Data Type 34
  Static Variable 36
Data_Type 30
DC analysis, example 123
DC operating point, UDN 136
DC sweep 15
Debug menu 117
debug version 117
debugging 44
Default Type 32
Default Value 34
Default_Type 30
Default_Value 30
definition file
  gain model 116
  UDN 136
deltaTemp 54, 74
derivative 40
Description
  Name Table 31
  Parameter Table 34
  Port Table 32
  Static Variable 36
digital, initialization 132
digital example 130
digital load 132
digital node 134
digital OR gate 130
digital simulation 17, 19
Digital_State_t 92, 130
Digital_Strength_t 80
Direction 30, 32
directory structure 23
DISMANTLE_ARGS 48
DLL
  CML 9
  directory 23, 47
  executing 43
  Simple 113
DLL_MAIN.C 115
doc 24

E

EQUAL 54, 75
EVENT 142
event 17
event-driven code model
  example 130, 133
Evt_Udn_Info_t 50, 136
example
  DC analysis 123
  digital 130
  gain 113
  INIT routine 121
  node bridge 140
  User-Defined Node 135
examples 24
eXtended Description Language 10

F

fast pointers 38, 39, 122
file location 114
FREE 55
frequency, complex axis 16
frequency analysis 40

G

g, port type 32
Gain example 113
gd, port type 32
Georgia Tech 10
getVar 55, 76
getVarPtr 55, 77, 122, 146
gMIN 54, 77
gmin stepping 14

H

h, port type 32
hardware protection key 8
hardware requirements 7
hd, port type 32
hybrid 9
hybrid model 140

I

i, port type 32
id, port type 32
ID string 114
Ident.h 24
ident.h 114
IFS 27, 38
  creating 29
  parameter name 52
  port name 52
IFS file 10
  gain example 115
  node bridge 140
ifspec.ifs 115
imagFreq 40, 54, 77
imaginary 121
include 24
INIT 13, 36, 38, 39, 53, 78
  digital 132
  example 121
  node bridge 141
INITIALIZE_ARGS 49
INPUT 37, 53, 78, 147
INPUT_SIZE 37
INPUT_STATE 53, 79
INPUT_STRENGTH 53, 80
INPUT_STRUCT_PTR 54, 80
INPUT_STRUCT_PTR_ARRAY 54, 81
INPUT_STRUCT_PTR_ARRAY_SIZE 54, 82
INPUT_TYPE 53
installing 7
installing The CMSDK 7
instance variables 13, 37, 146
integral 42
integration 12, 13, 41
Interface Specification File 29

INVERT_ARGS 49
IS directory 117
isBYPASS 55
isINIT 55, 82
isMODEAC 55, 83
isMODEINITFIX 55, 83
isMODEINITJCT 55, 83
isMODEINITPRED 38, 43, 55, 84
isMODEINITSMSIG 55, 84
isMODEINITTRAN 38, 41, 55, 84, 126
isMODETRAN 55, 85
isMODETRANOP 55, 85
isMODEUIC 55, 86
IsSpice3 21
IsSpice4
- analyses 16
- arbitrary source 14
- CKTcircuit data structure 13
- memory usage 13
- simulation flow 14

iteration 42

K

Kirchoff's current law 11

L

lastSTATE 55
lastSTATEptr 39, 55, 86, 123
lastStatePtr 147
license levels 114
Limits 30
- Parameter Table 35

LOAD 53, 87, 132
local variables 120

M

macromodel 10
macromodeling 21
MakeDLL.Mak 24
makefile.cml 24
MALLOC 55
MALLOCED_PTR 54, 88
memory 13
memory allocation 36, 39, 43, 131
Microsoft Visual C++ 7
MIF_AC 38, 40, 116, 125
Mif_Complex_t 67
MIF_DC 38, 40
Mif_Info_t 148
Mif_Private_t 58
MIF_TRAN 38, 41, 126
MOD 27
MOD file 10
mod_to_c.cml 24
mod_to_c.mak 24, 26, 43
model directory 23
Model Definition File 37
model directory 23
model parameter 118
modeling, behavioral 9
models, SPICE 3 13
modified nodal analysis 11
modpath.lst 24, 28, 114

N

Name, Static Variable 36
NAME_TABLE 30
newState 12, 39, 55, 88, 122, 147
Newton-Raphson 12
newVar 13, 37, 55, 89, 122, 145
nmake.exe 26, 43
nodal analysis 11
node bridge 18
- example 140

nonlinear differential equations 11
Null Allowed
- Parameter Table 35
- Port Table 30, 33

O

operating point 14, 15
  example 123
  UDN 135
OUTPUT 37, 41, 54, 90, 147
OUTPUT_CHANGED 54, 90, 133
OUTPUT_DELAY 54, 91
OUTPUT_SIZE 37
OUTPUT_STATE 54, 92, 134
OUTPUT_STRENGTH 54, 92, 134
OUTPUT_STRUCT_PTR 54, 93
OUTPUT_TYPE 54

P

PARAM 37, 53, 94, 116, 145
PARAM_NULL 35, 53, 95, 126
PARAM_SIZE 53, 94
Parameter Name 34
Parameter Table 30, 34
Parameter_Name 30
PARTIAL 37, 40, 54, 96, 147
partial derivative 40, 124
partial derivatives 12
PLOT_VAL_ARGS 49
pole-zero analysis 16, 125
polynomial sources 14
port, UDN 135
Port Name 30, 32
Port Table 30
port type
  d 32
  g 32
  gd 32
  h 32
  hd 32
  i 32
  id 32
  user-defined 32
  v 32
  vd 32
  vname 32
PORT_NULL 53, 97
PORT_SIZE 53, 97, 132
PORT_TABLE
  node bridge 140
postQuit 54, 98
predictor algorithm 12
primitive 10
PRINT_VAL_ARGS 49
private data structure 120
private data structures 13
project directory 28
project file
  debug info 29
  opening 29
  output generation 29
  UDN 48
project settings 149

Q

QuickWatch 118

R

RAD_FREQ 53, 98
real 121
Real DLL directory 135
real2 136
realFreq 40, 54, 98
REALLOC 55
reference designation
  code models 45
requirements
  software/hardware 7
RESOLVE_ARGS 49
rgain.mod 137

S

s_xfer 31
signal translation 140
Simp11.DLL 117
Simple.Mak 114
simulation process 14
small signal gain 40
software requirements 7
source stepping 14
SPICE 2
    polynomial controlled sources 14
SPICE 2G.6 21
SPICE 3
    model conversion 145
    models 13
SPICE Model Name 31
Spice_Model_Name 30
src 24
SRC directory
    Simple 113
state variable 10, 12
state variables 39, 41, 146
stateIntegrate 13, 41, 42, 56, 99
Static Variable Table 36
static variables 36, 37, 122
STATIC_VAR 37, 54, 101
STATIC_VAR_SIZE 54
storage 13
strength 80
STRUCT_MEMBER_ID 54, 102
STRUCT_PTR 54, 102
STRUCT_PTR_1 54, 103
STRUCT_PTR_2 54, 103
support files 28
switch-case arrangement 123
syntax, code models 45

T

T( ) 105
T(n) 53
TEMPERATURE 53, 106
temperature calculations 13
test 24
testing the DLL 117
The Name Table 31
The Port Table 32
thisSTATE 55, 104
thisSTATEptr 39, 55, 123
thisStatePtr 146
TIME 53, 142
time domain analysis 16
Tools menu 25
TOTAL_LOAD 53, 106
transient analysis 16
Transient simulation 14
tut_or.mod 130

U

UDN 46
    building 50
    definition file 47, 48
    example 140
    operation 135
udn_int_compare 49
udn_int_copy 49
udn_int_create 48
udn_int_dismantle 48
udn_int_initialize 49
udn_int_invert 49
udn_int_plot_val 49
udn_int_print_val 49
udn_int_resolve 49
udn_r2_compare 139
udn_r2_create 137
udn_r2_dismantle 137

udn_r2_initialize 138
udn_r2_invert 138
udn_real2_info 137
udn_XXX_compare 57, 107
udn_XXX_copy 57, 108
udn_XXX_create 57, 107
udn_XXX_dismantle 57, 108
udn_XXX_info 50
udn_XXX_initialize 57, 109
udn_XXX_invert 57, 110
udn_XXX_ipc_val 57
udn_XXX_plot_val 57
udn_XXX_plotval 111
udn_XXX_print_val 57, 110, 111
udn_XXX_resolve 57, 111
UDNpath.Lst 47
udnpath.lst 24, 28, 114
UIC 14
und_r2_copy 138
und_r2_resolve 139
User-Defined Node
  example 135
  functions 57
user-defined node 18
User-Defined Nodes 46
user-defined, port type 32
using the code model 45

V

v, port type 32
variables
  circuit (ckt) 148
  instance 146
  state 146
vd, port type 32
Vector 30
  Parameter Table 35
  Port Table 33
Vector Bounds
  Parameter Table 35
  Port Table 33
Vector_Bounds 30
VHDL 10
Visual C++ 10
vnam, port type 32

W

Windows NT 7

X

XDL 10
XSPICE 10, 21