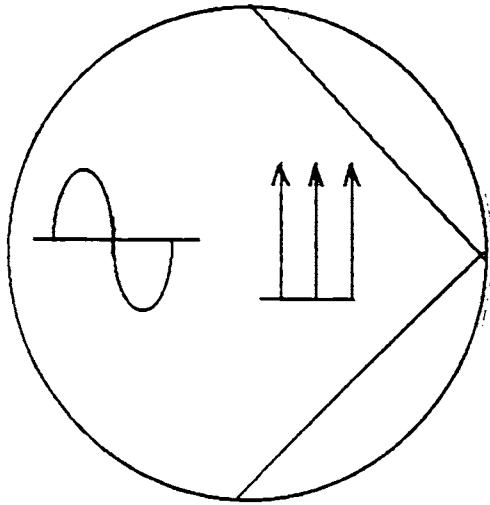


*XSPICE*TM

Code Model Subsystem Software Design Document



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

Georgia Tech Research Institute
Georgia Institute of Technology

Georgia Tech Research Corporation
Office of Technology Licensing
Atlanta, Georgia 30332-0415
Telephone (404) 894 - 6287
Facsimile (404) 894 - 9728

Contents

1 Scope	1
1.1 Identification	1
1.2 System Overview	1
1.3 Document Overview	3
1.4 Acknowledgements	4
2 Referenced Documents	5
3 Top-Level Design	7
3.1 CSCI Overview	7
3.1.1 CSCI Architecture	8
3.1.1.1 Code Model Toolkit	8
3.1.1.2 Code Model Library	9
3.1.1.3 User-Defined Node Library	10
3.1.2 System States and Modes	10
3.1.3 Memory and Processing Time Allocation	10
3.2 CSCI Design Description	11
3.2.1 Code Model Toolkit	11
3.2.1.1 Model Directory Generator	13
3.2.1.2 User-Defined Node Directory Generator	13
3.2.1.3 Simulator Directory Generator	14
3.2.1.4 Code Model Preprocessor	14
3.2.2 Code Model Library	17
3.2.3 User-Defined Node Library	19
4 Detailed Design	21
4.1 Code Model Toolkit	21
4.1.1 Model Directory Generator	22
4.1.2 User-Defined Node Directory Generator	23
4.1.3 Simulator Directory Generator	24

4.1.4	Code Model Preprocessor	25
4.1.4.1	Function main	26
4.1.4.2	Function init_error	28
4.1.4.3	Function print_error	29
4.1.4.4	Function preprocess_ifs_file	30
4.1.4.5	Function preprocess_mod_file	69
4.1.4.6	Function preprocess_lst_file	89
4.2	Code Model Library	104
4.2.1	Analog Code Models	104
4.2.1.1	Gain	105
4.2.1.2	Summer	106
4.2.1.3	Multiplier	107
4.2.1.4	Divider	109
4.2.1.5	Limiter	111
4.2.1.6	Controlled Limiter	113
4.2.1.7	PWL Controlled Source	116
4.2.1.8	Analog Switch	122
4.2.1.9	Zener Diode	124
4.2.1.10	Current Limiter	128
4.2.1.11	Hysteresis Block	132
4.2.1.12	Differentiator	136
4.2.1.13	Integrator	138
4.2.1.14	S-Domain Transfer Function	141
4.2.1.15	Slew Rate Block	148
4.2.1.16	Inductive Coupling	152
4.2.1.17	Magnetic Core	154
4.2.1.18	Controlled Sine Wave Oscillator	161
4.2.1.19	Controlled Triangle Wave Oscillator	164
4.2.1.20	Controlled Square Wave Oscillator	167
4.2.1.21	Controlled Oneshot	171
4.2.1.22	Capacitor	176
4.2.1.23	Capacitance Meter	178
4.2.1.24	Inductor	179
4.2.1.25	Inductance Meter	181
4.2.2	Hybrid Code Models	182
4.2.2.1	Digital-to-Analog Node Bridge	182
4.2.2.2	Analog-to-Digital Node Bridge	187
4.2.2.3	Controlled Digital Oscillator	191
4.2.3	Digital Code Models	195
4.2.3.1	Digital Buffer	195
4.2.3.2	Digital Inverter	197
4.2.3.3	Digital AND Gate	199

4.2.3.4	Digital NAND Gate	202
4.2.3.5	Digital OR Gate	205
4.2.3.6	Digital NOR Gate	208
4.2.3.7	Digital XOR Gate	211
4.2.3.8	Digital XNOR Gate	214
4.2.3.9	Digital Tristate Buffer	217
4.2.3.10	Digital Open-Collector Buffer	219
4.2.3.11	Digital Open-Emitter Buffer	221
4.2.3.12	Digital Pullup Resistor	223
4.2.3.13	Digital Pulldown Resistor	224
4.2.3.14	Digital D Flip Flop	225
4.2.3.15	Digital JK Flip Flop	228
4.2.3.16	Digital Toggle Flip Flop	231
4.2.3.17	Digital Set-Reset Flip Flop	234
4.2.3.18	Digital D Latch	238
4.2.3.19	Digital Set-Reset Latch	241
4.2.3.20	Digital State Machine	245
4.2.3.21	Digital Frequency Divider	257
4.2.3.22	Digital Random Access Memory	259
4.2.3.23	Digital Source	266
4.3	User-Defined Node Library	274
4.3.1	Node Type "real"	275
4.3.1.1	Function udn_real_create	275
4.3.1.2	Function udn_real_dismantle	276
4.3.1.3	Function udn_real_initialize	277
4.3.1.4	Function udn_real_invert	278
4.3.1.5	Function udn_real_copy	279
4.3.1.6	Function udn_real_resolve	280
4.3.1.7	Function udn_real_compare	281
4.3.1.8	Function udn_real_plot_val	282
4.3.1.9	Function udn_real_print_val	283
4.3.1.10	Function udn_real_ipc_val	284
4.3.2	Node Type "int"	285
4.3.2.1	Function udn_int_create	285
4.3.2.2	Function udn_int_dismantle	286
4.3.2.3	Function udn_int_initialize	287
4.3.2.4	Function udn_int_invert	288
4.3.2.5	Function udn_int_copy	289
4.3.2.6	Function udn_int_resolve	290
4.3.2.7	Function udn_int_compare	291
4.3.2.8	Function udn_int_plot_val	292
4.3.2.9	Function udn_int_print_val	293

CONTENTS

XSPICE Code Model Subsystem Software Design Document

4.3.2.10 Function udn_int_ipc_val	294
5 CSCI Data	295
5.1 Interface Specification Data	295
6 CSCI Data Files	301
7 Requirements Traceability	303
8 Notes	307
8.1 Glossary	307
8.2 Acronyms	309
8.3 Project Unique Identifiers	310

List of Figures

3.1 Code Model Subsystem External Interfaces.	7
---	---

List of Tables

6.1	Code Model Toolkit Data Files.	301
6.2	Code Model Toolkit Template Files.	302
7.1	Requirements Cross Reference.	303

1

Scope

1.1 Identification

This Software Design Document describes the XSPICE Code Model Subsystem CSCI (Computer Software Configuration Item) of the version 2 Automatic Test Equipment Software Support Environment system (ATESSE). This design is governed by the Software Requirements Specification for the Simulator of the Automatic Test Equipment Software Support Environment (ATESSE).

1.2 System Overview

The ATESSE is an integrated set of software tools designed to support all stages of the life cycle of software used to control Automatic Test Equipment (ATE) in testing analog and hybrid (analog/digital) circuit cards.

The ATESSE includes a mixed-mode (analog/digital) simulator called XSPICE which performs mathematical simulation of a circuit specified by the user. The XSPICE simulator takes input in the form of commands and circuit descriptions and produces output data which predicts the circuit's behavior. The simulator is based on the industry standard SPICE program developed at the University of California at Berkeley and is enhanced and modified to provide mixed-mode, board-level, and system-level simulation capabilities.

The XSPICE Code Model Subsystem described in this document works in conjunction with the XSPICE simulator to provide "code models" and "user-defined node" data types used in simulating circuits and systems.

Several predefined models and node types are delivered with the system. These components of the Code Model Subsystem are referred to as the Code Model Library and the User-Defined Node Library respectively. The following predefined code models are contained in the Code Model Library.

Analog Models:

```
Gain
Summer
Multiplier
Divider
Limiter
Controlled Limiter
Piecewise Linear Controlled Source
Analog Switch
Zener Diode
Current Limiter
Hysteresis Block
Differentiator
Integrator
S-Domain Transfer Function
Slew Rate Block
Inductive Coupling
Magnetic Core
Controlled Sine Wave Oscillator
Controlled Triangle Wave Oscillator
Controlled Square Wave Oscillator
Controlled Oneshot
Capacitor
Capacitance Meter
Inductor
Inductance Meter
```

Hybrid Models:

```
Digital-to-Analog Node Bridge
Analog-to-Digital Node Bridge
Controlled Digital Oscillator
```

Digital Models:

```
Buffer
Inverter
And
Nand
Or
Nor
Xor
Xnor
Tristate
Open-collector Buffer
```

```
Open-Emitter Buffer
Pullup
Pulldown
D Flip Flop
JK Flip Flop
Toggle Flip Flop
Set-Reset Flip Flop
D Latch
Set-Reset Latch
State Machine
Frequency Divider
RAM
Digital Source
```

The following predefined node types are contained in the User-Defined Node Library.

```
Real
Int
```

The set of available code models and node types in the XSPICE simulator can also be modified and extended by a user through the use of the Code Model Subsystem's "Code Model Toolkit". The Code Model Toolkit consists of the Model Directory Generator, the User-Defined Node Directory Generator, the Code Model Preprocessor, and the Simulator Directory Generator utilities. These utilities work with the host computer operating system software (UNIX) to assist the user in the process of creating new models, node types, and customized simulator executables.

1.3 Document Overview

This document describes the design of the XSPICE Code Model Subsystem CSCI (Computer Software Configuration Item). It includes descriptions of the Code Model Toolkit used in adding user-written models and user-defined nodes to the simulator and descriptions of the library of code models and library of node types supplied with the simulator.

Section 2 is an annotated bibliography of associated documents. Section 3 provides a top-level description of the design of each of the components. Section 4 provides a detailed description of each of the components. Section 5 describes data items. Section 6 describes data files. Section 7 provides traceability of the CSCs (Computer Software Components) that comprise the Code Model Subsystem back to the Software Requirements Specification document. Section 8 provides additional notes including a glossary, list of acronyms, and list of project-unique identifiers.

1.4 Acknowledgements

The XSPICE simulator is based on the SPICE3 program developed by the Electronics Research Laboratory, Department of Electrical Engineering and Computer Sciences, University of California at Berkeley.

2

Referenced Documents

1. Software Requirements Specification for the XSPICE Simulator of the Automatic Test Equipment Software Support Environment (ATESSE), F. L. Cox, W. B. Kuhn, J. P. Murray, S. D. Tynor, Georgia Tech Research Institute, Atlanta, GA, September, 1992.
2. Software User's Manual for the XSPICE Simulator of the Automatic Test Equipment Software Support Environment (ATESSE), F. L. Cox, W. B. Kuhn, H. W. Li, J. P. Murray, S. D. Tynor, M. J. Willis, Georgia Tech Research Institute, Atlanta, GA, September, 1992.
3. SPICE3C.1 Nutmeg Programmer's Manual, Department of Electrical Engineering and Computer Sciences, University of California, Berkeley, CA, April, 1987.
4. SPICE3 Version 3C1 User's Guide, Thomas L. Quarles, Department of Electrical Engineering and Computer Sciences, University of California, Berkeley, CA, April, 1989.
5. SPICE 3C1 Nutmeg Programmer's Guide, Department of Electrical Engineering and Computer Sciences, University of California, Berkeley, CA, April, 1989.
6. The Front End to Simulator Interface, Electronics Research Laboratory, College of Engineering, University of California, Berkeley, CA, April, 1989.
7. The SPICE3 Implementation Guide, Thomas L. Quarles, Department of Electrical Engineering and Computer Sciences, University of California, Berkeley, CA, April, 1989.
8. Adding Devices to SPICE3, Thomas L. Quarles, Department of Electrical Engineering and Computer Sciences, University of California, Berkeley, CA, April, 1989.

9. The C Programming Language, Second Edition, Brian Kernighan and Dennis Ritchie, Prentice-Hall, Englewood Cliffs, NJ, 1988.
10. Interface Design Document for the XSPICE Code Model Subsystem of the Automatic Test Equipment Software Support Environment (ATESSE), F. L. Cox, W. B. Kuhn, H. W. Li, J. P. Murray, S. D. Tynor, Georgia Tech Research Institute, Atlanta, GA, September, 1992.
11. Interface Design Document for the XSPICE Simulator of the Automatic Test Equipment Software Support Environment (ATESSE), F. L. Cox, W. B. Kuhn, H. W. Li, J. P. Murray, S. D. Tynor, Georgia Tech Research Institute, Atlanta, GA, September, 1992.
12. Software Design Document for the XSPICE Simulator of the Automatic Test Equipment Software Support Environment (ATESSE), F. L. Cox, W. B. Kuhn, H. W. Li, J. P. Murray, S. D. Tynor, Georgia Tech Research Institute, Atlanta, GA, September, 1992.
13. Program Design Specification (Volumes 1 and 2) for the Automatic Test Equipment Software Support Environment (ATESSE), F. L. Cox, R. M. Ingle, J. E. Doss, G. T. Fulton, A. M. Gilchrist, R. W. Kearney, W. B. Kuhn, D. A. Moreland, P. P. Warren, B. D. Williams, Georgia Tech Research Institute, Atlanta, GA, October 1988.
14. Data Base Design Document for the Automatic Test Equipment Software Support Environment (ATESSE), F. L. Cox, R. M. Ingle, J. E. Doss, G. T. Fulton, A. M. Gilchrist, R. W. Kearney, W. B. Kuhn, D. A. Moreland, P. P. Warren, B. D. Williams, Georgia Tech Research Institute, Atlanta, GA, October 1988.
15. Analysis of Performance and Convergence Issues for Circuit Simulation, Thomas L. Quarles, Department of Electrical Engineering and Computer Sciences, University of California, Berkeley, CA, April, 1989.
16. SPICE2: A Computer Program to Simulate Semiconductor Circuits, Lawrence W. Nagel, Electronics Research Laboratory, College of Engineering, University of California, Berkeley, CA, May, 1975.

3

Top-Level Design

This section describes the top-level design of the XSPICE Code Modeling Subsystem.

3.1 CSCI Overview

Figure 3.1 illustrates the interfaces between the Code Model Subsystem and the rest of the ATESSE system. A user interacts with the Code Model Subsystem through the Simulator Interface (SI) process which provides menus and forms for creating and compiling code models, and which automatically links the simulator according to the code models needed for particular schematics.

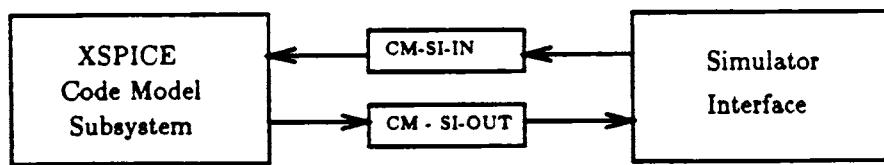


Figure 3.1 Code Model Subsystem External Interfaces.

Code Model Interface Specification files, Code Model Definition files, and User-Defined Node Definition files are created and read by the Simulator Interface editing facilities through the interfaces CM-SI-IN and CM-SI-OUT. The Code Model Subsystem is responsible for processing and compiling these files so that they can be linked with the simulator through interface SIM-CM-IN.

The XSPICE simulator and the Code Model Subsystem can also be used in a stand-alone fashion, independent of the ATESSE Simulator Interface process. In this case, the interfaces CM-SI-IN and CM-SI-OUT are replaced by the operating system command line interface, and text editing facilities.

3.1.1 CSCI Architecture

The Simulator Code Modeling Subsystem works with the XSPICE simulator to provide the simulator with an extensible set of device models over and above the resistors, capacitors, diodes, and transistors found in standard SPICE. The subsystem consists of three major CSCs:

- The Code Model Toolkit. A set of software tools to assist users in developing their own code models and user-defined node types.
- The Code Model Library. A set of prewritten code models handling a wide range of analog and digital functions.
- The User-Defined Node Library. A set of prewritten node types useful in simulating sampled-data systems.

3.1.1.1 Code Model Toolkit

The Code Model Toolkit assists in adding new code models and user-defined node types to the simulator. The process involves three main steps:

1. Create code models in “model directories”.
2. Create node types in “user-defined node directories”.
3. Create a simulator executable in a “simulator directory”.

The Code Model Toolkit facilitates this process by providing four tools, which work in conjunction with an ANSI C language compiler, an associated linker, and the UNIX “make” utility on the target computer system. The tools provided with the toolkit are:

- Model Directory Generator
- User-Defined Node Directory Generator
- Simulator Directory Generator
- Code Model Preprocessor

3.1.1.1.1 Model Directory Generator

The Model Directory Generator (`mkmoddir`) is a tool to create a directory in which a new code model will be written and compiled. A makefile and templates for two files required to define a code model are automatically placed in the new directory.

3.1.1.1.2 User-Defined Node Directory Generator

The User-Defined Node Directory Generator (`mkudndir`) is a tool to create a directory in which a node type will be written and compiled. A makefile and template for the file required to define a node type are automatically placed in the new directory.

3.1.1.1.3 Simulator Directory Generator

The Simulator Directory Generator (`mksimdir`) is a tool to create a directory in which a new simulator will be compiled and linked. A makefile and templates of the files that specify the directory pathnames of desired code models and node types are automatically placed in the new directory.

3.1.1.1.4 Code Model Preprocessor

The Code Model Preprocessor (`cmpp`) tool is run automatically by the UNIX “make” utility in the process of creating a code model or a simulator. For processing code models, this tool takes user written “.ifs” and “.mod” files and turns them into .c files. When used in conjunction with building a new simulator executable, it takes user written “.lst” files and turns them into a collection of include files needed to add the models into the simulator. Note that the Code Model Preprocessor is not used for processing user-defined node types. The standard C language preprocessor is sufficient for this case.

3.1.1.2 Code Model Library

The Code Model Library consists of source code for a set of predefined functional models. Each model resides in a separate directory and includes two source file components:

- An Interface Specification file (`ifspec.ifs`) defining the model’s name, ports, and parameters.
- A C function (`cfunc.mod`) containing the C language code that implements the model’s behavior.

The C function for each model is described in this document. A description of the Interface Specification file for each model may be found in the [Interface Design Document](#) for

the XSPICE Code Model Subsystem of the Automatic Test Equipment Software Support Environment (ATESSE).

3.1.1.3 User-Defined Node Library

The User-Defined Node Library consists of source code for a set of predefined node types. These node types are primarily included as examples. Each type resides in a separate directory and includes a single source file which defines primitive operations on the type.

The C code for each node type is described in this document. A description of the data structure used to interface the node type into the XSPICE simulator may be found in the Interface Design Document for the XSPICE Code Model Subsystem of the Automatic Test Equipment Software Support Environment (ATESSE).

3.1.2 System States and Modes

This section discusses the Code Model Preprocessor only. It is not applicable to the other components of this CSCI.

The Code Model Preprocessor has three modes of operation depending on which of three command-line switches is specified:

```
-ifs      Preprocess an Interface Specification file
-mod     Preprocess a model file
-lst      Preprocess model pathname and user-defined node-list files
```

Only a single command-line switch is allowed per invocation. If the switch is “-ifs”, the Code Model Preprocessor translates an Interface Specification file (ifspec.ifs) into a C language file suitable for compilation. If the switch is “-mod”, the preprocessor translates a model definition file (e.g., cfunc.mod) into a C language file by expanding special “accessor macros”. If the switch is “-lst”, the preprocessor reads files that list the pathnames to models and user-defined node types to be linked into the simulator (modpath.lst and udnpnpath.lst) and then writes files used in compiling and building a new simulator executable.

3.1.3 Memory and Processing Time Allocation

This section discusses the Code Model Preprocessor only. It is not applicable to other components of this CSCI.

Memory is allocated dynamically in the Code Model Preprocessor through “malloc” type storage allocators to allow arbitrary size Interface Specification and model list files to be processed. No attempt is made to free allocated memory within the program. Memory is freed by the operating system upon termination.

Since the Code Model Preprocessor is not a real-time process, processing time allocation is not applicable to this program.

3.2 CSCI Design Description

There are three major components of the Simulator Code Model Subsystem:

- The Code Model Toolkit. A set of software tools to assist users in developing their own code model devices and user-defined node types.
- The Code Model Library. A set of prewritten code models handling a wide range of analog and digital functions.
- The User Defined Node Library. A set of prewritten node types useful in simulating sampled-data systems.

The top-level design of each of these components is covered in a separate section below.

3.2.1 Code Model Toolkit

The Code Model Toolkit consists of four tools designed to assist in adding code models and user-defined node types to the simulator:

- Model Directory Generator (`mkmaddir`)
- User-Defined Node Directory Generator (`mkudndir`)
- Simulator Directory Generator (`mksimdir`)
- Code Model Preprocessor (`cmpp`)

The process of adding code models and user-defined nodes to the simulator with these tools involves three main steps:

1. Create code models in “model directories”.
2. Create node types in “user-defined node directories”.
3. Create a simulator executable in a “simulator directory”.

The complete process of creating a new code model involves the following steps:

1. Run “mkmoddir” to create a directory containing a “make” source file (Makefile) and templates for an “Interface Specification” file (ifspec.ifs) and a “Model Definition File” (cfunc.mod).
2. Move into the newly created directory.
3. Edit the Interface Specification template file to specify the model’s name, ports, parameters, and static variables.
4. Run “make” to preprocess the Interface Specification with “cmpp” and to compile the resulting .c file.
5. Edit the Model Definition File to code the model’s behavior.
6. Run make to preprocess the Model Definition File with cmpp and to compile the resulting .c file.

The process of creating a user-defined node is similar and involves the following steps:

1. Run “mkudndir” to create a directory containing a make source file (Makefile) and a template (udnfunc.c) for the user-defined node definition.
2. Move into the newly created directory.
3. Edit the user-defined node definition template file to code functions that perform basic operations on the type such as structure creation, comparision, copying, etc.
4. Run “make” to compile the definition file. Note that the Code Model Preprocessor (cmpp) is not involved in this step since the file udnfunc.c is a pure C language file, and the macros used are handled by the C preprocessor.

The process of creating a new simulator involves the following steps:

1. Run “mksimdir” to create a directory containing a make source file (Makefile) and a template for a “model pathname” file (modpath.lst) and “user-defined node pathname” file (udnpath.lst) that will identify which models and node types are to be included in the simulator.
2. Move into this new directory.
3. Edit the model list file to indicate the pathnames to directories containing the desired models.
4. Edit the user-defined node list file to indicate the pathnames to directories containing the desired node types.

5. Run “make” to preprocess the list files with cmpp, and create the simulator executable with the C compiler and linker.

The top-level design of the Model Directory Generator, User-Defined Node Directory Generator, Simulator Directory Generator, and the Code Model Preprocessor are described below.

3.2.1.1 Model Directory Generator

The Model Directory Generator (`mkmoddir`) is a UNIX shell script invoked as follows:

```
mkmoddir [directory] [-n spice-model-name] [-c c-function-name]
```

The first argument specifies the name desired for the new directory. The second argument specifies the name to be used for the model on SPICE .model cards. The third argument specifies the name to be used for the code model C function.

The shell script first checks the command line arguments and prompts for any not given. The UNIX “mkdir” command is then run to create the directory. Finally, three files are copied into the new directory using the UNIX “cp” command:

- Makefile
- ifspec.ifs
- cfunc.mod

The UNIX “sed” utility is used to customize the template file for the user-specified code model name.

3.2.1.2 User-Defined Node Directory Generator

The User-Defined Node Directory Generator (`mkudndir`) is a UNIX shell script invoked as follows:

```
mkudndir [directory] [-n node-type-name]
```

The first argument specifies the name desired for the new directory. The second argument specifies the name to be used for the user-defined node type on SPICE instance cards.

The shell script first checks the command line arguments and prompts for any not given. The UNIX “mkdir” command is then run to create the directory. Finally, two files are copied into the new directory using the UNIX “cp” command:

- Makefile
- udnfunc.c

The UNIX “sed” utility is used to customize the template file for the user-specified node type name.

3.2.1.3 Simulator Directory Generator

The Simulator Directory Generator (`mksimdir`) tool takes a single command-line argument specifying the name of the directory to be created:

```
mksimdir [directory]
```

The UNIX “mkdir” command is run to create the directory, and then the following files are copied into the new directory using the UNIX cp command:

- Makefile
- modpath.lst
- udnpath.lst

3.2.1.4 Code Model Preprocessor

The Code Model Preprocessor tool is an executable run automatically by “make” in the process of creating a code model or simulator. This tool takes user-written “.ifs” and “.mod” files and turns them into .c files in the process of creating a model. It also takes user-written “.lst” files and turns them into a collection of include files needed to add the models and user-defined node types into the simulator.

The `cmpp` utility takes one of three command-line switches:

- -ifs
- -mod
- -lst

The `-ifs` switch causes `cmpp` to process the Interface Specification file describing the model’s name, ports, parameters, and static variables (if any). `cmpp` expects to find an Interface Specification file named “`iifspec.ifs`” in the current directory. `cmpp` processes this file to

produce a C file (ifspec.c) that defines C language data structures required by the XSPICE simulator to access information about the model at runtime. A description of the Interface Specification file may be found in the Interface Design Document for the XSPICE Code Model Subsystem of the Automatic Test Equipment Software Support Environment (ATESSE).

The -mod switch causes cmpp to preprocess a model file containing one or more C functions that compute model outputs. If a filename is not specified, cmpp expects to find a model file named "cfunc.mod" in the current directory; otherwise it expects to find a model file named <filename>.mod. In either case, the model file is processed to create a .c file with the same prefix as the source. Model files use special "macros" that provide a simple method to access model inputs and parameters, and to assign model outputs, partial derivatives, AC gain values, etc. cmpp maps these access macros into appropriate C language structure references. A description of the model file format and macros may be found in the Interface Design Document for the XSPICE Code Model Subsystem of the Automatic Test Equipment Software Support Environment (ATESSE).

The -lst switch causes cmpp to preprocess the model and user-defined node-list files that specify pathnames to directories of models and node types to be linked into the simulator. cmpp expects to find files named "modpath.lst" and "udnpath.lst" in the current directory. These files are processed to create four include files (CMextrn.h, CMinfo.h, UDNextrn.h, and UDInfo.h) used to add the specified models and node types to the simulator, plus a make include file (objects.inc) used by the make utility to determine where to find object files that must be linked. After these files are generated, the Makefile in the simulator directory causes the make utility to compile simulator source file SPIinit.c and then link the required object modules.

cmpp -lst does not check to see if the models or node types in the specified directories have been successfully compiled. Before this command is run, the model and node files must exist, have been preprocessed with cmpp -ifs and cmpp -mod, and have been compiled. cmpp -lst will attempt to open the ifspec.ifs file in each model directory listed. Therefore, these files must also exist and must be current.

A description of the list file format may be found in the Interface Design Document for the XSPICE Code Model Subsystem of the Automatic Test Equipment Software Support Environment (ATESSE).

3.2.1.4.1 Code Organization

The major organization of the Code Model Preprocessor is illustrated by the following simplified “call tree”:

```
main

preprocess_ifs_file
    read_ifs_file
        read_ifs_table
            yyparse
    write_ifs_file
        write_comment
        write_includes
        write_mPTable
        write_pTable
        write_conn_info
        write_param_info
        write_inst_var_info
        write_SPICEdev

preprocess_mod_file
    read_ifs_file
        read_ifs_table
    mod_yyparse

preprocess_lst_file
    read_modpath
    read_udnpath
    read_model_names
    read_node_names
    check_uniqueness
    write_CMextrn
    write_CMinfo
    write_UDNextrn
    write_UDNinfo
    write_object_inc
```

The main() function is responsible for processing command-line arguments, checking their validity, and vectoring to one of three routines depending on the argument provided.

If the command-line argument is “-ifs”, function preprocess_ifs_file() is called to read the ifspec.ifs file and convert it to an ifspec.c file. Opening of the .ifs file is performed by read_ifs_file(). Reading and parsing is done by yyparse() (automatically generated by UNIX lex/yacc utilities based on source code in files ifs.lex.l and ifs.yacc.y) which creates an internal representation of the table’s contents in a structure of type “Ifs_Table.t”. Finally, function write_ifs_file() is called to write the ifspec.c file that is compiled and ultimately bound with the simulator. The ifspec.c file is written by several functions, beginning

with a comment at the top (written by `write_comment()`), followed by includes (written by `write_includes()`), and then by a sequence of data structures written by `write_mPTable()`, etc.

If the command-line argument is “-mod”, function `preprocess_mod_file()` is called to read the `cfunc.mod` file and produce a `cfunc.c` file. Optionally, this function will work on any `.mod` file, producing a `.c` file of the same name. In order to perform the required conversion, the information in the `ifspec.ifs` file for the model is first read through a call to `read_ifs_file()`. The `.mod` filename suffix is then converted to a `.c` suffix by `change_extension()`, and then `mod_yyparse()` is called to perform the conversion and write the `.c` file. As in the case of the Interface Specification file preprocessing, function `mod_yyparse()` is automatically created by UNIX lex/yacc utilities based on code in files `mod.lex.l` and `mod.yacc.y`.

If the command-line argument is “-lst”, function `preprocess_lst_file()` is called to read the `modpath.lst` and `udnpath.lst` files and generate five include files needed for compiling and linking the simulator. The file `modpath.lst` contains pathnames to each model to be included in the simulator. The file `udnpath.lst` contains pathnames to each user-defined node type to be included in the simulator. These two files are read by function `read_modpath()` and `read_udnpath()`.

Function `read_model_names()` is then called to get the names of global symbols associated with the data structures that link the model with the simulator core. Each model path is examined for validity, and `read_ifs_file()` is called for each path to get the names of the code model C function and the names of the models from the models’ `ifspec.ifs` files. Similarly, function `read_node_names()` gets the names of the global symbols associated with user-defined nodes from the nodes’ `udnfunc.c` source files.

The names are then checked by `check_uniqueness()` to guarantee that there are no name collisions. If there are collisions, an appropriate error is generated. Otherwise, the names are used in creating files `CMextrn.h`, `CMinfo.h`, `UDNextrn.h`, and `UDNinfo.h` by the corresponding `write....()` functions. Finally, a “make” include file is written by a call to `write_object_inc()`. This file (`object.inc`) defines the locations of the models and user-defined nodes in the file system and is used in linking the model and node object files with the simulator core.

3.2.2 Code Model Library

The following prewritten code models are contained in the Code Model Library. Details of each model’s operation can be found in Section 4 of this document. Interface Specifications for each model are documented in the [Interface Design Document for the XSPICE Code Model Subsystem of the Automatic Test Equipment Software Support Environment \(ATESSE\)](#).

Analog Models:

Gain
Summer
Multiplier
Divider
Limiter
Controlled Limiter
Piecewise Linear Controlled Source
Analog Switch
Zener Diode
Current Limiter
Hysteresis Block
Differentiator
Integrator
S-Domain Transfer Function
Slew Rate Block
Inductive Coupling
Magnetic Core
Controlled Sine Wave Oscillator
Controlled Triangle Wave Oscillator
Controlled Square Wave Oscillator
Controlled Oneshot
Capacitor
Capacitance Meter
Inductor
Inductance Meter

Hybrid Models:

Digital-to-Analog Node Bridge
Analog-to-Digital Node Bridge
Controlled Digital Oscillator

Digital Models:

Buffer
Inverter
And
Nand
Or
Nor
Xor
Xnor
Tristate
Open-collector Buffer
Open-Emitter Buffer
Pullup
Pulldown
D Flip Flop
JK Flip Flop

Toggle Flip Flop
Set-Reset Flip Flop
D Latch
Set-Reset Latch
State Machine
Frequency Divider
RAM
Digital Source

3.2.3 User-Defined Node Library

The following prewritten node types are contained in the User-Defined Node Library. Details of each node type's operation can be found in Section 4 of this document. The data structures that tie these node types into the simulator are documented in the Interface Design Document for the XSPICE Code Model Subsystem of the Automatic Test Equipment Software Support Environment (ATESSE).

Real
Int

4 Detailed Design

This chapter describes the detailed design of the functions that make up the following CSCs (Computer Software Components):

- Code Model Toolkit
- Code Model Library
- User-Defined Node Library

For each CSC, a general description of its functions plus Program Design Language (PDL) pseudo code is given.

4.1 Code Model Toolkit

The Code Model Toolkit (CM-CMT) consists of the following utilities:

- Model Directory Generator (CM-CMT-MDG)
- User-Defined Node Directory Generator (CM-CMT-UDNG)
- Simulator Directory Generator (CM-CMT-SDG)
- Code Model Preprocessor (CM-CMT-CMPP)

4.1.1 Model Directory Generator

Summary

The Model Directory Generator (CM-CMT-MDG) is a UNIX shell script that creates a directory for a new code model and installs a Makefile and template files into the directory.

Called By

The user invokes this shell script from the command line.

Returned Value

Exits with zero to indicate success and non-zero otherwise.

PDL Description

Get arguments from command line, if given.

Prompt for needed names for directory, model, and C function if not given on command line.

Check for valid names.

Create the directory.

Copy in the Makefile and file templates and run 'sed' to set the modname and funcname in the files to the names given by the user.

4.1.2 User-Defined Node Directory Generator

Summary

The User-Defined Node Directory Generator (CM-CMT-UDNG) is a UNIX shell script that creates a directory for a new node type and installs a Makefile and a template file into the directory.

Called By

The user invokes this shell script from the command line.

Returned Value

Exits with zero to indicate success and non-zero otherwise.

PDL Description

Get arguments from command line, if given.

Prompt for needed name for directory, and node type
if not given on command line.

Create the directory.

Copy in the Makefile and file templates and run 'sed' to set the node name
in the file to the name given by the user.

4.1.3 Simulator Directory Generator

Summary

The Simulator Directory Generator (CM-CMT-SDG) is a UNIX shell script that creates a directory for a new simulator and installs a Makefile and template files into the directory.

Called By

The user invokes this shell script from the command line.

Returned Value

Exits with zero to indicate success and non-zero otherwise.

PDL Description

```
Get directory name from command line if given.  
Prompt for directory name if not given on command line.  
Create the directory.  
Copy over the Makefile and file templates.
```

4.1.4 Code Model Preprocessor

The Code Model Preprocessor (CM-CMT-CMPP) is an executable process invoked by the Makefiles in model directories and simulator directories. The following sections detail the design of the functions comprising this process.

4.1.4.1 Function main

Summary

Function main checks the validity of its command-line arguments and then calls one of three major functions as appropriate to perform the processing depending on the type of file:

preprocess_ifs_file	Process a model's Interface Specification File.
preprocess_mod_file	Process a Model Definition File.
preprocess_lst_file	Process model and user-defined node pathname files.

Called By

Makefile in model or simulator directory created with "mkmoddir" or "mksimdir".

Returned Value

Exits with zero to indicate success and non-zero otherwise.

Global Variables

None.

PDL Description

```
main(argc, argv)
int argc;      /* IN - Number of command line arguments */
char *argv[];  /* IN - Command line argument text */
{
    Initialize the error handler with the name of the program found in the
    first element of 'argv'. [init_error()]

    If 'argc' < 2,
        Exit with error. [print_error()]

    If second element of 'argv' is "-ifs",
        If 'argc' is 2,
```

```
Preprocess the Interface Specification File.  
[preprocess_ifs_file()]

Else

    Exit with error. [print_error()]

Else if second element of 'argv' is "-lst",

    If 'argc' is 2,

        Preprocess the model and user-defined node pathname files.  
[preprocess_lst_files()]

    Else

        Exit with error. [print_error()]

    Else if second element of 'argv' is "-mod",

        If 'argc' is 2,

            Preprocess the Model Definition File.  
[preprocess_mod_file()]

        Else

            Exit with error. [print_error()]

    Else,
        Exit with error. [print_error()]

    Exit with success.
}
```

4.1.4.2 Function init_error

Summary

This function sets a global variable ('prog_name') to the string given by the caller. This global variable is used by function 'print_error()' to print the name of the program together with an error message string.

Called By

main() main.c

Returned Value

None.

Global Variables

```
char *prog_name;        /* OUT - Name of program executable */
```

PDL Description

```
void init_error(program_name)

char *program_name;    /* IN - Name of program executable */
{
    Assign 'program_name' to global variable 'prog_name'.
}
```

4.1.4.3 Function print_error

Summary

This function is a simple error handler that writes an error message to the 'stderr' stream together with the name of the program executable.

Called By

main()	main.c
check_uniqueness()	pp_lst.c
read_udnpath()	pp_lst.c
read_ifs_file()	read_ifs.c

Returned Value

None.

Global Variables

```
char *prog_name;      /* IN - Name of program executable */
```

PDL Description

```
void print_error(msg)  
  
char *msg;      /* IN - The message to print */  
{  
    Write the message supplied as argument to the 'stderr' stream  
    prefixed by the name of the executable 'prog_name'.  
}
```

4.1.4.4 Function preprocess_ifs_file

Summary

Function `preprocess_ifs_file` is the top-level driver function for preprocessing an Interface Specification file (`ifspec.ifs`). This function calls `read_ifs_file()` requesting it to read and parse the Interface Specification file and place the information contained in it into a data structure of type `Ifs_Table_t` (See Chapter 5). Then `write_ifs_c_file()` is called to write the information out in a C file suitable for compiling and linking with the simulator.

Called By

`main()` `main.c`

Returned Value

None.

PDL Description

```
void preprocess_ifs_file()
{
    Ifs_Table_t ifs_table; /* Internal representation of IFS file data */

    Read the entire ifspec.ifs file and load the data into 'ifs_table'.
    [read_ifs_file()]

    Write the ifspec.c file required by the XSPICE simulator.
    [write_ifs_c_file()]
}
```

4.1.4.4.1 Function read_ifs_file

Summary

Function `read_ifs_file()` opens the Interface Specification file (`ifspec.ifs`) for read access and calls `read_ifs_table()` with the assigned file pointer to read and parse the file. Upon return from `read_ifs_table()`, the file is closed.

Called By

`preprocess_ifs_file()` pp_ifs.c

Returned Value

Zero on success, and non-zero on failure.

Global Variables

`char *current_filename; /* OUT - Filename being processed */`

PDL Description

```
Status_t read_ifs_file(filename, mode, ifs_table)

    char *filename;          /* IN - File to read */
    int mode;                /* IN - Get names only or get everything? */
    Ifs_Table_t *ifs_table;  /* OUT - Table to put info in */

{
    Open the ifs file 'filename' for read access.

    If error on opening file.

        Issue error message and return. [print_error()]

        Set global 'current_filename' to 'filename' for use by yacc functions.

        Get the information from the file into 'ifs_table'. [read_ifs_table()]

        Close file and return status returned from call to read_ifs_table().
}
```

4.1.4.4.1.1 Function `read_ifs_table`

Summary

Function `read_ifs_table()` calls `yyparse()` to read and parse the Interface Specification file contents and place the information into an internal data structure. Function `yyparse()` is automatically generated by UNIX lex/yacc.

Called By

`read_ifs_file()` `read_ifs.c`

Returned Value

Returns a success status if the IFS file was successfully parsed. Otherwise, returns failure.

Global Variables

```
int yylineno;                        /* OUT - current linenumber */
FILE *yyin;                        /* OUT - input file pointer for the parser */
Boolean_t parser_just_names;    /* OUT - return after the function and SPICE name are parsed */
```

PDL Description

```
static Status_t read_ifs_table(fp, mode, ifs_table)

FILE *fp;                        /* File to read from */
int mode;                        /* Get names only or get everything? */
Ifs_Table_t *ifs_table;        /* Table to put info in */

{
    Initialize 'yylineno' to 1, 'yyin' to 'fp' and 'parser_just_names'
    depending on 'mode'.

    Call yyparse() to parse the Interface Specification file.

    If parse failed, generate an error message [print_error()] and return ERROR,
    otherwise return OK.
}
```

4.1.4.4.1.1.1 Function yyparse

This function is automatically generated by UNIX lex/yacc based on definitions of tokens and grammar in files ifs_lex.l and ifs_yacc.y.

4.1.4.4.1.1.1.1 Interface Specification Lexer

Summary

This section documents the source for the Interface Specification File lexer. The UNIX Lex utility generates the function yylex() from this source description. yylex() is called by the Interface Specification parser to get tokens from the Interface Specification source file.

Called By

yyparse() ifs_yacc.y

Returned Value

Returns the token code of the last token scanned. The global variable 'yytext' holds the text of that token. 'yytext' is only needed if the return value is TOK_IDENTIFIER or TOK_STRING_LITERAL. 'yyval' holds the decoded integer value if the return value is TOK_INT_LITERAL, and 'yddval' holds the decoded double value if the return value is TOK_REAL_LITERAL. In all other cases, the token code is sufficient to describe the token. Returns 0 at end of file.

Global Variables

```
FILE *yyin;       /* IN - file pointer for the source IFS file */
char *yytext;      /* OUT - text of the last token */
double ydval;     /* OUT - real value of the last token */
int yyval;        /* OUT - integer value of the last token */
```

PDL Description

The lexer is a finite state automaton that matches text on the input stream using the regular expressions on the left and executes the corresponding code on the right. Most rules simply return the token code, while some (e.g., string literals) do more processing. The token code constants are derived from the %token directives in the corresponding Yacc source, ifs_yacc.y. Yacc generates a header file, ifs_tok.h, which is referenced by this file.

The state machine has a number of distinct states: 0, CTYPE, BOOL, DTYPE, DIR. It starts in state 0 and changes its state in certain rules in order to allow context sensitive parsing. This allows keywords in one context to be treated as non-keywords in others. Rules that contain a state name in angle braces (e.g., "<BOOL>yes...") are active only when the state machine has been placed in that state. Rules without explicit states are always active.

In the following PDL, the pattern {W} is used to indicate 'white space', {D} is 'decimal digit', {I} is 'alpha numeric', {Z} is 'alpha numeric with underscores', and {E} is 'real exponent'.

```
/*
   {Read until comment terminator "/*"
encountered}

allowed_types{W}*:          {BEGIN CTYPE; return TOK_ALLOWED_TYPES;}
vector{W}*:                   {BEGIN BOOL;  return TOK_ARRAY;}
vector_bounds{W}*:            {return TOK_ARRAY_BOUNDS;}
c_function_name{W}*:          {return TOK_C_FUNCTION_NAME;}
port_name{W}*:                {return TOK_PORT_NAME;}
port_table{W}*:               {return TOK_PORT_TABLE;}
data_type{W}*:                {BEGIN DTYPE; return TOK_DATA_TYPE;}
default_type{W}*:              {BEGIN CTYPE; return TOK_DEFAULT_TYPE;}
default_value{W}*:             {return TOK_DEFAULT_VALUE;}
description{W}*:               {return TOK_DESCRIPTION;}
direction{W}*:                 {BEGIN DIR;  return TOK_DIRECTION;}
static_var_name{W}*:            {return TOK_STATIC_VAR_NAME;}
static_var_table{W}*:           {return TOK_STATIC_VAR_TABLE;}
limits{W}*:                    {return TOK_LIMITS;}
name_table{W}*:                {return TOK_NAME_TABLE;}
null_allowed{W}*:               {BEGIN BOOL; return TOK_NULL_ALLOWED;}
parameter_name{W}*:              {return TOK_PARAMETER_NAME;}
parameter_table{W}*:             {return TOK_PARAMETER_TABLE;}
spice_model_name{W}*:            {return TOK_SPICE_MODEL_NAME;}

<BOOL>yes                  {return TOK_BOOL_YES;}
<BOOL>no                   {return TOK_BOOL_NO;}
true                         {return TOK_BOOL_YES;}
false                        {return TOK_BOOL_NO;}

<CTYPE>v                     {return TOK_CTYPE_V;}
<CTYPE>vd                   {return TOK_CTYPE_VD;}
<CTYPE>vnam                 {return TOK_CTYPE_VNAM;}
<CTYPE>i                     {return TOK_CTYPE_I;}
<CTYPE>id                   {return TOK_CTYPE_ID;}
<CTYPE>g                     {return TOK_CTYPE_G;}
<CTYPE>gd                   {return TOK_CTYPE_GD;}
<CTYPE>h                     {return TOK_CTYPE_H;}
<CTYPE>hd                   {return TOK_CTYPE_HD;}
<CTYPE>d                     {return TOK_CTYPE_D;}
```

```

<DIR>in          {return TOK_DIR_IN;}
<DIR>out         {return TOK_DIR_OUT;}
<DIR>inout       {return TOK_DIR_INOUT;}

<DTYPE>real       {return TOK_DTYPE_REAL;}
<DTYPE>int        {return TOK_DTYPE_INT;}
<DTYPE>boolean    {return TOK_DTYPE_BOOLEAN;}
<DTYPE>complex    {return TOK_DTYPE_COMPLEX;}
<DTYPE>string     {return TOK_DTYPE_STRING;}
<DTYPE>pointer    {return TOK_DTYPE_POINTER;}

"<"              {return TOK_LANGLE;}
">"              {return TOK_RANGLE;}
"["
"]"              {return TOK_RBRACKET;}
","              {return TOK_COMMAS;}
"--"             {return TOK_DASH; }

"\\"              {Read until closing quote terminates string.
return TOK_STRING_LITERAL; }

{I}+{Z}*          {return TOK_IDENTIFIER; }

[+-]?{D}+        {Convert alpha to integer and
                  assign to yyival.
                  return TOK_INT_LITERAL; }

[+-]?{D}+"."{D}*(({E}))? | 
[+-]?{D}+"."{D}+(({E}))? | 
[+-]?{D}+(({E}))?      {Convert alpha to double and
                  assign to yydval.
                  return TOK_REAL_LITERAL; }

.                ; /* ignore anything else */
\n               ; /* ignore anything else */

```

4.1.4.4.1.1.1.2 Interface Specification Parser

Summary

This section documents the grammar used in the yacc source to effect the reading of the Interface Specification file. Data read from the file is saved in appropriate locations in a data structure of type Ifs_Table_t (called TBL in this specification) supplied by the caller of yyparse().

During processing, a large, dynamically sized, item buffer array is used to hold the results of the productions until they are assigned into TBL. This item buffer is an array of unions of all possible types that will be needed.

Called By

read_ifs_table() read_ifs.c

Returned Value

Returns zero to indicate a successful parse. Otherwise returns a non-zero value.

Global Variables

```
double yydval;                    /* IN - real value of the last token */
int yyival;                      /* IN - integer value of the last token */
char *yytext;                    /* IN - text of the last token */
Boolean_t parser_just_names;    /* IN - return as soon as the function and SPICE name
                                  are parsed */

static Boolean_t sav_model_name; /* INOUT - Has the SPICE model name been read yet? */
static Boolean_t sav_function_name; /* INOUT - Has the C function name been read yet? */
static Boolean_t did_default_type; /* INOUT - Has a default type field been read yet? */
static Boolean_t did_allower_type; /* INOUT - Has an allowed type field been read yet? */
static int num_items;            /* INOUT - Number of items in the current row */
static int item;                /* INOUT - Number of the current item */
static int item_offset;        /* INOUT - Offset of the first item in the current row */
static Boolean_t num_items_fixed; /* INOUT - Once a complete row has been parsed all
                                  subsequent rows need to be the same length */
Ifs_Table_t *parser_ifs_table; /* OUT - The parsed IFS table */
```

PDL Description

The PDL description corresponds closely to the Yacc grammar which comprises the actual source text. The 'start' production (the root of the grammar) is 'ifs_file'. Actions embedded

in the productions are delimited by curly braces. Non-terminal productions (e.g., 'ifs_file') are named in lowercase, terminal tokens (e.g., 'TOK_COMMA') are named in all uppercase. In the action code, \$1 is used to refer to the value generated by first non-terminal of the current production, \$2 refers to the second non-terminal. Embedded action code is considered a non-terminal for the purposes of such reference.

```

int yyparse()
{
    ifs_file          : {Allocate and initialize data structures}
                        list_of_tables
                        ;
    list_of_tables     : table
                        | list_of_tables table
                        ;
    table             : TOK_NAME_TABLE
                        {Set table context to TBL_NAME}
                        name_table
                        | TOK_PORT_TABLE
                        {Set table context to TBL_PORT and
                         initialize variables}
                        port_table
                        {Set number of connections in TBL to num_items}
                        | TOK_PARAMETER_TABLE
                        {Set table context to TBL_PARAMETER
                         and initialize variables [INIT()].}
                        parameter_table
                        {Set number of parameters in TBL to num_items}
                        | TOK_STATIC_VAR_TABLE
                        {Set table context to TBL_STATIC_VAR
                         and initialize variables [INIT()].}
                        static_var_table
                        {Set number of inst_vars in TBL to num_items}
                        ;
    name_table         : /* empty */
                        | name_table name_table_item
                        ;
    name_table_item    : TOK_C_FUNCTION_NAME identifier
                        {Set C function name in TBL to yytext.
                         Mark that we have got the function name.
                         If just supposed to get names, and already got
                         model name, return 0}
                        | TOK_SPICE_MODEL_NAME identifier
                        {Set model name in TBL to yytext.
                         Mark that we have got the model name.
                         If just supposed to get names, and already got
                         C function name, return 0}
                        | TOK_DESCRIPTION string

```

```

        {Set description in TBL to yytext.}
;

port_table      : /* empty */
| port_table port_table_item
;

port_table_item : TOK_PORT_NAME list_of_ids
{Check/update count of items in table [check_end_item_num()].
 Copy port names into TBL.}
| TOK_DESCRIPTION list_of_strings
{Check/update count of items in table [check_end_item_num()].
 Copy description strings into TBL.}
| TOK_DIRECTION list_of_directions
{Check/update count of items in table [check_end_item_num()].
 Copy directions into TBL.}
| TOK_DEFAULT_TYPE list_of_ctypes
{Check/update count of items in table [check_end_item_num()].
 Copy default type into TBL.
 If got allowed types, check against
 default type [check_default_type()].}
| TOK_ALLOWED_TYPES list_of_ctype_lists
{Check/update count of items in table [check_end_item_num()].
 Copy the allowed types into TBL [assign_ctype_list()].
 If got default type, check against
 allowed types [check_default_type()].}
| TOK_ARRAY list_of_bool
{Check/update count of items in table [check_end_item_num()].
 Copy the is_array booleans into TBL.}
| TOK_ARRAY_BOUNDS list_of_array_bounds
{Check/update count of items in table [check_end_item_num()].
 Copy the bounds info and has_conn_ref info
 into TBL [ASSIGN_BOUNDS()].}
| TOK_NULL_ALLOWED list_of_bool
{Check/update count of items in table [check_end_item_num()].
 Copy the null_allowed info into TBL.}
;

parameter_table : /* empty */
| parameter_table parameter_table_item
;

parameter_table_item : TOK_PARAMETER_NAME list_of_ids
{Check/update count of items in table [check_end_item_num()].
 Copy the parameter names into TBL.}
| TOK_DESCRIPTION list_of_strings
{Check/update count of items in table [check_end_item_num()].
 Copy the parameter descriptions into TBL.}
| TOK_DATA_TYPE list_of_dtotypes
{Check/update count of items in table [check_end_item_num()].
 Copy the types into TBL and verify they
 are not pointer types [check_dtype_not_pointer()].}
;
```

```

| TOK_DEFAULT_VALUE list_of_values
| {Check/update count of items in table [check_end_item_num()].
|   Set default info in TBL [assign_value()].}
| TOK_LIMITS list_of_ranges
| {Check/update count of items in table [check_end_item_num()].
|   Set limits info in TBL [assign_limits()].}
| TOK_ARRAY list_of_bool
| {Check/update count of items in table [check_end_item_num()].
|   Set is_array info in TBL.}
| TOK_ARRAY_BOUNDS list_of_array_bounds
| {Check/update count of items in table [check_end_item_num()].
|   Set array bounds info in TBL [ASSIGN_BOUNDS()].}
| TOK_NULL_ALLOWED list_of_bool
| {Check/update count of items in table [check_end_item_num()].
|   Set null_allowed info in TBL.}
;

static_var_table      : /* empty */
| static_var_table static_var_table_item
;

static_var_table_item : TOK_STATIC_VAR_NAME list_of_ids
| {Check/update count of items in table [check_end_item_num()].
|   Set inst_var name info in TBL.}
| TOK_DESCRIPTION list_of_strings
| {Check/update count of items in table [check_end_item_num()].
|   Set inst_var description info in TBL.}
| TOK_DATA_TYPE list_of_dtotypes
| {Check/update count of items in table [check_end_item_num()].
|   Set inst_var type info in TBL.}
| TOK_ARRAY list_of_bool
| {Check/update count of items in table [check_end_item_num()].
|   Set inst_var is_array info in TBL.}
;

list_of_ids           : /* empty */
| list_of_ids identifier
| {Check/increase size of TBL struct arrays.
|   Increment item count [check_item_num()].
|   Record identifier in item buffer.}
;

list_of_array_bounds : /* empty */
| list_of_array_bounds int_range
| {Check/increase size of TBL struct arrays.
|   Increment item count [check_item_num()].
|   Record range in item buffer.}
| list_of_array_bounds identifier
| {Check/increase size of TBL struct arrays.
|   Increment item count [check_item_num()].
|   Record bounds info in item buffer.}
;

```

```
list_of_strings      : /* empty */
| list_of_strings string
{Check/increase size of TBL struct arrays.
 Increment item count [check_item_num()].
 Record strings in item buffer.}
;

list_of_directions : /* empty */
| list_of_directions direction
{Check/increase size of TBL struct arrays.
 Increment item count [check_item_num()].
 Record direction info in item buffer.}
;

direction          : TOK_DIR_IN
{Set value of production result to IN}
| TOK_DIR_OUT
{Set value of production result to OUT}
| TOK_DIR_INOUT
{Set value of production result to INOUT}
;

list_of_bool         : /* empty */
| list_of_bool bool
{Check/increase size of TBL struct arrays.
 Increment item count [check_item_num()].
 Record booleans in item buffer.}
;

list_of_ctypes       : /* empty */
| list_of_ctypes ctype
{Check/increase size of TBL struct arrays.
 Increment item count [check_item_num()].
 Record ctypes info in item buffer.}
;

ctype               : TOK_CTYPE_V
{Set value of production result's kind field
 to VOLTAGE}
| TOK_CTYPE_VD
{Set value of production result's kind field
 to DIFF_VOLTAGE}
| TOK_CTYPE_VNAM
{Set value of production result's kind field
 to VSOURCE_CURRENT}
| TOK_CTYPE_I
{Set value of production result's kind field
 to CURRENT}
| TOK_CTYPE_ID
{Set value of production result's kind field
 to DIFF_CURRENT}
```

```

| TOK_CTYPE_G
{Set value of production result's kind field
 to CONDUCTANCE}
| TOK_CTYPE_GD
{Set value of production result's kind field
 to DIFF_CONDUCTANCE}
| TOK_CTYPE_H
{Set value of production result's kind field
 to RESISTANCE}
| TOK_CTYPE_HD
{Set value of production result's kind field
 to DIFF_RESISTANCE}
| TOK_CTYPE_D
{Set value of production result's kind field
 to DIGITAL}
| identifier
{Set value of production result's kind field
 to USER_DEFINED. and id field to identifier}
;

list_of_dtypes : /* empty */
| list_of_dtypes dtype
{Check/increase size of TBL struct arrays.
 Increment item count [check_item_num()].
 Record dtypes in item buffer.}
;

dtype : TOK_DTYPE_REAL
{Set value of production result to REAL}
| TOK_DTYPE_INT
{Set value of production result to INTEGER}
| TOK_DTYPE_BOOLEAN
{Set value of production result to BOOLEAN}
| TOK_DTYPE_COMPLEX
{Set value of production result to COMPLEX}
| TOK_DTYPE_STRING
{Set value of production result to POINTER}
| TOK_DTYPE_POINTER
{Set value of production result to REAL}
;

list_of_ranges : /* empty */
| list_of_ranges range
{Check/increase size of TBL struct arrays.
 Increment item count [check_item_num()].
 Record ranges in item buffer.}
;

int_range : TOK_DASH
{Set value of production result.is_named to FALSE.
 Set has_bounds in production result to FALSE.}
| TOK_LBRACKET int_or_dash maybe_comma int_or_dash
;

```

```

TOK_RBRACKET
{Set value of production.is_named to FALSE.
 Set bounds info in production result.}
;

maybe_comma
: /* empty */
| TOK_COMMA
;

int_or_dash
: TOK_DASH
{Set value of production.has_bound to FALSE.}
| integer_value
{Set value of production.has_bound to TRUE
 and record bound value in production result.}
;

range
: TOK_DASH
{Set value of production result is_named field
 to FALSE, and has bounds fields to FALSE.}
| TOK_LBRACKET number_or_dash maybe_comma
number_or_dash TOK_RBRACKET
{Set value of production result's is_named
 field to FALSE, and record the number_or_dash
 results in the bounds fields.}
;

number_or_dash
: TOK_DASH
{Set value of production result's has_bound field
 to FALSE.}
| number
{Set value of production result's has_bound field
 to TRUE, and record the number in the bound field}
;

list_of_values
: /* empty */
| list_of_values value_or_dash
{Check/increase size of TBL struct arrays.
 Increment item count [check_item_num()].
 Record value_or_dash result in item buffer.}
;

value_or_dash
: TOK_DASH
{Set value of production result's has_value
 field to FALSE}
| value
;

value
: string
{Set value of production result's has_value
 field to TRUE, kind to STRING, and u.bavalue
 to the string production result}
| bool
;

```

```

        {Set value of production result's has_value
         field to TRUE, kind to BOOLEAN, and u.bavalue
         to the bool production result}
| complex
{Set value of production result's has_value
 field to TRUE, kind to COMPLEX, and u.bavalue
 to the complex production result}
| number
;

complex : TOK_LANGLE real maybe_comma real TOK_RANGLE
{Set value of production result's real field
 to the first real production result, and imag
 field to the second real production result}
;

list_of_ctype_lists : /* empty */
| list_of_ctype_lists delimited_ctype_list
{Check/increase size of TBL struct arrays.
 Increment item count [check_item_num()].
 Record delimited_ctype_list result in item buffer.}
;

delimited_ctype_list : TOK_LBRACKET ctype_list TOK_RBRACKET
{Assign value of ctype_list production to production
 result.}
;

ctype_list : ctype
{Allocate a linked list element and assign to
 production result.
 Store ctype in element and set next to NULL.}
| ctype_list maybe_comma ctype
{Allocate a linked list element and assign to
 production result.
 Store ctype in element and set next to ctype_list.}
;

bool : TOK_BOOL_YES {Assign TRUE to production result}
| TOK_BOOL_NO {Assign FALSE to production result}
;

string : TOK_STRING_LITERAL {Copy 'yytext' to production result}
;

identifier : TOK_IDENTIFIER {Copy 'yytext' to production result}
;

number : real
{Set value of production result has_value
 field to TRUE, kind field to REAL, and ivalue
 field to the real value}
;
```

```
| integer_value
;

integer_value      : integer
                    {Set value of production results has_value
                     field to TRUE, kind field to INTEGER, and ivalue
                     field to the integer value}
;

real               : TOK_REAL_LITERAL {Assign 'yydval' to production result}
;

integer            : TOK_INT_LITERAL {Assign 'yyival' to production result}
;
}
```

4.1.4.4.1.1.2.1 Static Function `assign_ctype_list`

Summary

Copies the parsed list of allowed types, 'ctype_list', into the `allowed_types` list in the specified connection description, 'conn'.

Called By

`yyparse()` `ifs_yacc.c`

Returned Value

None.

PDL Description

```
static void assign_ctype_list (conn, ctype_list)

Conn_Info_t *conn; /* IN - pointer to the destination connection descriptor */
Ctype_List_t *ctype_list; /* IN - parsed list of types */
{
    Count the number of types in 'ctype_list'.
    Allocate an array of that size for 'conn's allowed_type list.
    Allocate an array of that size for 'conn's allowed_port_type list.
    For each element in 'ctype_list'
        Check that each element is compatible with the others [get_class_type()].
        Check that the type is compatible with the 'conn's direction
        [check_port_type_direction()].
        Assign the type's kind field to the appropriate element in conn->allowed_port_type
        Assign the type's id field to the appropriate element in conn->allowed_type
}
```

4.1.4.4.1.1.2.2 Static Function assign_limits

Summary

Assign the fields of 'range' into the appropriate fields of 'param'.

Called By

yyparse() ifa_yacc.c

Returned Value

None.

PDL Description

```
static void assign_limits (type, param, range)
Data_Type_t type; /* IN - The type of the range (INTEGER, REAL) */
Param_Info_t *param; /* IN - The parameter whose limits are the target */
Range_t range; /* IN - The source range */
{
    Ensure that 'range' is not a named range [yyerror()];
    Assign the fields of 'range' into the appropriate fields of 'param' [assign_value()];
}
```

4.1.4.4.1.1.2.3 Static Function assign_value

Summary

Assign a value and check that the data types are compatible.

Called By

```
assign_limits()      ifs_yacc.c
yparse()           ifs_yacc.c
```

Returned Value

None.

PDL Description

```
static void assign_value (type, dest_value, src_value)

Data_Type_t type;          /* IN - The type of the source value (INTEGER, REAL) */
Value_t    *dest_value;    /* IN - Pointer to the target of the assignment */
My_Value_t src_value;     /* IN - The source value */
{
    Check that the type of 'src_value' is compatible with the type of 'dest_value'
    [yyerror()];
    Depending on the 'type', assign the appropriate field from
    'src_value' into the corresponding field of 'dest_value'.
}
```

4.1.4.4.1.1.2.4 Static Function check_default_type

Summary

Check that the given connection's default type is a member of that connection's allowed type set.

Called By

yyparse() ifs_yacc.c

Returned Value

None.

Global Variables

Ifs_Table_t *parser_ifs_table; /* OUT - The parsed IFS table */

PDL Description

```
static void check_default_type (conn)
{
    Conn_Info_t conn; /* IN - the connection to be checked */
    If the default type is a not member of the allowed_type set,
    generate an error message [yyerror()].
```

4.1.4.4.1.1.1.2.5 Static Function `check_dtype_not_pointer`

Summary

Check that the given data type, 'dtype', is not a POINTER.

Called By

`yyparse()` `ifs_yacc.c`

Returned Value

None.

PDL Description

```
static void check_dtype_not_pointer (dtype)
Data_Type_t dtype; /* IN - the type to be checked */
{
    If 'dtype' is a POINTER, generate an error message [yyerror()].
```

4.1.4.4.1.1.2.6 Static Function check_end_item_num

Summary

Called at the end of a row of items in a table, this function checks that the proper number of items were parsed, or if no proper number has yet been determined, sets the proper number to the current number.

Called By

yyparse() ifs_yacc.c

Returned Value

None.

Global Variables

```
Ifs_Table_t *parser_ifs_table;        /* OUT - The parsed IFS table */  
static int num_items;                /* INOUT - Number of items in the current row */  
static int item;                    /* IN - Number of the current item */  
static Boolean_t num_items_fixed;    /* INOUT - Once a complete row has been parsed all
```

PDL Description

```
static void check_end_item_num ()  
  
{  
    If 'num_items_fixed', then check that the current number of items is  
    the expected number [yyerror()].  
  
    Otherwise, set the expected number to the current number, set  
    'num_items_fixed' to TRUE, and set the size of the current table to that size.  
}
```

4.1.4.4.1.1.2.7 Static Function check_item_num

Summary

Check that the current table is large enough for one more item – grow the table if it is not.

Called By

yyparse() ifs_yacc.c

Returned Value

None.

Global Variables

```
Ifs_Table_t *parser_ifs_table;       /* OUT - The parsed IFS table */  
static int item;                        /* IN - Number of the current item */  
static int item_offset;                /* IN - Offset of the first item in the current row */  
static int allocated_size[4];        /* INOUT - the allocated size of each table */
```

PDL Description

```
static void check_item_num ()  
  
{  
    If there are too many items for the per-row item buffer, generate an error [yyerror()].  
    Otherwise, if the current table is too small, grow it.  
}
```

4.1.4.4.1.1.1.2.8 Static Function check_port_type_direction

Summary

Check that 'port_type' is compatible with the direction 'dir'.

Called By

assign_ctype_list ifs_yacc.c

Returned Value

None.

PDL Description

```
static void check_port_type_direction (dir, port_type)
{
    Dir_t dir;          /* IN - The desired direction */
    Port_Type_t port_type; /* IN - The port's type (VOLTAGE, CURRENT, etc.) */
    {
        If 'dir' is not compatible with 'port_type', generate an error [yyerror()].
    }
}
```

4.1.4.4.1.1.2.9 Static Function `find_conn_ref`

Summary

Return the connection index of the connection named 'name'.

Called By

`ASSIGN_BOUNDS` `ifs_yacc.c`

Returned Value

Returns the connection index of the connection named 'name'.

Global Variables

`Ifs_Table_t *parser_ifs_table; /* OUT - The parsed IFS table */`

PDL Description

```
static void find_conn_ref (name)

char *name; /* IN - the name of the connection */
{
    Loop over all connections and compare names. If a name matches
    'name', return than index.

    Otherwise, generate an error message [yyerror()].
}
```

4.1.4.4.1.1.2.10 Static Function get_ctype_class

Summary

Return the connection type class (POINTER, BOOLEAN, DOUBLE) based on 'type' (VOLTAGE, DIGITAL, etc.).

Called By

assign_ctype_list ifs_yacc.c

Returned Value

Returns the computed connection type class.

PDL Description

```
static void get_ctype_class (type)
{
    Port_type_t type; /* IN - the type */
    If type is USER_DEFINED, return POINTER.
    If type is DIGITAL, return BOOLEAN.
    Otherwise, return DOUBLE.
}
```

4.1.4.4.1.1.2.11 Static Macro ASSIGN_BOUNDS

Summary

Assigns the range at index 'i' in the per-row item buffer into the table named 'struct_name' in the parser table.

Called By

`yyparse()` `ifs_yacc.c`

Returned Value

None.

Global Variables

`Ifs_Table_t *parser_ifs_table; /* OUT - The parsed IFS table */`

PDL Description

```
#define ASSIGN_BOUNDS (struct_name, i)

C-Identifier struct_name; /* IN - the name of the structure comprising the table */
int i;                    /* IN - the offset in the pre-row buffer */
{
    Assigns the range at index 'i' in the per-row item buffer into the table
    named 'struct_name' in the parser table. If the range is a named
    range, dereference the name to the appropriate connection [find_conn_ref()].
}
```

4.1.4.4.1.1.2.12 Static Macro INIT

Summary

Initialize the per-row item buffer. Called at the beginning of a table.

Called By

yyparse() ifs_yacc.c

Returned Value

None.

Global Variables

```
static int item_offset;                /* OUT - Offset of the first item in the current row */
static int num_items;                 /* OUT - Number of items in the current row */
static int item;                      /* OUT - Number of the current item */
static Boolean_t num_items_fixed;    /* OUT - Once a complete row has been parsed all
```

PDL Description

```
#define INIT (n)

int n; /* IN - offset of the first item in the row */
{
  Initialize 'item', 'item_offset' and 'num_items' to 'n'. Set
  'num_items+fixed' to FALSE.
}
```

4.1.4.4.1.1.2.13 Function yyerror

Summary

Writes an error message, 'str', to the standard error stream including the name of name of the file being parsed and the last line parsed and the text of the last token.

Called By

```
yyparse()           ifs_yacc.y
Most auxiliary functions in ifs_yacc.y
```

Returned Value

None.

Global Variables

```
char *prog_name;          /* IN - name of current program */
static char *current_filename; /* IN - name of current file */
char *yytext;              /* IN - current token */
int yylineno;              /* IN - current linenumber */
```

PDL Description

```
int yyerror (str)

char *str;
{
    Write an error message, 'str', to the standard error stream including
    the name of name of the file being parsed and the last line parsed and
    the text of the last token.
}
```

4.1.4.4.1.1.2.14 Function yywrap

Summary

Mandatory function called by yyparse() and mod_yyparse() at the end of file.

Called By

yyparse()	ifa_yacc.y
mod_yyparse()	mod_yacc.y

Returned Value

Returns 1 to indicate that it is OK to finish parsing.

PDL Description

```
int yywrap ()  
{  
    Returns 1.  
}
```

4.1.4.4.2 Function write_ifs_c_file

Summary

Function `write_ifs_c_file` is a top-level driver function for creating a C file (`ifspec.c`) that defines the model Interface Specification in a form usable by the simulator. The `ifspec.c` output file is opened for writing, and then each of the following functions is called in order to write the necessary statements and data structures into the file:

```
write_comment
write_includes
write_mPTable
write_conn_info
write_param_info
write_inst_var_info
write_SPICEdev
```

The output file is then closed.

Called By

```
preprocess_ifs_file()          pp_ifs.c
```

Returned Value

Zero on success, and non-zero on failure.

PDL Description

```
status_t write_ifs_c_file(filename, ifs_table)
{
    char *filename;           /* IN - File to write to */
    Ifs_Table_t *ifs_table;   /* IN - Table of Interface Specification data */
    {
        Open the ifspec.c file for write access.

        Write out a comment section at the top of the file.
        [write_comment()]

        Put in the # includes.
        [write_includes()]
    }
}
```

Write the SPICE3 required XXXmPTable structure.
[write_mPTable()]

Write the SPICE3 required XXXpTable structure.
[write_pTable()]

Write out the connector table required for the code model element parser.
[write_conn_info()]

Write out the parameter table required for the code model element parser.
[write_param_info()]

Write out the instance variable table required for the code model element parser.
write_inst_var_info()

Write out the information structure for this model.
[write_SPICEdev()]

Close the ifspec.c file and return.

}

4.1.4.4.2.1 Function write_comment

Summary

Function write_comment places a comment at the top of the ifspec.c file warning the user that this file is automatically generated and should not be edited.

Called By

write_ifs_c_file() writ_ifs.c

Returned Value

None.

PDL Description

```
static void write_comment(fp, ifs_table)

FILE      *fp;          /* IN - File to write to */
Ifs_Table_t *ifs_table; /* IN - Table of Interface Specification data */

{
    Write a comment header at the top of the file indicating that the
    file was automatically generated and should not be edited.
}
```

4.1.4.4.2.2 Function write_includes

Summary

Function `write_includes` writes the C header files required in `ifspec.c`.

Called By

`write_ifs_c_file()` `writ_ifs.c`

Returned Value

None.

PDL Description

```
static void write_includes(fp)
FILE      *fp;          /* IN - File to write to */
{
    Write the #include files needed.
}
```

4.1.4.4.2.3 Function write_mPTable

Summary

Function `write_mPTable` writes the model parameter information using SPICE's IFparm structure type. This table defines the parameters to be input/output variables using SPICE's "IOP" macro so that these variables can be set or queried from SPICE. These model parameters are derived from the Interface Specification's PARAMETER table.

Called By

`writes_ifs_c_file()` `writ_ifs.c`

Returned Value

None.

PDL Description

```
static void write_mPTable(fp, ifs_table)

FILE      *fp;          /* IN - File to write to */
Ifs_Table_t *ifs_table; /* IN - Table of Interface Specification data */

{
    If 'num_param' member of 'ifs_table' indicates there were no
    parameters defined in the IFS file, just return.

    Write the structure beginning.

    Write out an entry for each parameter in the table.

    Finish off the structure.
}
```

4.1.4.4.2.4 Function write_pTable

Summary

Function `write_pTable` writes the instance parameter information using SPICE's IFparm structure type. This table defines the parameters as output-only variables using SPICE's "OP" macro. These instance parameters are derived from the Interface Specification file's STATIC_VAR table and define the parameters that can be queried using the SPICE 3C1 .save feature.

Called By

`write_ifs_c_file()` `writ_ifs.c`

Returned Value

None.

PDL Description

```
static void write_pTable(fp, ifs_table)

FILE      *fp;          /* IN - File to write to */
Ifs_Table_t *ifs_table; /* IN - Table of Interface Specification data */

{
    If 'num_inst_var' member of 'ifs_table' indicates there were no
    static instance variables in the IFS file, just return.

    Write the structure beginning.

    Write out an entry for each parameter in the table.

    Finish off the structure.
}
```

4.1.4.4.2.5 Function write_conn_info

Summary

Function write_conn_info writes information used by the Simulator's new MIF package to interpret and error check a model's connection list in a SPICE deck. This information is derived from the Interface Specification file's PORT table.

Called By

write_ifs_c_file() writ_ifs.c

Returned Value

None.

PDL Description

```
static void write_conn_info(fp, ifs_table)
{
    FILE      *fp;          /* IN - File to write to */
    Ifs_Table_t *ifs_table; /* IN - Table of Interface Specification data */
{
    If 'num_conn' member of 'ifs_table' indicates there were no
    port connections defined in the IFS file, just return.

    Loop through number of connections.

        Write out structure for port type.

        Write out structure for port type string.

        Now write the main connTable structure.

        Finish off the structure.
}
```

4.1.4.4.2.6 Function write_param_info

Summary

Function `write_param_info` writes information used by the Simulator's new MIF package to interpret and error check a model's parameter list in a SPICE deck. This information is derived from the Interface Specification file's PARAMETER table. It is essentially a superset of the IFparm information written by `write_mPTable()`. The IFparm information written by `write_mPTable()` is required to work with SPICE's device set and query functions. The information written by `write_param_info` is more extensive and is required to parse and error check the SPICE input deck.

Called By

`write_ifs_c_file()` `writ_ifs.c`

Returned Value

None.

PDL Description

```
static void write_param_info(fp, ifs_table)

FILE      *fp;          /* IN - File to write to */
Ifs_Table_t *ifs_table; /* IN - Table of Interface Specification data */

{
    If 'num_param' member of 'ifs_table' indicates there were no
    parameters defined in the IFS file, just return.

    Write the structure beginning.

    Write out an entry for each parameter in the table.

    Finish off the structure.
}
```

4.1.4.4.2.7 Function write_inst_var_info

Summary

Function `write_inst_var_info` writes information used by the Simulator's new MIF package to allocate space for, and to output (using SPICE3's `.save` feature), variables defined in the Interface Specification file's STATIC_VAR table. It is essentially a superset of the IFparm information written by `write_mPTable()`. The IFparm information written by `write_pTable()` is required to work with SPICE's device query functions. The information written by `write_inst_var_info` is more extensive.

Called By

`write_ifs_c_file()` `writ_ifs.c`

Returned Value

None.

PDL Description

```
static void write_inst_var_info(fp, ifs_table)

FILE      *fp;          /* IN - File to write to */
Ifs_Table_t *ifs_table; /* IN - Table of Interface Specification data */

{
    If 'num_inst_var' member of 'ifs_table' indicates there were no
    static instance variables defined in the IFS file, just return.

    Write the structure beginning.

    Write out an entry for each parameter in the table.
    Finish off the structure.
}
```

4.1.4.4.2.8 Function write_SPICEdev

Summary

Function `write_SPICEdev` writes the global `XXX_info` structure used by SPICE to define a model. Here “`XXX`” is the name of the code model as explained in the Software Design Document for the Simulator of the Automatic Test Equipment Software Support Environment (ATESSE). This structure contains the name of the model, a pointer to the C function that implements the model, and pointers to all of the data structures created by the functions described in the preceding sections.

Called By

`write_ifs_c_file()` `writ_ifs.c`

Returned Value

None.

PDL Description

```
static void write_SPICEdev(fp, ifs_table)

FILE      *fp;          /* IN - File to write to */
Ifs_Table_t *ifs_table; /* IN - Table of Interface Specification data */
{
    Write out the structure beginning.

    Write the IFdevice structure.

    Write the function pointer entries.

    Write the sizeof info used in dynamic allocation of instance and model
    structures in SPICE.
}
```

4.1.4.5 Function preprocess_mod_file

Summary

Function preprocess_mod_file is the top-level driver function for preprocessing a code model file (cfunc.mod). This function calls read_ifs_file() requesting it to read and parse the Interface Specification file (ifspec.ifs) and place the information contained in it into an internal data structure. It then calls mod_yyparse() to read the cfunc.mod file and translate it according to the Interface Specification information. Function mod_yyparse() is automatically generated by UNIX lex/yacc utilities.

Called By

main() main.c

Returned Value

None.

Global Variables

```
char *current_filename;        /* OUT - Filename being processed */
```

PDL Description

```
void preprocess_mod_file(filename)
{
    char *filename;        /* The file to read */
    Ifs_Table_t ifs_table;    /* Internal representation of IFS file data */

    Read the entire ifspec.ifs file and load the data into 'ifs_table'.
    [read_ifs_file()]

    Open the .mod file specified in 'filename'.

    Set global 'current_filename' to 'filename' for use by yacc functions.

    Change the extension on 'filename' to .c and open the output file.

    Process .mod file.
    [mod_yyparse()]
}
```

4.1.4.5.1 Function mod_yyparse

This function is automatically generated by UNIX lex/yacc based on definitions of tokens and grammar in files mod_lex.l and mod_yacc.y.

4.1.4.5.1.1 Model Definition Lexer

Summary

This section documents the source for .l Model Definition File lexer. The UNIX Lex utility generates the function mod_yylex() from this source description. mod_yylex() is called by the Model Definition parser to get tokens from the Model Definition source file.

Called By

mod_yyparse() mod_yacc.y

Returned Value

Returns the token code of the last token scanned. The global variable 'yytext' holds the text of that token. 'yytext' is only needed if the return value is TOK_IDENTIFIER or TOK_MISC_C. In all other cases, the token code is sufficient to describe the token. Returns 0 at end of file.

Global Variables

```
FILE *yyin; /* IN - file pointer for the source .mod file */
char *yytext; /* OUT - text of the last token */
```

PDL Description

The lexer is a finite state automaton that matches text on the input stream using the regular expressions on the left and executes the corresponding code on the right. Most rules simply return the token code, while some (e.g., comments) do more processing. The token code constants are derived from the %token directives in the corresponding Yacc source, mod_yacc.y. Yacc generates a header file, mod_tok.h, which is referenced by this file.

In the following PDL, the pattern {I} is used to indicate 'alpha numeric' and {Z} is 'alpha numeric with underscores'.

```

int mod_yylex()
{
    /*"          {Read and echo the comment text until
                   closing "/*" found}

    ARGS          {return TOK_ARGS;}
    INIT          {return TOK_INIT;}
    ANALYSIS      {return TOK_ANALYSIS;}
    NEW_TIMEPOINT {return TOK_NEW_TIMEPOINT;}
    CALL_TYPE     {return TOK_CALL_TYPE;}
    TIME          {return TOK_TIME;}
    RAD_FREQ      {return TOK_RAD_FREQ;}
    TEMPERATURE   {return TOK_TEMPERATURE;}
    T              {return TOK_T;}
    LOAD          {return TOK_LOAD;}
    TOTAL_LOAD    {return TOK_TOTAL_LOAD;}
    MESSAGE        {return TOK_MESSAGE;}
    PARAM          {return TOK_PARAM;}
    PARAM_SIZE    {return TOK_PARAM_SIZE;}
    PARAM_NULL    {return TOK_PARAM_NULL;}
    PORT_SIZE     {return TOK_PORT_SIZE;}
    PORT_NULL     {return TOK_PORT_NULL;}
    PARTIAL        {return TOK_PARTIAL;}
    AC_GAIN        {return TOK_AC_GAIN;}
    OUTPUT_DELAY   {return TOK_OUTPUT_DELAY;}
    STATIC_VAR    {return TOK_STATIC_VAR;}
    STATIC_VAR_SIZE {return TOK_STATIC_VAR_SIZE;}
    INPUT          {return TOK_INPUT;}
    INPUT_STATE    {return TOK_INPUT_STATE;}
    INPUT_TYPE     {return TOK_INPUT_TYPE;}
    INPUT_STRENGTH {return TOK_INPUT_STRENGTH;}
    OUTPUT         {return TOK_OUTPUT;}
    OUTPUT_STATE   {return TOK_OUTPUT_STATE;}
    OUTPUT_STRENGTH {return TOK_OUTPUT_STRENGTH;}
    OUTPUT_TYPE    {return TOK_OUTPUT_TYPE;}
    OUTPUT_CHANGED {return TOK_OUTPUT_CHANGED;}

    "("           {return TOK_LPAREN;}
    ")"           {return TOK_RPAREN;}
    "["           {return TOK_LBRACKET;}
    "]"           {return TOK_RBRACKET;}
    ","           {return TOK_COMMA;}

    {I}+{Z}*      {return TOK_IDENTIFIER;}
    [ \t]          {Echo the tab.}
    \n            {Echo the newline.}
    .             {return TOK_MISC_C;}
}

```

4.1.4.5.1.2 Model Definition Parser/Translator

Summary

This section documents the grammar description used to effect the translation of the C Function Definition file (cfunc.mod) into a .c file. The UNIX Yacc utility generates the function mod_yyparse() from this description.

The Model Definition Parser translates model accessor macros into appropriate C language code. During the translation, a string buffer is used to build up long expressions from embedded macros. At the appropriate time, this buffer is written to the C file being created.

For clarity, statements, such as printf(), are included in the grammar description when their use is more concise than a PDL description of the actions. The output device for such statements is usually 'yyout' which is the translated C file being created. The text read by the lexical analyzer is 'yytext' and is often copied directly to the output with fputs().

Called By

preprocess_mod_file() pp_mod.c

Returned Value

Returns zero to indicate a successful parse. Otherwise returns a non-zero value.

Global Variables

```
FILE        *yyout;                    /* IN - file pointer for the translated C file */
char        *yytext;                    /* IN - text of the last token */
static char buffer[3000];            /* INOUT - string buffer for storing embedded C code */
```

PDL Description

The PDL description corresponds closely to the Yacc grammar which comprises the actual source text. The 'start' production (the root of the grammar) is 'mod_file'. Actions embedded in the productions are delimited by curly braces. Non-terminal productions (e.g., 'mod_file') are named in lowercase, terminal tokens (e.g., 'TOK_COMMA') are named in all uppercase. In the action code, \$1 is used to refer to the value generated by first non-terminal of the current production, \$2 refers to the second non-terminal. Embedded action code is considered a non-terminal for the purposes of such reference.

```

int mod_yyparse()
{
    mod_file      : /* empty */
    | mod_file c_code
    ;

    c_code        : /* empty */
    | c_code c_char
    | c_code macro
    ;

    buffered_c_code : {Initialize string 'buffer' to NULL [init_buffer()].}
    buffered_c_code2
    {Copy string 'buffer' to production result.}
    ;

    buffered_c_code2 : /* empty */
    | buffered_c_code2 buffered_c_char
    ;

    buffered_c_char : TOK_IDENTIFIER {Append 'yytext' to string 'buffer' [append().]}
    | TOK_MISC_C {Append 'yytext' to string 'buffer' [append().]}
    | TOK_COMMAS {Append 'yytext' to string 'buffer' [append().]}
    | TOK_LBRACKET
        {Append "[" to string 'buffer' [append().]}
        buffered_c_code2 TOK_RBRACKET
        {Append "]" to string 'buffer' [append().]}
    | TOK_LPAREN
        {Append "(" to string 'buffer' [append().]}
        buffered_c_code2 TOK_RPAREN
        {Append ")" to string 'buffer' [append().]}
    ;

    c_char         : TOK_IDENTIFIER {fputs (yytext, yyout);}
    | TOK_MISC_C {fputs (yytext, yyout);}
    | TOK_COMMAS {fputs (yytext, yyout);}
    | TOK_LBRACKET
        {putc ('[', yyout);}
        c_code TOK_RBRACKET
        {putc (']', yyout);}
    | TOK_LPAREN
        {putc ('(', yyout);}
        c_code TOK_RPAREN
        {putc (')', yyout);}
    ;

    macro          : TOK_INIT
        {fprintf (yyout, "private->circuit.init");}
    | TOK_ARGS
        {fprintf (yyout, "Mif_Private_t *private");}
    | TOK_ANALYSIS
        {fprintf (yyout, "private->circuit.anal_type");}

```

```

| TOK_NEW_TIMEPOINT
|     {fprintf (yyout, "private->circuit.anal_init");}
| TOK_CALL_TYPE
|     {fprintf (yyout, "private->circuit.call_type");}
| TOK_TIME
|     {fprintf (yyout, "private->circuit.time");}
| TOK_RAD_FREQ
|     {fprintf (yyout, "private->circuit.frequency");}
| TOK_TEMPERATURE
|     {fprintf (yyout, "private->circuit.temperature");}
| TOK_T TOK_LPAREN buffered_c_code TOK_RPAREN
|     {fprintf (yyout, "private->circuit.t[%s]", $3);}
| TOK_PARAM TOK_LPAREN subscriptable_id TOK_RPAREN
|     {Get the subscript, 'i', from the subscriptable_id result
|      [valid_subid()].
|      fprintf (yyout, "private->param[%d]->element[%s]",
|              i, subscript ($3)) [subscript()].
|      Append the type suffix (e.g. ".rvalue") [put_type()].
|      }
| TOK_PARAM_SIZE TOK_LPAREN id TOK_RPAREN
|     {Get the subscript, 'i', from the id [valid_id()].
|      fprintf (yyout, "private->param[%d]->size", i);}
| TOK_PARAM_NULL TOK_LPAREN id TOK_RPAREN
|     {Get the subscript, 'i', from the id [valid_id()].
|      fprintf (yyout, "private->param[%d]->is_null", i);}
| TOK_PORT_SIZE TOK_LPAREN id TOK_RPAREN
|     {Get the subscript, 'i', from the id [valid_id()].
|      fprintf (yyout, "private->conn[%d]->size", i);}
| TOK_PORT_NULL TOK_LPAREN id TOK_RPAREN
|     {Get the subscript, 'i', from the id [valid_id()].
|      fprintf (yyout, "private->conn[%d]->is_null", i);}
| TOK_PARTIAL TOK_LPAREN subscriptable_id TOK_COMM
|     subscriptable_id TOK_RPAREN
|     {Get the subscripts, 'i' and 'j' from the subscriptable_id's
|      [valid_subid()].
|      Verify correct directions for subscriptable_id's [check_dir()].
|      fprintf (yyout,
|              "private->conn[%d]->port[%s]->partial[%d].port[%s]",
|              i, subscript($3), j, subscript($5)) [subscript()].}
| TOK_AC_GAIN TOK_LPAREN subscriptable_id TOK_COMM
|     subscriptable_id TOK_RPAREN
|     {Get the subscripts, 'i' and 'j' from the subscriptable_id's
|      [valid_subid()].
|      Verify correct directions for subscriptable_id's [check_dir()].
|      fprintf (yyout,
|              "private->conn[%d]->port[%s]->ac_gain[%d].port[%s]",
|              i, subscript($3), j, subscript($5)) [subscript()].}
| TOK_STATIC_VAR TOK_LPAREN subscriptable_id TOK_RPAREN
|     {Get the subscript from the subscriptable_id [valid_subid()].
|      fprintf (yyout,
|              "private->inst_var[%d]->element[%s]",
|              i, subscript($3)) [subscript()].

```

```

        if (static var is not an array or subscriptable_id has
            as subscript) {
                Append the type info to yyout (e.g. ".rvalue") [put_type()].
            }
| TOK_STATIC_VAR_SIZE TOK_LPAREN subscriptable_id TOK_RPAREN
    {Get the subscript, 'i', from the id [valid_id()].
     sprintf (yyout, "private->inst_var[%d]->size",
              i, subscript($3)) [subscript()].}
| TOK_OUTPUT_DELAY TOK_LPAREN subscriptable_id TOK_RPAREN
    {Get the subscript, 'i', from the subscriptable_id
     [valid_subid()].
     Verify direction is OUT [check_dir()].
     fprintf (yyout,
              "private->conn[%d]->port[%s]->delay", i,
              subscript($3)) [subscript()].}
| TOK_CHANGED TOK_LPAREN subscriptable_id TOK_RPAREN
    {Get the subscript, 'i' from the subscriptable_id [valid_subid()].
     Verify direction is OUT [check_dir()].
     fprintf (yyout,
              "private->conn[%d]->port[%s]->changed", i,
              subscript($3)) [subscript()].}
| TOK_INPUT TOK_LPAREN subscriptable_id TOK_RPAREN
    {Get the subscript, 'i' from the subscriptable_id [valid_subid()].
     Verify direction is IN [check_dir()].
     fprintf (yyout,
              "private->conn[%d]->port[%s]->input",
              i, subscript($3)) [subscript()].
     Append data type ("".rvalue" or ".pvalue") depending
     on port type.}
| TOK_INPUT_TYPE TOK_LPAREN subscriptable_id TOK_RPAREN
    {Get the subscript, 'i' from the subscriptable_id [valid_subid()].
     Verify direction is IN [check_dir()].
     fprintf(yyout,
              "private->conn[%d]->port[%s]->type_str",
              i, subscript($3)) [subscript()].}
| TOK_OUTPUT_TYPE TOK_LPAREN subscriptable_id TOK_RPAREN
    {Get the subscript, 'i' from the subscriptable_id [valid_subid()].
     Verify direction is OUT [check_dir()].
     fprintf (yyout,
              "private->conn[%d]->port[%s]->type_str",
              i, subscript($3)) [subscript()].}
| TOK_INPUT_STRENGTH TOK_LPAREN subscriptable_id TOK_RPAREN
    {Get the subscript, 'i' from the subscriptable_id [valid_subid()].
     Verify direction is IN [check_dir()].
     fprintf (yyout,
              "((Digital_t)(private->conn[%d]->port[%s]->input",
              i, subscript($3)) [subscript()].
     Append data type ("".pvalue").
     fprintf (yyout, "))->strength");}
| TOK_INPUT_STATE TOK_LPAREN subscriptable_id TOK_RPAREN
    {Get the subscript, 'i' from the subscriptable_id [valid_subid()].
     Verify direction is IN [check_dir()].

```

```

        fprintf (yyout,
                  "((Digital_t*)(private->conn[%d]->port[%s]->input",
                  i, subscript($3)) [subscript()].
                  Append data type (" .pvalue").
                  fprintf (yyout, "")->state");}
| TOK_OUTPUT TOK_LPAREN subscriptable_id TOK_RPAREN
{Get the subscript, 'i' from the subscriptable_id [valid_subid()].
 Verify direction is OUT [check_dir()].
 fprintf (yyout,
          "private->conn[%d]->port[%s]->output",
          i, subscript($3)) [subscript()].
          Append data type (" .rvalue" or ".pvalue") depending
          on port type.}
| TOK_OUTPUT_STRENGTH TOK_LPAREN subscriptable_id TOK_RPAREN
{Get the subscript, 'i' from the subscriptable_id [valid_subid()].
 Verify direction is OUT [check_dir()].
 fprintf (yyout,
          "((Digital_t*)(private->conn[%d]->port[%s]->output",
          i, subscript($3));
          Append data type (" .pvalue").
          fprintf (yyout, "")->strength");}
| TOK_OUTPUT_STATE TOK_LPAREN subscriptable_id TOK_RPAREN
{Get the subscript, 'i' from the subscriptable_id [valid_subid()].
 Verify direction is OUT [check_dir()].
 fprintf (yyout,
          "((Digital_t*)(private->conn[%d]->port[%s]->output",
          i, subscript($3));
          Append data type (" .pvalue") [put_conn_type()].
          fprintf (yyout, "")->state");}
| TOK_OUTPUT_CHANGED TOK_LPAREN subscriptable_id TOK_RPAREN
{Get the subscript, 'i' from the subscriptable_id [valid_subid()].
 fprintf (yyout,
          "private->conn[%d]->port[%s]->changed", i,
          subscript($3)) [subscript()].
          subscript($3)) [subscript()]}.
| TOK_LOAD TOK_LPAREN subscriptable_id TOK_RPAREN
{Get the subscript, 'i' from the subscriptable_id [valid_subid()].
 fprintf (yyout,
          "private->conn[%d]->port[%s]->load", i,
          subscript($3)) [subscript()].}
| TOK_TOTAL_LOAD TOK_LPAREN subscriptable_id TOK_RPAREN
{Get the subscript, 'i' from the subscriptable_id [valid_subid()].
 fprintf (yyout,
          "private->conn[%d]->port[%s]->total_load", i,
          subscript($3)) [subscript()].}
| TOK_MESSAGE TOK_LPAREN subscriptable_id TOK_RPAREN
{Get the subscript, 'i' from the subscriptable_id [valid_subid()].
 fprintf (yyout,
          "private->conn[%d]->port[%s]->msg", i,
          subscript($3)) [subscript()].}
;
subscriptable_id : id

```

```
| id TOK_LBRACKET buffered_c_code TOK_RBRACKET
| {Set production result to id.
|   Set production result's has_subscript filed to TRUE.
|   Set production result's subscript field to
|     buffered_c_code.}
|
id : TOK_IDENTIFIER
{Set production result's has_subscript field
to FALSE.
Copy 'yytext' to production result's id field.}
;
}
```

4.1.4.5.1.2.1 Static Function append

Summary

Appends the argument string 'str' to the global string 'buffer'. Checks for overflow of the buffer and produces a fatal error if the buffer overflows.

Called By

mod_yyparse() mod_yacc.c

Returned Value

None.

Global Variables

```
static char buffer[3000]; /* INOUT - buffered C code string buffer */
static int buf_len;        /* INOUT - length of the string in 'buffer' */
```

PDL Description

```
static void append (str)
char *str; /* IN - string to be appended to 'buffer' */

{
    if 'str' cannot be appended to 'buffer' without overflow:
        Generated a "buffer overflow" error message [yyerror()].
        exit the program.
    else
        append 'str' to 'buffer'.
        increment 'buf_len' by the length of 'str'.
}
```

4.1.4.5.1.2.2 Static Function check_dir

Summary

Checks if the direction of the connection specified by 'conn_number' is the same as 'dir'. If not, generates an error message giving the 'context'.

Called By

`mod_yyparse()` `mod_yacc.c`

Returned Value

None.

Global Variables

```
Ifs_Table_t *mod_ifs_table; /* IN - Defined connections, parameters, static vars */
int mod_num_errors;                                                                   /* OUT - Number of errors detected while parsing */
```

PDL Description

```
static void check_dir (conn_number, dir, context)

int conn_number; /* IN - connection to be checked */
Dir_t dir;        /* IN - correct connection direction */
char *context;    /* IN - name of the MOD macro that is being checked */

{
    if ('conn_number' is a valid connection number and the associated
        connection's direction is not 'dir' or IINOUT,
        Emit an error message [yyerror()].
        Increment the error count 'mod_num_errors'.
}
```

4.1.4.5.1.2.3 Static Function check_id

Summary

Used by valid_id() and valid_subid() to check that the given 'sub_id' is a valid connection, parameter or static var (depending on the value of 'kind'). If 'do_subscript' is true, then also checks that the subscript in 'sub_id' is consistent with the corresponding connection, parameter or static var.

Called By

valid_id	mod_yacc.y
valid_subid	mod_yacc.y

Returned Value

The return value is -1 if 'sub_id' does not match any of the known connections, parameters, etc. Otherwise, the return value is the index of the connection, parameter or static var in the 'mod_ifs_table'. If a subscript check fails, an error message is generated and 'mod_num_errors' is incremented.

Global Variables

```
Ifs_Table_t *mod_ifs_table; /* IN - Defined connections, parameters, static vars */
int mod_num_errors; /* OUT - Number of errors detected while parsing */
```

PDL Description

```
static void check_id (sub_id, kind, do_subscript)
Sub_Id_t sub_id; /* IN - The id to be checked */
Id_Kind_t kind; /* IN - Is this id a connection, parameter, or static var? */
Boolean_t do_subscript; /* IN - Should the subscripts be checked? */

{
    Loop through the appropriate list in 'ifs_mod_table' (depending on 'kind')
    if the 'ifs_mod_table' element's name is the same as 'sub_id's id, then
        if 'do_subscript' is true, check for a valid subscript [check_subscript()].
    If none of the elements in 'ifs_mod_table' match, generated an error and increment
    'mod_num_errors' [yyerror()].
}
```

4.1.4.5.1.2.4 Static Function `check_subscript`

Summary

Check that the use of a subscript is appropriate.

Called By

`check_id` `mod_yacc.c`

Returned Value

None.

Global Variables

```
int mod_num_errors;                        /* OUT - Number of errors detected while parsing */
```

PDL Description

```
static void check_subscript (formal, actual, missing_actual_ok, context, id)

Boolean_t formal;                         /* IN - Is the formal definition an array? */
Boolean_t actual;                         /* IN - Does the use have a subscript? */
Boolean_t missing_actual_ok; /* IN - Is it OK to for the actual to not be subscripted
                                                  even if formal is an array? */
char *context;                             /* IN - What kind of element is this (e.g. "Port",
                                                  "Parameter", etc.) */
char *id;                                 /* IN - The name of the id being checked */

{
    if 'formal' and 'actual' are not compatible
        emit an error message and increment 'mod_num_errors' [yyerror()];
}
```

4.1.4.5.1.2.5 Static Function init_buffer

Summary

Initializes the buffered C code string buffer to the null string.

Called By

`mod_yyparse()` `mod_yacc.c`

Returned Value

None.

Global Variables

```
int mod_num_errors;        /* OUT - Number of errors detected while parsing */
static char buffer[3000]; /* OUT - buffered C code string buffer */
static int buf_len;       /* OUT - length of the string in 'buffer' */
```

PDL Description

```
static void init_buffer ()
{
    Set 'buf_len' to 0 and 'buffer' to the null string.
}
```

4.1.4.5.1.2.6 Static Function put_conn_type

Summary

Emit a structure field selector for the given connection type.

Called By

mod_yyparse() mod_yacc.c

Returned Value

None.

PDL Description

```
static void put_conn_type (fp, type)

FILE *fp;          /* IN - File to write to */
Port_Type_t type; /* IN - The type of the connection (e.g., USER_DEFINED, DIGITAL) */
{
    fprintf (fp, .%cvalue, ch); where 'ch' is the appropriate type code for 'type'.
}
```

4.1.4.5.1.2.7 Static Function put_type

Summary

Emit a structure field selector for the given data type.

Called By

mod_yyparse() mod_yacc.c

Returned Value

None.

PDL Description

```
static void put_type (fp, type)

FILE *fp;                        /* IN - File to write to */
Data_Type_t type;              /* IN - The type of the data (e.g., INTEGER, REAL, COMPLEX...) */
{
    fprintf (fp, .%cvalue, ch); where 'ch' is the appropriate type code for 'type'.
}
```

4.1.4.5.1.2.8 Static Function strcmpl

Summary

Case insensitive string comparison. Like the standard string comparison function, `strcmp()`, but case insensitive.

Called By

`check_id()` `mod_yacc.c`

Returned Value

Returns 0 if the strings are equal, 'negative' if the first string is lexically 'less' than the second. Otherwise it returns 'positive'.

PDL Description

```
static int strcmpl (s1, s2)

char *s1;      /* IN - String to be compared */
char *s2;      /* IN - String to be compared */
{
    Loop over the initial substrings in each string that compare equal
    case insensatively.

    If we reached the end of the second string but not the first. return 1.

    If we reached the end of the first string but not the second. return -1.

    If we reached the end of both strings, return 0..

    Otherwise, return the difference between the lower case equivalents
    of the first characters that were not equal.
}
```

4.1.4.5.1.2.9 Static Function subscript

Summary

Returns the subscript of a possibly subscripted id. Scalar values are implemented as arrays of 1 element, so their subscript is "0".

Called By

`mod_yyparse()` `mod_yacc.c`

Returned Value

The subscript of the 'id'.

PDL Description

```
static char *subscript (sub_id)

Sub_Id_t sub_id; /* the possibly subscripted id from which to extract the subscript */
{
    If there is a user supplied subscript in 'id', return it. Otherwise,
    return "0".
}
```

4.1.4.5.1.2.10 Static Function valid_id

Summary

Check that the given 'sub_id' is a valid connection, parameter or static var (depending on the value of 'kind').

Called By

mod_yyparse() mod_yacc.c

Returned Value

The return value is -1 if 'sub_id' does not match any of the known connections, parameters, etc. Otherwise, the return value is the index of the connection, parameter or static var in the 'mod_ifs_table'.

PDL Description

```
static void valid_id (sub_id, kind)

Sub_Id_t sub_id;                /* IN - The id to be checked */
Id_Kind_t kind;                /* IN - Is this id a connection, parameter, or static var? */
{
    Call check_id to do the work [check_id()];
}
```

4.1.4.5.1.2.11 Static Function valid_subid

Summary

Used by valid_id() and valid_subid() to check that the given 'sub_id' is a valid connection, parameter or static var (depending on the value of 'kind'). Also checks that the subscript in 'sub_id' is consistent with the corresponding connection, parameter or static var.

Called By

`mod_yyparse()` `mod_yacc.c`

Returned Value

The return value is -1 if 'sub_id' does not match any of the known connections, parameters, etc. Otherwise, the return value is the index of the connection, parameter or static var in the 'mod_ifs_table'.

PDL Description

```
static void valid_subid (id, kind)

Sub_Id_t sub_id;          /* IN - The id to be checked */
Id_Kind_t kind;           /* IN - Is this id a connection, parameter, or static var? */
{
    Call check_id to do the work [check_id()];
}
```

4.1.4.6 Function preprocess_lst_file

Summary

Function preprocess_lst_file is the top-level driver function for preprocessing a simulator model path list file (modpath.lst). This function calls read_ifs_file() requesting it to read and parse the Interface Specification file (ifspec.ifs) to extract the model name and associated C function name and place this information into an internal data structure. It then calls check_uniqueness() to verify that the model names and function names are unique with respect to each other and to the models and functions internal to SPICE 3C1. Following this check, it calls write_CMextrn(), write_CMinfo(), and write_make_include() to write out the C include files CMextrn.h and CMinfo.h required by SPICE function SPLinit.c to define the models known to the simulator, and to write out an include file used by the make utility to locate the object modules necessary to link the models into the simulator.

Called By

main() main.c

Returned Value

None.

Type Definitions

```
typedef struct {
    char      *path_name;    /* Pathname read from model path file */
    char      *spice_name;   /* Name of model from ifspec.ifs */
    char      *cfunc_name;   /* Name of C fcn from ifspec.ifs */
    Boolean_t  spice_unique; /* True if spice name unique */
    Boolean_t  cfunc_unique; /* True if C fcn name unique */
} Model_Info_t;

typedef struct {
    char      *path_name;    /* Pathname read from udn path file */
    char      *node_name;    /* Name of node type */
    Boolean_t  unique;       /* True if node type name unique */
} Node_Info_t;
```

PDL Description

```
void preprocess_list_files()
{
    int          num_models; /* The number of models */
    Model_Info_t *model_info; /* Info about each model */

    int          num_nodes; /* The number of user-defined nodes */
    Node_Info_t *node_info; /* Info about each user-defined node type */

    Get number of and list of models from model pathname file into
    'num_models' and 'model_info'.
    [read_modpath()]

    Get number of and list of node types from udn pathname file into
    'num_nodes' and 'node_info'.
    [read_udnpath()]

    Get the spice and C function names from the ifspec.ifs files.
    [read_model_names()]

    Get the user-defined node type names.
    [read_node_names()]

    Check to be sure the names are unique.
    [check_uniqueness()]

    Write out the CMextrn.h file used to compile SPIinit.c.
    [write_CMextrn()]

    Write out the CMinfo.h file used to compile SPIinit.c.
    [write_CMinfo()]

    Write out the UDNextrn.h file used to compile SPIinit.c.
    [write_UDNextrn()]

    Write out the UDNinfo.h file used to compile SPIinit.c.
    [write_UDNinfo()]

    Write the make_include file used to link the models and
    user-defined node functions with the simulator.
    [write_objects_inc()]
}
```

4.1.4.6.1 Function `read_modpath`

Summary

This function opens the modpath.lst file, reads the pathnames from the file, and puts them into an internal data structure for future processing.

Called By

`preprocess_lst_files()` pp_lst.c

Returned Value

Zero on success. Non-zero otherwise.

PDL Description

```
static Status_t read_modpath(num_models, model_info)

int *num_models; /* OUT - Number of model pathnames found */
Model_Info_t **model_info; /* OUT - Info about each model */
{
    Initialize 'num_models' to zero in case of error.

    Open the model pathname file.

    Read the pathnames from the file, one line at a time until EOF,
        If line was too long for buffer, exit with error.

        Strip white space including newline.

        Strip trailing '/' if any.

        If blank line, continue.

        Make sure pathname is short enough to add a filename at the end.

        Allocate and initialize a new model info structure in 'model_info'.

        Put pathname into info structure.

        Increment 'num_models'.

    Close model pathname file and return.
}
```

4.1.4.6.2 Function `read_udnpath`

Summary

This function opens the `udnpath.lst` file, reads the pathnames from the file, and puts them into an internal data structure for future processing.

Called By

`preprocess_lst_files()` pp_lst.c

Returned Value

Zero on success. Non-zero otherwise.

PDL Description

```
static Status_t read_udnpath(num_nodes, node_info)

int      *num_nodes;    /* OUT - Number of node pathnames found */
Node_Info_t **node_info; /* OUT - Info about each node */

{
    Initialize 'num_nodes' to zero in case of error.

    Open the node pathname file.

    Output a warning message and return if file not found. [print_error()]

    Read the pathnames from the file, one line at a time until EOF.

    If line was too long for buffer, exit with error.

    Strip white space including newline.

    Strip trailing '/' if any.

    If blank line, continue.

    Make sure pathname is short enough to add a filename at the end.

    Allocate and initialize a new node info structure.

    Put pathname into info structure.
```

XSPICE Code Model Subsystem
Software Design Document

Detailed Design
Code Model Toolkit

```
    Increment 'num_nodes'.  
    Close node pathname file and return.  
}
```

4.1.4.6.3 Function `read_model_names`

Summary

This function opens each of the models and copies the names of the model and the C function into the internal model information structure.

Called By

`preprocess_lst_files()` pp_lst.c

Returned Value

Zero on success. Non-zero otherwise.

PDL Description

```
static Status_t read_model_names(num_models, model_info)

int          num_models;    /* IN      - Number of model pathnames */
Model_Info_t *model_info; /* INDOUT - Info about each model */

{
    For each model found in model pathname file, read the Interface
    Specification file to get the SPICE and C function names into 'model_info'
    as follows.

    Form the full pathname to the Interface Spec file.

    Read the SPICE and C function names from the Interface Spec file.
    [read_ifs_file()]

    Transfer the names into 'model_info'.

    If all found,
        Return OK.

    Else
        Return ERROR.
}
```

4.1.4.6.4 Function `read_node_names`

Summary

This function opens each of the user-defined node definition files and copies the names of the node into the internal information structure.

Called By

`preprocess_lst_files()` pp_lst.c

Returned Value

Zero on success. Non-zero otherwise.

PDL Description

```
static Status_t read_node_names(num_nodes, node_info)

int          num_nodes;    /* IN - Number of node pathnames */
Node_Info_t *node_info;   /* INOUT - Info about each node */

{
    For each node found in node pathname file, read the udnfunc.c
    file to get the node type names into 'node_info' as follows,
    Form the full pathname to the node definition file.

    Read the udn node type name from the file.
    [read_udn_type_name()]

    Transfer the names into the 'node_info' structure.

    If all found,
        Return OK.

    Else
        Return ERROR.
}
```

4.1.4.6.4.1 Function `read_udn_type_name`

Summary

This function reads a User-Defined Node Definition File until the definition of a structure of type `Evt_Udn_Info_t` is found, and then gets the name of the node type from the first member of the structure.

Called By

`read_node_names()` pp_lst.c

Returned Value

Zero on success. Non-zero otherwise.

PDL Description

```
static Status_t read_udn_type_name(path, node_name)

char *path;          /* IN - The path to the node definition file */
char **node_name;   /* OUT - The node type name found in the file */
{

    Open the file specified in 'path' from which the node type name will be read.

    Read the file until the definition of the Evt_Udn_Info_t struct
    is found, then get the name of the node type from the first
    member of the structure as follows.

        Read until the string "Evt_Udn_Info_t" is encountered outside of comment
        delimiters.

        If found, read until open quote found, and then read the name until
        the closing quote is found.

        Assign the name into 'node_name'.

    Close the file and return.
}
```

4.1.4.6.5 Function check_uniqueness

Summary

Function `check_uniqueness` determines if model names and function names are unique with respect to each other and to the models and functions internal to SPICE 3C1.

Called By

`preprocess_lst_files()` pp_lst.c

Returned Value

Zero if all names unique, non-zero otherwise.

PDL Description

```
static Status_t check_uniqueness(num_models, model_info, num_nodes, node_info)

int          num_models;    /* IN - Number of model pathnames */
Model_Info_t *model_info;  /* INOUT - Info about each model */
int          num_nodes;    /* IN - Number of node type pathnames */
Node_Info_t *node_info;   /* INOUT - Info about each node type */
{
    Define the list of model names used internally by XSPICE.
    These names (except 'poly') are defined in src/sim/IMP/IMPdomodel.c and
    are case insensitive.

    Define the list of node type names used internally by the simulator.
    These names are defined in src/sim/include/MIFtypes.h and are case
    insensitive.

    Normalize all model and node names in 'model_info' and 'node_info'
    to lower case since SPICE is case insensitive in parsing decks.

    Loop through all model names in 'model_info' and report errors if same
    as SPICE model name or same as another model name in list.
    [print_error()]

    Loop through all C function names in 'model_info' and report errors if
    duplicates found.  [print_error()]
```

```
Loop through all node type names in 'node_info' and report errors if
same as internal name or same as another name in list. [print_error()]
```

```
Return status.
```

```
}
```

4.1.4.6.6 Function write_CMextrn

Summary

Function write_CMextrn writes the CMextrn.h file used in compiling SPIUnit.c immediately prior to linking the simulator and code models. CMextrn.h contains a C language “extern” declaration for each code model’s “SPICEdev” data structure.

Called By

preprocess_lst_files() pp_lst.c

Returned Value

Zero on success. Non-zero otherwise.

PDL Description

```
static Status_t write_CMextrn(num_models, model_info)
{
    int         num_models;      /* IN - Number of model pathnames */
    Model_Info_t *model_info;   /* IN - Info about each model */
{
    Open the file to be written (CMextrn.h).

    Write out the extern statements.

    Close the file and return.
}
```

4.1.4.6.7 Function write_CMinfo

Summary

Function `write_CMinfo` writes the `CMinfo.h` file used in compiling `SPInit.c` immediately prior to linking the simulator and code models. `CMinfo.h` contains an entry for each model's "SPICEdev" data structure. These entries are used in the construction of the SPICE3 "DEVICES" array of "SPICEdev" pointers that defines the models known to the simulator.

Called By

`preprocess_lst_files()` `pp_lst.c`

Returned Value

Zero on success. Non-zero otherwise.

PDL Description

```
static Status_t write_CMinfo(num_models, model_info)
{
    int          num_models;    /* IN - Number of model pathnames */
    Model_Info_t *model_info;  /* IN - Info about each model */
    {
        Open the file to be written (CMinfo.h).

        Write out the SPICEdev pointers for the models in 'model_info'.

        Close the file and return.
    }
}
```

4.1.4.6.8 Function write_UDNextrn

Summary

Function `write_UDNextrn` writes the `UDNextrn.h` file used in compiling `SPInit.c` immediately prior to linking the simulator and user-defined nodes. `UDNextrn.h` contains a C language “extern” declaration for each node type’s “`Evt_Udn_Info_t`” data structure.

Called By

`preprocess_lst_files()` pp_lst.c

Returned Value

Zero on success. Non-zero otherwise.

PDL Description

```
static Status_t write_UDNextrn(num_nodes, node_info)

int          num_nodes;    /* IN - Number of node pathnames */
Node_Info_t   *node_info;  /* IN - Info about each node */
{
    Open the file to be written (UDNextrn.h).

    Write out the extern statements.

    Close the file and return.
}
```

4.1.4.6.9 Function write_UDNinfo

Summary

Function `write_UDNinfo` writes the `UDNinfo.h` file used in compiling `SPIinit.c` immediately prior to linking the simulator and user-defined nodes. `UDNinfo.h` contains an entry for each node type's "`Evt_Udn_Info_t`" data structure. These entries are used in the construction of the XSPICE "`g_evt_udn_info`" array of "`Evt_Udn_Info_t`" pointers that defines the node types known to the simulator.

Called By

`preprocess_lst_files()` pp_lst.c

Returned Value

Zero on success. Non-zero otherwise.

PDL Description

```
static Status_t write_UDNinfo(num_nodes, node_info)

int          num_nodes;    /* IN - Number of node pathnames */
Node_Info_t   *node_info;  /* IN - Info about each node */
{
    Open the file to be written (UDNinfo.h).

    Write out the Evt_Udn_Info_t pointers for the nodes in 'node_info'.

    Close the file and return.
}
```

4.1.4.6.10 Function write_object_inc

Summary

Function `write_object_inc` writes an include file used by the make utility to locate the object modules needed to link the simulator with the code models and user-defined node types.

Called By

`preprocess_lst_files()` pp_lst.c

Returned Value

Zero on success. Non-zero otherwise.

PDL Description

```
static Status_t write_objects_inc(num_models, model_info, num_nodes, node_info)

int      num_models;    /* IN - Number of model pathnames */
Model_Info_t *model_info; /* IN - Info about each model */
int      num_nodes;     /* IN - Number of udn pathnames */
Node_Info_t *node_info; /* IN - Info about each node type */
{
    Open the file to be written (objects.inc).

    Write out the pathnames to the model directories in 'model_info'.

    Write out the pathnames to the node directories in 'node_info'.

    Close the file and return.
}
```

4.2 Code Model Library

This section specifies the design of the Code Model Library cfunc.mod functions. The Interface Specification that defines ports and parameters for each model is defined in the Interface Design Document for the Simulator Code Model Subsystem of the Automatic Test Equipment Software Support Environment (ATESSE). Refer to this document when reading the behavioral descriptions of the models in the following sections.

In each of the following PDL descriptions, items in single quotes ('item') reference the name of a model attribute, an input to or an output from that model.

4.2.1 Analog Code Models

The following sections describe the functional behavior of the analog code models of the Code Model Library.

4.2.1.1 Gain

Summary

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

Called By

MIFload() MIF/MIFload.c

Returned Value

None.

PDL Description

```
void cm_gain(ARGS)

ARGS = Mif_Private_t *private; /* INOUT - Code model inputs,
.. outputs, and parameters */

{
    If ARGS indicates that this is a DC or TRANSIENT analysis,
        Output the value (input + 'in_offset') * 'gain' + 'out_offset'.
        Output the partial derivative.

    Else if this is an AC analysis,
        Output the AC gain (complex value = ('gain', 0)).
}
```

4.2.1.2 Summer

Summary

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

Called By

MIFload() MIF/MIFload.c

Returned Value

None.

PDL Description

```
void cm_summer(ARGS)
ARGS = Mif_Private_t *private; /* INOUT - Code model inputs,
.. outputs, and parameters */

{
    If ARGS indicates that this is a DC or TRANSIENT analysis,
        Accumulate the inputs by reading each input 'in[i]',
        offsetting it by individual parameter 'in_offset[i]',
        multiplying by the individual gain value, 'in_gain[i]', and
        summing with the previous inputs. At the same time, output the
        individual partial values of the output w.r.t. the inputs.

        Output total result = 'out_gain' * accumulated input values +
        'out_offset'.

    Else if this is an AC analysis,
        Output the AC gain for each input. AC gain for input "i"
        equals the complex value ('in_gain[i]' * 'out_gain', 0).

}
```

4.2.1.3 Multiplier

Summary

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

Called By

MIFload() MIF/MIFload.c

Returned Value

None.

PDL Description

```
void cm_mult(ARGS)
ARGS = Mif_Private_t *private; /* INOUT - Code model inputs,
.. outputs, and parameters */

{
    Calculate multiplication of inputs and gains for
    all types of analyses. This is accomplished by multiplying
    all values of inputs ('in[i]' + 'in_offset[i]') together, and
    separately multiplying all input gains ('in_gain[i]') together.

    If ARGS indicates that this is a DC or TRANSIENT analysis,
        Compute each partial derivative as the product of all 'in_gain[i]'
        values and 'in[i]' inputs divided by the individual input value.
        First make sure that no division by zero will occur. If a
        division by zero is possible for one of the input partials, set
        that partial to zero.

        Output the total summer output = 'out_gain' *
        (product of all input gains) * (product of all input values) +
        'out_offset'.

    Else if this is an AC analysis,
```

Real portion of each AC gain is the product of all 'in_gain[i]',
'in[i]' and 'out_gain' divided by the individual input value.

Output each AC gain value.

}

4.2.1.4 Divider

Summary

This function is a two-quadrant divider. It takes two inputs: num (numerator) and den (denominator). Divide offsets its inputs, multiplies them by their respective gains, divides the results, multiplies the quotient by the output gain, and offsets the result. The denominator is limited to a value above zero via a user specified lower limit. This limit is approached through a quadratic smoothing function, the domain of which may be specified as a fraction of the lower limit value (default), or as an absolute value. This model will operate in DC, AC, and Transient analysis modes.

Called By

MIFload() MIF/MIFload.c

Returned Value

None.

PDL Description

```
void cm_divide(ARGS)
ARGS = Mif_Private_t *private; /* INOUT - Code model inputs,
.. outputs, and parameters */

{
    Retrieve frequently used parameters, and assign to internal variables.

    Set 'den_domain' smoothing domain to either an absolute value
    or a percentage of the 'den_lower_limit' parameter.

    Set the upper threshold value.

    Set the lower threshold value.

    Read inputs--denominator and numerator, and apply 'num_offset',
    'num_gain', 'den_offset', and 'den_gain' values to these inputs.

    If the offset-and-amplified value of the denominator is less
    than the upper threshold value and greater than zero,

        See if the denominator is greater than threshold_lower. If so,
```

Invoke the smoothing function [cm_smooth_corner()] to parabolically smooth between the upper threshold value and the 'den_lower_limit' value. This function will also set the value of the partial derivative of the internal denominator value w.r.t. the input

Else

The denominator value is in the hard-limited region; in this case, set the denominator value to 'out_lower_limit' and set the partial derivative to zero.

Else if the offset-and-amplified value of the denominator is greater than -(the upper threshold value) and less than zero,

See if the denominator is less than -(threshold_lower). If so,

Invoke the smoothing function [cm_smooth_corner()] to parabolically smooth between -(the upper threshold value) and -('den_lower_limit'). This function will also set the value of the partial derivative of the internal denominator value w.r.t. the input.

Else

The denominator value is in the hard-limited region; in this case, set the denominator value to -('out_lower_limit') and set the partial derivative to zero.

Else

No limiting is needed; the denominator value does not approach zero; use the offset-and-amplified version of the denominator value without change, and do not set the partial derivative to zero.

If ARGS indicates that this is a DC or TRANSIENT analysis,

Compute and output values and partials.

Else if ARGS indicates that this is an AC analysis,

Output the AC values; these are complex numbers the real portions of which are just the partial derivatives of the output w.r.t. the numerator and the denominator. The imaginary portion of both of these gains is zero.

}

4.2.1.5 Limiter

Summary

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

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

Called By

MIFload() MIF/MIFload.c

Returned Value

None.

PDL Description

```
void cm_limit(ARGS)
ARGS = Mif_Private_t *private; /* INOUT - Code model inputs,
.. outputs, and parameters */

{
    Retrieve frequently used parameters, and assign to internal variables.

    If 'fraction' indicates that the range is to be a fractional value,
    set the range equal to ('out_upper_limit' - 'out_lower_limit') *
    limit_range.

    Set the upper threshold based on the limit_range value.

    Set the lower threshold based on the limit_range value.

    Compute the non-limited output. This is equal to
    ('in' + 'in_offset') * 'gain'.
```

If the unlimited output is less than the lower threshold,

and if the unlimited output is also greater than the lower threshold minus the limit_range,

Smooth the output value parabolically and obtain the partial derivative of the output w.r.t. the input through the use of the smoothing function [cm_smooth_corner()].

Else if the output is below the lower threshold minus the limit_range,

Hard-limit the output to 'out_lower_limit' and set the partial to zero.

Else

If the unlimited output is greater than the upper threshold,

and if the unlimited output is less than the upper threshold minus the limit_range,

Smooth the output value parabolically and obtain the partial derivative of the output w.r.t. the input through the use of the smoothing function [cm_smooth_corner()].

Else if the output is above the upper threshold plus the limit range,

Hard-limited the output to 'out_upper_limit' and set the partial to zero.

Else if the unlimited output is between the threshold values,

No limiting is needed.

If ARGS indicates that this is a DC or TRANSIENT analysis,

Output the values and partials.

Else if ARGS indicates that this is an AC analysis,

Output the AC gain values.

}

4.2.1.6 Controlled Limiter

Summary

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

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

Note also that the Controlled Limiter code tests the input values of `cntl_lower` and `cntl_upper` to make sure that they are spaced far enough apart to guarantee the existence of a linear range between them. The range is calculated as the difference between (`cntl_upper - upper_delta - limit_range`) and (`cntl_lower + lower_delta + limit_range`), and must be greater than or equal to zero. Note that when the `limit_range` is specified as a fractional value, the `limit_range` used in the above is taken as the calculated fraction of the difference between `cntl_upper` and `cntl_lower`. Still, the potential exists for too great a `limit_range` value to be specified for proper operation, in which case the model will return an error message.

Called By

`MIFload()` `MIF/MIFload.c`

Returned Value

None.

PDL Description

```
void cm_climit(ARGS)
ARGS = Mif_Private_t *private; /* I/MOUT - Code model inputs,
.. outputs, and parameters */

{
    Retrieve frequently used parameters.
```

Find upper & lower output limits. These are equal to ('cntl_upper' - 'upper_delta') and ('cntl_lower' - 'lower_delta'), respectively.

Set the internal value of limit_range to a fixed value, if required by the 'fraction' parameter.

Set values for threshold_upper, threshold_lower, and linear_range. Note that linear_range is the range over which the output value will not be limited.

Test the linear region & make sure there IS one...
if not, output an error message [cm_message_send()] and return.

Compute what the unlimited output would be...

If the unlimited output value is less than threshold_lower,

and if the unlimited output is greater than (out_lower_limit - limit_range),

Smooth the output between the unlimited value and out_lower_limit, and obtain the partial derivative through the use of the smoothing function [cm_smooth_corner()]. Also, obtain a value for the partial of the output w.r.t. 'cntl_lower' [cm_smooth_discontinuity()] which represents a smooth transition from 0 at out = out_lower_limit to 1 at out = (out_lower_limit - limit_range).

Else if the unlimited output is less than (out_lower_limit - limit_range),

Hard-limit the output to out_lower_limit and set the the partial derivative of the output w.r.t the input to zero, and the partial of the output w.r.t. 'cntl_lower' to 1.

Else

If the unlimited output is greater than threshold_upper,

and if the unlimited output is less than (out_upper_limit + limit_range),

Smooth the output between the unlimited value and out_upper_limit, and obtain the partial derivative through the use of the smoothing function [cm_smooth_corner()]. Also, obtain a value for the partial of the output w.r.t. 'cntl_upper' [cm_smooth_discontinuity()] which represents a smooth

```
transition from 0 at out = out_upper_limit to 1 at
out = (out_upper_limit+limit_range).

Else if the unlimited output is greater than (out_upper_limit +
limit_range.

    Hard-limit the output to out_upper_limit and set
    the derivative to zero.

Else

    No limiting is needed.

If ARGS indicates that this is a DC or Transient analysis,
    Output 'out' value and derivatives...
Else if this is an AC analysis,
    Output AC gain values.

}
```

4.2.1.7 PWL Controlled Source

Summary

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

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

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

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

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

"Fraction" is a switch that is either TRUE or FALSE. When TRUE (the default setting), the simulator assumes that the specified `input_domain` value is to be interpreted as a fractional figure. Otherwise, it is interpreted as an absolute value. Thus, if `fraction=TRUE` and `input_domain=0.10`, the simulator assumes that the smoothing radius about each coordinate point is to be set equal to 10% of the length of either the `x_array` segment above each coordinate point, or the `x_array` segment below each coordinate point. The specific segment length chosen will be the smallest of these two for each coordinate point.

On the other hand, if `fraction=FALSE` and `input=0.10`, then the simulator will begin smoothing the transfer function at 0.10 volts (or amperes) below each `x_array` coordinate,

and will continue the smoothing process for another 0.10 volts (or amperes) above each x_array coordinate point. Since the overlap of smoothing domains is not allowed, checking is done by the model to insure that the specified input_domain value is not excessive.

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

Called By

MIFload() MIF/MIFload.c

Returned Value

None.

PDL Description

```
static double limit_x_value(x_lower, x_upper, x_input, fraction,
                            *last_x_value)

double x_lower; /* IN - The lower value of the current x segment */
double x_upper; /* IN - The upper value of the current x segment */
double x_input; /* IN - The current x value */
double fraction; /* IN - The fraction of the current segment length
                   ..      to be used as the maximum allowed x excursion */
double
*last_x_input; /* INOUT - The previous value of x_input */

{
    Calculate the maximum allowable difference (max_x_delta) between
    the previous value of x_input and the current value, based on the
    value of fraction and the length of the current segment (defined as
    equal to (x_upper - x_lower)).
```

If the change in `x_input` since the previous call is greater than `max_x_delta`,

Add or subtract `max_x_delta` from the previous value of `*last_x_value`, depending on the direction that the new `x_input` would have gone had no limiting been necessary.

Alert the simulator to the fact that the calling code model has not converged. This will cause the simulator to re-call the model at an earlier time step to attempt to obtain a solution once again.

else if the change in `x_input` is less than `max_x_delta`,

set `*last_x_value` to `x_input`.

Return the value of `x_input` and exit the routine.

}

`void cm_pwl(ARGs)`

`ARGs = Mif_Private_t *private; /* INOUT - Code model inputs,
.. outputs, and parameters */`

{

Retrieve frequently used parameters 'input_domain' and the size of the '`x_array`'.

If `ARGs` indicates that this is the initial call to the model,

allocate storage for the `last_x_value` variable (which is used to prevent excessive changes in the input from analysis point to analysis point) [`malloc()`], and for the '`x_array`' and the '`y_array`' holding variables. If allocation fails, alert the simulator [`cm_message_send()`] Then retrieve the individual `x` and `y` values.

Else if this is not the first call to the model,

Retrieve `x` and `y` values.

See if `input_domain` is an absolute rather than a fractional value. If so, test against breakpoint segment sizes and make sure that it is no larger than 50% of the size of the smallest specified `x` segment length [`cm_message_send()`]. If this were the case, then overlap would occur between the smoothing domains for at least one of the `x` segments.

Retrieve x_input = 'in'.

If this is the first call to the model, set last_x_value to x_input.

Add internal limiting to x_input as follows:
if the last_x_value is less than 'x[0]',

Test the delta between x_input and 'x[0]'. Depending on this value, either accept it as a reasonable increment from the previous input (last_x_value), or force the simulator to back up and use a smaller delta since the previous call [limit_x_value()]. The fractional value of the ('x[1]' - 'x[0]') segment to which the input change is constrained is set by the constant definition FRACTION (not to be confused with the parameter 'fraction').

else if the last_x_value is greater than 'x[size-1]',

Test the delta between x_input and 'x[size-1]'. Depending on this value, either accept it as a reasonable increment from the previous input (last_x_value), or force the simulator to back up and use a smaller delta since the previous call [limit_x_value()]. The fractional value of the ('x[size-1]' - 'x[size-2]') segment to which the input change is constrained is set by the constant definition FRACTION (not to be confused with the parameter 'fraction').

else if the last_x_value is somewhere between 'x[0]' and 'x[size-1]',

Beginning with the segment ('x[0]' -> 'x[1]'), find out within which segment x_input can be found.
Test the delta between x_input and 'x[i]'. Depending on this value, either accept it as a reasonable increment from the previous input (last_x_value), or force the simulator to back up and use a smaller delta since the previous call [limit_x_value()]. The fractional value of the ('x[i-1]' - 'x[i]') segment to which the input change is constrained is set by the constant definition FRACTION (not to be confused with the parameter 'fraction').

Regardless of whether the 'in' value was accepted or not, assign the calculated, limited value of x_input to 'in' for the next matrix calculation.

For the limited value of x_input, determine segment boundaries within which x_input resides:

```
If x_input is less than the midpoint between 'x[0]' and 'x[1]',  
    Calculate the slope for this region, then calculate the output  
    value.  
  
Else if x_input is greater than the midpoint between 'x[size-2]'  
and 'x[size-1]',  
    Calculate the slope for this region, then calculate the output  
    value.  
  
Else x_input is somewhere between the two outside midpoints.  
  
    Progressively step through the possible segment midpoints,  
    and find which of them is greater than x_input. Once this  
    midpoint is found, the approximate position of x_input is  
    known to be less than the current midpoint, but greater  
    than the midpoint of the segment with length  
    ('x[i]' - 'x[i-1]'). Note that this means that x_input  
    inhabits a region of the x domain which also includes  
    a breakpoint. The breakpoint corresponds to the common  
    point between segment ('x[i]'-'x[i-1]) and segment  
    ('x[i+1]'-'x[i]').  
  
    Calculate the length of the segments above and below the  
    breakpoint in the current region.  
  
    If the 'fraction' parameter is TRUE,  
        calculate a fixed value for the smoothing domain about  
        the breakpoint, based on the smallest of the two segments,  
        ('x[i]'-'x[i-1]) and ('x[i+1]'-'x[i]'). If 'fraction'  
        is FALSE, this is unnecessary.  
  
    Set up threshold values about breakpoint.  
  
    Determine where x_input is within region & determine  
    output and partial values:  
  
    If x_input is less than the lower threshold value,  
        calculate the slope of the lower segment, and use  
        y = slope * (x_input - 'x[i]') + y[i] to obtain  
        the output value.  
  
    else if x_input is less than the upper threshold,  
        calculate a smoothed value for the output and the  
        output partial derivative [cm_smooth_corner()].  
  
    else x_input is in the upper segment.  
        calculate the slope of the upper segment, and use
```

y = slope * (x_input - 'x[i]') + y[i] to obtain
the output value.

If ARGS indicates that this is a DC or TRANSIENT analysis,

Output the y value and the partial of the output w.r.t. the
input.

Else this is an AC analysis;

Output AC the gain value (= the partial of the output w.r.t. the
input).

}

4.2.1.8 Analog Switch

Summary

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

Called By

MIFload() MIF/MIFload.c

Returned Value

None.

PDL Description

```
void cm_aswitch(ARGS)
ARGS = Mif_Private_t *private; /* INOUT - Code model inputs,
.. outputs, and parameters */

{
    Retrieve frequently used parameters.

    Set the minimum 'ON' resistance to 1.0e-3 ohms.

    If the difference between the "on" and "off" controlling
    input values is less than 1.0e-12, send an error to the
    simulator and return from the function.

    If the 'log' parameter indicates that the analog switch resistance
    is to vary in a logarithmic fashion w.r.t. the 'cntl_in' value,

        Determine r, the logarithmically-varying resistance of
        the switch as a function of 'cntl_in'.

    Calculate the partial derivative of the output current
```

w.r.t. the input voltage,
and the partial of the output current w.r.t. 'cntl_in'.

Else if a linear variation of the resistance w.r.t. 'cntl_in' is
desired,

Determine r, the linearly-varying resistance value of the
switch, as a function of 'cntl_in'.

Calculate the partial derivative of the output current
w.r.t. the input voltage,
and the partial of the output current w.r.t. 'cntl_in'.

If ARGS indicates that this is a DC or TRANSIENT analysis,

Output DC & Transient values of output current.
Note that the minus signs are required because current
is positive flowing INTO rather than OUT OF a component node.

Else if this is an AC analysis,

Output the AC gain values.
Minus signs are required because current is positive flowing
INTO rather than OUT OF a component node.

}

4.2.1.9 Zener Diode

Summary

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

The breakdown voltage, breakdown current, and breakdown resistance parameters define the DC characteristics of the zener in the breakdown region and are usually explicitly given on the data sheet.

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

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

Called By

MIFload() MIF/MIFload.c

Returned Value

None.

PDL Description

```
void cm_zener(ARGS).  
  
ARGS = Mif_Private_t *private; /* INOUT - Code model inputs,  
.. outputs, and parameters */  
  
{
```

Retrieve frequently used parameters.

If ARGS indicates that this is the initialization call to the model,

Allocate storage for the 'previous_voltage' value [cm_analog_alloc()] and set the value to zero.

Else

Retrieve 'previous_voltage' [cm_analog_get_ptr()]; the value of the voltage across the zener.

If the 'limit_switch' parameter is TRUE,

And if the absolute value of the 'previous_voltage' is greater than 1.0,

Set the allowed increment for this pass to 10% of the previous voltage.

Else if 'previous_voltage' is less than zero,

set the allowed increment to -0.1.

Else

set the allowed increment to +0.1.

If the current value of 'v_zener' is greater than the previous value plus the increment variable just calculated, then

Set the v_zener value and the 'previous_voltage' value to ('previous_voltage' + increment).

Let the simulator know that convergence has not occurred [cm_analog_not_converged()].

else

No limiting is needed. Just set 'previous_voltage' to the current input voltage value.

Compute voltage at boundary of regions 1 & 2.

(Note that three distinct regions are defined for the model. Region 1 defines the forward voltage characteristic, region 3 defines that voltage domain over which the zener exhibits the breakdown characteristic, and region 2 is the reverse-biased region at voltages below those necessary to cause breakdown to occur).

Compute voltage at boundary of regions 2 & 3.

Compare v_1_2 and v_2_3 to determine if a 3 segment model is possible.

If the boundary points are such that the three distinct regions are possible (i.e., if the boundary voltage between region 1 and 2 is greater than the boundary voltage between regions 2 and 3),

Compute v0 for region 3. For this, assume that i0 is 1e-6.

Compute the ordinate at the boundary of regions 1 & 2.

Compute a & b coefficients for linear section in region 2

Compute the constant necessary to match the ordinate at the region 2/3 boundary.

Compute the required zener current:

If v_zener is greater than or equal to the region 1/2 boundary voltage,

compute i_zener using the forward bias characteristic.

Else

If v_zener is greater than the region 2/3 boundary voltage (\Rightarrow v_zener is in region 2),

Compute i_zener by using the equation
 $i_{zener} = a * v_{zener} + b$

Else v_zener is in the breakdown region:

Compute i_zener based on breakdown characteristic.

Else,

Must use a 2 segment model; in this case, if v_zener is greater than zero, we use the standard forward-biased characteristic. If v_zener is less than zero, we use the breakdown region characteristic. Thus, region 2 is removed.

Determine i0 for the reverse region.

Determine the slopes at the region endpoints.

Determine the zener current & first partial derivative.
Use a linear conductance in one region to match the slopes at the single boundary (if they do not agree).

Add the equivalent of a resistor in parallel with the active

```
device model to simulate the reverse leakage on the zener.  
If ARGS indicates that this is a DC or Transient analysis,  
    Output DC & Transient Values.  
  
Else  
    Output the AC Gain.  
  
}
```

4.2.1.10 Current Limiter

Summary

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

Depending on the polarity of the current flow, either a sourcing or a sinking resistance value (r_{out_source} , r_{out_sink}) is applied to govern the v_{out}/i_{out} relationship. The chosen resistance will continue to control the output current until it reaches a maximum value specified by either i_{limit_source} or i_{limit_sink} . The latter mimics the current limiting behavior of many operational amplifier output stages.

During all operation, the output current is reflected either in the v_{pos_pwr} port current or the v_{neg_pwr} current, depending on the polarity of i_{out} . Thus, realistic power consumption as seen in the supply rails is included in the model.

The user-specified smoothing parameters relate to model operation as follows: v_{pwr_range} controls the voltage below v_{pos_pwr} and above v_{neg_pwr} inputs beyond which v_{eq} [= gain * ($v_{in} + v_{offset}$)] is smoothed; i_{source_range} specifies the current below i_{limit_source} at which smoothing begins, as well as specifying the current increment above $i_{out}=0.0$ at which i_{pos_pwr} begins to transition to zero; i_{sink_range} serves the same purpose with respect to i_{limit_sink} and i_{neg_pwr} that i_{source_range} serves for i_{limit_source} & i_{pos_pwr} ; r_{out_domain} specifies the incremental value above and below $(v_{eq} - v_{out})=0.0$ at which r_{out} will be set to r_{out_source} and r_{out_sink} , respectively. For values of $(v_{eq} - v_{out})$ less than r_{out_domain} and greater than $-r_{out_domain}$, r_{out} is interpolated smoothly between r_{out_source} & r_{out_sink} .

Called By

MIFload() **MIF/MIFload.c**

Returned Value

None.

PDL Description

```
void cm_climit(ARGS)

ARGS = Mif_Private_t *private; /* INPUT - Code model inputs,
.. outputs, and parameters */

{
    Define variables.

    Retrieve frequently used parameters...

    Retrieve the voltage at the 'out' port.

    Test to see if pos_pwr or neg_pwr are connected...
    if not, assign large voltage values to the variables
    pos_pwr and neg_pwr...

    If ARGS indicates that this is not the first call to the model,
        Compute Veq plus the derivatives [cm_climit_fcn()]. Note that
        the behavior of the cm_climit_fcn() function is virtually
        identical to that of the entire controlled limiter code
        model. A review of the PDL for that code model should
        readily explain the nature of the expected outputs. Note
        that the returned veq value from cm_climit_fcn() is taken
        as the voltage that would appear on the output of this
        code model if zero-valued sourcing and sinking resistances
        were specified for the model.

    Else
        Set the initial-pass values of veq to the average of the power
        inputs; also set the three partials of output w.r.t. 'in'
        'pos_pwr' and 'neg_pwr' to zero.

        Calculate Rout:
        If 'r_out_src' equals 'r_out_sink',
            Set r_out to 'r_out_src', and the partial of r_out w.r.t.
            the output voltage to zero.

        Else
            Set r_out to smoothly vary between 'r_out_src' and 'r_out_sink',
            depending on the values of the difference between the ideal
            output (veq) and the actual voltage at the output ('out').
            Smoothing domain is +- 'r_out_domain' [cm_smooth_discontinuity()].

        Calculate the threshold voltages about the 'i_limit_sink' and
        'i_limit_source' parameter values.
```

```
Calculate i_out & derivatives. i_out is set to (veq - vout) / r_out.  
  
Preset i_pos_pwr, i_neg_pwr & partials to 0.0.  
  
Determine the operating point of i_out for limiting, and limit:  
  
If i_out is less than zero (limiter is sinking current),  
  
    If i_out is less than the lower threshold,  
  
        If i_out is less than 'i_limit_sink' minus 'limit_range',  
  
            fix i_out at '-i_limit_sink', and set derivatives to zero.  
  
        Else  
  
            i_out in lower smoothing region. Obtain a smoothed  
            value for i_out [cm_smooth_corner()], and set the  
            partial derivatives according to the returned values  
            from the function.  
  
    Else i_out is in the lower linear region...calculate i_neg_pwr.  
  
        if i_out is greater than -2.0 * 'i_sink_range',  
  
            i_out is near 0.0...smooth i_neg_pwr [cm_smooth_corner()].  
  
        Else  
  
            Not near i_out=0.0, so set i_neg_pwr = -i_out, and set the  
            partial derivatives as appropriate.  
  
Else i_out is sourcing :  
  
    If i_out is greater than the upper threshold voltage,  
  
        and If i_out is greater than 'i_limit_source' plus  
        'i_source_range',  
  
            fix i_out at 'i_limit_source', and set the partial  
            derivatives to zero.  
  
    Else  
        i_out is in the upper smoothing region. Obtain a smoothed  
        value for i_out [cm_smooth_corner()], and set the  
        partial derivatives according to the returned values  
        from the function.  
  
Else i_out is in the upper linear region...calculate i_pos_pwr:  
  
    If i_out is less than twice 'i_source_range',
```

```
i_out is near 0.0...smooth i_pos_pwr [cm_smooth_corner()].  
Else  
    Not near i_out=0.0, so set i_pos_pwr = -i_out, and set the  
    partial derivatives as appropriate.  
If ARGS indicates that this is a DC or Transient Analysis,  
    Output the values and derivatives.  
Else if this is an AC Analysis,  
    Output the AC gain values.  
}
```

4.2.1.11 Hysteresis Block

Summary

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

Called By

MIFload() MIF/MIFload.c

Returned Value

None.

PDL Description

```
void cm_hyst(ARGS)
ARGS = Mif_Private_t *private; /* INOUT - Code model inputs,
.. outputs, and parameters */

{
    Declare variables.

    Retrieve frequently used parameters...

    Calculate Hysteresis Linear Region Slopes & Derived Values.

    Define slope of rise and fall lines when not being smoothed.
    through the use of the formula slope = ('out_upper_limit' -
    'out_lower_limit')/('in_high' - 'in_low').

    Set the four "breakpoints" for the hysteresis transfer function.
```

These are x_rise_linear and x_rise_zero for the positive-going direction, and x_fall_linear and x_fall_zero for the negative-going direction.

If ARGS indicates that a fractional rather than an absolute value is represented by the 'input_domain' parameter, obtain an absolute value by multiplying 'input_domain' by ('in_high' - 'in_low').

Retrieve the 'in' value.

If this is the first call to the model,

Allocate storage for the current hysteresis state. This variable lets the model know which "leg" of the hysteresis curve the model is on at any time [cm_analog_alloc()].

Else

Retrieve the pointers to the current hysteresis state and the previous state (hyst_state and old_hyst_state, respectively) [cm_analog_get_ptr()].

Preset *hyst_state to equal *old_hyst_state.

If the previous hysteresis state indicated that the model was operating on the lower leg (X_RISING) of the hysteresis curve,

And if 'in' was less than the first breakpoint (x_rise_linear) minus the calculated input_domain,

Set the output to 'out_lower_limit'.

Set the partial of the output w.r.t. 'in' to zero.

Else

If 'in' was less than x_rise_linear plus the input_domain,

Calculate the output based on a smoothed transition from the hard-limited region to the linear rising region [cm_smooth_corner()]. Set the partial derivative based on output from cm_smooth_corner().

Else

If 'in' was less than x_rise_zero minus the input_domain,

Calculate the output based on the linear rise from out_lower_limit to out_upper_limit. Set the partial derivative to the slope.

```
Else

    If 'in' was less than x_rise_zero plus input_domain,
        Calculate the output based on a smoothed transition from
        the linear rising region to the hard-limited region
        [cm_smooth_corner()]. Set the partial derivative based
        on output from cm_smooth_corner().

    Else

        The input has transitioned into the X_FALLING region
        by moving above x_rise_zero plus input_domain. Set
        the output to 'out_upper_limit', set the partial
        derivative to zero, and set the hysteresis state
        to X_FALLING for the next call.

Else the previous hysteresis state indicated that the model
was operating on the upper leg (X_FALLING) of the hysteresis curve,

    If 'in' was greater than the x_fall_linear
    plus the calculated input_domain,
        Set the output to 'out_upper_limit'.
        Set the partial of the output w.r.t. 'in' to zero.

    Else

        If 'in' was greater than x_rise_linear minus the input_domain,
            Calculate the output based on a smoothed transition from
            the hard-limited region to the linear falling region
            [cm_smooth_corner()]. Set the partial derivative based
            on output from cm_smooth_corner().

        Else

            If 'in' was greater than x_fall_zero plus the input_domain,
                Calculate the output based on the linear fall from
                out_upper_limit to out_lower_limit. Set the partial
                derivative to the slope.

            Else

                If 'in' was greater than x_rise_zero minus input_domain,
                    Calculate the output based on a smoothed transition from
                    the linear falling region to the hard-limited region
                    [cm_smooth_corner()]. Set the partial derivative based
```

on output from cm_smooth_corner().

Else

The input has transitioned into the X_RISING region by moving below x_fall_zero minus input_domain. Set the output to 'out_lower_limit', set the partial derivative to zero, and set the hysteresis state to X_RISING for the next call.

If ARGS indicates that this is a DC or Transient analysis,

Output DC & Transient Analyses values.

Else this is an AC analysis:

Output the AC gain value.

}

4.2.1.12 Differentiator

Summary

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

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

Called By

`MIFload()` `MIF/MIFload.c`

Returned Value

None.

PDL Description

```
void cm_d_dt(ARGS)

ARGS = Mif_Private_t *private; /* INOUT - Code model inputs,
.. outputs, and parameters */

{
    Define variables.

    Retrieve the 'gain' parameter. This is the only parameter used
    by all analysis types.

    If ARGS indicates that this is a DC or Transient analysis,
        Retrieve parameters used in DC or Transient analyses only.
}
```

```
If ARG$ indicates that this is the first call to the model,  
    Allocate storage for the current and previous values  
    of the input [cm_analog_alloc()].  
  
Else,  
    Retrieve the pointers to the previous and current  
    input value storage locations [cm_analog_get_ptr()].  
  
If this is the first timepoint calculation (TIME = 0.0),  
    return a zero d/dt value.  
  
If not,  
    calculate d_dt: out = gain * (*in - *in_old) /  
    (current timepoint - previous timepoint) + out_offset  
    calculate the partial derivative = gain / (current timepoint -  
    previous timepoint)  
    Smooth the output if it is within 'limit_range' of  
    'out_lower_limit' or 'out_upper_limit' [cm_smooth_corner()].  
    Output values for DC & Transient, and partial derivative of  
    the output w.r.t. the input.  
    Also make one call to the cm_integrate routine to make sure that  
    this derivative value is fairly accurate...an integration call  
    forces an evaluation of the truncation error criterion, and can  
    in turn force the simulator to abandon the current time point  
    for one which is closer to the previous one in order to improve  
    accuracy [cm_analog_integrate()].  
  
Else  
    This is an AC Analysis...output (0.0, s*gain). The value of  
    "s" is obtained from the macro RAD_FREQ.  
}
```

4.2.1.13 Integrator

Summary

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

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

Called By

`MIFload()` MIF/MIFload.c

Returned Value

None.

PDL Description

```
void cm_int(ARGS)
{
    Mif_Private_t *private; /* INPUT - Code model inputs,
                           .. cputs, and parameters */

    Define variables.

    Retrieve parameters used by all analyses (in this case, only 'gain').
```

```
If ARGS indicates that this is a DC or Transient analysis,  
    Retrieve parameters used only in DC or Transient analyses.  
  
If ARGS indicates that this is the first call to the model,  
    Allocate storage for the previous input and output  
    values [cm_analog_alloc()].  
  
Else,  
    Retrieve the pointers to the previous input and output  
    value storage locations [cm_analog_get_ptr()].  
  
Read input value for current time, and calculate pseudo-input  
which includes input offset and gain according to the equation  
•in = 'gain' * ('in' + 'in_offset').  
  
If this is the first timepoint calculation...  
    Set the output equal to the initial condition value, 'out_ic',  
    and set the partial derivative of the output w.r.t. the input  
    to zero.  
  
Else it is NOT the first calculation:  
    Call the cm_integrate function to calculate the  
    output value and the partial derivative [cm_integrate()].  
  
Smooth output if it is within limit_range of  
out_lower_limit or out_upper_limit:  
    If the output value is less than ('out_lower_limit'-'limit_range'),  
        Set the output to 'out_lower_limit', and the partial to zero.  
    Else  
        If the output value is less than ('out_lower_limit'+  
            'limit_range'),  
            Smooth the output value between 'out_lower_limit' and  
            the actual value, and obtain a smoothed derivative  
            value [cm_smooth_corner()].  
        Else  
            If the output value is greater than ('out_upper_limit' +  
                'limit_range'),
```

Set the output to 'out_upper_limit', and the partial derivative to zero.

Else

If the output value is greater than ('out_upper_limit' - 'limit_range'),

Smooth the output value between 'out_upper_limit' and the actual value, and obtain a smoothed derivative value [cm_smooth_corner()].

If ARGS indicates that this is a DC or Transient analysis,

Post the output and derivative values.

Else this is an AC analysis:

Output the AC gain value (0.0,gain/s).

}

4.2.1.14 S-Domain Transfer Function

Summary

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

- The degree of the numerator polynomial cannot exceed that of the denominator polynomial in the variable "s".
- The coefficients for each term must be stated explicitly. That is, if a coefficient is zero, it must be included as an input to the num_coeff or den_coeff array.

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

```
.model filter s_xfer(gain=0.139713 num_coeff=[1.0 0.0 0.07464102]
+           den_coeff=[1.0 0.0998942 0.001170077])
```

specifies a transfer function of the form:

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

Likewise, the following equation:

```
.model filter s_xfer(gain=66.6 num_coeff=[0.666]
+           den_coeff=[1.0 6.66 666.0 1.0])
```

specifies a transfer function of the form:

$$N(s) = 66.6 \frac{0.666}{s^3 + 6.66s^2 + 666.0s + 1.0}$$

The s-domain transfer function includes gain and input offset parameters to allow for tailoring of the required signal. There are no limits on the internal signal values or on the output value of the s-domain transfer function, so the user is cautioned to specify gain and

coefficient values that will not cause the model to produce excessively large values. In AC analysis, the value returned is equal to the real and imaginary components of the total s-domain transfer function at each frequency of interest.

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

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

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

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

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

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

Called By

MIFload() MIF/MIFload.c

Returned Value

None.

PDL Description

```
static Mif_Complex_t cm_complex_div(Mif_Complex_t x, Mif_Complex_t y)

Mif_Complex_t x; /* IN - The numerator value */
Mif_Complex_t y; /* IN - The denominator value */

{

    Declare variables.

    Calculate the magnitude and phase of x.

    Calculate the magnitude and phase of y.

    Calculate the output magnitude as (mag_x / mag_y).

    Calculate the output phase as (phase_x - phase_y).

    Calculate the real portion of the output.

    Calculate the imaginary portion of the output.

    Return the complex output value.

}

void cm_s_xfer(ARGS)

ARGS = Mif_Private_t *private; /* INOUT - Code model inputs,
.. outputs, and parameters */

{

    Retrieve frequently used parameters.

    If the denormalized frequency is not specified, preset it
    to 1.0. Otherwise, retrieve the value of 'denormalized_freq'.

    If the number of terms in the numerator coefficient vector ('num_coeff')
    is greater than the number of terms in the denominator coefficient
    vector ('den_coeff'),

        Send an error message to the simulator [cm_message_send()], and
        return.

    If ARGS indicates that this is the initial call to the function,
        Create an array of pointers for the integrator current and
        the previous output values (two arrays in all). There will
}
```

be as many integrators in the model as there are denominator coefficient terms.

Allocate rotational storage for in & out, gain, denominator coefficients and their previous values [calloc()].

Else

Set pointers to storage locations for in, out, and integrators [cm_analog_get_ptr()].

Retrieve previous timepoint input values as necessary in subsequent analysis sections.

Gain has to be stored each time since it could possibly change if the highest order denominator coefficient isn't zero.

If ARGS indicates that the current timepoint is zero,

Set initial conditions:

Initialize integrators to int_ic condition values.
Note...do NOT set the highest order value...this represents the "calculated" input to the actual highest integrator...it is NOT a true state variable.

If necessary, denormalize all of the coefficient values.

Test denominator highest order coefficient...if that value is other than 1.0, then divide all denominator coefficients and the gain by that value...

If ARGS indicates that this is a DC or TRANSIENT analysis,

If this is specifically a DC analysis,

If the zero-th order denom coefficient is zero,

Since a term does not exist for the zero-th order denom coeff,
Avoid division by zero (i.e., num_coeff[0]/den_coeff[0]):
Output initial conditions instead.

Else

Zero-th order denominator term is not zero: calculate the steady state value of the output as:
'out' = 'gain' * ('in' + 'in_offset') *
('num_coeff[0]' / 'den_coefficient[0]').

Else this is a Transient analysis:

Read the input value for the current time, and calculate the pseudo-input, which includes input offset and gain.

Calculate the "new" input to the first integrator, based on the pseudo-input summed with the old values of each subsequent integrator multiplied by their respective denominator coefficients and then subtracted from the input. Note that this value, which is similar to a state variable, is stored in *(integrator[den_size-1]).

Propagate the "new" input through each integrator in succession and calculate each integrator output value [cm_analog_integrate()] (It should be noted that this step "distributes" the input through the Controller Canonical topology used in this model; each pass through the loop integrates the input to that stage, which in turn produces the next stage's input).

Calculate the actual output from the model by multiplying each integrator-stage output by the numerator coefficient values, and summing.

Output the values for DC & Transient analyses.

Else this is an AC analysis:

Calculate the real & imaginary portions of the AC gain at the current RADFREQ point:

First calculate the numerator real & imaginary components:

For each of the numerator coefficients,

Divide i by 2, placing the resulting integer portion in the variable divide_integer and the fractional part in the variable frac.

Now divide divide_integer by 2 a second time. If the fractional part of this division is greater than zero,

then this is a NEGATIVE coefficient value for this iteration:

If frac is greater than zero,

This is an odd power of "s" => accumulate this coefficient term into the imaginary portion of the numerator

Else

This is an even power of "s" => accumulate this coeff. term into the real portion of the numerator

Else if the fractional part of divide_integer/2 is zero,

then this is a POSITIVE coefficient value for this iteration:

If frac is greater than zero,

This is an odd power of "s" => accumulate this coefficient term into the imaginary portion of the numerator

Else

This is an even power of "s" => accumulate this coeff. term into the real portion of the numerator

Calculate denominator real & imaginary components in the same fashion.

For each of the denominator coefficients,

Divide i by 2, placing the resulting integer portion in the variable divide_integer and the fractional part in the variable frac.

Now divide divide_integer by 2 a second time. If the fractional part of this division is greater than zero,

then this is a NEGATIVE coefficient value for this iteration:

If frac is greater than zero,

This is an odd power of "s" => accumulate this coefficient term into the imaginary portion of the denominator.

Else

This is an even power of "s" => accumulate this coeff. term into the real portion of the denominator.

```
Else if the fractional part of divide_integer/2 is zero.  
    then this is a POSITIVE coefficient value for  
        this iteration:  
  
    If frac is greater than zero,  
        This is an odd power of "s" => accumulate this coefficient  
            term into the imaginary portion of the denominator.  
  
    Else  
        This is an even power of "s" => accumulate this coeff.  
            term into the real portion of the denominator.  
  
Divide numerator values by denominator values using the complex  
divide function [cm_complex_div()].  
  
Output the ac_gain values.  
  
Free all allocated memory.  
}
```

4.2.1.15 Slew Rate Block

Summary

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

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

Called By

MIFload() MIF/MIFload.c

Returned Value

None.

PDL Description

```
void cm_slew(ARGS)

ARGS = Mif_Private_t *private; /* INOUT - Code model inputs,
.. outputs, and parameters */

{
    Retrieve frequently used parameters (used by all analyses).

    If ARGS indicates that this is the first time through model,
        Allocate storage for current and previous input values
        [cm_analog_alloc()], and for the current and previous
        output values.
```

If ARGS indicates that this is a DC analysis,

 Retrieve the pointers for the inputs and output values
 [cm_analog_get_ptr()].

 Set old values for the input and output equal to the
 current input value: for DC, no slewing occurs, so the
 model acts as a unity-gain buffer.

 Set the partial derivative to 1.0.

 Output 'out' and the partial.

Else

If ARGS indicates that this is a TRANSIENT analysis,

 Retrieve the rise and fall slope value parameters.

 Retrieve the previous input and output values.

If ARGS indicates that this is the first timepoint calculation,

 Set old values for the input and output equal to the
 current value of 'in'.

 Set the partial derivative to 1.0.

 Output 'out' and the partial.

Else

 Determine the slope of the input from:

 slope = ('in' - in_old)/(current_time - previous_timepoint);

 If the calculated slope value is positive,

 Determine the preliminary slewed output as:

 out_slew = out_old + 'rise_rise' * (current_time - previous_time)

 If the output was slewing in the negative direction and
 the current input is less than the slewed output, but
 rising (slope positive),

 This means that the input has changed directions and
 the "slewed" response hasn't caught up to the input yet.

 Continue negative slewing until the slewed output
 meets the positive sloping input. Output slewed

```
    output value, but set d/dt=0.

Else

If model is in a standard slewing state,

(Aside: Two conditions for slewing, if the slope is greater
than the positive slew rate, or if the input slope
is less than the positive slew rate and the slewed output
is less than the input. This second condition occurs
if the input levels off and the slewed output hasn't
caught up to the input yet.)

Output slewed output value. Set d/dt.

Else if neither of the above conditions is true,

No slewing, output=input, d/dt = 1.

Else

If the calculated slope value is negative,

Calculate the slewed output value according to:
out_slew = out_old - 'fall_slope' * (current_time - previous_time)

If the output was slewing in a positive direction and
the current input is greater than the slewed output, but
falling (slope negative),

(See comment above in positive input slope section).

Continue positive slewing until the slewed output
meets the negative sloping input. Output positive
slewed value, set d/dt = 0.

Else

If model is in a standard slewing state,

Output slewed output value. Set d/dt = 0.

Else

No slewing. Set output = input, d/dt = 1.

Output values for Transient analysis.

Else

If ARGS indicates that this is an AC analysis,
```

Output the AC gain (which always equals 1+j0; note that this is not an accurate reflection of the AC gain in the case of an AC frequency/amplitude combination which would in fact cause slewing to occur, but for the vast majority of cases, a real gain of 1.0 is acceptable for this model).

}

4.2.1.16 Inductive Coupling

Summary

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

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

The most common use for inductive couplings will be as a building block in the construction of transformer models. To create a transformer with a single input and a single output, the user would require two inductive coupling models plus one magnetic core model. The process of building up such a transformer is described under the description of the magnetic core model, below.

Called By

MIFload() MIF/MIFload.c

Returned Value

None.

PDL Description

```
void cm_lcouple(ARGS)
ARGS = Mif_Private_t *private; /* I/MOUT - Code model inputs,
.. outputs, and parameters */

{
    Retrieve the 'num_turns' parameter.
```

```
If ARGS indicates that this is a DC or Transient analysis,  
  
    If ARGS indicates that this is the initialization pass  
    for the model,  
  
        Allocate storage for the flux input value [cm_analog_alloc()],  
        and retrieve the pointer for the old flux value  
        [cm_analog_get_ptr()].  
  
    Else,  
  
        Retrieve previous timepoint flux value [cm_analog_get_ptr()].  
        Also retrieve fake input flux and output voltage values  
        for use in truncation error checking.  
  
    Retrieve inputs...  
  
    (The input from electrical side is a current.  
    the input from core side is a flux  
    represented as a current...note  
    that a negative is introduced,  
    because current INTO the positive  
    node would normally result in  
    a NEGATIVE output_voltage...  
    the minus sign corrects this.)  
  
Calculate and post the output value for mmf according to:  
output_mmf = num_turns * input_current.  
  
Calculate and post the output value for output_voltage  
as follows:  
If the current analysis time is zero,  
  
    output_voltage = 0.0  
    Return a zero d/dt value.  
  
Else  
  
    output_voltage = num_turns * (*in_flux - *in_flux_old) /  
    (current_timepoint - previous_timepoint).  
    Calculate value of d_dt = -num_turns / (current_time - previous_time).  
  
Else ARGS indicates that this is an AC analysis:  
  
    Output AC gain values:  
  
    For mmf_out to 1, AC gain = 0 + j('num_turns' * RAD_FREQ).  
    For 1 to mmf_out, AC gain = 'num_turns' + j0.  
}
```

4.2.1.17 Magnetic Core

Summary

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

PWL Mode (mode = 1)

The core model in PWL mode takes as input a voltage which it treats as a magnetomotive force (mmf) value. This value is divided by the total effective length of the core to produce a value for the Magnetic Field Intensity, H. This value of H is then used to find the corresponding Flux Density, B, using the piecewise linear relationship described by the user in the H_array / B_array coordinate pairs. B is then multiplied by the cross-sectional area of the core to find the Flux value, which is output as a current. The pertinent mathematical equations are described below:

$$H = mmf/L$$

where L = Length and H, the Magnetic Field Intensity, is expressed in ampere-turns/meter.

$$B = f(H)$$

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

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

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

Called By

MIFload() MIF/MIFload.c

Returned Value

None.

PDL Description

```
void cm_core(ARGS)

ARGS = Mif_Private_t *private; /* INOUT - Code model inputs.
.. outputs, and parameters */

{
    Retrieve the 'mode' parameter.

    If 'mode' indicates that the model is operating in PWL
    mode,

    Retrieve frequently used parameters 'input_domain', 'area',
    and 'length', plus the size of the H_array.

    Allocate storage for breakpoint domain & range values
    [calloc()], and pass the simulator an error message if
    allocation fails.
    (These are the H and B fields, H is the driver in the
     model...the B field is the resulting value (modeled
     as a current)

    Retrieve H the and B array ('H_array[i]' and 'B_array[i]')
    values.

    See if input_domain is an absolute rather than a relative
    value...if so, test to make sure that it is not greater
    than half of the smallest of the H_array segments (if
    this were so, the smoothing domain would not fit into
    the smallest segment, and unexpected results would be
    forthcoming). If it IS greater than half of one of
    the segments, send an error message to the simulator.

    Retrieve mmf_input value (taken from the 'mc' port).

    Calculate H_input value from mmf_input...

    Determine segment boundaries within which H_input resides
```

```
If the H input is below lowest H_array segment midpoint,  
Calculate dout_din as the slope defined by the two  
lowest (H,B) coordinate points,  
and use the slope to calculate what the B output value  
will be from the equation:  
B_out = B[0] + (H_input - H[0]) * dout_din.  
  
Else  
  
if the H input is above highest H_array segment midpoint,  
Calculate dout_din as the slope defined by the two  
highest (H,B) coordinate points,  
and use the slope to calculate what the B output value  
will be from the equation:  
B_out = B[size-1] + (H_input - H[size-1]) * dout_din.  
  
Else  
  
H_input within bounds of end midpoints...  
must determine position progressively & then  
calculate required output.  
  
The input is between the two outer H-coordinate  
endpoints. Starting with the next-to-lowest  
H value, test the input until we know in which  
coordinate point region the input resides...  
  
Progressively step through the possible segment midpoints,  
and find which of them is greater than H_input. Once this  
midpoint is found, the approximate position of H_input is  
known to be less than the current midpoint, but greater  
than the midpoint of the segment with length  
('H[i]' - 'H[i-1]'). Note that this means that H_input  
inhabits a region of the H domain which also includes  
a breakpoint. The breakpoint corresponds to the common  
point between segment ('H[i]'-'H[i-1]) and segment  
('H[i+1]'-'H[i]').  
  
Calculate the length of the segments above and below the  
breakpoint in the current region.  
  
If the 'fraction' parameter is TRUE,  
calculate a fixed value for the smoothing domain about
```

the breakpoint, based on the smallest of the two segments, ('H[i]'-'H[i-1]') and ('H[i+1]'-'H[i]'). If 'fraction' is FALSE, this is unnecessary.

Set up threshold values about breakpoint.

Determine where H_input is within region & determine output and partial values:

If H_input is less than the lower threshold value,

calculate the slope of the lower segment, and use
 $B = \text{slope} * (H_{\text{input}} - 'H[i]') + B[i]$ to obtain
the output value.

else if H_input is less than the upper threshold,

calculate a smoothed value for the output and the
output partial derivative [cm_smooth_corner()].

else H_input is in the upper segment.

calculate the slope of the upper segment, and use
 $B = \text{slope} * (H_{\text{input}} - 'H[i]') + B[i]$ to obtain
the output value.

Calculate value of flux_out as $\text{flux_out} = B_{\text{out}} * \text{'area'}$.

Adjust dout_din value to reflect area and length multipliers
according to the equation: $dout_{\text{din}} = dout_{\text{din}} * \text{'area' / 'length'}$.

If ARGS indicates that this is a DC or Transient analysis,

Output DC & Transient Values.

Else,

Output the AC gain value.

Else if 'mode' indicates that the model is to operate in
HYSTERESIS mode,

Retrieve frequently used parameters...

Calculate Hysteresis Linear Region Slopes & Derived Values.

Define slope of rise and fall lines when not being smoothed.
through the use of the formula $\text{slope} = (\text{'out_upper_limit'} - \text{'out_lower_limit'}) / (\text{'in_high'} - \text{'in_low'})$.

Set the four "breakpoints" for the hysteresis transfer function.
These are x_rise_linear and x_rise_zero for the positive-going
direction, and x_fall_linear and x_fall_zero for the negative-
going direction.

If ARGS indicates that a fractional rather than an absolute value
is represented by the 'input_domain' parameter, obtain an absolute
value by multiplying 'input_domain' by ('in_high' - 'in_low').

Retrieve the 'in' value.

If this is the first call to the model,

Allocate storage for the current hysteresis state [cm_analog_alloc()],
and retrieve the pointer for the old state [cm_analog_get_ptr()].
This variable lets the model know which "leg" of the hysteresis
curve the model is on at any time.

Else

Retrieve the pointers to the current hysteresis [cm_analog_get_ptr()]
state and the previous state (hyst_state and old_hyst_state,
respectively).

Preset *hyst_state to equal *old_hyst_state.

If the previous hysteresis state indicated that the model was operating
on the lower leg (X_RISING) of the hysteresis curve,

And if 'in' was less than the first breakpoint (x_rise_linear)
minus the calculated input_domain,

Set the output to 'out_lower_limit'.

Set the partial of the output w.r.t. 'in' to zero.

Else

If 'in' was less than x_rise_linear plus the input_domain,

Calculate the output based on a smoothed transition from
the hard-limited region to the linear rising region
[cm_smooth_corner()]. Set the partial derivative based
on output from cm_smooth_corner().

Else

If 'in' was less than x_rise_zero minus the input_domain,

Calculate the output based on the linear rise from
out_lower_limit to out_upper_limit. Set the partial

```
derivative to the slope.

Else

If 'in' was less than x_rise_zero plus input_domain,
Calculate the output based on a smoothed transition from
the linear rising region to the hard-limited region
[cm_smooth_corner()]. Set the partial derivative based
on output from cm_smooth_corner().

Else

The input has transitioned into the X_FALLING region
by moving above x_rise_zero plus input_domain. Set
the output to 'out_upper_limit', set the partial
derivative to zero, and set the hysteresis state
to X_FALLING for the next call.

Else the previous hysteresis state indicated that the model
was operating on the upper leg (X_FALLING) of the hysteresis curve,
If 'in' was greater than the x_fall_linear
plus the calculated input_domain,
Set the output to 'out_upper_limit'.
Set the partial of the output w.r.t. 'in' to zero.

Else

If 'in' was greater than x_rise_linear minus the input_domain,
Calculate the output based on a smoothed transition from
the hard-limited region to the linear falling region
[cm_smooth_corner()]. Set the partial derivative based
on output from cm_smooth_corner().

Else

If 'in' was greater than x_fall_zero plus the input_domain,
Calculate the output based on the linear fall from
out_upper_limit to out_lower_limit. Set the partial
derivative to the slope.

Else

If 'in' was greater than x_rise_zero minus input_domain,
Calculate the output based on a smoothed transition from
```

the linear falling region to the hard-limited region [cm_smooth_corner()]. Set the partial derivative based on output from cm_smooth_corner().

Else

The input has transitioned into the X_RISING region by moving below x_fall_zero minus input_domain. Set the output to 'out_lower_limit', set the partial derivative to zero, and set the hysteresis state to X_RISING for the next call.

If ARGS indicates that this is a DC or Transient analysis,

Output DC & Transient Analyses values.

Else this is an AC analysis:

Output the AC gain value.

}

4.2.1.18 Controlled Sine Wave Oscillator

Summary

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

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

Called By

`MIFload()` `MIF/MIFload.c`

Returned Value

None.

PDL Description

```
void cm_sine(ARGS)

ARGS = Mif_Private_t *private; /* INOUT - Code model inputs,
.. outputs, and parameters */

{
    Retrieve frequently used parameters 'out_low' and 'out_high'.
    In addition, retrieve the sizes of the 'freq_array' and
    'cntl_array' vectors.

    If the 'freq_array' and 'cntl_array' sizes are not equal,
        Send an error message to the simulator [cm_message_send()]
        and return.

    If this is the initial pass through the model,
        Allocate memory for the phase value [cm_analog_alloc()].
```

```
If ARGS indicates that this is a DC analysis,  
  
    Output the average of the output high and output low voltages.  
    Output partial derivative (output wrt input) = zero.  
    Set the phase equal to zero.  
  
Else  
  
If ARGS indicates that this is a TRANSIENT analysis,  
  
    Retrieve the previous phase value [cm_analog_get_ptr()].  
  
    Allocate storage for control input and frequency breakpoint  
    range values [calloc()].  
  
    Retrieve 'cntl_array[i]' and 'freq_array[i]' values.  
  
    Retrieve 'cntl_in' input value.  
  
Determine segment boundaries within which cndl_input resides:  
If 'cntl_input' is less than 'cntl_array[0]',  
  
    Calculate the frequency as: freq = ('cntl_input' - 'cntl_array[0]') *  
        (slope of 'freq_array' vs. 'cntl_array' segment) + 'freq_array[0]'.  
  
    If this frequency is less than zero,  
  
        send a message to the simulator to that effect  
        [cm_message_send()], and clamp the frequency to 1e-6 Hz.  
  
Else  
  
    If 'cntl_input' is greater than 'cntl_array[size-1]',  
  
        Calculate the frequency as: freq = ('cntl_input' -  
            'cntl_array[size-1]') * (slope of 'freq_array' vs. 'cntl_array'  
            segment) + 'freq_array[size-1]'.  
  
    Else  
  
        The input value is between the 'cntl_array' endpoints.  
        Loop through each possible segment defined by the  
        'cntl_array' breakpoint values until the one in which  
        'cntl_in' resides is found. At that point, calculate the  
        slope of that segment, then calculate the frequency  
        value corresponding to 'cntl_in' by the following  
        equation: freq = ('cntl_input' - 'cntl_array[i]') *  
            (slope of 'freq_array' vs. 'cntl_array' segment)  
            + 'freq_array[i-1]'.
```

Calculate the peak value of the wave, the center of the wave, the instantaneous phase and the radian value of the phase.

Output the value of the sinewave and the partial derivative of the output w.r.t. the control input (zero).

Else

If ARGS indicates that this is an AC analysis,

This model has no AC capabilities. Set all AC values to zero,
and exit.

}

4.2.1.19 Controlled Triangle Wave Oscillator

Summary

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

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

Called By

`MIFload()` MIF/MIFload.c

Returned Value

None.

PDL Description

```
void cm_triangle(ARGS)
ARGS = Mif_Private_t *private; /* INOUT - Code model inputs,
.. outputs, and parameters */

{
    Retrieve frequently used parameters.

    If the 'freq_array' and 'cntl_array' sizes are not equal,
        Send an error message to the simulator [cm_message_send()]
        and return.

    If this is the initial pass through the model,
        Allocate memory for the phase, t1, t2 and t_end
        values [cm_analog_alloc()].
```

```
If ARG$ indicates that this is a DC analysis,  
    Retrieve and initialize time values [cm_analog_get_ptr()].  
    Output DC values of output (= 'out_low') and partial (=zero).  
Else  
If ARG$ indicates that this is a TRANSIENT analysis,  
    Retrieve pointers to previous values [cm_analog_get_ptr()],  
    and preset time1, time2 and t_start.  
    Allocate storage for 'cntl_array' breakpoint domain & freq.  
    range values [calloc()].  
    Retrieve 'cntl_array' and 'freq_array' values.  
    Retrieve 'cntl_input' value.  
    Determine segment boundaries within which cntl_input resides.  
    If 'cntl_input' is less than 'cntl_array[0]',  
        Calculate the frequency as: freq = ('cntl_input' - 'cntl_array[0]') *  
        (slope of 'freq_array' vs. 'cntl_array' segment) + 'freq_array[0]'.  
        If this frequency is less than zero,  
            send a message to the simulator to that effect  
            [cm_message_send()], and clamp the frequency to 1e-6 Hz.  
    Else  
        If 'cntl_input' is greater than 'cntl_array[size-1]',  
            Calculate the frequency as: freq = ('cntl_input' -  
            'cntl_array[size-1]') * (slope of 'freq_array' vs. 'cntl_array'  
            segment) + 'freq_array[size-1]'.  
        Else  
            The input value is between the 'cntl_array' endpoints.  
            Loop through each possible segment defined by the  
            'cntl_array' breakpoint values until the one in which  
            'cntl_in' resides is found. At that point, calculate the  
            slope of that segment, then calculate the frequency  
            value corresponding to 'cntl_in' by the following  
            equation: freq = ('cntl_input' - 'cntl_array[i]') *  
            (slope of 'freq_array' vs. 'cntl_array' segment)  
            + 'freq_array[i-1]'.
```

```
Calculate instantaneous phase:  
Instantaneous phase is the old phase + frequency/(delta time)  
  
If the current time is greater than time1, but less than time2,  
  
    calculate time2 and set the temporary breakpoint  
    [cm_analog_set_temp_bkpt()].  
  
    Set output value.  
  
Else  
  
    Otherwise, calculate time1 and time2 and set their respective  
    breakpoints [cm_analog_set_temp_bkpt()].  
  
    Set output value.  
  
}  
  
Set partials (model supposes no appreciable connection between  
the control input and the output, hence all partials are set  
to zero).  
  
Set the stored time values for this call for time1, time2, and  
t_end (=t_start).  
  
Else  
  
If ARGS indicates that this is an AC analysis,  
  
    This model has no AC capabilities, so post all AC gains as  
    equal to zero.  
}
```

4.2.1.20 Controlled Square Wave Oscillator

Summary

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

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

Called By

MIFload() MIF/MIFload.c

Returned Value

None.

PDL Description

```
void cm_square(ARGS)

ARGS = Mif_Private_t *private; /* INOUT - Code model inputs,
.. outputs, and parameters */

{
    Retrieve frequently used parameters.

    If the 'freq_array' and 'cntl_array' sizes are not equal,
        Send an error message to the simulator [cm_message_send()]
        and return.

    If ARGS indicates that this is the first iteration,
        Allocate memory for the phase, and for the rolling values
        of T1, T2, T3 and T4 (as defined in graphic, above)
```

[cm_analog_alloc()].

If ARGS indicates that this is a DC analysis,

Initialize T1, T2, T3 and T4 times to negative values [cm_analog_get_ptr()] so that the algorithm can correctly set them on the next pass.

Set output = output_low and partial = 0.

Else

If ARGS indicates that this is a TRANSIENT analysis,

Retrieve previous values [cm_analog_get_ptr()], and set local variables to the values stored previously.

Allocate storage for 'cntl_array' breakpoint domain & 'freq_array' range values [calloc()].

Retrieve x ('cntl_array') and y ('freq_array') values for control and frequency values, respectively.

Retrieve 'cntl_input' value.

Determine segment boundaries within which cntl_input resides.

If 'cntl_input' is less than 'cntl_array[0]',

Calculate the frequency as: freq = ('cntl_input' - 'cntl_array[0]') * (slope of 'freq_array' vs. 'cntl_array' segment) + 'freq_array[0]'.

If this frequency is less than zero,

send a message to the simulator to that effect [cm_message_send()], and clamp the frequency to 1e-6 Hz.

Else

If 'cntl_input' is greater than 'cntl_array[size-1]',

Calculate the frequency as: freq = ('cntl_input' - 'cntl_array[size-1]') * (slope of 'freq_array' vs. 'cntl_array' segment) + 'freq_array[size-1]'.

Else

The input value is between the 'cntl_array' endpoints.
Loop through each possible segment defined by the
'cntl_array' breakpoint values until the one in which
'cntl_in' resides is found. At that point, calculate the

slope of that segment, then calculate the frequency value corresponding to 'cntl_in' by the following equation: freq = ('cntl_input' - 'cntl_array[i]') * (slope of 'freq_array' vs. 'cntl_array' segment) + 'freq_array[i-1]'.

Determine the instantaneous phase, the integer number of cycles, and the fractional part into the current cycle.
Instantaneous phase is the old phase + frequency/(delta time)

Calculate values for time1, time2, time3, and time4 and output depending on what part of the cycle the iteration is in.

If the current time is greater than time1 and less than time2,

Set time3 and time4 and set their temporary breakpoints. Set the breakpoint for time2 if necessary.

Set the output to 'out_low' + the appropriate value dependent on the risetime and 'out_high' and 'out_low' values.

Else

If the current time is greater than time2 and less than time3,

Set time4 temporary breakpoints. Set time3 breakpoint if necessary.

Set the output to 'out_high'.

Else

If the current time is greater than time3 and less than time4){

Set time1 and time2 and their respective breakpoints. Set time4 breakpoint if necessary.

Set the output to 'out_high' minus the appropriate value dependent on the falltime and the 'out_high' and 'out_low' values.

Else

Set time1 and time2. Set breakpoint for time2 and for time1 if appropriate.

Set the output to 'out_low'.

Set the partial of out w.r.t. cntl_in to zero.

Set the time values for storage.

Else

if ARGS indicates that this is an AC analysis,

This model has no AC capabilities...set all AC gains to zero.

}

4.2.1.21 Controlled Oneshot

Summary

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

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

Called By

`MIFload()` MIF/MIFload.c

Returned Value

None.

PDL Description

```
void cm_oneshot(ARGS)

ARGS = Mif_Private_t *private; /* INPUT - Code model inputs,
.. outputs, and parameters */

{
    Retrieve frequently used parameters...

    Set minimum rise and fall_times of 1.0e-12.

    Ensure that the control array is the same size as the
    pulse-width array; if not, send an error message to the
    simulator [cm_message_send()] and return.

    If ARGS indicates that this is the first call to the
    model,
```

Allocate memory for T1, T2, T3, T4, set, state,
clock, locked and output_old [cm_analog_alloc()].

If ARGS indicates that this is a DC analysis,

Retrieve pointers to all stored values [cm_analog_get_ptr()].

Initialize all time values to -1.0, and all other values
to zero except output_old. Set that to 'out_low'.

Set the output to 'out_low'. Also, if the model
makes use of an 'in' port, set the partial of the output
w.r.t. 'in' to zero. Likewise, if the 'clear' port
exists, set the partial of the output w.r.t. 'clear'
to zero.

Else,

if ARGS indicates that this is a TRANSIENT analysis,

Retrieve previous values, set local variables to be
equal to them [cm_analog_get_ptr()].

Note that these pointer values are immediately dumped into
other variables because the previous values can't change-
can't rewrite the old values.

If the 'clear' port exists for the model, and if its
value is TRUE,

Clear all time values to -1.0, and all other values
to zero except output_old. Set that to 'out_low'.

Else,

Allocate storage for the breakpoint domain & freq. range
values [calloc()] and return an allocation error if
a failure occurs [cm_message_send()].

Retrieve 'cntl_array' and 'pw_array' values.

Retrieve the 'cntl_in' value if it exists,
and the 'clk' value.

Determine segment boundaries within which 'cntl_in' resides.
If 'cntl_input' is less than 'cntl_array[0]',

Calculate the pulse width as: pw=('cntl_input'-'cntl_array[0]') *
(slope of 'pw_array' vs. 'cntl_array' segment) + 'pw_array[0]'.

If this pulse width is less than zero,

send a message to the simulator to that effect
[cm_message_send()], and clamp the pulse width to 1e-6 Hz.

Else

If 'cntl_input' is greater than 'cntl_array[size-1]',
Calculate the pulse width as: pw = ('cntl_input' -
'cntl_array[size-1]') * (slope of 'pw_array' vs. 'cntl_array'
segment) + 'pw_array[size-1]'.

Else

The input value is between the 'cntl_array' endpoints.
Loop through each possible segment defined by the
'cntl_array' breakpoint values until the one in which
'cntl_in' resides is found. At that point, calculate the
slope of that segment, then calculate the pulse width
value corresponding to 'cntl_in' by the following
equation: pw = ('cntl_input' - 'cntl_array[i]') *
(slope of 'pw_array' vs. 'cntl_array' segment)
+ 'pw_array[i-1]'.

If the model is supposed to trigger on a positive-going
clock edge,

And if the set1 variable is not set to 1 (which implies
that the oneshot is not is a 'set' condition),

look for:

1. a rising edge trigger and
2. the clock to be higher than the trigger value.
...if these exist,

then trigger the oneshot
by setting state1 and set1 to 1.

Else

Look for a neg edge before resetting the trigger.
if this is a neg edge, change set1 to 0.

Else

If the oneshot is supposed to trigger on a negative-going
clock edge,

And if the oneshot has not been set,

Look for:

1. A falling edge trigger and

2. The clock to be lower than the trigger value.
...if these exist,

trigger the oneshot by setting
state1 and set1 to 1.

Else

Look for a positive edge before resetting the trigger.
If this is a positive edge, change set1 to 0.

The model can only set the breakpoints for the output
pulse if state1 is high and the output is low, and
locked = 0 :

If oneshot is triggered and the output is low

Set the temporary breakpoints [cm_analog_set_temp_bkpt()],
and the storage variables that hold their time values.

If the oneshot is not retriggerable, disallow
setting of a new trigger point by setting locked1
to 1.

Set the current state to zero, and
set the output = output_low.

Else

If the oneshot had been triggered and the output is high,

Set time3 and time4 and their temporary breakpoints
[cm_analog_set_temp_bkpt()]

Set output = 'out_high', and set the state to 0.

Reset the state if it's 1 and the locked flag is 1. This
means that the clock tried to retrigger the oneshot, but
the retrig flag prevented it from doing so */

Set the value for the output depending on the current time, and
the values of time1, time2, time3, and time4:

If the current time is less than T1,

set the output to 'out_low'.

Else,

If the current time is between T1 and T2,

XSPICE Code Model Subsystem
Software Design Document

Detailed Design
Code Model Library

```
Interpolate the output based on the rise time to
'out_high'.

Else

If the current time is between T2 and T3,
    Set the output to 'out_high'.

Else

    if the current time is between T3 and T4,
        Interpolate the output based on the fall time to
        'out_low'.

    Else

        Set output to 'out_low'.

        If the model is not retriggerable,
            allow retriggering at this point by returning
            the locked1 value to 0.

Set the variables which need to be stored for the
next iteration [cm_analog_get_ptr()].
Set output, and all partials.

Else

If ARGS indicates that this is an AC analysis,
    This model has no AC capability, set everything to zero.
}
```

4.2.1.22 Capacitor

Summary

The capacitor is an alternative capacitor element to the one supplied in the standard SPICE3C1 simulator. This model provides the ability to add true initial conditions on it at time=0.0, and to support the ramptime option of the simulator.

Called By

MIFload() MIF/MIFload.c

Returned Value

None.

PDL Description

```
void cm_capacitor(ARGS)
ARGS = Mif_Private_t *private; /* INOUT - Code model inputs,
.. outputs, and parameters */

{
    Get the ramp factor from the .option ramptime [cm_analog_ramp_factor()].
    The ramp factor is a value from zero to one that is used to
    slowly ramp initial conditions up to their intended values from
    an initial value of zero. This aids in convergence.

    If ARGS indicates that this is the first call to this model,
        Create storage for the initial voltage on the capacitor
        [cm_analog_alloc()].
        Store the value of the initial condition voltage ('ic')
        multiplied by the ramp factor into this location. Note
        that the model can be called multiple times with the
        INIT value set to TRUE; each time, the ramp factor will
        be increased until it finally reaches 1.0.

    Else
        Retrieve the previous value of the voltage output by the
        model.
```

```
If ARGS indicates that this is a DC analysis,  
    Compute the output as 'ic' * ramp_factor.  
    Output the partial of the output voltage w.r.t. the input  
    current (at DC, this is equal to zero).  
  
Else if ARGS indicates that this is an AC analysis,  
    Output the AC gain value as the complex value:  
    (0, -1/(2*pi*freq*c)), where 'c' is the capacitance  
    parameter.  
  
Else if ARGS indicates that this is a TRANSIENT analysis,  
    If the ramp factor is still less than 1.0,  
        This means that the DC portion of the TRANSIENT analysis  
        is not complete yet.  
        Compute the output as 'ic' * ramp_factor.  
        Output the partial of the output voltage w.r.t. the input  
        current (at DC, this is equal to zero).  
  
    Else  
        Calculate the output voltage as the integral of the input  
        current divided by the capacitance parameter value  
        [cm_analog_integrate()]. The cm_analog_integrate function  
        also returns a partial derivative value, which must be  
        divided by 'c' to produce the final partial, since the  
        cm_analog_integrate function produces the partial of  
        the output w.r.t. (input current/capacitance) => must  
        remove the extra 'c' multiplier.  
        Output the transient results and the partial derivative.  
    }  
}
```

4.2.1.23 Capacitance Meter

Summary

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

Called By

MIFload() MIF/MIFload.c

Returned Value

None.

PDL Description

```
void cm_cmeter(ARGS)
ARGS = Mif_Private_t *private; /* INOUT - Code model inputs,
... outputs, and parameters */

{
    If this is the initial pass

        Retrieve the sum total of capacitance values [cm_netlist_get_c()],
        and store it into a static variable.

    Else just get the static variable value back from memory.

    Output the total value of the capacitance, scaled by the gain.

}
```

4.2.1.24 Inductor

Summary

The inductor is an alternative inductor element to the one supplied to the user with the standard SPICE3C1 simulator. This model provides the ability to add true initial conditions on it at time=0.0, and to support the ramptime option of the simulator.

Called By

MIFload() MIF/MIFload.c

Returned Value

None.

PDL Description

```
void cm_inductor(ARGS)
ARGS = Mif_Private_t *private; /* INOUT - Code model inputs,
.. outputs, and parameters */

{
    Get the ramp factor from the .option ramptime [cm_analog_ramp_factor()].
    The ramp factor is a value from zero to one that is used to
    slowly ramp initial conditions up to their intended values from
    an initial value of zero. This aids in convergence.

    If ARGS indicates that this is the first call to this model,
        Create storage for the initial current through the inductor
        [cm_analog_alloc()].
        Store the value of the initial condition current ('ic')
        multiplied by the ramp factor into this location. Note
        that the model can be called multiple times with the
        INIT value set to TRUE; each time, the ramp factor will
        be increased until it finally reaches 1.0.

    Else
        Retrieve the previous value of the current output by the
        model.
```

```
If ARGS indicates that this is a DC analysis,  
    Compute the output as 'ic' * ramp_factor.  
  
    Output the partial of the output current w.r.t. the input  
    voltage (at DC, this is equal to zero).  
  
Else if ARGS indicates that this is an AC analysis,  
    Output the AC gain value as the complex value:  
    (0, (2*pi*freq*'l')), where 'l' is the inductance  
    parameter.  
  
Else if ARGS indicates that this is a TRANSIENT analysis,  
    If the ramp factor is still less than 1.0,  
        This means that the DC portion of the TRANSIENT analysis  
        is not complete yet.  
        Compute the output as 'ic' * ramp_factor.  
        Output the partial of the output current w.r.t. the input  
        voltage (at DC, this is equal to zero).  
  
    Else  
        Calculate the output current as the integral of the input  
        voltage divided by the inductance parameter value  
        [cm_analog_integrate()]. The cm_analog_integrate function  
        also returns a partial derivative value, which must be  
        divided by 'l' to produce the final partial, since the  
        cm_analog_integrate function produces the partial of  
        the output w.r.t. (input current/inductance) => must  
        remove the extra 'l' multiplier.  
  
        Output the transient results and the partial derivative.
```

}

4.2.1.25 Inductance Meter

Summary

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

Called By

MIFload() MIF/MIFload.c

Returned Value

None.

PDL Description

```
void cm_lmeter(ARGS)
{
    ARGS = Mif_Private_t *private; /* INOUT - Code model inputs,
                                     .. outputs, and parameters */

    If this is the initial pass
        Retrieve the sum total of inductance values [cm_netlist_get_l()],
        and store it into a static variable.

    Else just get the static variable value back from memory.

    Output the total value of the inductance, scaled by the gain.

}
```

4.2.2 Hybrid Code Models

The following sections describe the functional behavior of the hybrid code models of the Code Model Library.

4.2.2.1 Digital-to-Analog Node Bridge

Summary

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

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

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

Called By

MIFload() MIF/MIFload.c

Returned Value

None.

PDL Description

```
void cm_dac_bridge(ARGS)
ARGS = Mif_Private_t *private; /* IINOUT - Code model inputs,
.. outputs, and parameters */

{
    Determine "width" of the node bridge by reading in the size
    of the 'in' array.

    Read in remaining model parameters.
```

Test to see if out_low and out_high were specified, but
out_undef was not...
if so,

calculate the value of out_undef as the mean of
out_high and out_low.

If ARGS indicates that this is the initial call to the model,

Allocate storage for the previous and current input
[cm_event_alloc()].

Allocate storage for outputs and for the next-breakpoint
time value [cm_analog_alloc()].

Read current input values.

Output initial analog values based on input states:
For each input,

 Output initial analog levels based on input values:

 If input state is ZERO, output 'out_low'.

 If input state is UNDEF, output 'out_undef'.

 If input state is HIGH, output 'out_high'.

 Also assign an 'input_load' value to each input
(Note that this acts as a pseudo-impedance value such
that models that alter their behavior based on the perceived
load they are driving can ascertain the total load
that they need to deal with).

Else

This is not an initialization pass...read in parameters.

Retrieve the old and new input pointers [cm_event_get_ptr()].

Retrieve the old and new output pointers, plus the pointer
to the next breakpoint time value [cm_analog_get_ptr()].

Read the current input values.

If ARGS indicates that this call to the model was made by
the event-driven simulator,

 Then test each current input value against its previous

value. If a change has occurred, then post the current timepoint as a permanent analog breakpoint value [cm_analog_set_perm_bkpt()].

Else if ARGS indicates that this call the the model was made by the analog simulator,

First calculate level_inc, rise_slope, fall_slope and time_inc incremental values for later use.

Then for each input to the model,

If ARGS indicates that the current time is zero (i.e., that this is a DC call),

If input state is ZERO, output 'out_low'.

If input state is UNDEF, output 'out_undef'.

If input state is HIGH, output 'out_high'.

Else if the current time is NOT equal to zero,

If the current input state equals the state it had at the last call,

And if the current input state is ZERO,

And if the previous output was greater than 'out_low',

Calculate the new output as:

out[i] = out_old[i] - fall_slope*time_inc,
and if this falls below 'out_low',
set it equal to 'out_low'.

Else set the current output to 'out_low'.

If the current input state is ONE,

And if the previous output was less than 'out_high',

Calculate the new output as:

out[i] = out_old[i] + rise_slope*time_inc,
and if this falls above 'out_high',
set it equal to 'out_high'.

Else set the current output to 'out_high'.

If the current input state is UNDEF,

And if the previous output was less than
'out_undef',

Calculate the new output as:
out[i] = out_old[i] + rise_slope*time_inc,
and if this falls above 'out_undef',
set it equal to 'out_undef'.

Else

If the previous output was greater
than 'out_undef',

Calculate the new output as:
out[i] = out_old[i] - fall_slope*time_inc,
and if this falls below 'out_undef',
set it equal to 'out_undef'.

Else set the current output to 'out_undef'.

Else

There HAS been a change in the input since the last
call:

There HAS been a change in this digital input
since the last analog access...need to use the
old value of input to complete the breakpoint
slope before changing directions...This calculation is
identical to the one just described for the non-
changed input case.

Calculate the required new output value based on previous
input and current input.

If the current input equals ONE,

Calculate the required time before the rise to
'out_high' will be complete, and post this
analog breakpoint [cm_analog_set_perm_bkpt()].

If the current input equals UNKNOWN,

And if the current output is less than 'out_undef',

Calculate the amount of time necessary to
rise to 'out_undef', and post this as a
permanent analog breakpoint
[cm_analog_set_perm_bkpt()].

Else if the current output is greater than
'out_undef',

Calculate the amount of time necessary to fall to 'out_undef', and post this as a permanent analog breakpoint [cm_analog_set_perm_bkpt()].

If the current input equals ZERO,

Calculate the required time before the fall to 'out_low' will be complete, and post this analog breakpoint [cm_analog_set_perm_bkpt()].

Output all values.

}

4.2.2.2 Analog-to-Digital Node Bridge

Summary

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

Called By

MIFload() MIF/MIFload.c

Returned Value

None.

PDL Description

```
void cm_adc_bridge(ARGS)
ARGS = Mif_Private_t *private; /* INOUT - Code model inputs,
.. outputs, and parameters */

{
    Retrieve the "width" of the node bridge, as well as the
    values of the 'in_high' and 'in_low' parameters.

    If ARGS indicates that this is the initialization pass through
    the model,

        Allocate storage for the analog inputs [cm_analog_alloc()].

        Allocate storage for the digital outputs [cm_event_alloc()].

    Else if this is not an initialization pass,

        Retrieve the pointers to the current and old input values
        [cm_analog_get_ptr()]

        Retrieve the pointers to the current and old output values
```

[cm_event_get_ptr()].

Read the current input values.

If the current time point is not 0.0 (i.e., if this is not a DC operating point call),

And if this is a call by the analog simulator,

Loop through all inputs...

If the current input is less than or equal to 'in_low'

And if the old output is not ZERO,

Then queue up an event-driven breakpoint at the current timepoint [cm_event_queue()].

Else do nothing.

Else

If the current input is greater than or equal to 'in_high',

And if the old output is not ONE,

Then queue up an event-driven breakpoint at the current timepoint [cm_event_queue()].

Else do nothing.

Else,

If the old output is not equal to UNKNOWN,

Then queue up an event-driven breakpoint at the current timepoint [cm_event_queue()].

Else do nothing.

Discrete call...lots to do ...

This is where we actually output stuff

Loop through all inputs...

If the current input is less than or equal to 'in_low'.

Set the current output to ZERO.

```
If this output has changed since the last call,  
    Post the new output value,  
        And post the delay value for the change  
        ('rise_delay').  
    Else post the fact that this output has not  
        changed.  
Else  
    Set the current output to UNKNOWN.  
    If this output has changed since the last call,  
        Post the new output value,  
            If the old value was ZERO,  
                Post the delay value for the change  
                ('fall_delay').  
            Else  
                Post the delay value for the change  
                ('rise_delay').  
        Else post the fact that this output has not  
            changed.  
  
    Regardless of the output value, post an output  
    strength of STRONG.  
  
If the current timepoint is zero,  
    Loop through all inputs...  
    If this input is less than 'in_low',  
        Set the output to ZERO.  
Else  
    If this input is greater than 'in_high',  
        Set the output to ONE.  
Else
```

set the output to UNKNOWN.

Set this output's strength to STRONG.

}

4.2.2.3 Controlled Digital Oscillator

Summary

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

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

Called By

`MIFload()` `MIF/MIFload.c`

Returned Value

None.

PDL Description

```
void cm_d_osc(ARGS)

ARGS = Mif_Private_t *1 i1; /* INOUT - Code model inputs,
.. outputs, and parameters */

{
    Retrieve the size of the 'cntl_array' and the 'freq_array'.
    Also retrieve the value of 'duty_cycle'.

    Check and make sure that the control array is the
    same size as the frequency array; if not, return a message
    to the simulator to that effect [cm_message_send()].
    If ARGS indicates that this is the initialization pass for
    this model,
        Allocate storage for internal variables phase, t1 and t3
        [cm_analog_alloc()].
```

Else

 Retrieve previous value pointers [cm_analog_get_ptr()].

If ARGS indicates that this is an ANALOG call,

 If ARGS indicates that this is an AC analysis,

 Return from model: This model does not function
 in AC analysis mode.

 Else

 If the current time is zero (i.e., this is either a
 DC analysis or the preliminary call of a Transient analysis),

 Retrieve & normalize phase value.

 Set phase value to init_phase

 Preset time values to harmless values...

 For all time values:

 Allocate storage for breakpoint domain & freq. range values
 [calloc()].

 Retrieve 'cntl_array[i]' and 'freq_array[i]' values
(i.e., controlling input and frequency coordinate pairs).

 Retrieve 'cntl_in' value.

 Determine segment boundaries within which 'cntl_in' resides:

 If 'cntl_in' is below the lowest cntl_array value,

 Calculate the value of the frequency by extrapolating the
 lower linear region.

 Else,

 If 'cntl_in' is above the highest cntl_array value,

 Calculate the value of the frequency by extrapolating
 the upper linear region.

 Else,

 'cntl_in' within bounds of end coordinate points.

 Determine the 'cntl_in' position progressively,

And once the region in which 'cntl_in' resides is found, calculate the required frequency by interpolating between the adjacent 'cntl_array', 'freq_array' coordinate pairs.

If the calculated frequency value is less than zero,

Clamp to 1e-16 & issue a warning

Calculate the instantaneous phase by summing the frequency value multiplied by the time delta since the last call with the previous phase value. Note that this phase value is not used in calculations, as it is not certain whether or not this timepoint will be accepted. Consequently, all calculations are done on the old_phase value, which is the last ACCEPTED phase value.

Convert the last accepted phase to an integer and subtract the integer value from it...the result is a number less than 1.0 which represents the fraction of the current cycle period which has transpired...Dphase is that fraction into the cycle for the period.

If the current time is greater than T1, but less than T3,

Add the dphase to the last accepted phase value.

If this value is less than T3,

Queue up falling edge of the oscillator [cm_event_queue()].

Else if the current time is between T3 and T1,

Adjust the value of the phase delta, if required.

And calculate and queue up the next value of T1 [cm_event_queue()].

Else

The current cycle is over...must recalculate the value of T1 for the new cycle, and queue that time point [cm_event_queue()].

In any event, free the storage space allocated for the local x and y variable arrays.

If ARGS indicates that this is an event-driven call,

```
And if this is a DC analysis,
    Retrieve & normalize the phase value .
    Preset time values to harmless values...
For any transient analysis:
    Calculate the time variables and the output value.
Set the output strength to STRONG.
If the current time is equal to T1,
    Post 'out' = ZERO.
    Post the output delay as equal to 'fall_delay'.
Else
    If the current time equals T3,
        Post 'out' = ONE.
        Post the output delay as equal to 'rise_delay'.
    Else,
        If the current time is not equal to zero,
            Post a no-change for 'out'.
        If the current time is between T1 and T3,
            Post the current output value of ONE,
        Else
            Post the current output value of ZERO,
}
}
```

4.2.3 Digital Code Models

The following sections describe the functional behavior of the digital code models of the Code Model Library.

4.2.3.1 Digital Buffer

Summary

The buffer is a single-input, single-output digital buffer which produces as output a time-delayed copy of its input. The delays associated with an output rise and those associated with an output fall may be different. The model also posts an input load value (in Farads) based on the parameter `input_load`; the output of this model does NOT, however, respond to the total loading it sees on its output; it will always drive the output strongly with the specified delays.

Called By

`MIFload()` MIF/MIFload.c

Returned Value

None.

PDL Description

```
void cm_d_buffer(ARGS)
{
    ARGS = Mif_Private_t *private; /* INOUT - Code model inputs,
        .. outputs, and parameters */

    {
        Declare variables.

        If ARGS indicates that this is the Initial pass,
            Allocate storage for the previous and current output values
            [cm_event_alloc()].
        Define the loading on 'in'.
        Else,
    }
```

Retrieve the pointers to the current and previous output values [cm_event_get_ptr()].

If ARGs indicates that this is a DC analysis,

Output the current 'in' value without delays.

Else this is a Transient analysis:

If the current input is a ZERO,

Set 'out' to ZERO,

And post 'fall_delay' as the output delay value.

If the current input is a ONE,

Set 'out' to ONE,

And post 'rise_delay' as the output delay value.

If the current input is UNKNOWN,

Set 'out' to UNKNOWN.

If the previous 'out' value was a ZERO,

Post 'rise_delay' as the output delay value.

Else the previous 'out' value must have been a ONE, so

Post 'fall_delay' as the output delay value.

Set the output strength value (always strong for this model.

}

4.2.3.2 Digital Inverter

Summary

The inverter is a single-input, single-output digital inverter which produces as output an inverted, time-delayed copy of its input. The delays associated with an output rise and those associated with an output fall may be specified independently. The model also posts an input load value (in Farads) based on the parameter `input_load`; the output of this model does NOT, however, respond to the total loading it sees on its output; it will always drive the output strongly with the specified delays.

Called By

`MIFload()` `MIF/MIFload.c`

Returned Value

None.

PDL Description

```
void cm_d_inverter(ARGS)
ARGS = Mif_Private_t *private; /* INPUT - Code model inputs,
.. outputs, and parameters */

{
    Declare variables.

    If ARGS indicates that this is the Initial pass,
        Allocate storage for the previous and current output values
        [cm_event_alloc()].
        Define the loading on 'in'.

    Else,
        Retrieve the pointers to the current and previous output
        values [cm_event_get_ptr()].
        If ARGS indicates that this is a DC analysis,
            Output the current 'in' value without delays.
```

Else this is a Transient analysis:

If the current input is a ONE,

Set 'out' to ZERO,

And post 'fall_delay' as the output delay value.

If the current input is a ZERO,

Set 'out' to ONE,

And post 'rise_delay' as the output delay value.

If the current input is UNKNOWN,

Set 'out' to UNKNOWN.

If the previous 'out' value was a ZERO,

Post 'rise_delay' as the output delay value.

Else the previous 'out' value must have been a ONE, so

Post 'fall_delay' as the output delay value.

Set the output strength value (always strong for this model.

}

4.2.3.3 Digital AND Gate

Summary

The digital AND gate is an n-input, single-output AND gate which produces an active “1” value if and only if all of its inputs are also “1” values. If *any* of the inputs is a “0”, the output will also be a “0”; if neither of these conditions holds, the output will be unknown. The delays associated with an output rise and those associated with an output fall may be specified independently. The model also posts an input load value (in Farads) based on the parameter `input_load`; the output of this model does *not*, however, respond to the total loading it sees on its output; it will always drive the output strongly with the specified delays.

Called By

`MIFload()` `MIF/MIFload.c`

Returned Value

None.

PDL Description

```
void cm_d_and(ARGS)
ARGS = Mif_Private_t *private; /* INOUT - Code model inputs,
.. outputs, and parameters */

{
    Declare variables.

    Retrieve the 'in' size value.

    If this is the initial pass,

        Allocate storage for the current and previous outputs
        [cm_event_alloc()].
        Set loading for each input.

    Else,

        Retrieve the pointers to the previous values [cm_event_get_ptr()].
```

Initialize the output to ONE.

Cycle through all of the inputs,

If this input isn't floating...

If the current input is a 0, set the output to ZERO,
and stop searching through the inputs.

If the current input is UNKNOWN, set the output
to UNKNOWN, but continue looking at the remaining inputs.

Else

At least one input is floating, so set the output to
UNKNOWN, and stop searching through the inputs.

If this is a DC analysis,

Post the 'out' value without delays.

Else this is a transient analysis:

If the current calculated output is not equal to
the old output value,

If the current calculated value of 'out' is ZERO,

Post the output.

Post the output delay value of 'fall_delay'.

If the current calculated value of 'out' is ONE,

Post the output.

Post the output delay value of 'rise_delay'.

If the current calculated value of 'out' is UNKNOWN,

Post the output.

If the old output was a ZERO,

Post 'rise_delay' as the delay value.

Else the old output must have been a ONE:

Post 'fall_delay' as the delay value.

Else the calculated output is not different from the previous output value:

Post a no-change condition for the output.

Output the strength value, regardless of the output state.

}

4.2.3.4 Digital NAND Gate

Summary

The digital NAND gate is an n-input, single-output NAND gate which produces an active “0” value if and only if all of its inputs are “1” values. If *any* of the inputs is a “0”, the output will be a “1”; if neither of these conditions holds, the output will be unknown. The delays associated with an output rise and those associated with an output fall may be specified independently. The model also posts an input load value (in Farads) based on the parameter `input_load`; the output of this model does *not*, however, respond to the total loading it sees on its output; it will always drive the output strongly with the specified delays.

Called By

`MIFload()` `MIF/MIFload.c`

Returned Value

None.

PDL Description

```
void cm_d_nand(ARGS)
ARGS = Mif_Private_t *private; /* INOUT - Code model inputs,
.. outputs, and parameters */

{
    Declare variables.

    Retrieve the 'in' size value.

    If this is the initial pass,

        Allocate storage for the current and previous outputs
        [cm_event_alloc()].
        Set loading for each input.

    Else,

        Retrieve the pointers to the previous values [cm_event_get_ptr()].
}
```

Initialize the output to ZERO.

Cycle through all of the inputs,

If this input isn't floating...

If the current input is a 0, set the output to ONE,
and stop searching through the inputs.

If the current input is UNKNOWN, set the output
to UNKNOWN, but continue looking at the remaining inputs.

Else

At least one input is floating, so set the output to
UNKNOWN, and stop searching through the inputs.

If this is a DC analysis,

Post the 'out' value without delays.

Else this is a transient analysis:

If the current calculated output is not equal to
the old output value,

If the current calculated value of 'out' is ZERO,

Post the output.

Post the output delay value of 'fall_delay'.

If the current calculated value of 'out' is ONE,

Post the output.

Post the output delay value of 'rise_delay'.

If the current calculated value of 'out' is UNKNOWN,

Post the output.

If the old output was a ZERO,

Post 'rise_delay' as the delay value.

Else the old output must have been a ONE:

Post 'fall_delay' as the delay value.

Else the calculated output is not different from the previous output value:

Post a no-change condition for the output.

Output the strength value, regardless of the output state.

}

4.2.3.5 Digital OR Gate

Summary

The digital OR gate is an n-input, single-output OR gate which produces an active “1” value if at least one of its inputs is a “1” value. The gate produces a “0” value if all inputs are “0”; if neither of these two conditions holds, the output is unknown. The delays associated with an output rise and those associated with an output fall may be specified independently. The model also posts an input load value (in Farads) based on the parameter `input_load`; the output of this model does NOT, however, respond to the total loading it sees on its output; it will always drive the output strongly with the specified delays.

Called By

`MIFload()` MIF/MIFload.c

Returned Value

None.

PDL Description

```
void cm_d_or(ARGS)
ARGS = Mif_Private_t *private; /* INOUT - Code model inputs,
.. outputs, and parameters */

{
    Declare variables.

    Retrieve the 'in' size value.

    If this is the initial pass,

        Allocate storage for the current and previous outputs
        [cm_event_alloc()].
        Set loading for each input.

    Else,
        Retrieve the pointers to the previous values [cm_event_get_ptr()].
}
```

Initialize the output to ZERO.

Cycle through all of the inputs,

If this input isn't floating...

If the current input is a 1, set the output to ONE,
and stop searching through the inputs.

If the current input is UNKNOWN, set the output
to UNKNOWN, but continue looking at the remaining inputs.

Else

At least one input is floating, so set the output to
UNKNOWN, and stop searching through the inputs.

If this is a DC analysis,

Post the 'out' value without delays.

Else this is a transient analysis:

If the current calculated output is not equal to
the old output value,

If the current calculated value of 'out' is ZERO,

Post the output.

Post the output delay value of 'fall_delay'.

If the current calculated value of 'out' is ONE,

Post the output.

Post the output delay value of 'rise_delay'.

If the current calculated value of 'out' is UNKNOWN,

Post the output.

If the old output was a ZERO,

Post 'rise_delay' as the delay value.

Else the old output must have been a ONE:

Post 'fall_delay' as the delay value.

Else the calculated output is not different from the previous output value:

Post a no-change condition for the output.

Output the strength value, regardless of the output state.

}

4.2.3.6 Digital NOR Gate

Summary

The digital NOR gate is an n-input, single-output NOR gate which produces an active “0” value if at least one of its inputs is a “1” value. The gate produces a “0” value if all inputs are “0”; if neither of these two conditions holds, the output is unknown. The delays associated with an output rise and those associated with an output fall may be specified independently. The model also posts an input load value (in Farads) based on the parameter `input_load`; the output of this model does NOT, however, respond to the total loading it sees on its output; it will always drive the output strongly with the specified delays.

Called By

`MIFload()` `MIF/MIFload.c`

Returned Value

None.

PDL Description

```
void cm_d_nor(ARGS)
{
    Mif_Private_t *private; /* INPUT - Code model inputs,
                           .. outputs, and parameters */

    Declare variables.

    Retrieve the 'in' size value.

    If this is the initial pass,

        Allocate storage for the current and previous outputs
        [cm_event_alloc()].
        Set loading for each input.

    Else,

        Retrieve the pointers to the previous values [cm_event_get_ptr()].
}
```

Initialize the output to ONE.

Cycle through all of the inputs,

If this input isn't floating...

If the current input is a 1, set the output to ZERO,
and stop searching through the inputs.

If the current input is UNKNOWN, set the output
to UNKNOWN, but continue looking at the remaining inputs.

Else

At least one input is floating, so set the output to
UNKNOWN, and stop searching through the inputs.

If this is a DC analysis,

Post the 'out' value without delays.

Else this is a transient analysis:

If the current calculated output is not equal to
the old output value,

If the current calculated value of 'out' is ZERO,

Post the output.

Post the output delay value of 'fall_delay'.

If the current calculated value of 'out' is ONE,

Post the output.

Post the output delay value of 'rise_delay'.

If the current calculated value of 'out' is UNKNOWN,

Post the output..

If the old output was a ZERO,

Post 'rise_delay' as the delay value.

Else the old output must have been a ONE:

Post 'fall_delay' as the delay value.

Else the calculated output is not different from the previous output value:

Post a no-change condition for the output.

Output the strength value, regardless of the output state.

}

4.2.3.7 Digital XOR Gate

Summary

The digital XOR gate is an n-input, single-output XOR gate which produces an active “1” value if an odd number of its inputs are also “1” values. The delays associated with an output rise and those associated with an output fall may be specified independently. The model also posts an input load value (in Farads) based on the parameter `input_load`; the output of this model does *not*, however, respond to the total loading it sees on its output; it will always drive the output strongly with the specified delays. Note also that to maintain the technology-independence of the model, any UNKNOWN input, or any floating input causes the output to also go UNKNOWN.

Called By

`MIFload()` `MIF/MIFload.c`

Returned Value

None.

PDL Description

```
static void cm_toggle_bit(Digital_State_t *bit)

Digital_State_t *bit; /* INOUT - The bit value to be changed */
{
    If the bit value is not unknown,
        And if its input value is ONE,
            Set the bit to ZERO.
        Else its input value is ZERO:
            Set the bit to ONE
}
```

```
void cm_d_xor(ARGS)
ARGS = Mif_Private_t *private; /* IINOUT - Code model inputs,
.. outputs, and parameters */

{
    Declare variables.

    Retrieve the 'in' vector size value.

    If this is the initial pass,

        Allocate storage for the current and previous outputs
        [cm_event_alloc()];

        Set loading for each input.

    Else,
        Retrieve the pointers to the previous values [cm_event_get_ptr()].
        Initialize the 'out' value to ZERO.

        Cycle through all of the inputs.

            If this input isn't floating...

                If the current input is a 1,
                    Toggle the output value [cm_toggle_bit()].

                Else the current input is UNKNOWN:
                    Set the output to UNKNOWN, and stop searching
                    through the input states.

            Else
                At least one input is floating, so set the output to
                UNKNOWN, and stop searching through the inputs.

        If this is a DC analysis,
            Post the 'out' value without delays.

        Else this is a transient analysis:
            If the current calculated output is not equal to
            the old output value,
                If the current calculated value of 'out' is ZERO,
```

```
Post the output.

Post the output delay value of 'fall_delay'.

If the current calculated value of 'out' is ONE,
    Post the output.

    Post the output delay value of 'rise_delay'.

    If the current calculated value of 'out' is UNKNOWN,
        Post the output.

        If the old output was a ZERO,
            Post 'rise_delay' as the delay value.

            Else the old output must have been a ONE:
                Post 'fall_delay' as the delay value.

        Else the calculated output is not different from the
        previous output value:
            Post a no-change condition for the output.

        Output the strength value, regardless of the output state.
```

}

4.2.3.8 Digital XNOR Gate

Summary

The digital XNOR gate is an n-input, single-output XNOR gate which produces an active “0” value if an odd number of its inputs are also “1” values. It produces a “1” output when an even number of “1” values occurs on its inputs. The delays associated with an output rise and those associated with an output fall may be specified independently. The model also posts an input load value (in Farads) based on the parameter input_load; the output of this model does *not*, however, respond to the total loading it sees on its output; it will always drive the output strongly with the specified delays. Note also that to maintain the technology-independence of the model, any UNKNOWN input, or any floating input causes the output to also go UNKNOWN.

Called By

MIFload() MIF/MIFload.c

Returned Value

None.

PDL Description

```
static void cm_toggle_bit(Digital_State_t *bit)

Digital_State_t *bit; /* IINOUT - The bit value to be changed */
{
    If the bit value is not unknown,
        And if its input value is ONE,
            Set the bit to ZERO.
        Else its input value is ZERO:
            Set the bit to ONE
}
```

```
void cm_d_xnor(ARGS)

ARGS = Mif_Private_t *private; /* IINOUT - Code model inputs,
.. outputs, and parameters */

{
    Declare variables.

    Retrieve the 'in' vector size value.

    If this is the initial pass,

        Allocate storage for the current and previous outputs
        [cm_event_alloc()];

        Set loading for each input.

    Else,

        Retrieve the pointers to the previous values [cm_event_get_ptr()];

        Initialize the 'out' value to ONE.

        Cycle through all of the inputs.

        If this input isn't floating...

            If the current input is a 1,

                Toggle the output value [cm_toggle_bit()];

            Else the current input is UNKNOWN:

                Set the output to UNKNOWN, and stop searching
                through the input states.

        Else

            At least one input is floating, so set the output to
            UNKNOWN, and stop searching through the inputs.

    If this is a DC analysis,

        Post the 'out' value without delays.

    Else this is a transient analysis:

        If the current calculated output is not equal to
        the old output value,

            If the current calculated value of 'out' is ZERO,
```

```
Post the output.

Post the output delay value of 'fall_delay'.

If the current calculated value of 'out' is ONE.

Post the output.

Post the output delay value of 'rise_delay'.

If the current calculated value of 'out' is UNKNOWN.

Post the output.

If the old output was a ZERO,

Post 'rise_delay' as the delay value.

Else the old output must have been a ONE:

Post 'fall_delay' as the delay value.

Else the calculated output is not different from the
previous output value:

Post a no-change condition for the output.

Output the strength value, regardless of the output state.
```

}

4.2.3.9 Digital Tristate Buffer

Summary

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

Called By

MIFload() MIF/MIFload.c

Returned Value

None.

PDL Description

```
void cm_d_tristate(ARGS)
ARGS = Mif_Private_t *private; /* INPUT - Code model inputs,
.. outputs, and parameters */

{
    Declare variables.

    Retrieve the 'enable' value.

    Retrieve the 'in' value and set 'out' to this value.

    Set the 'out' delay value to parameter 'delay'.
```

Set the 'in' and 'enable' load values to 'in_load'
and 'enable_load', respectively.

If 'enable' equals ZERO,

 Set the 'out' strength to HI_IMPEDANCE.

Else if 'enable' is UNKNOWN,

 Set the 'out' strength to UNDETERMINED.

Else

 Set the 'out' strength to STRONG.

}

4.2.3.10 Digital Open-Collector Buffer

Summary

The digital open-collector buffer provides a standard open-collector non-inverting buffer whose output may be tied to that of any other open-collector device along with a pullup resistor to provide wired NOR operation. It can also be used as a building block for a tristate buffer.

Called By

MIFload() MIF/MIFload.c

Returned Value

None.

PDL Description

```
void cm_d_open_collector(ARGS)
ARGS = Mif_Private_t *private; /* IINOUT - Code model inputs,
.. outputs, and parameters */

{
    Declare variables.

    If this is the initial pass,
        Allocate storage for the current and previous outputs
        [cm_event_alloc()].
        Set loading for each input.

    Else,
        Retrieve the pointers to the previous values [cm_event_get_ptr()].
        If ARGS indicates that this is a DC Analysis,
            Set the output state to the input state.
        If the output state is a ONE,
```

```
    Set the output strength to HI_IMPEDANCE
Else if the output state is ZERO,
    Set the output strength to STRONG.
Else
    Set the output strength to UNDETERMINED.
Else this is a Transient Analysis:
If the input state is a ZERO,
    Set the output state to ZERO.
    Set the output strength to STRONG.
    Set the output delay to 'fall_delay'.
If the input state is a ONE,
    Set the output state to ONE.
    Set the output strength to HI_IMPEDANCE.
    Set the output delay to 'rise_delay'.
If the input state is UNKNOWN,
    Set the output state to UNKNOWN.
    Set the output strength to UNDETERMINED.
If the previous output state was a ZERO,
    Set the output delay to 'rise_delay'.
Else
    Set the output delay to 'fall_delay'.
}
```

4.2.3.11 Digital Open-Emitter Buffer

Summary

The digital open-emitter buffer provides a standard open-emitter non-inverting buffer whose output may be tied to that of any other open-emitter device along with a pullup resistor to provide wired OR operation. It can also be used as a building block for a tristate buffer.

Called By

MIFload() MIF/MIFload.c

Returned Value

None.

PDL Description

```
void cm_d_open_emitter(ARGS)
ARGS = Mif_Private_t *private; /* INOUT - Code model inputs,
.. outputs, and parameters */

{
    Declare variables.

    If this is the initial pass,
        Allocate storage for the current and previous outputs
        [cm_event_alloc()].
        Set loading for each input.

    Else,
        Retrieve the pointers to the previous values [cm_event_get_ptr()].
        If ARGS indicates that this is a DC Analysis,
            Set the output state to the input state.
            If the output state is a ONE,
                Set the output strength to STRONG
```

```
Else if the output state is ZERO,  
    Set the output strength to HI_IMPEDANCE.  
Else  
    Set the output strength to UNDETERMINED.  
Else this is a Transient Analysis:  
    If the input state is a ZERO,  
        Set the output state to ZERO.  
        Set the output strength to HI_IMPEDANCE.  
        Set the output delay to 'fall_delay'.  
    If the input state is a ONE,  
        Set the output state to ONE.  
        Set the output strength to STRONG.  
        Set the output delay to 'rise_delay'.  
    If the input state is UNKNOWN,  
        Set the output state to UNKNOWN.  
        Set the output strength to UNDETERMINED.  
    If the previous output state was a ZERO,  
        Set the output delay to 'rise_delay'.  
    Else  
        Set the output delay to 'fall_delay'.  
}
```

4.2.3.12 Digital Pullup Resistor

Summary

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

Called By

MIFload() MIF/MIFload.c

Returned Value

None.

PDL Description

```
void cm_d_pullup(ARGS)
{
    ARGS = Mif_Private_t *private; /* INOUT - Code model inputs,
                                    .. outputs, and parameters */

    {
        Output load value for node to which pullup is connected.

        Set the output state to ONE.

        Set the output strength to RESISTIVE; this plus the
        previous will allow the simulator to resolve the final
        state of the node to which pullup is connected.

    }
}
```

4.2.3.13 Digital Pulldown Resistor

Summary

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

Called By

MIFload() MIF/MIFload.c

Returned Value

None.

PDL Description

```
void cm_d_pulldown(ARGS)
ARGS = Mif_Private_t *private; /* INOUT - Code model inputs,
.. outputs, and parameters */

{
    Output load value for node to which pulldown is connected.

    Set the output state to ZERO.

    Set the output strength to RESISTIVE; this plus the
    previous will allow the simulator to resolve the final
    state of the node to which pulldown is connected.

}
```

4.2.3.14 Digital D Flip Flop

Summary

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

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

Called By

MIFload() MIF/MIFload.c

Returned Value

None.

PDL Description

```
static void cm_toggle_bit(Digital_State_t *bit)

Digital_State_t *bit; /* INOUT - The bit value to be changed */
{
    If the bit value is not unknown,
        And if its input value is ONE,
            Set the bit to ZERO.
        Else its input value is ZERO:
            Set the bit to ONE
}
```

```
void cm_d_dff(ARGS)
ARGS = Mif_Private_t *private; /* INOUT - Code model inputs,
.. outputs, and parameters */

{
    Declare variables

    If ARGS indicates that this is the Initial pass,
        Allocate storage for the previous and current output values
        [cm_event_alloc()].
        Define the loading on the input nodes.

    Else,
        Retrieve the pointers to the current and previous output
        values [cm_event_get_ptr()].
        Load current input values.
        If 'set' or 'reset' are not connected, set them to zero.
        Determine analysis type and output appropriate values:
        If this is a DC analysis,
            Output initial conditions w/o delays [cm_toggle_bit()].
        Else, if this is a transient analysis,
            Find out which input has changed. The order of testing is
            to first check 'set', then 'reset', then 'clk', and
            finally 'data':
            If 'set' value has changed from the previous value,
                Output values and delays for a set or set release.
            Else,
                If 'reset' value has changed,
                    Output values and delays for a reset or reset release.
                Else,
                    If 'set' & 'reset' haven't changed, but 'clk' has,
                        Output values and store new data, as appropriate
```

```
[cm_toggle_bit()].
Else, if neither 'set', 'reset' nor 'clk' have changed,
'Data' value must have changed;
Return previous output value.

Add additional rise or fall delays, if appropriate.

Output the strength values ('out' and 'Mout' values
are always STRONG).

}
```

4.2.3.15 Digital JK Flip Flop

Summary

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

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

Called By

MIFload() MIF/MIFload.c

Returned Value

None.

PDL Description

```
static void cm_toggle_bit(Digital_State_t *bit)

Digital_State_t *bit; /* INOUT - The bit value to be changed */

{
    If the bit value is not unknown,
        And if its input value is ONE,
            Set the bit to ZERO.
        Else its input value is ZERO:
            Set the bit to ONE
}
```

```
static Digital_State_t cm_eval_jk_result(Digital_State_t j_input,
                                         Digital_State_t k_input,
                                         Digital_State_t old_output)

Digital_State_t j_input;      /* IN - The j bit input state      */
Digital_State_t k_input;      /* IN - The k bit input state      */
Digital_State_t old_output;   /* IN - The previous output state */

{
    Evaluate the nine possible input state combinations of
    the j and k inputs, and output next-state value based
    on the following:

    j = 0, k = 0 => out = old_output
    j = 0, k = 1 => out = 0
    j = 1, k = 0 => out = 1
    j = 1, k = 1 => out = !(old_output)
    j and/or k = UNKNOWN => out = UNKNOWN

    This function uses [cm_toggle_bit()] to invert the state
    of the old_output for the case of j = k = 1.

}
```

```
void cm_d_jkff(ARGs)

ARGs = Mif_Private_t *private; /* IINOUT - Code model inputs,
                                .. outputs, and parameters */

{
    Declare variables

    If ARGs indicates that this is the Initial pass,

        Allocate storage for the previous and current output values
        [cm_event_alloc()].
        Define the loading on the input nodes.

    Else,
        Retrieve the pointers to the current and previous output
        values [cm_event_get_ptr()].
        Load current input values.

    If 'set' or 'reset' are not connected, set them to zero.
```

Determine the analysis type and output appropriate values:
If this is a DC analysis,

Output w/o delays [cm_toggle_bit()].

Else, this must be a transient analysis:

Find out which input has changed. The order of testing is
to first check 'set', then 'reset', then 'clk', and
finally 'j' and 'k':

If 'set' value has changed from the previous value,

Output values and delays for a set or set release.

Else,

If 'reset' value has changed,

Output values and delays for a reset or reset release.

Else,

If 'set' & 'reset' haven't changed, but 'clk' has,

Output values and store new data, as appropriate
[cm_toggle_bit()].

Else, if neither 'set', 'reset' nor 'clk' have changed,

'j' or 'k' values must have changed;

Return previous output value.

Add additional rise or fall delays, if appropriate.

Output strength values (STRONG in all cases).

}

4.2.3.16 Digital Toggle Flip Flop

Summary

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

Note that any UNKNOWN inputs other than t immediately cause the output to go UNKNOWN.

Called By

MIFload() MIF/MIFload.c

Returned Value

None.

PDL Description

```
Digital_State_t *bit; /* INOUT - The bit value to be changed */
{
    If the bit value is not unknown,
        And if its input value is ONE,
            Set the bit to ZERO.
        Else its input value is ZERO:
            Set the bit to ONE
}
```

```
void cm_d_tff(ARGS)
ARGS = Mif_Private_t *private; /* INOUT - Code model inputs,
.. outputs, and parameters */

{
    Declare variables

    If ARGS indicates that this is the Initial pass,
        Allocate storage for the previous and current output values
        [cm_event_alloc()].
        Define the loading on the input nodes.

    Else,
        Retrieve the pointers to the current and previous output
        values [cm_event_get_ptr()].
        Load current input values.

        If 'set' or 'reset' are not connected, set them to zero.

        Determine analysis type and output appropriate values:
        If this is a DC analysis,
            Output w/o delays [cm_toggle_bit()].
        Else, this is a transient analysis,
            Find out which input has changed. The order of testing is
            to first check 'set', then 'reset', then 'clk', and
            finally 't':
            If 'set' value has changed from the previous value,
                Output values and delays for a set or set release.

            Else,
                If 'reset' value has changed,
                    output values and delays for a reset or reset release.

            Else,
                If 'set' & 'reset' haven't changed, but 'clk' has,
                    Output values and store new data, as appropriate
```

XSPICE Code Model Subsystem
Software Design Document

Detailed Design
Code Model Library

```
[cm_toggle_bit()].  
  
Else, if neither 'set', 'reset' nor 'clk' have changed,  
    't' value must have changed;  
  
    Return previous output value.  
  
Add additional rise or fall delays, if appropriate.  
Output strength values (all STRONG).  
}
```

4.2.3.17 Digital Set-Reset Flip Flop

Summary

The digital sr-type flip flop is a one-bit, edge-triggered storage element which will store data whenever the clk input line transitions from low to high (ZERO to ONE). The value stored (i.e., the "out" value) will depend on the s and r input pin values:

```
*  
out=ONE           if s=ONE and r=ZERO;  
out=ZERO          if s=ZERO and r=ONE;  
out=previous value if s=ZERO and r=ZERO;  
out=UNKNOWN       if s=ONE and r=ONE;
```

In addition, asynchronous set and reset signals exist, and each of the three methods of changing the stored output of the model have separate load values and delays associated with them. The user may also specify separate rise and fall delay values that are added to those specified for the input lines; these allow for more faithful reproduction of the output characteristics of different IC fabrication technologies.

Note that any UNKNOWN inputs other than s and r immediately cause the output to go UNKNOWN.

Called By

MIFload() MIF/MIFload.c

Returned Value

None.

PDL Description

```
static void cm_toggle_bit(Digital_State_t *bit)  
  
Digital_State_t *bit; /* IINOUT - The bit value to be changed */  
{  
    If the bit value is not unknown,  
        And if its input value is ONE,  
            Set the bit to ZERO.
```

Else its input value is ZERO:

 Set the bit to ONE

}

```
static Digital_State_t cm_eval_sr_result(Digital_State_t s_input,
                                         Digital_State_t r_input,
                                         Digital_State_t old_output)
```

```
Digital_State_t s_input; /* IN - The s bit input state */
Digital_State_t r_input; /* IN - The r bit input state */
Digital_State_t old_output; /* IN - The previous output state */
```

{

 Evaluate the nine possible input state combinations of
 the s and r inputs, and output next-state value based
 on the following:

```
s = 0, r = 0 => out = old_output
s = 0, r = 1 => out = 0
s = 1, r = 0 => out = 1
s = 1, r = 1 => out = UNKNOWN
s and/or r = UNKNOWN => out = UNKNOWN
```

This function uses [cm_toggle_bit()] to invert the state
of the old_output for the case of s = r = 1.

}

```
void cm_d_srff(ARGS)
```

```
ARGS = Mif_Private_t *private; /* IMOUT - Code model inputs,
.. outputs, and parameters */
```

{

 Declare variables

 If ARGS indicates that this is the Initial pass,

 Allocate storage for the previous and current output values
 [cm_event_alloc()].

 Define the loading on the input nodes.

```
Else,  
  
    Retrieve the pointers to the current and previous output  
    values [cm_event_get_ptr()].  
  
Load current input values.  
  
If 'set' or 'reset' are not connected, set them to zero.  
  
Determine analysis type and output appropriate values:  
If this is a DC analysis,  
  
    Output w/o delays [cm_toggle_bit()].  
  
Else, this is a transient analysis,  
  
    Find out which input has changed. The order of testing is  
    to first check 'set', then 'reset', then 'clk', and  
    finally 's' and 'r':  
  
    If 'set' value has changed from the previous value,  
  
        Output values and delays for a set or set release  
        [cm_eval_sr_result(), cm_toggle_bit()].  
  
    Else,  
  
        If 'reset' value has changed,  
  
            Output values and delays for a reset or reset release  
            [cm_eval_sr_result(), cm_toggle_bit()].  
  
        Else,  
  
            If 'set' & 'reset' haven't changed, but 'clk' has,  
  
                Output values and store new data based on states  
                of 's' and 'r' inputs, as appropriate  
                [cm_eval_sr_result(), cm_toggle_bit()].  
  
            Else, if neither 'set', 'reset', nor 'clk' have changed,  
  
                's' or 'r' (or both) value must have changed  
                [cm_eval_sr_result(), cm_toggle_bit()];  
  
                Return previous output value.
```

**XSPICE Code Model Subsystem
Software Design Document**

**Detailed Design
Code Model Library**

Add additional rise or fall delays, if appropriate.

Output strength values (always STRONG).

}

4.2.3.18 Digital D Latch

Summary

The digital d-type latch is a one-bit, level-sensitive storage element which will output the value on the data line whenever the enable input line is high (ONE). The value on the data line is stored (i.e., held on the out line) whenever the enable line is low (ZERO).

In addition, asynchronous set and reset signals exist, and each of the four methods of changing the stored output of the d_dlatch (i.e., data changing with enable=ONE, enable changing to ONE from ZERO with a new value on data, raising set and raising reset) have separate delays associated with them. The user may also specify separate rise and fall delay values that are added to those specified for the input lines; these allow for more faithful reproduction of the output characteristics of different IC fabrication technologies.

Note that any UNKNOWN inputs other than on the data line when enable=ZERO immediately cause the output to go UNKNOWN.

Called By

MIFload() MIF/MIFload.c

Returned Value

None.

PDL Description

```
static void cm_toggle_bit(Digital_State_t *bit)

Digital_State_t *bit; /* INOUT - T...L... value to be changed */

{
    If the bit value is not unknown,
        And if its input value is ONE,
            Set the bit to ZERO.
        Else its input value is ZERO:
            Set the bit to ONE
}
```

```
void cm_d_dlatch(ARGS)
ARGS = Mif_Private_t *private; /* IINOUT - Code model inputs,
.. outputs, and parameters */

{
    Declare variables

    If ARGS indicates that this is the Initial pass,
        Allocate storage for the previous and current output values
        [cm_event_alloc()].
        Define the loading on the input nodes.

    Else,
        Retrieve the pointers to the current and previous output
        values [cm_event_get_ptr()].
        Load current input values.

        If 'set' or 'reset' are not connected,
            Set their state values to zero.

        Determine the analysis type and output appropriate values:
        If this is a DC analysis,
            output w/o delays [cm_toggle_bit()].
        Else, this is a transient analysis:
            Find out which input has changed. The order of testing is
            to first check 'set', then 'reset', then 'enable', and
            finally 'data':
            If 'set' has changed,
                Set the appropriate outputs and/or delays [cm_toggle_bit()].
            Else, if 'reset' has changed,
                Set the appropriate outputs and/or delays
                [cm_toggle_bit()].
            Else if 'enable' has changed,
                Set the appropriate outputs and/or delays
```

[cm_toggle_bit()].

Else if 'set', 'reset', and 'enable' are all unchanged,
then test the 'data' value for change...

If 'data' has changed and 'enable' is still active,

Post new values [cm_toggle_bit()].

Else, no change.

Add additional rise or fall delays, if appropriate.

Output strength values (all STRONG).

}

4.2.3.19 Digital Set-Reset Latch

Summary

The digital sr-type latch is a one-bit, level-sensitive storage element which will output the value dictated by the state of the s and r pins whenever the enable input line is high (ONE). This value is stored (i.e., held on the out line) whenever the enable line is low (ZERO). The particular value chosen is as shown below:

```
s=ZERO, r=ZERO => out=current value (i.e., not change in output)
s=ZERO, r=ONE  => out=ZERO
s=ONE,  r=ZERO => out=ONE
s=ONE,  r=ONE  => out=UNKNOWN
```

Asynchronous set and reset signals exist, and each of the four methods of changing the stored output of the d_srLatch (i.e., s/r combination changing with enable=ONE, enable changing to ONE from ZERO with an output-changing combination of s and r, raising set and raising reset) have separate delays associated with them. The user may also specify separate rise and fall delay values that are added to those specified for the input lines; these allow for more faithful reproduction of the output characteristics of different IC fabrication technologies.

Note that any UNKNOWN inputs other than on the s and r lines when enable=ZERO immediately cause the output to go UNKNOWN.

Called By

MIFload() MIF/MIFload.c

Returned Value

None.

PDL Description

```
static void cm_toggle_bit(Digital_State_t *bit)

Digital_State_t *bit; /* I/MOUT - The bit value to be changed */
{
    If the bit value is not unknown,
```

```
And if its input value is ONE,  
Set the bit to ZERO.  
  
Else its input value is ZERO:  
Set the bit to ONE  
  
}  
  
  
static Digital_State_t cm_eval_sr_result(Digital_State_t s_input,  
                                         Digital_State_t r_input,  
                                         Digital_State_t old_output)  
  
Digital_State_t s_input;      /* IN - The s bit input state      */  
Digital_State_t r_input;      /* IN - The r bit input state      */  
Digital_State_t old_output;   /* IN - The previous output state  */  
{  
    Evaluate the nine possible input state combinations of  
    the s and r inputs, and output next-state value based  
    on the following:  
  
    s = 0, r = 0 => out = old_output  
    s = 0, r = 1 => out = 0  
    s = 1, r = 0 => out = 1  
    s = 1, r = 1 => out = UNKNOWN  
    s and/or r = UNKNOWN => out = UNKNOWN  
  
    This function uses [cm_toggle_bit()] to invert the state  
    of the old_output for the case of s = r = 1.  
  
}  
  
  
void cm_d_srLatch(ARGs)  
  
ARGs = Mif_Private_t *private; /* INOUT - Code model inputs,  
                                .. outputs, and parameters */  
  
{  
    Declare variables  
  
    If ARGs indicates that this is the Initial pass,  
  
        Allocate storage for the previous and current output values  
        [cm_event_alloc()].  
  
    Define the loading on the input nodes.
```

Else,

 Retrieve the pointers to the current and previous output
 values [cm_event_get_ptr()].

. Load current input values.

If 'set' or 'reset' are not connected, set them to zero.

Determine analysis type and output appropriate values:
If this is a DC analysis,

 Output w/o delays [cm_toggle_bit()].

Else, this is a transient analysis.

 Find out which input has changed. The order of testing is
 to first check 'set', then 'reset', then 'enable', and
 finally 's' and 'r':

 If 'set' has changed,

 Set the appropriate outputs and/or delays
 [cm_eval_sr_result(), cm_toggle_bit()].

 Else, if 'reset' has changed,

 Set the appropriate outputs and/or delays
 [cm_eval_sr_result(), cm_toggle_bit()].

 Else if 'enable' has changed,

 Set the appropriate outputs and/or delays
 [cm_eval_sr_result(), cm_toggle_bit()].

 Else if 'set', 'reset', and 'enable' are
 all unchanged,

 then test 's' and 'r' inputs for change...

 If 's' or 'r' have changed and enable is still active,

 Post new values [cm_eval_sr_result(),
 cm_toggle_bit()].

 Else, no change.

Add additional rise or fall delays, if appropriate.

Output strength values.

}

4.2.3.20 Digital State Machine

Summary

The digital state machine provides for straightforward descriptions of clocked combinational logic blocks with a variable number of inputs and outputs and with an unlimited number of possible states. The model can be configured to behave as virtually any type of counter or clocked combinational logic block and can be used to replace very large digital circuit schematics with an identically functional but faster representation.

The d_state model is configured through the use of a state definition file (state.in) which resides in a directory of the user's choosing. The file defines all states to be understood by the model, plus input bit combinations which trigger changes in state. An example state.in file is shown below:

```
----- begin file -----  
  
* This is an example state.in file. This file  
* defines a simple 2-bit counter with one input. The  
* value of this input determines whether the counter counts  
* up (in = 1) or down (in = 0).  
  
0 0s 0s 0 -> 3  
    1 -> 1  
  
1 0s 1z 0 -> 0  
    1 -> 2  
  
2 1z 0s 0 -> 1  
    1 -> 3  
  
3 1z 1z 0 -> 2  
3 1z 1z 1 -> 0  
----- end file -----
```

Several attributes of the above file structure should be noted. First, ALL LINES IN THE FILE MUST BE ONE OF FOUR TYPES. These are:

1. A comment, beginning with a “*” in the first column
2. A header line, which is a complete description of the current state, the outputs corresponding to that state, an input value, and the state that the model will assume should that input be encountered. The first line of a state definition must ALWAYS be a header line.

3. A continuation line, which is a partial description of a state, consisting of an input value and the state that the model will assume should that input be encountered. Note that continuation lines may only be used after the initial header line definition for a state.
4. A line containing nothing but whitespace (space, formfeed, newline, carriage return, tab, vertical tab).

A line which is not one of the above will cause a file-loading error.

Note that in the example shown, whitespace (any combination of blanks, tabs, commas) is used to separate values, and that the character “->” is used to underline the state transition implied by the input preceding it. This particular pair of characters is not critical in itself, and can be replaced with any other character(s) or non-broken combination of characters that the user prefers (e.g., “==>”, “>>”, “:”, “resolves_to”, etc.)

The order of the output and input bits in the file is important; the first column is always interpreted to refer to the “zeroth” bit of input and output. Thus, in the file above, the output from state 1 sets out[0] to “0s”, and out[1] to “1z”.

The state numbers need not be in any particular order, but a state definition (which consists of the sum total of all lines which define the state, its outputs, and all methods by which a state can be exited) must be made on contiguous lines; a state definition cannot be broken into sub-blocks and distributed randomly throughout the file. On the other hand, the state definition can be broken up by as many comment lines as the user desires.

Header lines may be used throughout the state.in file, and continuation lines can be avoided completely if the user so chooses; continuation lines are primarily provided as a convenience.

Called By

MIFload() **MIF/MIFload.c**

Returned Value

None.

PDL Description

```
static char *CNVgettok(char **s)

char **s; /* INOUT - The pointer to the input string; this is
.. updated once a token is copied, such that the
.. next token is pointed to. */

{

    Declare variables.

    Allocate space large enough to hold the input [malloc()].
    Skip any whitespace.

    Check the next token:
    If the token is a null ('\0'), free space and return a NULL.

    Else,
        Copy each character into the 'buf' internal variable space,
        and add a null to the end of the character string.

        Skip over any additional whitespace from the end of the
        token to the beginning of the next token.

        Create a new string (ret_str) of exactly the same length as that used
        by buf, and copy the token into it.

        Free the space used by buf and return ret_str.

}

static char *CNVget_token(char **s, Cnv_Token_Type_t *type)

char **s; /* INOUT - The pointer to the input string;
.. this is updated once a token is copied,
.. such that the next token is pointed to. */
Cnv_Token_Type_t *type; /* OUT - The pointer is used to pass back
.. information about whether a string has
.. valid token data or not. */

{
    Obtain the token value [CNVgettok()].

    If the value returned was a NULL,

        Set the type to CNV_NO_TOK and return a NULL value.

    Else
```

```
Set the type to CNV_STRING_TOK and return the token.  
}  
  
static int cnv_get_spice_value(str,p_value)  
char *str; /* IN - The value text e.g. 1.2K */  
float *p_value; /* OUT - The numerical value */  
  
{  
    Scan the input string looking for an alpha character that is not  
    'e' or 'E'. Such a character is assumed to be an engineering  
    suffix as defined in the Spice 2G.6 user's manual.  
  
    Determine the scaling factor by matching the letter value  
    found against the accepted SPICE exponential notation  
    (e.g., t => E+12 g => E+9, etc.)  
  
    Use the string scan function to retrieve the mantissa of the  
    value. If the sscanf() fails, return FAIL value.  
  
    Multiply by the exponential determined previously, and return.  
}
```

```
static void cm_inputs_mask_and_retrieve(short base,  
                                         int bit_offset,  
                                         Digital_t *out)  
  
short base; /* IN - The input word from which to  
.. retrieve the 2-bit data value. */  
int bit_offset; /* IN - An integer representing which */  
Digital_t *out; /* OUT - The returned state of the bit */  
  
{  
  
    Based of which of the eight possible 2-bit values that are  
    desired (0 to 7), right shift the BASE integer so that  
    the two bits in question are on the far right.  
  
    Mask off the remaining bit values, leaving only the lowest-order  
    two bits in base.  
  
    If the remaining value is a ZERO, set out to ZERO.  
  
    If the remaining value is a ONE, set out to ONE.  
  
    If the remaining value is a 2, set output to UNKNOWN.
```

}

```
static int cm_set_indices(State_Table_t *states)

State_Table_t *states; /* IINOUT - This is the data structure which
.. maintains all state information as well as
.. the current operating state of the machine,
.. etc. See the type definition for the
.. State_Table_t in the cfunc.mod code for
.. the d_state. */

{

    For the entire depth of the state table,
    Check each state against the current active state.
    If the states match,
        And if the index0 value has not been set for the
        table,
            Set the current iteration number as the first occurrence
            of the current state.
        If a state is found which is in a non-contiguous portion
        of the table,
            Return from this routine with a TRUE value.
        Else the current entry is contiguous with previous
        entries for this state, so update the indexN value
        to the current index.
    Return a FALSE from this routine if it has finished successfully.

}

static int cm_compare_to_inputs(State_Table_t *states,
                                int index,int bit_number,
                                Digital_State_t in)

State_Table_t *states; /* IINOUT - This is the data structure which
.. maintains all state information as well as
.. the current operating state of the machine,
.. etc. See the type definition for the
.. State_Table_t in the cfunc.mod code for
```

```

        .. the d_state.          */

int index;      /* IN - This is the word number index value
.. into the inputs[] array                      */

int bit_number; /* IN - This is the bit number (0 to 7) of
.. the bit in the inputs[] array to which
.. we want to compare.                         */

Digital_State_t in; /* IN - The input value which we want to
.. compare the inputs[] entry to.             */

{
    Calculate the indices into the inputs[] array within the
    state table.

    Based on these values, retrieve the inputs[] bit pointed to
    [cm_inputs_mask_and_retrieve()].
    Compare to the passed intput vales;
    If they match, return a 0.
    If they don't, return a 1.
}

```

```

static int cm_inputs_mask_and_store(short *base,int bit_offset,
                                    int bit_value)

short *base;      /* INOUT - The base location into which the
.. bit data is to be stored.                     */
int bit_offset;   /* IN - Which of the 8 bit locations is
.. to be the storage location.                  */
int bit_value;   /* IN - The integer value to be stored.
.. Note that 0 => ZERO, 1 => ONE, and
.. 2 => UNKNOWN.                                */

{
    Depending on the value of the bit_offset,
    Clear the two bit values corresponding to the bit_offset
    location, and left-shift the integer value of the bit_value
    by the appropriate number of bit locations.

    Store the bit value by ORing the left-shifted value with the
    *base value.
}

```

}

```
static void cm_store_inputs_value(State_Table_t *states,int index,
                                 int bit_number, int in_val)

State_Table_t *states; /* INOUT - This is the data structure which
.. maintains all state information as well as
.. the current operating state of the machine,
.. etc. See the type definition for the
.. State_Table_t in the cfunc.mod code for
.. the d_state. */

int index;           /* IN - This is the word number index value
.. into the inputs[] array */

int bit_number;      /* IN - This is the bit number (0 to 7) of
.. the bit in the inputs[] array to which
.. we want to compare. */

int in_val;          /* IN - Input value to be stored. */

{

    Obtain offset value from word_number, word_width &
    bit_number.

    Retrieve entire base_address bits integer.

    For each offset, mask off the bits and store values .

    Store modified base value,
}

static int cm_bits_mask_and_store(short *base,int bit_offset,int bit_value)

short *base;           /* INOUT - Base integer into which
.. the bit value is 'stored' */
int bit_offset;        /* Which of the 4 possible bit
.. location is to be used for the
.. store operation. */
int bit_value;         /* The value to be stored. Note
.. that this value takes up 4 bits
.. rather than 2 because of the
.. 12 possible states that are
.. to be described. */


```

```

{
    Depending on the bit_offset value,
        Clear the four bits that need to be written to,
        and left-shift the value to be stored, if necessary.

    Store the bit_value by ORing the shifted value with the
    base integer value.

}

static void cm_bits_mask_and_retrieve(short base,int bit_offset,
                                      Digital_t *out)

short base;                      /* IN - The base integer from
.. which the data is to be retrieved. */
int bit_offset;                  /* IN - The number of the bit
.. location desired (of 4 possible) */
Digital_t *out;                  /* OUT - The output to which the
.. value is written. */

{
    Depending on the bit_offset, right shift the
    base value by zero, 4, 8, or 12, as appropriate.

    Based on the masked value of the right-shifted base,
    assign one of the 12 possible state + strength
    combinations to the *out output, and exit.

}

static void cm_get_bits_value(State_Table_t *states,int index, int bit_number,
                             Digital_t *out)

State_Table_t *states; /* INOUT - This is the data structure which
.. maintains all state information as well as
.. the current operating state of the machine,
.. etc. See the type definition for the
.. State_Table_t in the cfunc.mod code for
.. the d_state. */

int index;                      /* IN - This is the word number index value
.. into the bits[] array */

```

XSPICE Code Model Subsystem
Software Design Document

Detailed Design
Code Model Library

```
int bit_number;      /* IN - This is the bit number (0 to 3) of
.. the bit in the bits[] array which
.. we want to retrieve.          */
Digital_t *out;     /* OUT - The output value           */
{
    Calculate the bit_index and bit_offset values.

    Retrieve the address of the *base integer.

    Retreive the value of that bits data [cm_bits_mask_and_retrieve()].
}
```

```
static void cm_store_bits_value(State_Table_t *states,int index,
                                int bit_number, int in_val)

State_Table_t *states; /* INOUT - This is the data structure which
.. maintains all state information as well as
.. the current operating state of the machine,
.. etc. See the type definition for the
.. State_Table_t in the cfunc.mod code for
.. the d_state.          */

int index;           /* IN - This is the word number index value
.. into the bits[] array           */
int bit_number;      /* IN - This is the bit number (0 to 3) of
.. the bit in the bits[] array in which
.. we want to store a value.          */
int in_val;          /* IN - The value to be stored.           */
{
    Calculate the bit index and offset values.

    Retrieve the value of the current bit into the base local integer.

    Store the bit value into the base integer [cm_bits_mask_and_store()]

    Copy the new base integer value back into the bits array.

}
```

```
static int cm_read_state_file(FILE *state_file,State_Table_t *states)
```

```
FILE estate_file; /* IN - The name of the state input file */

State_Table_t *states; /* INOUT - This is the data structure which
.. maintains all state information as well as
.. the current operating state of the machine,
.. etc. See the type definition for the
.. State_Table_t in the cfunc.mod code for
.. the d_state. */

{

    For each non-null string in the state file,
        Count the number of tokens in this string, and
        Set the string type based on this.

        If the number of tokens is not what is expected for
        either a State Header line or a State Continuation Line,
        Return a '1' value to the calling routine and exit.

        Reset the current string pointer to the beginning of this line.

        If this is a State Header line,
            Retrieve and store the State number, the input values, and
            the output values [cnv_get_spice_value(), cm_store_bits_value(),
            cm_store_inputs_value()]

        Else for a Continuation line,
            Set the output bit values to the previous ones
            [cm_get_bits_value()] and store in the new location
            [cm_store_bits_value()].
            Store the new input values and the states to which
            they point [cm_store_inputs_value()].
            Return a '0' if all went well to this point.

}

void cm_d_state(ARGS)
ARGS = Mif_Private_t *private; /* INOUT - Code model inputs,
.. outputs, and parameters */

{
    Set up required state variables.

    If this is the initial pass,
```

Open the state.txt file and count the number of vectors in it.

Increment the counter if not a comment until EOF is reached...
This counts the total number of non-comment lines in the state
file. The variable "i" holds the resulting "depth" value of
the state file.

Allocate storage for states [cm_event_alloc()].

Store depth value

Retrieve widths for bits[] and inputs[]
(from the width of input and output busses).

Assign storage for arrays to pointers in states table.

Initialize state, bits, inputs & next_state to zero.

Allocate storage for clk, clk_old, reset & reset_old
[cm_event_alloc()].

Send file pointer and the four storage pointers
to "cm_read_state_file()". This will return after
reading the contents of state.in, and if no
errors have occurred, the "state" and "bits"
"inputs", and "next_state" vectors will be loaded
and the num_inputs, num_outputs and depth
values supplied [cm_read_state_file()].

If a problem occurred in load...send an error msg
[cm_message_send()].

Reset arrays to zero and return from the model.

Close state_file.

Declare load values on all input lines.

Else, if this isn't an initial pass,

Retrieve previous values [cm_event_get_ptr()].

Determine analysis type and output appropriate values

If this is a DC analysis...output w/o delays.

Set current state to default.

Set indices for this state [cm_set_indices()],
and output an error message if a failure occurs

```
[cm_message_send()]

Output new values...

Retrieve output value and post [cm_get_bits_value()].
Else, this is a transient analysis.

Load current input values, and
if reset is not connected, set to zero.

Find input that has changed...
RESET and CLK are the only possible inputs to have changed...
this section of code determines which one and outputs
according to its findings...best to see the code:

If reset has changed...

Reset has changed; calculate and post appropriate outputs
[cm_set_indices(), cm_message_send(), cm_get_bits_value()].
Else, Test clk value for change.

CLK has changed; calculate and post appropriate outputs.
[cm_compare_to_inputs(), cm_set_indices(),
cm_message_send(), cm_get_bits_value()].
Else, input value must have changed...

Return previous output values.

}
```

4.2.3.21 Digital Frequency Divider

Summary

The digital frequency divider is a programmable step-down divider which accepts an arbitrary divisor (div_factor), a duty-cycle term (high_cycles), and an initial count value (i_count). The generated output is synchronized to the rising edges of the input signal. Rise delay and fall delay on the outputs may also be specified independently.

Called By

MIFload() MIF/MIFload.c

Returned Value

None.

PDL Description

```
void cm_d_fdiv(ARGS)

ARGS = Mif_Private_t *private; /* INOUT - Code model inputs,
.. outputs, and parameters */

{
    Setup the required state variables.

    If this is the initial pass,
        Allocate storage [cm_event_alloc()] and
        Declare load values.

    Else, retrieve previous values.
        Retrieve storage for the outputs [cm_event_get_ptr()].
        Output the strength of freq_out (always strong)...
        Retrieve parameters.

    Determine analysis type and output appropriate values:
        If this is a DC analysis...output w/o delays.
```

Read initial count value, normalize, and if it is out of bounds, set to "zero" equivalent. This sets up the output in the right phase for a given range of initial count values.

Else, this is a transient analysis.

Load current input value...

If the input has provided an edge...

An edge has been provided...revise the count value.

If new count value is equal to the div_factor+1 value, need to normalize count to "1", and raise output.

If new count value is equal to the high_cycles+1 value, drop the output to ZERO.

If no edge has occurred, output does not change.

}

4.2.3.22 Digital Random Access Memory

Summary

The digital RAM is an M-wide, N-deep random access memory element with programmable select lines, tristated data_out lines, and a single write/read line. The width of the RAM words (M) is set through the use of the word_width parameter. The depth of the RAM (N) is set by the number of address lines input to the device. The value of N is related to the number of address input lines (P) by the following equation:

$$2^P = N$$

There is no reset line into the device. However, an initial value for all bits may be specified by setting the ic parameter to either 0 or 1. In reading a word from the RAM, the read_delay value is invoked, and output will not appear until that delay has been satisfied. Separate rise and fall delays are not supported for this device.

Note that UNKNOWN inputs on the address lines are not allowed during a write. In the event that an address line does indeed go unknown during a write, THE ENTIRE CONTENTS OF THE RAM WILL BE SET TO UNKNOWN. This is in contrast to the data_in lines being set to unknown during a write; in that case, only the selected word will be corrupted, and this is corrected once the data lines settle back to a known value. Note that protection is added to the write_en line such that extended UNKNOWN values on that line are interpreted as ZERO values; this is the equivalent of a read operation and will not corrupt the contents of the RAM. A similar mechanism exists for the select lines; if they are unknown, then it is assumed that the chip is not selected.

Detailed timing-checking routines are not provided in this model, other than for the enable_delay and select_delay restrictions on read operations. The user is advised, therefore, to carefully check the timing into and out of the RAM for correct read and write cycle times, setup and hold times, etc. for the particular device they are attempting to model.

Called By

MIFload() **MIF/MIFload.c**

Returned Value

None.

PDL Description

```
static int cm_address_to_decimal(Digital_State_t *address,
                                 int address_size,int *total)

Digital_State_t *address; /* IN - Address input value */

int address_size; /* IN - The number of bits in the
.. address vector. */

int *total; /* OUT - The output value */

{

    Calculate the decimal equivalent of the address value...this
    is a standard binary-to-decimal algorithm. Return a zero
    value for the error unless one of the address values is
    UNKNOWN.

}

static int cm_mask_and_store(short *base,int ram_offset,
                            Digital_State_t out)

short *base; /* I/MOUT - The base short integer into which
.. the data will be stored */

int ram_offset; /* IN - The integer value (from 0 to 7) of
.. the 2-bits which will be used to store
.. the data. */

Digital_State_t out; /* IN - The actual state value to be stored. */

{

    Depending on which storage location is to be used,

        If the out value is a ZERO, zero the bit values by
        ANDing with an appropriate mask.

        If the out value is a ONE, first zero the bit values,
        then store ONE by ORing with an appropriate mask.

}

static int cm_mask_and_retrieve(short *base,int ram_offset,
                               Digital_State_t *out)
```

```

short *base;           /* IINOUT - The base short integer into which
.. the data will be stored */
```

```

int ram_offset;        /* IN - The integer value (from 0 to 7) of
.. the 2-bits which will be used to store
.. the data. */
```

```

Digital_State_t *out; /* OUT - The actual state value to be read. */
```

```

{
```

For each possible value of ram_offset,

Mask the appropriate bit and test for the stored value at that location. Depending on the value (binary 0, 1, or 2), set *out, and return.

```

}
```



```

static void cm_initialize_ram(Digital_State_t out,
                           int word_width,int bit_number,
                           int word_number,short *ram)
```

```

Digital_State_t out;      /* IN - The value to be stored */
```

```

int word_width;          /* IN - The default word width
.. of the RAM */
```

```

int bit_number;          /* IN - The number of the bit within
.. the RAM word which is to be
.. written. */
```

```

int word_number;          /* IN - The number of the word
.. within the RAM which is to be
.. accessed. */
```

```

short *ram;               /* The pointer to the RAM storage
.. array. */
```

```

{
```

Determine the ram_offset and ram_index values.

Retrieve the current value from the ram[] array, and store it in 'base'.

Store the 'out' value [cm_mask_and_store()] into the 'base' integer.

Copy the new 'base' value into the ram[] array.

}

```
static void cm_store_ram_value(Digital_State_t out,int word_width,
                               int bit_number,
                               Digital_State_t *address,
                               int address_size, short *ram)
```

```
Digital_State_t out;      /* IN - The value to be stored      */
```

```
int word_width;          /* IN - The default word width
... of the RAM           */
```

```
int bit_number;          /* IN - The number of the bit within
... the RAM word which is to be
... written.                */
```

```
int *address;            /* IN - The address of the bit to
... which the out value is to be
... stored.                  */
```

```
short *ram;              /* The pointer to the RAM storage
... array.                   */
```

{

Convert the address values into a decimal index
[cm_address_to_decimal()].

If no error occurred,

Obtain the ram_offset and ram_index values.

Copy the value stored in the ram_indexed ram[]
storage location into 'base'.

Store the 'out' value into the 'base' integer
[cm_mask_and_store()].

Copy the new 'base' value into ram[].

}

```
static void cm_get_ram_value(int word_width,int bit_number,
                            Digital_State_t *address,
```

```

        int address_size,short *ram,
        Digital_State_t *out)

int word_width;          /* IN - The default word width
.. of the RAM */           */

int bit_number;          /* IN - The number of the bit within
.. the RAM word which is to be
.. retrieved. */            */

int *address;            /* IN - The address of the bit to
.. be accessed. */          */

short *ram;               /* The pointer to the RAM storage
.. array. */                */

Digital_State_t *out;     /* OUT - The value retrieved */      */

{

    First obtain the integer word index value from the
    current address inputs [cm_address_to_decimal()].
    If no error occurred,
        Calculate the ram_offset and ram_index values.
        Retrieve the value in ram[ram_index] and store it
        in 'base'.
        Retrieve the specific bit value stored in 'base'
        [cm_mask_and_retrieve()].
    Else
        Return an UNKNOWN value if the specified address
        is nonsensical.

}

void cm_d_ram(ARGS)
{
    ARGS = Mif_Private_t *private; /* INOUT - Code model inputs,
.. outputs, and parameters */
    {
        Retrieve device size values & other parameters.
}

```

```
Calculate RAM word size from address size.

Setup required state variables:

If this is the initial pass, allocate storage.

Allocate storage for ram memory [cm_event_alloc()] and

Declare load values.

Else, retrieve previous values.

Retrieve storage for the outputs [cm_event_get_ptr()].
Retrieve ram base addresses.

Retrieve inputs

Determine whether we are selected or not...

Retrieve the bit equivalents for the select lines.

Compare each bit in succession with the value of each of the
select input lines.

Shift the values select_value bits for next compare...

Determine analysis type and output appropriate values.

If this is a DC analysis, output w/o delays.

Initialize RAM to ic value.

This section initializes all two-bit ram locations
to the value represented by PARAM(ic). This would
be either 0, 1, or UNKNOWN (2)...

If we are selected,

Output STRONG initial conditions.

Else,

Change output to high impedance.

If this is a transient analysis,

Find input that has changed...

If select value has changed,
```

And if we are selected,

And if our write line is inactive,

Need to retrieve new output [cm_get_ram_value()].

For each output bit in the word,
retrieve the state value.

If our write line is active,

Store word & output current value [cm_initialize_ram()].

If our write line is unknown,

Entire ram goes unknown!!![cm_initialize_ram()]

If the model is not selected,

The output goes tristate unless already so.

If select value didn't change,

Either a change in write_en, a change in address
or a change in data values must have occurred...
all of these are treated similarly.

Calculate action to take in each case and post.

}

4.2.3.23 Digital Source

Summary

The digital source provides for straightforward descriptions of digital signal vectors in a tabular format. The model reads input from the input file, and at the times specified in the file generates the inputs along with the strengths listed.

The format of the input file is as shown below. Note that comment lines are delineated through the use of a single “*” character in the first column of a line. This is similar to the way the SPICE program handles comments.

```
* T      c      n      n      n      . . .
* i      l      o      o      o      . . .
* m      o      d      d      d      . . .
* e      c      e      e      e      . . .
* k      a      b      c      . . .

0.0000  Uu    Uu    Us    Uu    . . .
1.234e-9 0s    1s    1s    Oz    . . .
1.376e-9 0s    0s    1s    Oz    . . .
2.5e-7   1s    0s    1s    Oz    . . .
2.5006e-7 1s    1s    1s    Oz    . . .
5.0e-7   0s    1s    1s    Oz    . . .
```

Note that in the example shown, whitespace (any combination of blanks, tabs, commas) is used to separate the time and strength/state tokens. The order of the input columns is important; the first column is always interpreted to mean “time”. The second through the N’th columns map to the out[0] through out[N-2] output nodes. A non-commented line which does not contain enough tokens to completely define all outputs for the digital_source will cause an error. Also, time values must increase monotonically, or an error will result in reading the source file.

Errors will also occur if a line exists in source.in which is not either a comment or vector line. The only exception to this is in the case of a line that is completely blank; this is treated as a comment (note that such lines often occur at the end of text within a file; ignoring these in particular prevents nuisance errors on the part of the simulator).

Called By

MIFload() MIF/MIFload.c

Returned Value

None.

PDL Description

```
static char *CNVgettok(char **s)

char **s; /* INOUT - The pointer to the input string; this is
.. updated once a token is copied, such that the
.. next token is pointed to. */

{
    Declare variables.

    Allocate space large enough to hold the input [malloc()].
    Skip any whitespace.

    Check the next token:
    If the token is a null ('\0'), free space and return a NULL.

    Else,
        Copy each character into the 'buf' internal variable space,
        and add a null to the end of the character string.

        Skip over any additional whitespace from the end of the
        token to the beginning of the next token.

        Create a new string (ret_str) of exactly the same length as that used
        by buf, and copy the token into it.

        Free the space used by buf and return ret_str.

}
```

```
static char *CNVget_token(char **s, Cnv_Token_Type_t *type)

char **s; /* INOUT - The pointer to the input string;
.. this is updated once a token is copied,
.. such that the next token is pointed to. */
Cnv_Token_Type_t *type; /* OUT - The pointer is used to pass back
.. information about whether a string has
```

```
    ... valid token data or not.      */

{

    Obtain the token value [CNVgettok()].
    If the value returned was a NULL,
        Set the type to CNV_NO_TOK and return a NULL value.
    Else
        Set the type to CNV_STRING_TOK and return the token.
}

static int cnv_get_spice_value(str,p_value)

char  *str;          /* IN - The value text e.g. 1.2K */
float *p_value;     /* OUT - The numerical value      */

{
    Scan the input string looking for an alpha character that is not
    'e' or 'E'. Such a character is assumed to be an engineering
    suffix as defined in the Spice 2G.6 user's manual.

    Determine the scaling factor by matching the letter value
    found against the accepted SPICE exponential notation
    (e.g., t => E+12 g => E+9, etc.)

    Use the string scan function to retrieve the mantissa of the
    value. If the sscanf() fails, return FAIL value.

    Multiply by the exponential determined previously, and return.
}

static int cm_source_mask_and_retrieve(short *base,int ram_offset,
                                       Digital_State_t *out)

short *base;          /* INOUT - The base short integer into which
.. the data will be stored           */
int bit_offset;       /* IN - The integer value (from 0 to 7) of
.. the 2-bits which will be used to store
.. the data.                         */
Digital_State_t *out; /* OUT - The actual state value to be read. */


```

```
{  
  
    For each possible value of ram_offset,  
  
        Mask the appropriate bit and test for the stored value  
        at that location. Depending on the value (binary  
        0, 1, or 2), set sout, and return.  
  
}  
  
  
static int cm_mask_and_store(short *base,int ram_offset,  
                            Digital_State_t out)  
  
short *base;           /* IINOUT - The base short integer into which  
.. the data will be stored */  
  
int bit_offset;        /* IN - The integer value (from 0 to 7) of  
.. the 2-bits which will be used to store  
.. the data. */  
  
Digital_State_t bit_value; /* IN - The actual state value to be stored. */  
  
{  
  
    Depending on which storage location is to be used,  
  
        If the out value is a ZERO, zero the bit values by  
        ANDing with an appropriate mask.  
  
        If the out value is a ONE, first zero the bit values,  
        then store ONE by ORing with an appropriate mask.  
  
}  
  
  
static void cm_get_source_value(int word_width,int bit_number,int index,  
                                short *bits, Digital_t *out)  
  
int word_width;         /* IN - The width of the source vector */  
  
int bit_number;         /* IN - The number of the bit within  
.. a given source word that we wish  
.. to retrieve. */  
  
int index;              /* IN - The specific source word that  
.. is desired. */  
  
short *bits;             /* IN - The storage vector for all  
.. of the data. */
```

```
short *out;           /* OUT - the output integer representing
.. the retrieved data.          */

{
    Determine the bit_index and bit_offset values.

    Retrieve the word pointed to by the bit_index from *bits.

    Mask off the *base word and store the desired bit value
    in *out [cm_source_mask_and_retrieve()].
}

static int cm_read_source(FILE *source, short *bits, double *timepoints,
                         Source_Table_Info_t *info)

FILE *source;           /* IN - The name of the source file          */

short *bits;            /* OUT - The storage vector into which
.. the source state and strength
.. information will be encoded.          */

double *timepoints;     /* OUT - The storage vector into which
.. the timepoint information will
.. be written.          */

Source_Table_Info_t *info; /* IN - This is the data structure which
.. contains the current index, the
.. width, and the depth of the source
.. information.          */

{
    For each non-null string in the state file,
        If this is not a blank line,
            And if this is not a comment string,
                Count the number of tokens in this string, and
                Set the string type based on this.

                If the number of tokens is not what is expected for
                either a State Header line or a State Continuation Line,
                Return a '1' value to the calling routine and exit.

                Reset the current string pointer to the beginning of this line.

    For the expected width of the line,
```

```
    Retrieve each token [CNVget_token()].  
  
    If this is the first token,  
  
        Store the timepoint value [cnv_get_spice_value()].  
  
        Check to make sure that each timepoint is  
        greater than the previous timepoint, else  
        return a '1' value to the calling routine.  
  
    else this token represents a bit value:  
  
        Determine the integer representation of this  
        token's state+strength. If the token is  
        unrecognized, return a '1'.  
  
        If the token is a valid bit state+strength,  
  
            Determine where the interger representation  
            needs to be stored, and store it  
            [cm_source_mask_and_store()].  
  
    Return a '0' if all went well to this point.
```

}

```
void cm_d_source(ARGS)  
  
ARGS = Mif_Private_t *private; /* INOUT - Code model inputs,  
.. outputs, and parameters */  
  
{  
    Declare variables  
  
    If this is the initial pass,  
  
        open the source file and count the number of vectors in it.  
  
        Allocate storage for index, bits, & timepoints  
        [cm_event_alloc(), cm_event_get_ptr()].  
  
        Initialize info values...  
  
        Retrieve width of the source.  
  
        Initialize bits & timepoints to zero.  
  
        Send file pointer and the two array storage pointers  
        to [cm_read_source()]. This will return after
```

reading the contents of source.in, and if no errors have occurred, the " bits" and " timepoints" vectors will be loaded and the width and depth values supplied.

If a problem occurred in load...send error msg [cm_message_send()].

Reset bits & timepoints to zero.

Close source file.

Else, if not an initial pass, retrieve previous values.

Retrieve info [cm_event_get_ptr()].

Get old values to new...

Retrieve bits [cm_event_get_ptr()].

Set old values to new...

Retrieve timepoints [cm_event_get_ptr()].

Set old values to new...

If this is a DC call...

If the current time index indicates that time is zero,
Set DC value. This is done for the first call only.

Reset current breakpoint [cm_event_queue()].

Output new values...

Retrieve output value [cm_get_source_value()].

Increment breakpoint.

Set next breakpoint as long as depth
has not been exceeded [cm_event_queue()].

Else set breakpoint for first time index:

Set next breakpoint as long as depth
has not been exceeded [cm_event_queue()].

Otherwise, if this isn't the very first call..

Retrieve last index value and branch to appropriate
routine based on the last breakpoint's relationship

```
to the current time value.

If breakpoint has not occurred,
    Output hasn't changed...do nothing this time...
    Except for queueing up the next breakpoint again
    [cm_event_queue()].
Otherwise,
    Breakpoint has been reached or exceeded.

    If the breakpoint has been reached,
        Reset current breakpoint.

        Output new values...

        Retrieve output value [cm_get_source_value()].
        Increment breakpoint.

        Set next breakpoint as long as depth
        has not been exceeded [cm_event_queue()].
Otherwise..
    Last source file breakpoint has been exceeded...
    do not change the value of the output.

}
```

4.3 User-Defined Node Library

The following two simple user-defined node types are provided in the XSPICE User-Defined Node Library (CM-UDNL):

```
real
int
```

4.3.1 Node Type "real"

The "real" node type (CM-UDNL-REAL) provides for event-driven simulation with double-precision floating point data. This type is useful for evaluating sampled-data filters and systems. The type implements all optional functions for user-defined nodes, including inversion and node resolution. For inversion, the sign of the value is reversed. For node resolution, the resultant value at a node is the sum of all values output to that node.

4.3.1.1 Function udn_real_create

Summary

This function performs the create operation for the real node type.

Called By

EVTsetup_data()	EVT/EVTsetup.c
EVTcreate_output_event()	EVT/EVTload.c
EVTnode_copy()	EVT/EVTnode_copy.c

Returned Value

None.

PDL Description

```
void udn_real_create(CREATE_ARGS)
{
    Allocate space for a 'double' data type and assign to MALLOCED_PTR.
}
```

4.3.1.2 Function udn_real_dismantle

Summary

This function performs the dismantle operation for the real node type. User-defined node “dismantle” functions are not called in this version of XSPICE. Future versions may call this function to reclaim memory during or after a simulation.

Called By

None.

Returned Value

None.

PDL Description

```
void udn_real_dismantle(DISMANTLE_ARGS)
{
    Do nothing. There are no internally allocated things to dismantle.
}
```

4.3.1.3 Function udn_real_initialize

Summary

This function performs the initialize operation for the real node type.

Called By

EVTsetup_data() EVT/EVTsetup.c

Returned Value

None.

PDL Description

```
void udn_real_initialize(INITIALIZE_ARGS)
{
    Cast STRUCT_PTR to a pointer to type double and assign a value
    of zero.
}
```

4.3.1.4 Function udn_real_invert

Summary

This function performs the invert operation for the real node type. The inverted value is the negative of the input value.

Called By

EVTiter()	EVT/EVTiter.c
EVTprocess_output()	EVT/EVTLload.c
EVTnode_insert()	EVT/EVTtermInsert.c

Returned Value

None.

PDL Description

```
void udn_real_invert(INVERT_ARGS)
{
    Cast STRUCT_PTR to a 'double' pointer and change the sign of the value.
}
```

4.3.1.5 Function udn_real_copy

Summary

This function performs the copy operation for the real node type.

Called By

EVTprocess_output()	EVT/EVTdequeue.c
EVTiter()	EVT/EVTiter.c
EVTprocess_output()	EVT/EVTload.c
EVTnode_copy()	EVT/EVTnode_copy.c

Returned Value

None.

PDL Description

```
void udn_real_copy(COPY_tRGS)
{
    Cast INPUT_STRUCT_PTR and OUTPUT_STRUCT_PTR to pointers of type
    'double' and copy the value.
}
```

4.3.1.6 Function udn_real_resolve

Summary

This function performs the resolve operation for the real node type. The resolved value is the sum of all values output to the node.

Called By

EVTiter() EVT/EVTiter.c

Returned Value

None.

PDL Description

```
void udn_real_resolve(RESOLVE_ARGS)
{
    Cast INPUT_STRUCT_PTR_ARRAY to a pointer to a vector of 'doubles'
    and OUTPUT_STRUCT_PTR to a 'double' pointer.

    For number of structures indicated by INPUT_STRUCT_PTR_ARRAY_SIZE,
        Sum the values in the elements of INPUT_STRUCT_PTR_ARRAY.

    Assign the result to OUTPUT_STRUCT_PTR.
}
```

4.3.1.7 Function udn_real_compare

Summary

This function performs the create operation for the real node type. The values are considered equal if they are identical.

Called By

EVTprocess_output()	EVT/EVTdequeue.c
EVTdump()	EVT/EVTDump.c
EVTiter()	EVT/EVTitler.c
EVTprocess_output()	EVT/EVTload.c
EVTnode_compare()	EVT/EVTop.c

Returned Value

None.

PDL Description

```
void udn_real_compare(COMPARE_ARGS)
{
    Cast STRUCT_PTR_1 and STRUCT_PTR_2 to 'double' pointers and test
    their values for equality, setting EQUAL to TRUE or FALSE accordingly.
}
```

4.3.1.8 Function udn_real_plot_val

Summary

This function outputs the plot value for the real node type.

Called By

EVTdump() EVT/EVTdump.c

Returned Value

None.

PDL Description

```
void udn_real_plot_val(PLOT_VAL_ARGS)
{
    Cast STRUCT_PTR to a 'double' pointer and assign PLOT_VAL the
    value of the double.
}
```

4.3.1.9 Function udn_real_print_val

Summary

This function outputs the print value for the real node type.

Called By

EVTdump() EVT/EVTdump.c

Returned Value

None.

PDL Description

```
void udn_real_print_val(PRINT_VAL_ARGS)
{
    Allocate 30 characters of space for the print value string.
    Cast STRUCT_PTR to a 'double' pointer and format the double value into
    a string.
}
```

4.3.1.10 Function udn_real_ipc_val

Summary

This function determines the IPC value for the "real" node type.

Called By

EVTdump() EVT/EVTdump.c

Returned Value

None.

PDL Description

```
void udn_real_ipc_val(IPC_VAL_ARGS)
{
    Assign STRUCT_PTR to IPC_VAL.
    Assign the size of a 'double' to IPC_VAL_SIZE.
}
```

4.3.2 Node Type "int"

The "int" node type (CM-UDNL-INT) provides for event-driven simulation with integer data. This type is useful for evaluating roundoff error effects in sampled-data systems. The type implements all optional functions for user-defined nodes, including inversion and node resolution. For inversion, the sign of the integer value is reversed. For node resolution, the resultant value at a node is the sum of all values output to that node.

4.3.2.1 Function udn_int_create

Summary

This function performs the create operation for the int node type.

Called By

EVTsetup_data()	EVT/EVTsetup.c
EVTcreate_output_event()	EVT/EVTload.c
EVTnode_copy()	EVT/EVTnode_copy.c

Returned Value

None.

PDL Description

```
void udn_int_create(CREATE_ARGS)
{
    Allocate space for an 'int' data type and assign to MALLOCED_PTR.
}
```

4.3.2.2 Function udn_int_dismantle

Summary

This function performs the dismantle operation for the int node type. User-defined node “dismantle” functions are not called in this version of XSPICE. Future versions may call this function to reclaim memory during or after a simulation.

Called By

None.

Returned Value

None.

PDL Description

```
void udn_int_dismantle(DISMANTLE_ARGS)
{
    Do nothing.  There are no internally allocated things to dismantle.
}
```

4.3.2.3 Function udn_int_initialize

Summary

This function performs the initialize operation for the int node type.

Called By

EVTsetup_data() EVT/EVTsetup.c

Returned Value

None.

PDL Description

```
void udn_int_initialize(INITIALIZE_ARGS)
{
    Cast STRUCT_PTR to a pointer to type int and assign a value
    of zero.
}
```

4.3.2.4 Function udn_int_invert

Summary

This function performs the invert operation for the int node type. The inverted value is the negative of the input value.

Called By

EVTiter()	EVT/EVTiter.c
EVTprocess_output()	EVT/EVTLload.c
EVTnode_insert()	EVT/EVTtermInsert.c

Returned Value

None.

PDL Description

```
void udn_int_invert(INVERT_ARGS)
{
    Cast STRUCT_PTR to an 'int' pointer and change the sign of the value.
}
```

4.3.2.5 Function udn_int_copy

Summary

This function performs the copy operation for the int node type.

Called By

EVTprocess_output()	EVT/EVTdequeue.c
EVTiter()	EVT/EVTitler.c
EVTprocess_output()	EVT/EVTload.c
EVTnode_copy()	EVT/EVTnode_copy.c

Returned Value

None.

PDL Description

```
void udn_int_copy(COPY_ARGS)
{
    Cast INPUT_STRUCT_PTR and OUTPUT_STRUCT_PTR to pointers of type
    'int' and copy the value.
}
```

4.3.2.6 Function udn_int_resolve

Summary

This function performs the resolve operation for the int node type. The resolved value is the sum of all values output to the node.

Called By

EVTiter() EVT/EVTiter.c

Returned Value

None.

PDL Description

```
void udn_int_resolve(RESOLVE_ARGS)
{
    Cast INPUT_STRUCT_PTR_ARRAY to a pointer to a vector of 'ints'
    and OUTPUT_STRUCT_PTR to an 'int' pointer.

    For number of structures indicated by INPUT_STRUCT_PTR_ARRAY_SIZE,
        Sum the values in the elements of INPUT_STRUCT_PTR_ARRAY.

    Assign the result to OUTPUT_STRUCT_PTR.
}
```

4.3.2.7 Function udn_int_compare

Summary

This function performs the create operation for the int node type. The values are considered equal if they are identical.

Called By

EVTprocess_output()	EVT/EVTdequeue.c
EVTDump()	EVT/EVTDump.c
EVTiter()	EVT/EVTiter.c
EVTprocess_output()	EVT/EVTload.c
EVTnode_compare()	EVT/EVTop.c

Returned Value

None.

PDL Description

```
void udn_int_compare(COMPARE_ARGS)
{
    Cast STRUCT_PTR_1 and STRUCT_PTR_2 to 'int' pointers and test
    their values for equality, setting EQUAL to TRUE or FALSE accordingly.
}
```

4.3.2.8 Function udn_int_plot_val

Summary

This function outputs the plot value for the int node type.

Called By

EVTdump() EVT/EVTdump.c

Returned Value

None.

PDL Description

```
void udn_int_plot_val(PLOT_VAL_ARGS)
{
    Cast STRUCT_PTR to an 'int' pointer and assign PLOT_VAL the
    value of the int.
}
```

4.3.2.9 Function udn_int_print_val

Summary

This function outputs the print value for the int node type.

Called By

EVTdump() EVT/EVTdump.c

Returned Value

None.

PDL Description

```
void udn_int_print_val(PRINT_VAL_ARGS)
{
    Allocate 30 characters of space for the print value string.
    Cast STRUCT_PTR to an 'int' pointer and format the int value into
    a string.
}
```

4.3.2.10 Function udn_int_ipc_val

Summary

This function determines the IPC value for the "int" node type.

Called By

EVTdump() EVT/EVTdump.c

Returned Value

None.

PDL Description

```
void udn_int_ipc_val(IPC_VAL_ARGS)
{
    Assign STRUCT_PTR to IPC_VAL.
    Assign the size of an 'int' to IPC_VAL_SIZE.
}
```

5

CSCI Data

This section discusses the Code Model Preprocessor only. It is not applicable to the other components of this CSCI.

5.1 Interface Specification Data

The principal data structure employed in the Code Model Preprocessor is an internal representation of data found in the Interface Specification file. This structure is of type `Ifs_Table_t` declared in `cmpp.h`. All three modes of the preprocessor begin execution by reading the Interface Specification file to create this structure. Each mode then uses the information in the structure to perform its particular actions. The `Ifs_Table_t` structure contains four substructures, one for each major section of the Interface Specification file.

The `Ifs_Table_t` structure is declared and initialized in each of the three major functions:

```
process_ifs_file()  
process_mod_file()  
process_lst_file()
```

and is passed between functions in their argument lists, and as a global variable when used by the `yyparse()` functions created by lex/yacc.

Its definition, and the definition of its substructures are shown below.

```

typedef struct {

    Name_Info_t    name;          /* The name table entries */
    int            num_conn;      /* Number of entries in the port table(s) */
    Conn_Info_t   *econn;         /* Array of port/connection info structs */
    int            num_param;     /* Number of entries in the parameter table(s) */
    Param_Info_t  *param;        /* Array of parameter info structs */
    int            num_inst_var; /* Number of entries in the static/inst var table(s) */
    Inst_Var_Info_t *inst_var;   /* Array of static/inst var info structs */

} Ifs_Table_t;

/*
 * Information about the name of the model
 */
typedef struct {

    char      *c_fcn_name;      /* Name used in the C function */
    char      *model_name;      /* Name used in a spice deck */
    char      *description;     /* Description of the model */

} Name_Info_t;

/*
 * Information about a port/connection
 */
typedef struct {

    char      *name;           /* Name of this port/connection */
    char      *description;    /* Description of this port/connection */
    Dir_t     direction;       /* IN, OUT, or INOUT */
    Port_Type_t default_port_type; /* The default port type */
    char      *default_type;   /* The default type in string form */
    int       num_allowed_types; /* The size of the allowed type arrays */
    Port_Type_t *allowed_port_type; /* Array of allowed types */
    char      *allowed_type;   /* Array of allowed types in string form */
    Boolean_t is_array;        /* True if connection is an array */
    Boolean_t has_conn_ref;   /* True if there is associated with an array conn */
    int       conn_ref;        /* Subscript of the associated array conn */
    Boolean_t has_lower_bound; /* True if there is an array size lower bound */
    int       lower_bound;     /* Array size lower bound */
    Boolean_t has_upper_bound; /* True if there is an array size upper bound */
    int       upper_bound;     /* Array size upper bound */
    Boolean_t null_allowed;   /* True if null is allowed for this connection */

} Conn_Info_t;

```

```

/*
 * Information about a parameter
 */

typedef struct {

    char          *name;           /* Name of this parameter */
    char          *description;   /* Description of this parameter */
    Data_Type_t   type;           /* Data type, e.g., REAL, INTEGER, ... */
    Boolean_t     has_default;   /* True if there is a default value */
    Value_t       default_value; /* The default value */
    Boolean_t     has_lower_limit; /* True if there is a lower limit */
    Value_t       lower_limit;   /* The lower limit for this parameter */
    Boolean_t     has_upper_limit; /* True if there is a upper limit */
    Value_t       upper_limit;   /* The upper limit for this parameter */
    Boolean_t     is_array;       /* True if parameter is an array */
    Boolean_t     has_conn_ref;  /* True if there is associated with an array conn */
    int           conn_ref;      /* Subscript of the associated array conn */
    Boolean_t     has_lower_bound; /* True if there is an array size lower bound */
    int           lower_bound;   /* Array size lower bound */
    Boolean_t     has_upper_bound; /* True if there is an array size upper bound */
    int           upper_bound;   /* Array size upper bound */
    Boolean_t     null_allowed;  /* True if null is allowed for this parameter */

} Param_Info_t;

/*
 * Information about a static-instance variable
 */

typedef struct {

    char          *name;           /* Name of this variable */
    char          *description;   /* Description of this variable */
    Data_Type_t   type;           /* Data type, e.g. REAL, INTEGER, ... */
    Boolean_t     is_array;       /* True if parameter is an array */

} Inst_Var_Info_t;

/*
 * The boolean type
 */

typedef enum {
    FALSE,        /* False */
    TRUE,         /* True */
} Boolean_t;

```

```
/*
 * The direction of a port/connector
 */

typedef enum {
    IN,           /* Input only */
    OUT,          /* Output only */
    INOUT,         /* Input and output */
} Dir_t;

/*
 * The type of a port
 */

typedef enum {
    VOLTAGE,        /* v - Single-ended voltage */
    DIFF_VOLTAGE,   /* vd - Differential voltage */
    CURRENT,        /* i - Single-ended current */
    DIFF_CURRENT,   /* id - Differential current */
    VSOURCE_CURRENT,/* vnam - Vsorce name for input current */
    CONDUCTANCE,    /* g - Single-ended VCIS */
    DIFF_CONDUCTANCE,/* gd - Differential VCIS */
    RESISTANCE,     /* h - Single-ended ICVS */
    DIFF_RESISTANCE,/* hd - Differential ICVS */
    DIGITAL,        /* d - Digital */
    USER_DEFINED,   /* <identifier> - Any user defined type */
} Port_Type_t;

/*
 * The type of a parameter or Static_Var
 */

typedef enum {
    BOOLEAN,        /* Boolean_t type */
    INTEGER,        /* int type */
    REAL,           /* double type */
    COMPLEX,        /* Complex_t type */
    STRING,          /* char * type */
    POINTER,        /* void * type */
    /* NOTE: POINTER should not be used for Parameters - only
     * Static_Vars - this is enforced by the cmpp.
     */
} Data_Type_t;
```

```
/*
 * The complex type
 */

typedef struct {
    double real;      /* Real part */
    double imag;      /* Imaginary part */
} Complex_t;

/*
 * Values of different types.
 *
 * Note that a struct is used instead of a union for conformity
 * with the use of the Mif_Value_t type in the simulator where
 * the type must be statically initialized. ANSI C does not
 * support useful initialization of unions.
 *
 */
typedef struct {

    Boolean_t   bvalue;          /* For BOOLEAN parameters */
    int         ivalue;          /* For INTEGER parameters */
    double      rvalue;          /* For REAL parameters */
    Complex_t   cvalue;          /* For COMPLEX parameters */
    char        *svalue;          /* For STRING parameters */

} Value_t;
```

6

CSCI Data Files

The files shown in Table 6.1 are accessed or created by the Code Model Preprocessor.

Command-line Switch	Input Files	Output Files
-ifs	ifspec.ifs	ifspec.c
-mod	ifspec.ifs cfunc.mod	cfunc.c
-lst	ifspec.ifs modpath.lst udnpath.lst	CMinfo.h CMextrn.h UDNinfo.h UDNextrn.h objects.inc

Table 6.1 Code Model Toolkit Data Files.

Files in the center column are created by a user and serve as input to the preprocessor. Files on the right are created by the preprocessor to serve as input to compilation/linking of code models and the simulator.

In creating directories, the Model Directory Generator (`mkmoddir`), User-Defined Node Directory Generator (`mkudndir`), and the Simulator Directory Generator (`mksimdir`) tools copy the files shown in Table 6.2 into the new directories to act as templates for the user to build from.

Tool	Template Files Copied
mkmoddir	ifspec.ifs cfunc.mod Makefile
mkudndir	udnfunc.c Makefile
mksimdir	modpath.lst Makefile

Table 6.2 Code Model Toolkit Template Files.

7

Requirements Traceability

Requirements governing the design of the Simulator Code Model Subsystem are specified in the Software Requirements Specification for the Simulator of the Automatic Test Equipment Software Support Environment (ATESSE). Table 7.1 cross references CSCs in the Code Model Subsystem to requirements.

Project Unique Identifier	SRS Requirement Section
CM-CMT	3.2.2.4.3.3
CM-CMT-MDG	3.2.2.4.3.3.1
CM-CMT-UDNG	3.2.2.4.2.7
CM-CMT-SDG	3.2.2.4.3.3.2
CM-CMT-CMPP	3.2.2.4.3.3.1 3.2.2.4.3.3.2

Table 7.1 Requirements Cross Reference.

Project Unique Identifier	SRS Requirement Section
CM-CML	3.2.2.4.4
CM-CML-GAIN	3.2.2.4.4.1.1
CM-CML-SUM	3.2.2.4.4.1.2
CM-CML-MULT	3.2.2.4.4.1.3
CM-CML-DIV	3.2.2.4.4.1.4
CM-CML-LIM	3.2.2.4.4.1.9
CM-CML-CLIM	3.2.2.4.4.1.11
CM-CML-PCS	3.2.2.4.4.1.7
CM-CML-ASW	3.2.2.4.4.1.14
CM-CML-ZEN	3.2.2.4.4.1.15
CM-CML-ILIM	3.2.2.4.4.1.10
CM-CML-HYST	3.2.2.4.4.1.8
CM-CML-DIFF	3.2.2.4.4.1.6
CM-CML-INT	3.2.2.4.4.1.5
CM-CML-SDT	None
CM-CML-SLW	None
CM-CML-LCP	3.2.2.4.4.1.20
CM-CML-CORE	3.2.2.4.4.1.19
CM-CML-SINE	3.2.2.4.4.1.12
CM-CML-TRI	3.2.2.4.4.1.13
CM-CML-SQR	None
CM-CML-OS	3.2.2.4.4.1.21
CM-CML-CAP	3.2.2.4.2.2
CM-CML-CMET	3.2.2.4.4.1.16
CM-CML-IND	3.2.2.4.2.2
CM-CML-LMET	3.2.2.4.4.1.17
CM-CML-DAC	3.2.2.4.4.2.4
CM-CML-ADC	3.2.2.4.4.2.3
CM-CML-DOSC	3.2.2.4.4.2.1

Table 7.1: Requirements Cross Reference (continued).

Project Unique Identifier	SRS Requirement Section
CM-CML-DBUF	3.2.2.4.4.3.1
CM-CML-DINV	3.2.2.4.4.3.2
CM-CML-AND	3.2.2.4.4.3.3
CM-CML-DNND	3.2.2.4.4.3.4
CM-CML-DOR	3.2.2.4.4.3.5
CM-CML-DNOR	3.2.2.4.4.3.6
CM-CML-XOR	3.2.2.4.4.3.7
CM-CML-XNR	3.2.2.4.4.3.8
CM-CML-DTS	None
CM-CML-DOC	None
CM-CML-DOE	None
CM-CML-DPU	None
CM-CML-DPD	None
CM-CML-DDFF	3.2.2.4.4.3.9
CM-CML-DJKF	3.2.2.4.4.3.10
CM-CML-DTFF	3.2.2.4.4.3.11
CM-CML-DSRF	3.2.2.4.4.3.12
CM-CML-DDL	3.2.2.4.4.3.13
CM-CML-DSRL	3.2.2.4.4.3.14
CM-CML-DSTT	3.2.2.4.4.3.15
CM-CML-DFRQ	3.2.2.4.4.3.16
CM-CML-DRAM	3.2.2.4.4.3.17
CM-CML-DSRC	None
CM-UDNL	None
CM-UDNL-REAL	None
CM-UDNL-INT	None

Table 7.1: Requirements Cross Reference (concluded).

8

Notes

8.1 Glossary

ATESSE	Automatic Test Equipment Software Support Environment. An integrated set of software tools designed to aid in development of programs for testing mixed-mode (analog/digital) printed circuit cards.
C	A programming language developed in the early 70's at Bell Labs. It is the standard programming language used for system development on Unix machines.
Code Model Library	The set of code models supplied with the XSPICE simulator.
Code Model Preprocessor	A code model development tool that assists in adding new code models to the XSPICE simulator. It is automatically run by "make" to convert user-written files into C language source files that are compiled and linked with the simulator.
Code Model Toolkit	A set of tools that assist in adding new code models to the XSPICE simulator.
Interprocess Communication	Communication between two separate programs running concurrently on one or more computers.
lex	A UNIX operating system utility used to develop parsers.
Make	A UNIX software development tool that automates the compilation and linking of a program.

Makefile	A file that instructs the UNIX make utility how to compile and link a program.
malloc	A standard C library function for dynamically allocating memory in a program.
Model Directory Generator	A code model development tool that assists in adding new code models to the XSPICE simulator. It creates a directory in which a code model will be created.
Simulator Directory Generator	A code model development tool that assists in adding new code models to the XSPICE simulator. It creates a directory in which a copy of the simulator will be built.
SPICE	An analog simulation program developed at the University of California at Berkeley.
User-Defined Node Directory Generator	A development tool that assists in adding new user-defined nodes to the XSPICE simulator. It creates a directory in which a new node type will be created.
User-Defined Node Library	The set of user-defined node types supplied with the XSPICE simulator.
yacc	Yet Another Compiler Compiler. A UNIX operating system utility used in translating one language to another.

8.2 Acronyms

ANSI	American National Standards Institute
ATESSE	Automatic Test Equipment Software Support Environment
CDRL	Contract Deliverable Requirement List
CSCI	Computer Software Configuration Item
IFS	Interface Specification File
IPC	Interprocess Communication
MNA	Modified Nodal Analysis
PWL	Piecewise-Linear
SPICE	Simulation Program with Integrated Circuit Emphasis
UDN	User-Defined Node
XSPICE	Extended SPICE

8.3 Project Unique Identifiers

CM	Code Model Subsystem
CM-CML	Code Model Library
CM-CML-ADC	Analog-to-digital node bridge code model
CM-CML-AND	Digital AND gate code model
CM-CML-ASW	Analog switch code model
CM-CML-CAP	Capacitor code model
CM-CML-CLIM	Controlled limiter code model
CM-CML-CMET	Capacitance meter code model
CM-CML-CORE	Magnetic core code model
CM-CML-DAC	Digital-to-analog node bridge code model
CM-CML-DBUF	Digital buffer code model
CM-CML-DDFF	Digital D flip flop code model
CM-CML-DDL	Digital D latch code model
CM-CML-DFRQ	Digital frequency divider code model
CM-CML-DIFF	Differentiator code model
CM-CML-DINV	Digital inverter code model
CM-CML-DIV	Divider code model
CM-CML-DJKF	Digital JK flip flop code model
CM-CML-DNND	Digital NAND gate code model
CM-CML-DNOR	Digital NOR gate code model
CM-CML-DOC	Digital open-collector buffer code model
CM-CML-DOE	Digital open-emitter buffer code model
CM-CML-DOR	Digital OR gate code model
CM-CML-DOSC	Controlled digital oscillator code model
CM-CML-DPD	Digital pulldown code model
CM-CML-DPU	Digital pullup code model

CM-CMT-CMPP	Code Model Preprocessor
CM-CMT-MDG	Model Directory Generator
CM-CMT-SDG	Simulator Directory Generator
CM-CMT-UDNG	User-Defined Node Directory Generator
CM-SI-IN	Input to the Code Model Subsystem from the Simulator Interface
CM-SI-OUT	Output to the Simulator Interface from the Code Model Subsystem
CM-UDNL	User-Defined Node Library
CM-UDNL-INT	User-defined node type "int"
CM-UDNL-REAL	User-defined node type "real"
SI	ATESSE Version 2 Simulator Interface
SIM	Simulator
SIM-CM-IN	Input to the Simulator from the Code Model Subsystem