

# **SOFTWARE DESIGN DOCUMENT**

**for the**

## **XSPICE SIMULATOR**

**of the**

### **AUTOMATIC TEST EQUIPMENT SOFTWARE SUPPORT ENVIRONMENT (ATESSE)**

**Contract No. F09603-89-G-0077-0002**

**CDRL Sequence No. A00B**

**December 1992**

**Prepared for:**

**Department of the Air Force  
Warner Robins Air Logistics Center  
Robins Air Force Base, Georgia 31098**

**Prepared by:**

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

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

Copyright 1992  
Georgia Tech Research Corporation  
All Rights Reserved.  
This material may be reproduced by or for the U.S. Government  
pursuant to the copyright license under the clause at DFARS  
252.227-7013 (Oct. 1988)

# Contents

<b>1 Scope</b>	<b>1</b>
1.1 Identification . . . . .	1
1.2 System Overview . . . . .	1
1.3 Document Overview . . . . .	3
1.4 Acknowledgement . . . . .	3
<b>2 Referenced Documents</b>	<b>5</b>
<b>3 Preliminary Design</b>	<b>7</b>
3.1 CSCI Overview . . . . .	7
3.1.1 CSCI Architecture . . . . .	9
3.1.1.1 SPICE 3C1 Architecture . . . . .	9
3.1.1.1.1 Code Organization . . . . .	10
3.1.1.1.2 Code Structure . . . . .	11
3.1.1.1.3 Data Structures . . . . .	12
3.1.1.1.4 Static Function Pointer Structures . . . . .	13
3.1.1.1.5 Static Device Data Structures . . . . .	14
3.1.1.1.6 Dynamic Structures . . . . .	15
3.1.1.1.6.1 Deck Structure . . . . .	16
3.1.1.1.6.2 Circuit Structure . . . . .	16
3.1.1.1.6.3 Plot Structure . . . . .	18
3.1.1.1.7 Entry Points . . . . .	18
3.1.1.1.8 Make files . . . . .	19
3.1.1.2 XSPICE Architecture . . . . .	19
3.1.1.2.1 Code Organization . . . . .	20
3.1.1.2.2 Code Structure . . . . .	23
3.1.1.2.3 Data Structures . . . . .	23
3.1.1.2.3.1 Static Function Pointer Structures . . . . .	23
3.1.1.2.3.2 Static Device Data Structures . . . . .	23
3.1.1.2.3.3 Dynamic Structures . . . . .	23

3.1.1.2.3.4	New Global Structures . . . . .	24
3.1.1.2.4	Entry Points . . . . .	24
3.1.1.2.5	Make Files . . . . .	24
3.1.2	System States and Modes . . . . .	25
3.1.2.1	Program Startup . . . . .	25
3.1.2.2	Circuit Description Read-in . . . . .	28
3.1.2.3	Subcircuit Expansion . . . . .	28
3.1.2.4	Circuit Description Parsing . . . . .	28
3.1.2.5	Command Processing . . . . .	29
3.1.2.6	Simulation Setup . . . . .	29
3.1.2.7	Simulation Execution . . . . .	30
3.1.2.8	Results Output . . . . .	31
3.1.2.9	Program Exit . . . . .	32
3.1.3	Memory and Processing Time Allocation . . . . .	32
3.2	CSCI Design Description . . . . .	33
3.2.1	Model Interface . . . . .	33
3.2.1.1	Example Code Model . . . . .	35
3.2.1.2	Circuit Description Syntax . . . . .	36
3.2.1.2.1	Use of Parentheses . . . . .	37
3.2.1.2.2	Node Names . . . . .	37
3.2.1.2.3	Instance Parameters . . . . .	37
3.2.1.2.4	Vector Ports . . . . .	37
3.2.1.2.5	Connection Types . . . . .	37
3.2.1.2.5.1	Basic Analog Types . . . . .	38
3.2.1.2.5.2	Bidirectional Analog Types . . . . .	38
3.2.1.2.5.3	Digital and User-Defined Types . . . . .	38
3.2.1.2.5.4	Overriding Default Port Types . . . . .	39
3.2.1.2.5.5	Mixing Types . . . . .	39
3.2.1.2.5.6	The NULL Type . . . . .	40
3.2.1.2.6	Model Names . . . . .	40
3.2.1.2.7	Parameter Values . . . . .	40
3.2.1.3	Interface Specification . . . . .	41
3.2.1.3.1	Name Table . . . . .	42
3.2.1.3.2	Port Table . . . . .	43
3.2.1.3.2.1	Port Name . . . . .	43
3.2.1.3.2.2	Direction . . . . .	43
3.2.1.3.2.3	Default Type . . . . .	43
3.2.1.3.2.4	Allowed Types . . . . .	44
3.2.1.3.2.5	Vector . . . . .	44
3.2.1.3.2.6	Vector Bounds . . . . .	44
3.2.1.3.2.7	Null Allowed . . . . .	45
3.2.1.3.2.8	Description . . . . .	45

3.2.1.3.3	Parameter Table	45
3.2.1.3.3.1	Parameter Name	45
3.2.1.3.3.2	Description	45
3.2.1.3.3.3	Data Type	46
3.2.1.3.3.4	Default Value	46
3.2.1.3.3.5	Limits	46
3.2.1.3.3.6	Vector	46
3.2.1.3.3.7	Vector Bounds	47
3.2.1.3.3.8	Null Allowed	47
3.2.1.4	C Function and Accessor Macros	47
3.2.1.4.1	Design Approach	47
3.2.1.4.2	Access Macros	49
3.2.1.4.2.1	Circuit Data Macros	49
3.2.1.4.2.2	Parameter Data	50
3.2.1.4.2.3	Port Data Macros	51
3.2.1.4.2.4	Input Data Macros	51
3.2.1.4.2.5	Output Data Macros	51
3.2.1.4.2.6	Partial Derivative Macros	52
3.2.1.4.2.7	AC Gains	52
3.2.1.4.2.8	Static Variable Macros	52
3.2.1.5	Model Interface Implementation	53
3.2.1.5.1	Device Models	53
3.2.1.5.2	Data Structures and Functions	53
3.2.1.5.3	Syntax Parsing	55
3.2.1.5.4	Matrix Load Equations	56
3.2.1.5.4.1	Newton-Raphson Iteration	56
3.2.1.5.4.2	Modified Nodal Analysis Formulation	56
3.2.1.5.4.3	Matrix Load Functions	57
3.2.1.5.4.4	Functions with Current Source Outputs	58
3.2.1.5.4.5	Functions with Voltage Source Outputs	58
3.2.1.5.5	Analog Simulation States and Integration	59
3.2.2	Code Model Support	61
3.2.2.1	Adding Breakpoints	61
3.2.2.2	Output of Messages	61
3.2.2.3	Allocating Local Storage	62
3.2.2.4	Accessing Local Storage	63
3.2.2.5	Truncation Error Timestep Control	64
3.2.2.6	Integration	64
3.2.2.7	Complex Math Computations	65
3.2.2.8	Automatic Partials	65
3.2.3	Event-Driven Simulation	66
3.2.3.1	Circuit Parsing	67

3.2.3.2	DC Convergence . . . . .	67
3.2.3.3	Transient Analysis Algorithm . . . . .	68
3.2.3.4	AC Analysis . . . . .	69
3.2.3.5	Sources . . . . .	69
3.2.3.6	Display of Results . . . . .	69
3.2.3.7	Digital States and Strengths . . . . .	70
3.2.3.8	Access Macros . . . . .	70
3.2.3.8.1	Input and Output Macros . . . . .	70
3.2.3.8.2	Output Delay Macro . . . . .	71
3.2.3.8.3	Loading Macros . . . . .	71
3.2.3.8.4	Output Message Macro . . . . .	72
3.2.3.9	Support Functions . . . . .	72
3.2.3.10	Use of NULL in Vectors . . . . .	72
3.2.4	Enhancements . . . . .	72
3.2.4.1	Adding new options . . . . .	73
3.2.4.2	Arbitrary Phase Sources . . . . .	74
3.2.4.2.1	Parsing Modifications . . . . .	74
3.2.4.2.2	Device Function Modifications . . . . .	74
3.2.4.3	Supply Ramping . . . . .	75
3.2.4.4	Convergence Limiting . . . . .	76
3.2.4.5	Convergence Debug Reporting . . . . .	77
3.2.4.6	Resistors to Ground . . . . .	78
3.2.5	Internal Code Models . . . . .	79
3.2.6	Internally Defined Nodes . . . . .	79
3.2.7	Interprocess Communication . . . . .	80
3.2.7.1	SPICE 3C1 Program Modifications . . . . .	81
3.2.7.1.1	Modifications to inp_spsource() Subtree . . . . .	82
3.2.7.1.2	Modifications to ft_dotsaves() Subtree . . . . .	82
3.2.7.1.3	Modifications to ft_dorun() Subtree . . . . .	82
3.2.7.1.4	Modifications to ft_cktcoms() Subtree . . . . .	82
3.2.7.2	Sending Results Delimiters . . . . .	82
3.2.7.3	Selecting Data to Return . . . . .	83
3.2.7.4	Returning Results . . . . .	83
3.2.7.5	Modifications to BeginPlot . . . . .	84
3.2.7.5.1	Voltages and V Source Currents . . . . .	84
3.2.7.5.2	Device Currents . . . . .	85
3.2.7.5.3	Parameter names for device types . . . . .	85
4	Detailed Design . . . . .	87
4.1	Model Interface . . . . .	87
4.1.1	Function MIF_INP2A . . . . .	89
4.1.1.1	Static Function MIFinit_inst . . . . .	92

4.1.1.2	Static Function MIFget_port_type . . . . .	93
4.1.1.3	Static Function MIFget_port . . . . .	94
4.1.2	Function MIFgetMod . . . . .	97
4.1.3	Function MIFgetValue . . . . .	99
4.1.4	Function MIFgettok . . . . .	101
4.1.5	Function MIFget_token . . . . .	102
4.1.6	Function MIFget_ctrl_src_type . . . . .	104
4.1.7	Function MIFmParam . . . . .	105
4.1.8	Function MIFmAsk . . . . .	107
4.1.9	Function MIFask . . . . .	108
4.1.10	Function MIFsetup . . . . .	109
4.1.11	Function MIFload . . . . .	112
4.1.11.1	Static Function MIFauto_partial . . . . .	117
4.1.12	Function MIFconvTest . . . . .	120
4.1.13	Function MIFtrunc . . . . .	122
4.1.13.1	Static Function MIFTerr . . . . .	123
4.1.14	Function MIFmDelete . . . . .	124
4.1.15	Function MIFdelete . . . . .	125
4.1.16	Function MIFdestroy . . . . .	126
4.2	Code Model Support . . . . .	127
4.2.1	Function cm_analog_alloc . . . . .	128
4.2.2	Function cm_analog_get_ptr . . . . .	130
4.2.3	Function cm_analog_integrate . . . . .	131
4.2.3.1	Static Function cm_static_integrate . . . . .	133
4.2.4	Function cm_analog_converge . . . . .	134
4.2.5	Function cm_analog_set_temp_bkpt . . . . .	136
4.2.6	Function cm_analog_set_perm_bkpt . . . . .	138
4.2.7	Function cm_ramp_factor . . . . .	140
4.2.8	Function cm_analog_not_converged . . . . .	142
4.2.9	Function cm_analog_auto_partial . . . . .	143
4.2.10	Function cm_message_get_errmsg . . . . .	144
4.2.11	Function cm_message_send . . . . .	145
4.2.12	Function cm_event_alloc . . . . .	146
4.2.13	Function cm_event_get_ptr . . . . .	148
4.2.14	Function cm_event_queue . . . . .	150
4.2.15	Function cm_netlist_get_c . . . . .	151
4.2.16	Function cm_netlist_get_l . . . . .	153
4.2.17	Function cm_smooth_corner . . . . .	155
4.2.18	Function cm_smooth_discontinuity . . . . .	156
4.2.19	Function cm_climit_fcn . . . . .	157
4.2.20	Function cm_smooth_pwl . . . . .	158
4.2.21	Function cm_complex_set . . . . .	159

4.2.22 Function cm_complex_add . . . . .	160
4.2.23 Function cm_complex_subtract . . . . .	161
4.2.24 Function cm_complex_multiply . . . . .	162
4.2.25 Function cm_complex_divide . . . . .	163
<b>4.3 Event-Driven Simulation . . . . .</b>	<b>164</b>
4.3.1 Function EVTaccept . . . . .	166
4.3.2 Function EVTbackup . . . . .	169
4.3.2.1 Static Function EVTbackup_node_data . . . . .	171
4.3.2.2 Static Function EVTbackup_state_data . . . . .	173
4.3.2.3 Static Function EVTbackup_msg_data . . . . .	174
4.3.2.4 Static Function EVTbackup_inst_queue . . . . .	175
4.3.2.5 Static Function EVTbackup_output_queue . . . . .	177
4.3.3 Function EVTcall_hybrids . . . . .	179
4.3.4 Function EVTdequeue . . . . .	180
4.3.4.1 Static Function EVTdequeue_output . . . . .	181
4.3.4.2 Static Function EVTdequeue_inst . . . . .	183
4.3.4.3 Static Function EVTprocess_output . . . . .	185
4.3.5 Function EVTinit . . . . .	187
4.3.5.1 Static Function EVTcheck_nodes . . . . .	188
4.3.5.2 Static Function EVTcount_hybrids . . . . .	189
4.3.5.3 Static Function EVTinit_info . . . . .	190
4.3.5.4 Static Function EVTinit_queue . . . . .	192
4.3.5.5 Static Function EVTinit_limits . . . . .	193
4.3.6 Function EVTiter . . . . .	194
4.3.7 Function EVTload . . . . .	197
4.3.7.1 Static Function EVTcreate_state . . . . .	200
4.3.7.2 Static Function EVTadd_msg . . . . .	202
4.3.7.3 Static Function EVTcreate_output_event . . . . .	204
4.3.7.4 Static Function EVTprocess_output . . . . .	206
4.3.8 Function EVTnext_time . . . . .	208
4.3.9 Function EVTnode_copy . . . . .	209
4.3.10 Function EVTop . . . . .	211
4.3.11 Function EVTop_save . . . . .	213
4.3.11.1 Static Function EVTnode_compare . . . . .	215
4.3.12 Function EVTprint . . . . .	216
4.3.12.1 Static Function get_index . . . . .	218
4.3.12.2 Static Function print_data . . . . .	219
4.3.13 Function EVTdump . . . . .	220
4.3.13.1 Static Function EVTsend_line . . . . .	223
4.3.14 Function EVTqueue_output . . . . .	224
4.3.15 Function EVTqueue_inst . . . . .	226
4.3.16 Function EVTsetup . . . . .	228

4.3.16.1 Static Function EVTsetup_queues . . . . .	230
4.3.16.2 Static Function EVTsetup_data . . . . .	231
4.3.16.3 Static Function EVTsetup_jobs . . . . .	233
4.3.16.4 Static Function EVTsetup_load_ptrs . . . . .	234
4.3.17 Function EVTtermInsert . . . . .	235
4.3.17.1 Static Function EVTinst_insert . . . . .	236
4.3.17.2 Static Function EVTnode_insert . . . . .	237
4.3.17.3 Static Function EVTport_insert . . . . .	239
4.3.17.4 Static Function EVTooutput_insert . . . . .	240
4.4 Enhancements . . . . .	241
4.4.1 Function ENHreport_conv_prob . . . . .	242
4.4.2 Function ENHtranslate_poly . . . . .	243
4.4.2.1 Static Function needs_translating . . . . .	244
4.4.2.2 Static Function count_tokens . . . . .	245
4.4.2.3 Static Function translate . . . . .	246
4.4.2.4 Static Function get_poly_dimension . . . . .	247
4.5 Internal Code Models . . . . .	248
4.5.1 Function icm_poly . . . . .	249
4.5.1.1 Static Function evterm . . . . .	251
4.5.1.2 Static Function nxtpwr . . . . .	252
4.6 Internally Defined Nodes . . . . .	254
4.6.1 Info Structure idn_digital_info . . . . .	255
4.6.1.1 Function idn_digital_create . . . . .	256
4.6.1.2 Function idn_digital_dismantle . . . . .	257
4.6.1.3 Function idn_digital_initialize . . . . .	258
4.6.1.4 Function idn_digital_invert . . . . .	259
4.6.1.5 Function idn_digital_copy . . . . .	260
4.6.1.6 Function idn_digital_resolve . . . . .	261
4.6.1.7 Function idn_digital_compare . . . . .	263
4.6.1.8 Function idn_digital_plot_val . . . . .	264
4.6.1.9 Function idn_digital_print_val . . . . .	265
4.6.1.10 Function idn_digital_ipc_val . . . . .	267
4.7 Interprocess Communication . . . . .	268
4.7.1 Function ipc_initialize_server . . . . .	269
4.7.2 Function ipc_terminate_server . . . . .	270
4.7.3 Function ipc_get_line . . . . .	271
4.7.4 Function ipc_send_line . . . . .	273
4.7.5 Function ipc_send_line_binary . . . . .	274
4.7.6 Function ipc_send_data_prefix . . . . .	275
4.7.7 Function ipc_send_data_suffix . . . . .	276
4.7.8 Function ipc_send_dcop_prefix . . . . .	277
4.7.9 Function ipc_send_dcop_suffix . . . . .	278

4.7.10 Function ipc_send_evtdict_prefix . . . . .	279
4.7.11 Function ipc_send_evtdict_suffix . . . . .	280
4.7.12 Function ipc_send_evtdata_prefix . . . . .	281
4.7.13 Function ipc_send_evtdata_suffix . . . . .	282
4.7.14 Function ipc_send_errchk . . . . .	283
4.7.15 Function ipc_send_end . . . . .	284
4.7.16 Function ipc_send_double . . . . .	285
4.7.17 Function ipc_send_complex . . . . .	286
4.7.18 Function ipc_send_event . . . . .	287
4.7.19 Function ipc_flush . . . . .	289
4.7.20 Function ipc_handle_stop . . . . .	290
4.7.21 Function ipc_handle_returni . . . . .	291
4.7.22 Function ipc_handle_mintime . . . . .	292
4.7.23 Function ipc_handle_vtrans . . . . .	293
4.7.24 Function ipc_send_std_files . . . . .	294
4.7.24.1 Static Function ipc_send_stdout . . . . .	295
4.7.24.2 Static Function ipc_send_stderr . . . . .	296
4.7.25 Function ipc_screen_name . . . . .	297
4.7.26 Function ipc_get_devices . . . . .	299
4.7.27 Function ipc_free_devices . . . . .	301
4.7.28 Function ipc_check_pause_stop . . . . .	302
4.7.29 BSD UNIX Sockets Transport Layer . . . . .	303
4.7.29.1 Function ipc_transport_initialize_server . . . . .	304
4.7.29.2 Function ipc_transport_get_line . . . . .	306
4.7.29.3 Function ipc_transport_send_line . . . . .	308
4.7.29.4 Function ipc_transport_terminate_server . . . . .	309
4.7.30 HP/Apollo Mailbox Transport Layer . . . . .	310
4.7.30.1 Function ipc_transport_initialize_server . . . . .	311
4.7.30.2 Function ipc_transport_get_line . . . . .	313
4.7.30.3 Function ipc_transport_send_line . . . . .	315
4.7.30.4 Function ipc_transport_terminate_server . . . . .	316
<b>5 CSCI Data</b>	<b>317</b>
5.1 Interface to Code Model Libraries . . . . .	317
5.1.1 Interface Specification Data . . . . .	318
5.1.2 C Function Data . . . . .	321
5.1.3 Linker Include Data . . . . .	324
5.2 Interface to User Defined Node Libraries . . . . .	324
5.2.1 C Function Data . . . . .	324
5.2.2 Linker Include Data . . . . .	326
5.3 Model and Instance Data Structures . . . . .	326
5.4 Event-Driven Simulation Data . . . . .	328

5.4.1	Evt_Count_t . . . . .	328
5.4.2	Evt_Info_t . . . . .	329
5.4.3	Evt_Queue_t . . . . .	330
5.4.4	Evt_Data_t . . . . .	334
5.4.4.1	Evt_Node_Data_t . . . . .	335
5.4.4.2	Evt_State_Data_t . . . . .	336
5.4.4.3	Evt_Msg_Data_t . . . . .	337
5.4.4.4	Evt_Statistic_Data_t . . . . .	337
5.4.5	Evt_Limit_t . . . . .	338
5.4.6	Evt_Job_t . . . . .	338
5.4.7	Evt_Option_t . . . . .	338
5.5	General Enhancements Data . . . . .	338
5.6	Global Data Structures . . . . .	340
5.6.1	User-Defined Node Data . . . . .	340
5.6.2	Model Interface Data . . . . .	340
5.6.3	Interprocess Communication Data . . . . .	341
<b>6</b>	<b>CSCI Data Files</b>	<b>343</b>
6.1	Data File to CSC/CSU Cross Reference . . . . .	343
6.2	Standard Streams . . . . .	343
6.3	SPICE Rawfile . . . . .	344
6.4	Interprocess Communication Channel File . . . . .	344
6.5	Batch Results File . . . . .	344
<b>7</b>	<b>Requirements Traceability</b>	<b>345</b>
<b>8</b>	<b>Notes</b>	<b>347</b>
8.1	Glossary . . . . .	347
8.2	Acronyms . . . . .	349
8.3	Project Unique Identifiers . . . . .	350
<b>APPENDICES</b>		
<b>A</b>	<b>SPICE 3C1 Annotated Call Tree</b>	<b>355</b>
<b>B</b>	<b>SPICE 3C1 Structure Typedefs</b>	<b>359</b>
B.1	Nutmeg Type Definitions . . . . .	359
B.2	SPICE Specific Type Definitions . . . . .	361
B.3	Model/Instance Type Definitions . . . . .	362
B.4	Input Parser Type Definitions . . . . .	364

## CONTENTS

## XSPICE Simulator Software Design Document

<b>C SPICE 3C1 Structure Declarations</b>	<b>365</b>
<b>C.1 Nutmeg Structure Declarations</b>	<b>365</b>
<b>C.2 SPICE Structure Declarations</b>	<b>366</b>

# List of Figures

1.1	XSPICE External Interfaces. . . . .	2
3.1	XSPICE Top-Level Diagram. . . . .	8
3.2	Internal Circuit Structure Linked Lists. . . . .	17
5.1	Event Evaluation Sequence. . . . .	331
5.2	Event Queue Structure. . . . .	334

# List of Tables

3.1	CKTcircuit Structure Major Elements. . . . .	17
3.2	XSPICE States and Modes. . . . .	26
6.1	XSPICE Files. . . . .	343
7.1	XSPICE Requirements Cross-Reference. . . . .	346

# 1 Scope

## 1.1 Identification

This Software Design Document describes the design of the Simulator Computer Software Configuration Item (CSCI) of version 2 of the 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 XSPICE Simulator is the tool that performs mathematical simulation of a circuit specified by the user. It takes input in the form of commands and circuit descriptions, and produces output data that 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 board-level and system-level simulation capability.

Interfaces between the Simulator and other CSCIs of the ATESSE system are illustrated in Figure 1.1.

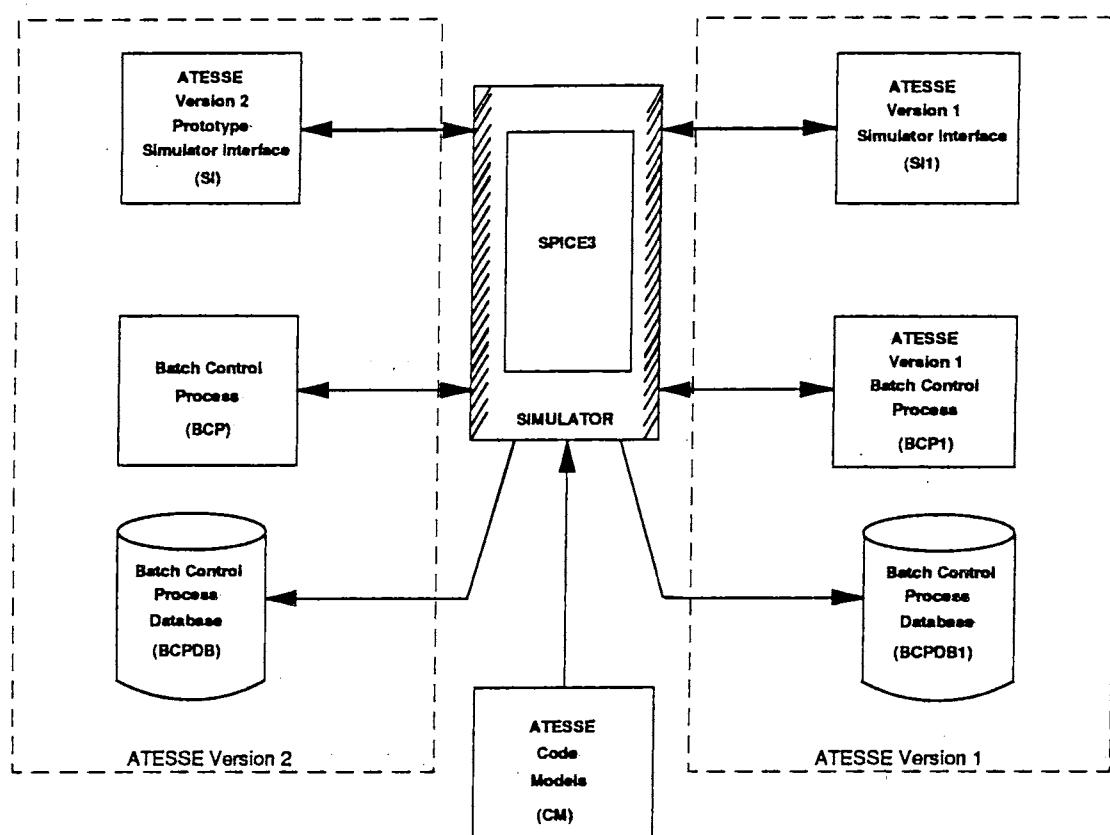


Figure 1.1 XSPICE External Interfaces.

### **1.3 Document Overview**

This document describes the design of the XSPICE Simulator CSCI. Chapter 2 is a bibliography of associated documents. Chapter 3 provides a top-level description of the design of each of the components. Chapter 4 provides a detailed description of each of the major Computer Software Components (CSCs) and Computer Software Units (CSUs) that make up the simulator. Chapter 5 describes important data items. Chapter 6 describes important data files. Chapter 7 provides traceability of requirements allocated down to the CSU level of each CSC back to the requirements. Chapter 8 provides additional notes including a glossary and list of acronyms. Finally, a set of appendices provides additional information on the analysis of the SPICE3C1 simulator.

### **1.4 Acknowledgement**

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 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, November, 1991.
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, 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 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.
11. 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.
12. Software 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.
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

# Preliminary Design

This chapter describes the top-level design of the simulator. It includes two major subsections. The first (CSCI Overview) provides a high-level look at the architecture and major control flow of the XSPICE simulator. The second (CSCI Design Description) provides a more in-depth look at the design, focusing on the major XSPICE packages (CSCIs), and describing their features and design.

## 3.1 CSCI Overview

The XSPICE Simulator is a modified and enhanced version of the SPICE 3C1 simulator developed at the University of California at Berkeley (UCB). A top-level diagram of the simulator and the associated Code Model Subsystem is shown in Figure 3.1. The major modifications made to the SPICE 3C1 simulator core include:

- Addition of a generalized code modeling facility that allows the set of devices known to the simulator to be easily extended. This facility is implemented in conjunction with the Code Model Toolset described in the Code Model Subsystem CSCI Software Design Document.
- Addition of an embedded event-driven simulation capability for efficient simulation of digital subcircuits and higher-level subsystems.
- Addition of interprocess communication (IPC) functions that allow the simulator to accept circuit descriptions from the ATESSE versions 1 and 2 Simulator Interface processes, and to return results to these processes for graphing. The IPC functions are also used to communicate with the ATESSE versions 1 and 2 Batch Control processes.

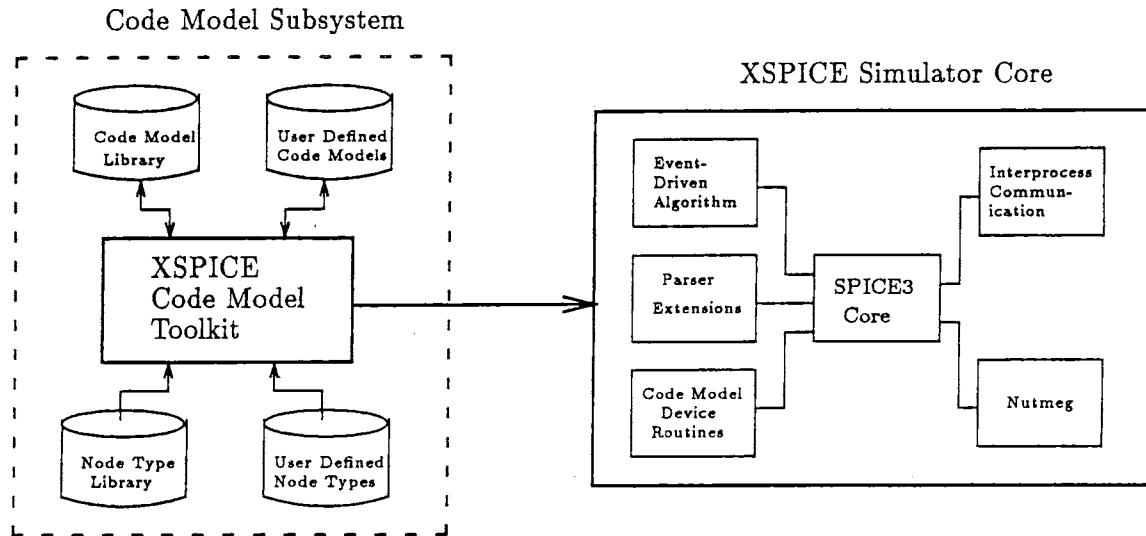


Figure 3.1 XSPICE Top-Level Diagram.

The process of incorporating new models into the simulator is facilitated by the Code Model Subsystem. This subsystem (described in a separate document) contains three major components:

**The Code Model Toolset**

A set of tools for creating and compiling models and user-defined node types and linking them with the simulator.

**The Code Model Library**

A set of predefined code models covering many important analog and digital functions.

**The User-Defined Node Library**

A set of predefined node types useful in simulating sampled-data systems.

The event-driven simulation capability provides for efficient simulation of digital subcircuits within a mixed-mode (analog/digital) design. A 12-state logic simulator is embedded within the XSPICE simulator. It is designed to take independent timesteps and to represent event-driven nodes without introducing entries into the analog simulation matrix. Special "Node Bridge" models perform the mapping between analog signals and digital states according to user-supplied parameters. The event-driven algorithm is generalized by implementing it independently from the data used to represent digital data, providing the ability to simulate with arbitrary user-defined data structures.

The IPC facility is designed to allow the simulator to function as an integral part of the ATESSE system. Depending on which version of the ATESSE system the simulator is configured to work with, the IPC uses either HP Apollo Aegis mailboxes (ATESSE version 1), or BSD UNIX sockets (ATESSE version 2).

Additional enhancements and modifications are incorporated as defined in the Software Requirements Specification for the Simulator of the Automatic Test Equipment Software Support Environment (ATESSE). These modifications are designed to target the simulator for use in board-level and system-level simulation, and provide compatibility with the ATESSE version 1 simulator.

Two different versions of the XSPICE executable can be built ("atesse\_xspice" and "xspice"). The first includes the IPC facility and is used whenever the simulator is used in the ATESSE system. The second version includes the standard Berkeley SPICE 3C1 "Nutmeg" command-line user interface and can be used as a standalone simulator, independent of the ATESSE system. The selection of which executable is built is made through arguments to the "make" command in a simulator directory as described in the Interface Design Document for the XSPICE Simulator of the Automatic Test Equipment Software Support Environment (ATESSE).

### 3.1.1 CSCI Architecture

The simulator is a modified version of SPICE 3C1, and as such, maintains the major architectural features of that simulator. The description of the simulator architecture below is divided into two major sections. The first section describes the principal components of the unmodified SPICE 3C1 simulator. Additional details of this architecture can be found in the references. The second section describes the architecture of the modifications and enhancements made to SPICE 3C1 to produce XSPICE.

#### 3.1.1.1 SPICE 3C1 Architecture

SPICE 3C1 is a rewrite of the FORTRAN-based SPICE 2G6 simulator developed at the University of California at Berkeley during the 1970's. The SPICE 3C1 version is written in C and includes an extensive new user interface called "Nutmeg" that provides for interactive selection of analysis and interactive graphing of results.

The SPICE 3C1 simulator is implemented in two major parts:

Nutmeg The command-line processor and interactive graphing routines.

SPICE The simulation routines.

Nutmeg is a command-line based user interface that provides for controlling simulations and for interactive graphing of results. Graphs are produced in resizable windows when run with the X Window System, and an expression capability is provided to allow addition, subtraction, and other operations on waveforms.

The SPICE routines provide an analog simulation capability largely upward compatible with SPICE 2G6. These routines read a SPICE deck, simulate the circuit, and write output voltage and current vectors (also called "plots") to a file or to memory.

On IBM PC platforms running under DOS, the Nutmeg and SPICE components are implemented as separate executables because of the limited memory available. On a UNIX or other workstation platform where sufficient memory is available, the SPICE routines are combined with Nutmeg and the resulting single executable is collectively called "spice". A "batch" mode simulator executable called "bspice" can also be built on UNIX platforms that excludes most of the Nutmeg interface to conserve memory.

### 3.1.1.1 Code Organization

The SPICE 3C1 code is organized in several hierarchical directories. These directories and their contents are summarized below.

spice3c1	- The root directory for UCB SPICE3c1
examples	- Example input decks
lib	- Library for mfbcap, news, and help files
helpdir	- Help files for Nutmeg and SPICE
scripts	- Miscellaneous script files
man	- Nroff source for man pages
man1	- Nutmeg, sconvert, SPICE man pages
man3	- Mfb man page
man5	- Mfbcap man page
spice3	- The root for the SPICE/Nutmeg source files
include	- C header files used by all CSCs
CKT	- 'Circuit' CSC
CP	- 'Command Processor' CSC
DEV	- 'Device' CSC root directory
ASRC	- 'Arbitrary Source'
BJT	- 'Bipolar Junction Transistor'
BSIM	- 'BSIM MOSFET'
CAP	- 'Capacitor'
CCCS	- 'Current Controlled Current Source'
CCVS	- 'Current Controlled Voltage Source'
CSW	- 'Current Controlled Switch'
DIO	- 'Diode'
IND	- 'Inductor'
ISRC	- 'Current Source'
JFET	- 'Junction FET'

```
MES      -  'MESFET'  
MOS1    -  'Level 1 MOSFET'  
MOS2    -  'Level 2 MOSFET'  
MOS3    -  'Level 3 MOSFET'  
RES     -  'Resistor'  
SW      -  'Voltage Controlled Switch'  
TRA     -  'Transmission Line'  
URC     -  'Uniform RC Transmission Line'  
VCCS    -  'Voltage Controlled Current Source'  
VCVS    -  'Voltage Controlled Voltage Source'  
VSRC    -  'Voltage Source'  
FTE     -  'Front End' CSC  
HLP     -  'Help' CSC  
INP     -  'Input' CSC  
MFB     -  'Model Frame Buffer' graphics display CSC  
MFB-PC   -  Same, but for IBM PC platform  
NI      -  'Numerical Iteration' CSC  
SMP     -  'Sparse Matrix Package' CSC
```

The source code for the simulator is contained under the "spice3" subdirectory. Each of the subdirectories under spice3 (with the exception of "include") may be considered a separate CSC.

### 3.1.1.1.2 Code Structure

The major structure of the SPICE3 code is illustrated by the following simplified "call tree".

Nutmeg:

```
main()  
  
SIMinit() -- Nutmeg and spice exchange function pointers  
  
inp_spsource() -- read SPICE deck and parse it  
    inp_subcktexpand() -- expand subcircuits  
    inp_dodeck() -- parse expanded deck  
    if_inpdeck() -- read SPICE deck  
        INPpas1() -- collect .model cards  
        INPpas2() -- parse element cards  
        INP2R() -- parse resistor element card  
  
cp_evloop() -- command processor evaluation loop  
    doblock() -- process a statement or block  
    docommand() -- process a single statement (e.g. 'run')  
    (*(cp_coms[i].cofunc)) -- A call to a command within  
                            the Nutmeg CSC (e.g.
```

```
        com_run() )
com_run() -- call dosim() to run simulation
dosim() -- open a results file
    if_run() -- prepare simulation data structs
    (*(ft_sim->doAnalyses)) -- A call to CKTdoJob()
                                in the SPICE CSC
```

SPICE:

```
CKTdoJob() -- run a simulation (called by Nutmeg)

CKTsetup() -- call each device's matrix setup function
CKTtemp() -- call each device's temperature setup function
CKTic()   -- call each device's initial condition setup
            function

DCop()   -- setup for a DC operating point analysis
CKTop()  -- run a DC operating point analysis
NIiter() -- iterate to a solution
CKTload() -- call each device's load function to
            build the matrix
SMPsolve() -- solve the matrix

CKTdump() -- output results to Nutmeg
    (*(SPfrontEnd->OUTpData)) -- A call to OUTpData() in
                                Nutmeg
```

### 3.1.1.3 Data Structures

There are a large number of type definitions and data structures in the SPICE 3C1 program. This section provides an overview of the most important ones. Additional typedefs and structures are discussed in the references.

The major data structures in SPICE can be divided into three categories:

- Static Function Pointer Structures
- Static Device Data Structures
- Dynamic Structures for SPICE Deck, Circuit, and Results Data

The static function pointer structures are essentially arrays of function pointers used to make function calls between SPICE and Nutmeg and within Nutmeg, as illustrated in the call-tree shown in the previous section.

The static device data structures hold information on each of the devices or models that the simulator knows about. This information is used in parsing the input deck, in setting up the dynamic structures for the circuit, and in calling the device model code during a simulation.

The dynamic structures are created during and after parsing, and hold the input deck, information about the circuit models and instances, the current state of the simulation (including the solution matrix), and the results saved at each analysis point.

These major data items are covered in more detail in the following subsections.

#### 3.1.1.1.4 Static Function Pointer Structures

The SPICE 3C1 simulator makes significant use of the C programming language's function pointer capability. Three important data structures are used throughout the program to make calls between SPICE and Nutmeg, as well as within Nutmeg for executing user commands. These three global variable structures are identified below, together with their respective types in standard C syntax.

```
IFfrontEnd    *SPfrontEnd;
IFsimulator   *ft_sim;
struct comm    cp_coms[];
```

SPfrontEnd is a pointer to a structure of type IFfrontEnd which is defined in include/IFsim.h. The structure holds pointers to functions in Nutmeg that the simulator calls. An example is shown in the call tree shown above, where the simulator calls the Nutmeg function "OUTpData()" using the statement:

```
(*(SPfrontEnd->OUTpData))();
```

In this case, the OUTpData member of the SPfrontEnd structure has the same name as the function OUTpData() to which it points. This is not always the case. In general, the name of the structure member may be different from the name of the function to which it points. An example of this case is shown in the call tree where the SPICE function CKTdoJob() is called from Nutmeg with the statement:

```
((ft_sim->doAnalysis))();
```

To determine the mapping from structure member name to actual function name, it is necessary to look at both the typedef for the structure and at the statement in the code that initializes the structure elements. In the case of the structure SPfrontEnd, the typedef is in include/IFsim.h, and the initialization is done in FTE/nutmeg.c, which statically allocates

an IFfrontEnd structure called “Nutmeginfo”. SPfrontEnd is set to point to Nutmeginfo through the call to SIMinit() in main().

The static structure pointed to by ft\_sim is the complement of SPfrontEnd. It contains function pointers used by Nutmeg to access functions that are part of the SPICE CSC. The IFsimulator typdef for this structure is found in include/IFsim.h, and the initialization is done in CKT/SPIinit.c which statically allocates an IFsimulator structure called “SIMinfo”. The global variable ft\_sim is set through the same SIMinit() call that sets SPfrontEnd.

The third important function pointer structure is cp\_coms[]. As illustrated in the call tree, cp\_coms is used by Nutmeg function “docommand()” which processes a user-typed command such as “run” and calls the appropriate C function. The structure cp\_coms contains additional data such as the character strings of known commands. Function docommand() scans the elements of cp\_coms[] to find a matching command string and then executes the command using the associated function pointer. In this case, all functions are referenced through the structure member “cofunc”, and the index into the cp\_coms[] array determines which command is executed:

```
(*(cp_coms[i].cofunc))();
```

The set of command strings and associated C function names can be found by examining FTE/cmdtab.c where cp\_coms[] is statically declared.

### 3.1.1.5 Static Device Data Structures

The SPICE 3C1 simulator includes a set of 22 built-in devices including resistors, capacitors, bipolar junction transistors, etc. Each of these devices has a defined syntax which the simulator must know in order to parse and error check the input deck, as well as to set up internal model and instance structures and to call appropriate analysis functions during a simulation. All of this device specific information is kept in SPICEdev type structures, one for each device type:

```
SPICEdev XXXinfo;
```

where XXX is a device prefix such as RES, CAP, BJT, etc.

The complete set of devices known to the simulator is accessed through a global array variable of such structures:

```
SPICEdev *DEVices[];
```

statically allocated in CKT/SPIinit.c. The individual XXXinfo structures are defined in DEV/XXX/XXX.c (e.g. DEV/RES/RES.c). Each XXXinfo structure includes three types of data:

- A substructure of type IFdevice called DEVpublic which holds information used by the SPICE deck parsing routines.
- A collection of pointers to device specific functions used in parsing and setting up and running the simulation.
- Two integers that define the size of dynamically allocated structures used to store data for models and instances of that device type referenced in the SPICE deck.

It is important to note here that there is not a one-to-one correspondence between the “elements” distinguished by SPICE on the basis of the first character of a SPICE-deck element card (e.g. r=resistor, c=capacitor, q=bipolar transistor, etc.) and the “devices” distinguished by different XXXinfo structures. There are more XXXinfo structures than element card prefixes, since some devices (such as MOSFETs) can have different model types on their .model card. This feature is exploited in the XSPICE simulator to allow all code models to use a single “A” type element card (one of the few unused element-card prefixes at the 3C1 level of SPICE). The individual code model referenced by that element card is determined by looking at an associated .model card.

### 3.1.1.1.6 Dynamic Structures

There are three major dynamic data structures in SPICE 3C1:

- A “deck” linked-list structure holding the text of the SPICE input deck.
- A “circuit” structure that holds the internal representation of the circuit including the models and instances, the nodes and voltage source branch dependent variables, the solution matrix, and the state of the simulation.
- A “plot” structure that holds computed results data during and after a simulation to allow interactive graphing of results.

All of these structures are built and dismantled using “malloc” type storage allocation functions. In general, they are passed between functions in argument lists and therefore may have different names in different functions. The “circuit” structure is the most involved of the three and includes several important substructures including linked-lists of models and instances that make up the circuit. These important substructures are discussed further in the subsections below.

### 3.1.1.6.1 Deck Structure

The deck structure is a linked list of type “line” created by Nutmeg as the SPICE deck is read. Type “line” is defined in include/FTEinp.h. As noted in that header file, an equivalent type is “card” defined in include/INPdefs.h. After the SPICE deck has been read, the linked list is manipulated to unwrap continuation lines, extract .options cards, and expand subcircuits. Error messages produced during parsing are also recorded in the deck structure.

### 3.1.1.6.2 Circuit Structure

During deck parsing, the textual description of the circuit with subcircuits expanded is used to create the internal “circuit” structure used during simulation. Since parsing is a grey area somewhere between the Nutmeg and SPICE CSCs, this structure is visible to both. However, Nutmeg treats the circuit structure as a “generic” pointer type ((void \*) or (char \*)), while SPICE sees it as type CKTcircuit defined in include/CKTdefs.h. In Nutmeg, the structure is generally attached to global pointer variable “ft\_curckt”, while in SPICE it is generally passed between functions in their argument lists with name “ckt”. This dual view of variable types is prevalent throughout the program.

The CKTcircuit structure includes a relatively large and diverse set of members. Some of the more important elements are shown in Table 3.1.

One of the most important members in the circuit structure is the array of GENmodel structure pointers called “CKThead”. This member holds the internal representation of the circuit during simulation.

Each element of the array points to the (possibly NULL) head of a linked list of model structures for a particular device type. In turn, each model structure in the list contains a GENinstance structure pointer which points to the (possibly NULL) head of a linked list of instance structures for instances of that model. The arrangement is illustrated in Figure 3.2.

As the input deck is parsed, the device type is determined from either the first character of the element card or from a .model card if appropriate. The device type is then used as the index “i” into the CKThead[] array. If the device has no model card associated with it (e.g. a simple resistor), then a default model is used and an instance structure is added below the default model. If the device has an associated .model card, the user supplied named set of parameters defines the model structure and the instance is added below that model.

During a simulation, the simulator operates on the circuit by looping through the CKThead[] array of devices, traversing the linked lists of models, and for each model, traversing the linked list of instances. A number of “driver” functions such as CKTsetup(), CKTtemp(), CKTload(), etc. are designed to loop through the CKThead[] array calling the appropriate

Type	Member	Description
GENmodel	*CKThead[]	The head of the linked lists of model structures for each device type.
STATistics	*CKTstat	A structure holding runtime statistics.
double	*CKTstates[8]	A collection of instance states from the current and previous timepoints used in numerical integration.
SMPmatrix	*CKTmatrix	The internal representation of the matrix used in solving the circuit.
double	CKTrhsOld	The solution vector.
CKTnode	*CKTnodes	The compact list of nodes and branches in the circuit.
double	CKTbreaks	The dynamically managed set of transient analysis timestep breakpoints.

Table 3.1 CKTcircuit Structure Major Elements.

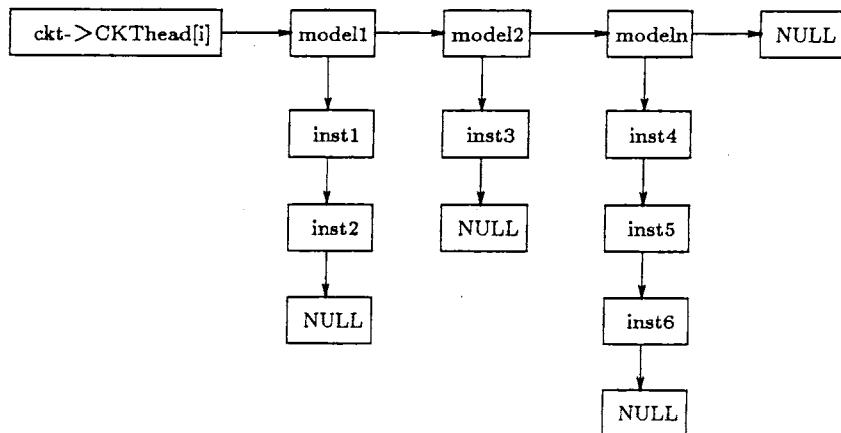


Figure 3.2 Internal Circuit Structure Linked Lists.

device specific functions through the pointers in the SPICEdev XXXinfo structure described above. The device specific functions then traverse the models and instances of that type.

The typedefs "GENmodel" and "GENinstance" used in conjunction with the dynamically created model and instance linked lists are defined in include/GENdefs.h. These typedefs are "prefixes" of larger device-specific XXXmodel and XXXinstance structures defined for each device in include/XXXdefs.h. (Here, as before, XXX is a device prefix such as RES, CAP, etc.) The XXXmodel and XXXinstance structures hold all the data for specific models and instances of their type during a simulation. Only the device specific functions can access the extended data structure information. Nutmeg accesses only the structure "prefix" elements available through the GENmodel and GENinstance types.

### 3.1.1.6.3 Plot Structure

As a simulation runs, results are computed in the CKTrhsOld solution vector at each iteration and are transferred to the "plot" structure following each successful analysis point (if the simulator is running interactively with the Nutmeg interface - otherwise the data is written directly to a file). The (anonymous typed) plot structure is named "runDesc" and is part of the OUTinterface.c routines, which are a part of Nutmeg residing in directory FTE. A companion structure named "dataDesc" is used to keep track of what data to save during an analysis. The dataDesc structure is used regardless of the destination of the output data.

### 3.1.1.7 Entry Points

There are several "main()" functions in the SPICE 3C1 source files. All of these main programs reside in the FTE directory. Files containing main() functions are:

nutmeg.c	SPICE + Nutmeg
bspice.c	Batch mode SPICE3
cspice.c	Modified version of bspice.c
sconvert.c	Program to convert old rawfile format to new format
spiced.c	SPICE daemon
test.c	A test program

The first two (nutmeg.c and bspice.c) are the principal interactive and batch SPICE 3C1 main programs respectively. The XSPICE interactive (standalone) and batch (IPC driven) simulators are implemented using the main() functions found in these files.

### 3.1.1.8 Make files

The SPICE executables are compiled on UNIX platforms using the UNIX "make" utility. Since there are multiple directories, there is a hierarchy of Makefile's. The top-level Makefile is in the SPICE 3C1 directory and is designed to change directory (cd) into the spice3 subdirectory and execute a "make" there. The Makefile in the spice3 directory then cd's into each of its subdirectories and executes a make in them, and so-on.

After all the .c files in a particular directory are compiled, they are archived into a library with the name <subdir>.a in the spice3 subdirectory. The last subdirectory to be made is FTE. The Makefile in this subdirectory links .o files and .a archives to create the appropriate executable within the FTE directory.

### 3.1.1.2 XSPICE Architecture

The XSPICE simulator is based on SPICE 3C1 but is heavily modified and extended to provide support for board-level and system-level modeling and simulation. In general, modifications made directly to the core SPICE 3C1 simulator are minimized; the implementation of board-level and system-level support consists primarily of code additions.

The additions provided in the XSPICE simulator consist of seven new CSCs and associated subdirectories added at the level of the 'include', INP, FTE, CKT, and related subdirectories. In addition, associated new include files are added to the existing include subdirectory. With the exception of the include subdirectory, no new files are added within existing sub-directories.

Since the set of code models linked into the XSPICE simulator are determined by the user, code models are treated in a separate CSCI, (the Code Model Subsystem). The architecture and directory structure for the Code Model Subsystem are detailed in the Software Design Document for the Code Model Subsystem of the Automatic Test Equipment Software Support Environment (ATESSE).

The new CSCs within the XSPICE Simulator are:

SIM-MIF	Model Interface
SIM-CM	Code Model support functions
SIM-EVT	Event-Driven simulation
SIM-ENH	Enhancements
SIM-ICM	Internal Code Models
SIM-IDN	Internally Defined Nodes
SIM-IPC	Interprocess Communication

The Model Interface CSC (SIM-MIF) implements a new "A" type SPICE deck element card that is used by all predefined and user-written code models. This CSC includes functions

analogous to those used by standard SPICE 3C1 device CSCs (e.g. RES, CAP, BJT, etc.) plus SPICE-deck parsing code analogous to that found in the SPICE 3C1 INP CSC.

The Code Model Support CSC (SIM-CM) provides support functions callable from code models including those for state allocation, integration, error reporting, etc.

The Event-Driven Simulation CSC (SIM-EVT) implements an embedded, event-driven simulation capability. The built-in 12-state logic simulation capability is the primary event-driven data type. However, the SIM-EVT CSC is implemented separately from the 12-state logic data type and provides the foundation for the implementation of “user-defined nodes”.

The Enhancements CSC (SIM-ENH) provides various enhancements to the base SPICE 3C1 simulator to make it easier to use, more compatible with the SPICE 2G6 simulator, and better suited for board-level and system-level simulation. Examples of the support provided are new phase parameters on time-varying voltage and current sources, and a new “ramptime” option that simulates the application of power to a board-level circuit.

The Internal Code Models CSC (SIM-ICM) includes special code models that are always included in the simulator. Currently this consists of the “POLY” device added for compatibility with SPICE2G6 macro-models.

The Internally Defined Nodes CSC (SIM-IDN) includes predefined event-driven node type definitions that are always included in the simulator. Currently this consists of the “digital” node type used for 12-state logic simulation.

The Interprocess Communication CSC (SIM-IPC) provides functions used to input SPICE decks from the ATESSE Simulator Interface or Batch Control processes, and to return results to those processes.

### 3.1.1.2.1 Code Organization

The directory structure of the SPICE 3C1 simulator is left largely unchanged in the XSPICE simulator except for the addition of new subdirectories for each of the CSCs described above. The new subdirectories are added at the level of the ‘include’, INP, FTE, CKT, etc. subdirectories.

In keeping with the organization of the original SPICE 3C1 code, header files used by the new CSCs are added to the include subdirectory using the CSC name as the prefix of the header filename. With the exception of this include directory, no files are added to the original SPICE 3C1 source directories. All new .c files are placed in one of the new CSC subdirectories.

The organization of the source code directories for the XSPICE simulator follows.

```

xspice          -- The XSPICE root directory
    bin          -- Binary executables and shell scripts
        cmpp       -- Code model preprocessor executable
        mkmoddir    -- Mkmoddir shell script
        mksimdir    -- Mksimdir shell script
        mkudndir    -- Mkudndir shell script
    xspice         -- Standalone simulator executable (includes nutmeg)
    atesse_xspice -- Simulator executable used by ATESSE
    lib           -- The release library
        sim          -- Subdirectory for simulator
            source     -- Source used by mksimdir Makefile
            object      -- Where the XSPICE core.o file resides
            helpdir     -- SPICE3C1 directory (unmodified)
            news         -- SPICE3C1 directory (unmodified)
            scripts      -- SPICE3C1 directory (unmodified)
            mfbcap      -- SPICE3C1 directory (unmodified)
            examples     -- Example XSPICE input decks and models
        cmt           -- Subdirectory for Code Model Toolkit
            cmpp        -- (empty)
            mkmoddir    -- Templates for mkmoddir
            mksimdir    -- Templates for mksimdir
            mkudndir    -- Templates for mkudndir
        cml           -- Subdirectory for Code Model Library
            gain         -- Gain block model
            limit        -- Limiter model
            ilimit       -- Current limiter model
            climit       -- Controlled limiter model
            ...
        udnl          -- Subdirectory for User-Defined Node library
            real         -- Real-valued event-driven data
            int          -- Integer-valued event-driven data
    include         -- The release directory for shared/runtime .h's
        sim          -- Header files used to compile code models
        cmt          -- (empty)
        cml          -- (empty)
    src            -- Source code (including .h source)
        sim          -- Simulator source
            include     -- C header files used by all CSCs
            notes        -- Design notes
            simdir       -- Directory used in building default executables
            CKT          -- 'Circuit' CSC
            CM           -- 'Code Model' support function CSC
            CP           -- 'Command Processor' CSC
            DEV          -- 'Device' CSC root directory
                ASRC        -- 'Arbitrary Source'
                BJT         -- 'Bipolar Junction Transistor'
                BSIM        -- 'BSIM MOSFET'
                CAP          -- 'Capacitor'

```

CCCS	-- 'Current Controlled Current Source'
CCVS	-- 'Current Controlled Voltage Source'
CSW	-- 'Current Controlled Switch'
DIO	-- 'Diode'
IND	-- 'Inductor'
ISRC	-- 'Current Source'
JFET	-- 'Junction FET'
MES	-- 'MESFET'
MOS1	-- 'Level 1 MOSFET'
MOS2	-- 'Level 2 MOSFET'
MOS3	-- 'Level 3 MOSFET'
RES	-- 'Resistor'
SW	-- 'Voltage Controlled Switch'
TRA	-- 'Transmission Line'
URC	-- 'Uniform RC Transmission Line'
VCCS	-- 'Voltage Controlled Current Source'
VCVS	-- 'Voltage Controlled Voltage Source'
VSRC	-- 'Voltage Source'
ENH	-- 'Enhancements' CSC
EVT	-- 'Event-driven' simulation CSC
FTE	-- 'Front End' CSC
HLP	-- 'Help' CSC
ICM	-- 'Internal Code Model' CSC
IDN	-- 'Internal User-Defined Nodes' CSC
INP	-- 'Input' CSC
IPC	-- 'Interprocess Communication' CSC
MFB	-- 'Model Frame Buffer' graphics display CSC
MIF	-- 'Model Interface' CSC
NI	-- 'Numerical Iteration' CSC
SMP	-- 'Sparse Matrix Package' CSC
cmt	-- Code Model Toolkit source
cmpp	-- Source for Code Model Preprocessor
mkmoddir	-- Source for mkmoddir script
mksimmdir	-- Source for mksimmdir script
mkudndir	-- Source for mkudndir script
cml	-- Code Model Library source
gain	-- Gain block model
limit	-- Limiter model
ilimit	-- Current limiter model
climit	-- Controlled limiter model
...	-- Other models
udnl	-- User-Defined Node Library source
real	-- Real-valued event-driven data
int	-- Integer-valued event-driven data

### 3.1.1.2.2 Code Structure

The major structure of the SPICE 3C1 code is also left unchanged in the ATESSE simulator, although some modifications have been made in existing files to incorporate new functionality.

In general, modifications to SPICE 3C1 source files are made without changing function parameter lists or data structures in order to simplify upgrading to future releases of SPICE3. The few exceptions include the addition of new members to the end of the CKTcircuit structure, a change in a function parameter list local to the OUTinterface package used for recording results, and the addition of several new members to the IFdevice structure. All modifications made to the original SPICE 3C1 code and data are bracketed by comments of the form “/\* gtri - ... \*/” and can be located with a utility such as UNIX “grep” by searching for the pattern “gtri”.

### 3.1.1.2.3 Data Structures

This section describes the limited modifications and additions to the existing SPICE 3C1 data structures.

#### 3.1.1.2.3.1 Static Function Pointer Structures

No modifications are made to the SPfrontEnd, ft\_sim, or cp\_coms structures.

#### 3.1.1.2.3.2 Static Device Data Structures

SPICEdev XXXinfo structures are created for each new predefined or user-created code model. These structures are generated automatically by the Code Model Toolset in the Code Model Subsystem CSCI.

The IFdevice typdef in include/iifsim.h has been modified to add additional information needed by the code model interface. This information includes a function pointer to the code model C function and information describing the expected connections, parameters, and instance variables. The XXXinfo structures for all existing SPICE 3C1 devices under the DEV subdirectory are modified to be compatible with the modified typedef. Since they are not code models, the entries in the new IFdevice members are all 0 or NULL for these devices.

#### 3.1.1.2.3.3 Dynamic Structures

No modifications are made to the deck structure used to hold the text description of the SPICE deck.

Two minor modifications are made to the circuit structure used to hold the simulator's internal representation of the circuit. The first modification consists of changing the declaration of the "CKThead" member at the beginning of the "CKTcircuit" structure to allow dynamic allocation of this array of pointers. This is required to provide for an unlimited number of code models to be linked without having to recompile functions in the XSPICE core. The second modification consists of adding two new substructures at the end of the CKTcircuit structure. The first ("evt") is used to hold event-driven simulation including states, queues, and results. The second ("enh") is used to hold data used in the general enhancements made to SPICE3.

New MIFmodel and MIFinstance structures for code models are defined in include/MIFdefs.h analogous to those used for SPICE 3C1 devices. These structures hold all information about individual code models (named parameter sets defined by a .model card) and instances used during a simulation, including information about the binding of individual code model ports, instance variables, parameter values, and data created for use by the code model support CSC.

The plot structure "dataDesc" is modified to hold special information needed to specify what data will be sent through the SIM-IPC CSC during a simulation. This structure and the effects of the modification are local to the OUTInterface.c file in FTE. No modifications are made to the plot structure "runDesc".

#### **3.1.1.2.3.4 New Global Structures**

New global data variables are introduced by the SIM-MIF, SIM-EVT, and SIM-IPC CSCs to allow data to be passed to and from modified SPICE 3C1 functions, without modifying the SPICE 3C1 function argument lists. These new variables are described in Section 5 of this document.

#### **3.1.1.2.4 Entry Points**

The main() function for the XSPICE simulator to be used with interprocess communication is in FTE/bspice.c. The program can also be compiled using FTE/nutmeg.c as the entry point file to create an interactive standalone simulator which uses the Nutmeg interface.

#### **3.1.1.2.5 Make Files**

The Makefiles have been modified as required for use in the HP/Apollo environment under BSD 4.3 Unix with X11 support. New Makefiles are also added within the new CSC directories.

The Makefile in FTE has been modified to allow creation of prelinked "core.o" and "bcore.o" files used by the Code Model Toolset in the creation of the executables "xspice" and

“atesse\_xspice”. The two “.o” files are created by typing “make core.o” and “make bcore.o” respectively. The Makefile automatically places the built cores into the “lib/object” directory where the Code Model Toolkit expects to find them.

The prelinked “core.o” file includes the full Nutmeg interface and is used when a user builds the stand-alone “xspice” executable to be run from the command line. The pre-linked “bcore.o” file includes only the object modules required to build the “atesse\_xspice” executable that works with the ATESSE system over interprocess communications. These two executables can be created by a user from the Makefile in a “simulator directory” by typing “make xspice” or “make atesse\_xspice” respectively. The default, if no argument is given is to make “xspice”. For more information, see the Software 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 provides an overview of the execution of the XSPICE simulator, indicating the major data manipulations and the principal CSCs involved in those manipulations. Since the simulator is implemented primarily as additions to the SPICE 3C1 basic architecture, this material applies to the underlying SPICE 3C1 program as well.

The basic SPICE 3C1 program consists of over 700 files and over 1400 functions. Only the most significant files and functions are described here. Table 3.2 outlines the major states/modes (the major operations performed by the program). A narrative of these operations follows.

#### 3.1.2.1 Program Startup

Various initializations are performed in main(), which is found in either FTE/nutmeg.c or FTE/bspice.c depending on whether the simulator includes the interactive interface or operates with the ATESSE system through interprocess communications. One of the most important initializations is the exchange of function pointers between Nutmeg and SPICE. This exchange is done by a call to SIMinit() from main(), which in turn calls SPIinit(). The source file containing SPIinit() defines the important DEVICES global data structure that determines which devices are included in the simulator. In the XSPICE simulator, this source file is recompiled each time a new simulator is generated with the Code Model Toolset.

State/Mode	CSC/File/Function
Program Startup	FTE/nutmeg.c/main() FTE/bspice.c/main() CKT/SPIInit.c/SPIInit()
Circuit Description Read-in	FTE/inp.c/inp_spsource() FTE/inp.c/inp_readall() IPC/IPC.c/ ipc_get_line() FTE/options.c/ inp_getopts()
Subcircuit Expansion	FTE/subckt.c/inp_subcktexpand()
Circuit Description Parsing	FTE/inp.c/inp_dodeck() FTE/spiceif.c/if_inpdeck() CKT/CKTinit.c/CKTinit() INP/INPpas1.c/INPpas1() INP/INPdomodel.c/INPdomodel() INP/INPpas2.c/INPpas2() INP/INP2R.c/INP2R() INP/INP2C.c/INP2C() ... MIF/MIF_INP2A.c/MIF_INP2A() INP/INPsymTab.c/INPtermInsert() EVT/EVTtermInsert.c/EVTtermInsert() EVT/EVTinit.c/EVTinit()
Command Processing	CP/front.c/cp_evloop() CP/front.c/docommand() FTE/runcoms.c/dosim() FTE/runcoms.c/com_run() FTE/runcoms.c/ft_dorun()
Simulation Setup	FTE/spiceif.c/if_run() CKT/CKTdoJob.c/CKTdoJob() CKT/CKTsetup.c/CKTsetup() DEV/RES/RESsetup.c/RESsetup() DEV/CAP/CAPsetup.c/CAPsetup() ... MIF/MIFsetup.c/MIFsetup() CKT/CKTtemp.c/CKTtemp() DEV/RES/REStemp.c/REStemp() DEV/CAP/CAPtemp.c/CAPtemp() ...

Table 3.2 XSPICE States and Modes.

State/Mode	CSC/File/Function
Simulation Setup (continued)	CKT/CKTic.c/CKTic() DEV/CAP/CAPgetic.c/CAPgetic() ... EVT/EVTsetup()
Simulation Execution	CKT/DCop.c/DCop() CKT/DCtrCurv.c/DCtrCurv() CKT/ACan.c/ACan() CKT/DCtran.c/DCtran() NI/NIiter.c/NIiter() CKT/CKTload.c/CKTload() DEV/RES/RESload.c/RESload() DEV/CAP/CAPload.c/CAPload() ... MIF/MIFload.c/MIFload() SMP/SMPsolve.c/SMPsolve() CKT/CKTconvTest.c/CKTconvTest() EVT/EVTop.c/EVTop() EVT/EVTiter.c/EVTiter() EVT/EVTnext_time.c/EVTnext_time() EVT/EVTaccept.c/EVTaccept() EVT/EVTbackup.c/EVTbackup()
Results Output	CKT/CKTdump.c/CKTdump() EVT/EVTdump.c/EVTdump() FTE/OUTinterface.c/OUTpData() IPC/IPC.c/ipc_send_double() IPC/IPC.c/ipc_send_complex() IPC/IPC.c/ipc_send_event() EVT/EVTop_save.c/EVTop_save() EVT/EVTiter.c/EVTiter() FTE/doplot.c/com_plot() FTE/doplot.c/plotit()

Table 3.2. XSPICE States and Modes (continued).

The main() function then processes command-line arguments that determine which mode the simulator will operate in. The actions of standard SPICE 3C1 command-line switches are documented in the SPICE 3C1 reports specified in the references. The XSPICE simulator modifies this command line processing in FTE/bspice.c for the “atesse\_xspice” executable to look for the new “-ipc” switch that enables Interprocess Communications. When the -ipc switch is used, the only other command line arguments that should be supplied

are the arguments to “-ipc” (see Interface Design Document for the XSPICE Simulator of the Automatic Test Equipment Software Support Environment (ATESSE)). If this switch is not specified, the “atesse\_xspice” executable works like the standard SPICE 3C1 bspice executable. The main() function in FTE/nutmeg.c does not accept the -ipc switch, so it is always the entry point for the “xspice” interactive mode version of XSPICE.

### 3.1.2.2 Circuit Description Read-in

Read-in of the SPICE deck begins at inp\_spsource() which declares a new “line” pointer variable named “deck” and immediately calls inp\_readall(). Function inp\_readall() is responsible for reading the input strings and building the linked-list for the deck. In the interactive mode simulator, the input deck is read from a file (or stdin), and new strings and linked list elements are allocated as each card is read. In the “atesse\_xspice” simulator with the -ipc switch specified, cards are read from the interprocess communications channel with a call to ipc\_get\_line.

After reading the deck into the linked-list deck structure, inp\_readall() manipulates the list to unwrap SPICE continuation cards, forming what SPICE 3C1 calls the “logical deck”. When the manipulation is complete, the deck structure contains a representation for both the original “physical deck”, and this new logical deck, all of which are basically character strings.

Next, inp\_readall() returns to inp\_spsource() which then calls inp\_getopts() to separate out the .options cards into a separate linked list deck structure.

### 3.1.2.3 Subcircuit Expansion

The deck without the .options cards is then passed to function inp\_subcktexpand() which expands the deck at the textual input level (this is different from SPICE2 which expanded subcircuits after parsing the deck and building the internal circuit structure). Function inp\_subcktexpand() extracts subcircuit definitions recursively and inserts them into the deck at the appropriate subcircuit element card locations. To guarantee unique names within subcircuits SPICE 3C1 prepends the name of the subcircuit to the node, element, and model names as the substitutions take place. In XSPICE, this algorithm is modified to correct a SPICE 3C1 error, and subcircuit names are added instead to the end of the node, element, or model names.

### 3.1.2.4 Circuit Description Parsing

Following subcircuit expansion, inp\_spsource() calls inp\_dodeck() to begin parsing. Function inp\_dodeck() calls if\_inpdeck() which acts as the top-level driver function for the parsing task. This function begins by allocating a new CKTcircuit structure through a call to

CKTinit() in the SPICE CSC. From this point forward, until a simulation is started, the circuit structure is held in global variable ft\_curckt.

Next, the first pass of the parser is executed by a call to INPpas1(). This function is responsible for identifying all .model cards and building a temporary structure that holds information on each .model card including the model's name, type, and text. This information is required by the second pass to resolve model references efficiently.

A second parser pass is performed by INPpas2(), which calls a separate parser function for each element card type according to the first character of the SPICE line. The individual element card parsers, such as INP2R(), INP2C(), and the new MIF\_INP2A() added by XSPICE, process the element card text and build the important linked list of XXXmodel and XXXinstance structures that represent the circuit internally during a simulation. These linked lists are attached to the "ckt" circuit structure in ft\_curckt. When analog nodes are parsed by XSPICE, the MIF\_INP2A() function creates the necessary data structures used by SPICE in building and setting up for the matrix solution by calling standard SPICE 3C1 functions such as INPtermInsert(). When event-driven nodes are parsed, a new EVTtermInsert() function is called to setup the internal representation of the event-driven nodes in the circuit.

The building of the internal representation for the event-driven circuits is completed by a call to EVTinit() from if\_inpdeck() after all instances in the circuit have been processed in INPpas2().

### 3.1.2.5 Command Processing

If the simulator is using the main() function in FTE/nutmeg.c, the top-level command processor function, cp\_evloop() is then called to wait for a user command. When a command is received, it is processed by docommand() by scanning the cp\_coms global data structure for a matching command string, and then calling the appropriate command through the function pointer associated with that string. For the command "run", the function com\_run() is invoked, which then calls dosim() to start the simulation. If the simulator is using the main() function in FTE/bspice.c, the function dosim() is invoked unconditionally through a call to ft\_dorun() in main().

### 3.1.2.6 Simulation Setup

Function dosim() sets up for a simulation run by opening a results output file (the SPICE "rawfile") and then calling if\_run(). Function if\_run() calls CKTdoJob() in the SPICE CSC to run the appropriate analysis. The circuit structure is broken out of the global variable ft\_curckt at this point and passed to CKTdoJob in its argument list as "ckt". From this point forward, the circuit structure is passed around between functions in the SPICE CSC in argument lists.

CKTdoJob() is the top-level driver function for running simulations in the SPICE CSC. Function CKTsetup() is called from CKTdoJob() to traverse all the device types in the circuit structure. For each device type (e.g. Resistor, Capacitor, Diode, Level 1 MOSFET, Level 2 MOSFET, and XSPICE code models) the global DEVices array that contains pointers to each XXXinfo structure is consulted to get the pointer to the device's setup function. For SPICE 3C1 devices, each device has its own individual setup function. For the new code model devices, the generic MIFsetup() function is used for all code models. The applicable setup functions are then responsible for traversing the linked lists of models of their type and of instances of each model. For each instance, the setup function adds current source equations and then builds the matrix structure and stores pointers into that structure for use later by the device's load function.

In the case of code models with analog nodes, the MIFsetup() function shared by all code models takes care of defaulting parameters and allocating space in the CKTstate vectors for storing old input values needed during a transient analysis to support timestep backup. In the case of code models with event-driven nodes, additional setup is done immediately prior to beginning the simulation (after the next two functions) as explained below.

Function CKTtemp() is called next from CKTdoJob() to perform a similar looping through device types and to make calls to individual device temperature functions. The individual temperature functions are responsible for updating any items in each instance's XXXinstance and/or model's XXXmodel structures according to the new circuit temperature. The XSPICE code model interface does not provide a temperature update function since all individual code models are responsible for dealing with temperature.

Function CKTic() is called from CKTdoJob() similarly. The individual device initial condition functions take care of analog node value initializations. In general, not all device types have an initial condition function since not all types have initial condition syntax.

The new XSPICE function EVTsetup() is then called to initialize the event-driven run-time data structures including the queues, state storage, and results data.

### 3.1.2.7 Simulation Execution

After calling the setup, temperature, initial condition function drivers, and the event-driven run-time data setup function, CKTdoJob() calls the analysis function for the required analysis type. DCop() is called for a basic DC operating point analysis. DCtrCurv() is called for a swept DC or "transfer function" analysis. ACan() is called for a small-signal AC analysis. DCtran() is called for a transient analysis. In each case, the simulator sets up and solves appropriate matrix equations. In the case of transient analysis, the solution of analog circuitry is subjected to an additional numerical integration "truncation error" assessment and other controls in timestepping to assure accurate results.

The basic solution of the analog circuit equations in DC and transient analysis modes is performed by NIiter() which repetitively solves matrix equations according to a Newton-Raphson non-linear equation solving algorithm. This algorithm involves guessing a solution to the node voltages and voltage source branch currents in the circuit, linearizing the circuit equations, and solving the linearized system of equations to produce the next guess. When all circuit variables in the solution vector converge to within some predefined amount, the iteration is stopped and the solution vector contents are taken as the solution values.

This Newton-Raphson process is performed by NIiter() by calling the CKTload() driver function to iterate through all devices with analog nodes in the same manner as CKTsetup(), CKTtemp, and CKTic(). Each device type has its own load function which is responsible for iterating through all models and instances of that device and inserting appropriate output and partial derivative terms into the matrix to build up the linearized circuit equation set. When all instances and models of all devices have loaded the matrix, SMPsolve() is called to solve the matrix. After each solution, NIiter() calls CKTconvTest() to check for convergence.

When a circuit also contains event-driven models, the event-driven analysis is interleaved with the analog solution described above. In the case of a DC operating point solution, the interleaving is a simple alternation between event-driven and analog solutions until all event-driven nodes reach stable states. This alternation is performed by EVTop() which is called in place of CKTop() when event-driven nodes exist in the circuit.

In the case of a transient analysis, the interleaving of analog and event-driven solution algorithms is more involved because of the need to coordinate event times with analog timesteps, and the possibility for timestep backups caused by non-convergence or excessive local truncation error. Normally the event-driven queues maintain a list of upcoming events that must be processed, and the time at which these events are to occur. Following completion of a successful analog timestep solution, the event queue is checked by a call to EVTnext\_time() for any events with a scheduled time less than or equal to the next timestep. These events, and any resulting events with time less than or equal to the next analog timepoint are then processed by calls to EVTiter(). The next analog timestep solution is then attempted as described above. If successful, function EVTaccept() is called to record the state of the event-driven simulation at this point. If unsuccessful, the DCtran() function determines a new analog time at which to attempt solution which is earlier than the analog time at which the solution failed. Function EVTbackup() is then called to undo the processing of any events that occurred prior to this new analog time.

### 3.1.2.8 Results Output

Following a successful analog solution, the analysis driver function (DCop, DCtran, etc.) calls CKTdump() to store or output analog results. CKTdump() in turn calls OUTpData() in FTE/OUTinterface.c to write the results to the “rawfile”, to save them in a “plot”

structure, or to send them to the ATESSE Simulator Interface process through a call to ipc\_send\_double() or ipc\_send\_complex(). For event-driven circuits, the analysis driver functions call EVTdump() to send the event-driven node solutions to the ATESSE Simulator Interface process through a call to ipc\_send\_event().

Function EVTop\_save() is called at the end of an operating point analysis and at the end of each step in a swept DC analysis to save the final event node values in the internal event-driven results structure (ckt->evt). For a transient analysis, the data is saved as it is computed in function EVTiter().

### 3.1.2.9 Program Exit

If the simulator is running in batch mode or is being used with the ATESSE through interprocess communication, a succession of function returns beginning at CKTdoJob() and ending at main() causes the program to wrap up its execution and terminate. If the program is running with the interactive Nutmeg interface, however, the program returns to cp\_evloop() where it waits for a user command. If the user issues the plot command, function com\_plot() is called through a function pointer in the cp\_coms data structure, which in turn calls plotit() to display a plot to the user. The interactive-mode simulator terminates when the user issues the “quit” command.

### 3.1.3 Memory and Processing Time Allocation

Memory is allocated dynamically throughout the program using UNIX “malloc” and associated storage allocators. This dynamic allocation allows the program to simulate circuits of arbitrary size.

Memory allocated in the SPICE 3C1 simulator is not consistently freed. Although CSCs such as the SPICE 3C1 and new XSPICE code model devices include functions to deallocate dynamic data structures, other CSCs such as INP do not. Therefore, the simulator should be exited and restarted periodically. The IPC package is implemented without a server loop to guarantee that this will be done when working in the ATESSE system.

Since the simulator is not a real-time process, allocation of processing time to CSCs is not applicable to the simulator and is not discussed here. However, overall execution time is a central measure of performance in simulators. Therefore, when tradeoffs are required between speed and memory usage in the XSPICE simulator development, speed is emphasized over conservation of memory.

### 3.2 CSCI Design Description

The previous sections have provided an overview of both the unmodified SPICE 3C1 simulator and the XSPICE simulator's additions and modifications. The remainder of this document concentrates only on the new simulator code. Descriptions of the SPICE 3C1 simulator code may be found in the documents specified in the references.

The seven new CSCs added to the SPICE 3C1 simulator are:

SIM-MIF	Model Interface
SIM-CM	Code Model support functions
SIM-EVT	Event-Driven simulation
SIM-ENH	Enhancements
SIM-ICM	Internal Code Models
SIM-IDN	Internally Defined Nodes
SIM-IPC	Interprocess Communications

The top-level designs of these new CSCs are described below. Descriptions of the major procedures involved in each are given in Chapter 4, "Detailed Design".

#### 3.2.1 Model Interface

The Model Interface (SIM-MIF) CSC works in conjunction with the Code Model Toolset to allow new models to be added to the simulator at the source code level. The SIM-MIF CSC builds on top of the SPICE 3C1 simulator's existing model implementation described in Adding Devices to SPICE3. However, the XSPICE model interface is designed to be more general and easier to use.

The SPICE 3C1 method of adding devices assumes that new devices will be similar to ones already known to the simulator - diodes, transistors, etc. This limits what can be modeled to those functions with a small number of fixed pins and with a syntax similar to existing devices. Moreover, the SPICE 3C1 approach describes over two dozen locations in the SPICE code that require modification to integrate a new model, including writing the parsing code for the new device. If the new model is similar to an existing device, the code for that existing device can be copied and modified as appropriate to create the new device, often without understanding all of the internal SPICE functions and data structures involved. However, if the device is not similar to an existing device, these functions and data structures must be understood in detail.

The SIM-MIF insulates a user from these details of SPICE, while defining an extended circuit description syntax that provides for models significantly more general than those in standard SPICE. A definition of this new syntax can be found in the Software User's

Manual for the Simulator of the Automatic Test Equipment Software Support Environment (ATESSE).

With the aid of the Code Model Toolset described in the XSPICE Code Model Subsystem design documents, a user is only required to create two files to add a new device to the simulator. One file contains an "Interface Specification" that defines the new device name, ports, parameters, etc. in a simple tabular format. The other is a C function that implements the device behavior. The Code Model Toolset takes these files and transforms them into a form that can be compiled and bound in with the modified SPICE 3C1 simulator core. The Code Model Toolset also allows a tailored copy of a simulator to be constructed which includes only a user-specified collection of predefined or user-written code models.

The tabular Interface Specification written by a user is converted by the XSPICE Code Model Toolset into a C language source file (ifspec.c) defining a static SPICEdev type structure similar to the XXXinfo structures used by standard SPICE 3C1 devices. Hence, new devices are handled by the simulator in the same way as standard SPICE 3C1 devices. The SPICEdev structure contains the information used by SIM-MIF code to parse and error check the input deck, as well as function pointers to setup, load, etc. functions used during the creation of the circuit structure and during simulation.

All code models written by a user share the SIM-MIF parser code and a standard set of SIM-MIF function pointers. One of the most important function pointers (the DEVload element) points to MIFload() which is responsible for setting up the proper data in the argument list of the user-written code model C function, calling the function, and then loading the matrix with the data computed by the function.

In addition to the new SIM-MIF CSC, a small number of modifications are made to existing SPICE source files. These modifications include:

- Changes to the subcircuit expansion code to make it work properly with the new syntax.
- Changes to the parsing functions to recognize the new "A" type SPICE deck element card.
- Adding include statements to CKT/SPIinit.c to include files that specify what code models to place in the simulator when it is linked. At the same time, SPI-init.o was removed from CKT/Makefile so that this file is not linked until the Code Model Toolset is used to build the simulator from spice\_core.o and the selected models. To force spice\_core.o to include the standard SPICE 3C1 devices specified in CKT/SPIinit.c, dummy structures are also added to FTE/spiceif.c.

### 3.2.1.1 Example Code Model

From a user's perspective, there are three main aspects to an XSPICE code model:

- The syntax employed in referencing the model in a SPICE circuit description (SPICE deck).
- The Interface Specification that defines the models inputs, output, and parameters.
- The Model Definition (or C function) that implements the model's behavior.

Examples for each of these for a simple "gain\_block" model are given below. The design of each is discussed in the following sections. Additional information can be found in the Software Design Document for the Code Model Subsystem of the Automatic Test Equipment Software Support Environment (ATESSE) and the Interface Design Document for the Code Model Subsystem of the Automatic Test Equipment Software Support Environment (ATESSE).

#### Circuit Description Syntax Example

```
a1 1 2 gain
.model gain gain_block (gain=10.0 in_offset=0.1)
```

#### Interface Specification Example

##### NAME\_TABLE:

C_Function_Name:	cm_gain_block
Spice_Model_Name:	gain_block
Description:	"A simple gain block""

##### PORT\_TABLE:

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

**PARAMETER\_TABLE:**

Parameter_Name:	in_offset	gain	out_offset
Description:	"input offset"	"gain"	"output offset"
Data_Type:	real	real	real
Default_Value:	0	1	0
Limits:	[-1e10, 1e10]	[-1e10, -]	[-1e10, 1e10]
Vector:	no	no	no
Vector_Bounds:	-	-	-
Null_Allowed:	yes	yes	yes

## C Function Example

```
void gain_block(ARGs)
{
    Complex_t ac_gain;

    if(ANALYSIS != AC) {
        OUTPUT(y) = PARAM(out_offset) + PARAM(gain)
                    * (INPUT(a) + PARAM(in_offset));
        PARTIAL(y,a) = PARAM(gain);
    }
    else {
        ac_gain.real = PARAM(gain);
        ac_gain.imag = 0.0;
        AC_GAIN(y,a) = ac_gain;
    }
}
```

### 3.2.1.2 Circuit Description Syntax

The new XSPICE Code Model syntax is patterned after SPICE as far as possible and uses the SPICE method of putting model parameters on a separate .model card. Example instance and .model cards are shown below.

```
a27 1 2 gain
.model gain gain_block (gain=10.0 in_offset=0.1)
```

In this example “a27” is the instance name. All code models begin with an “a” to indicate that they are code models. The instance name is followed by the connection list (“1” is the input node, and “2” is the output node), and then the name of the model (“gain”) that references the .model card. The order of the connections is defined by the order of the ports defined in the Interface Specification.

The .model card defines the actual code model used for the instance (“gain\_block”) and the parameters to the code model (“(gain=10.0 in\_offset=0.1)”). This construction is patterned after standard SPICE diode and transistor syntax.

### 3.2.1.2.1 Use of Parentheses

Note that the parentheses on the .model line are treated as white space and are not required. This will be true throughout the syntax in accordance with standard SPICE practice. Parentheses are used only to aid clarity in the deck and are typically used to surround a parameter list, or to surround pairs of differential nodes.

### 3.2.1.2.2 Node Names

Node names may be arbitrary strings, and are not restricted to being integers as was the case in SPICE 2G6, except for the special ground node which must always be 0.

### 3.2.1.2.3 Instance Parameters

For the present, no parameters are allowed on the instance line. This restriction may be removed in future versions of XSPICE to allow any parameter appropriate on a .model card to be specified on the instance card, and any parameter on the instance card.

### 3.2.1.2.4 Vector Ports

According to the XSPICE Requirements Document, the syntax must support arrays (or "vectors") of inputs and outputs. An example of an array input used to reference a three input NAND gate is shown below.

```
a27 [1 2 3] 4 nand3
.model nand3 nand_gate
```

In this example, the "input" to the NAND gate is an array of connections to nodes 1, 2, and 3, and the output is connected to node 4. Note that the parser can determine the dimension of the array as the SPICE deck is read, so that a single nand\_gate code model can serve for any number of inputs. Arrays of outputs can be handled similarly.

### 3.2.1.2.5 Connection Types

According to the XSPICE Requirements Document, the syntax must support an arbitrary mix of input and output types. The current implementation supports the following pre-defined types, plus user-defined types.

### 3.2.1.2.5.1 Basic Analog Types

```
v      voltage (ground referenced)
vd     voltage (differential)

i      current (ground referenced)
id     current (differential)

vnam   current through voltage source
```

Types v, vd, i, and id refer to port connections that are either inputs or outputs. Whether the port is an input or an output is defined in the separate Interface Specification table.

The special "vnam" type is used for compatibility with SPICE where an input is taken as the current through some voltage-defined branch in the circuit, usually a voltage source. This port can only be used with input ports.

### 3.2.1.2.5.2 Bidirectional Analog Types

```
g      conductance (vcis) (ground referenced)
gd     conductance (vcis) (differential)

h      resistance (icvs) (ground referenced)
hd     resistance (icvs) (differential)
```

The special types g, gd, h, and hd, are both inputs and outputs. Types g and gd are voltage controlled current sources, and types h and hd are current controlled voltage sources. The type gd will be the most commonly used and will find application in defining things like pwl conductances, the magnetics reluctance model, and the "output" port of a current limiter. For example, a pwl conductance has two pins which can be treated as a single differential port of type gd. With this mechanism, the SPICE card for a pwl conductance looks very much like a resistor. Note that the parentheses are removed to emphasize the similarity to a resistor.

```
a27 1 2 varistor
.model varistor pwl_conductance ( ...
```

### 3.2.1.2.5.3 Digital and User-Defined Types

```
d      digital
<name> user defined type
```

The special type d is a digital connection and comes exclusively in a single-ended variety (there is no meaning to subtracting the values of two digital nodes). For d type connections,

node names may be prefixed with a tilde character “~” to designate that their digital state should be inverted. Hence,

```
a27 [1 2 ~3] 4 nor3
.model nor3 nor_gate ( ...
```

refers to a nor gate with 3 inputs, one of which has a bubble. The code model does not need to know that the third input has been inverted and it is not informed of the inversion. The inversion is performed by the simulator before the model is called.

The special user-defined type will have a name specified by the user. This name must be unique from any of the pre-defined types and should probably be prefixed by something like “ud\_” to allow new built-in types to be added to the simulator in the future without creating name collisions. However, this prefixing is not enforced.

### 3.2.1.2.5.4 Overriding Default Port Types

In the XSPICE deck syntax, all models have default types and a list of allowable types (defined in an Interface Specification table) for their input and output port connections. The default types may be overridden by prefixing the connections with %<type designator>. For example if

```
a27 1 2 gain
.model gain gain_block (gain=10.0 in_offset=0.1)
```

refers to a gain block with voltage input on node 1 and voltage output on node 2, then

```
a27 %vd (1 5) 2 gain
.model gain gain_block (gain=10.0 in_offset=0.1)
```

refers to a gain block with differential input taken between nodes 1 and 5, and output connected to node 2.

### 3.2.1.2.5.5 Mixing Types

Interesting questions arise with vector connections. For example, can you have different types in a single vector? The current implementation allows different types (e.g. i, id, v, vd) as long as they have the same base representation type for the data passed to the code model C function (e.g. double). This helps solve some nasty structure definition and use problems in the code model C function while allowing things like voltages and currents to be multiplied by a generic multiplier code model. When two signals with different base data types must be interfaced, the user must provide an explicit “node bridge” type code model to perform the conversion.

### 3.2.1.2.5.6 The NULL Type

In order to support models such as flip-flops with optional set and clear inputs, a special node name of “null” is defined. This name is universal in that it matches any type (including differential types and the “vnam” type). The null connection tells the code model C function that there is no node connected to that input or output so that the code model can modify its functionality appropriately. For example, if

```
a27 1 2 3 4 5 6 dff1
.model dff1 d_flip_flop ( ...
```

calls a D flip-flop with connections in the following order:

```
clock, d, set, clear, q, not-q,
```

then

```
a27 1 2 null null 5 null dff1
.model dff1 d_flip_flop ( ...
```

calls a D flip-flop in which the set and clear inputs and the not-q output are not used.

### 3.2.1.2.6 Model Names

Model names may be of arbitrary length. It is recommended that users prefix model names and the associated C function name with a special string such as “ucm\_” to minimize chances of name collisions with internal model names and/or internal XSPICE function names. The prefix “ucm\_” is the default prefix used by the Code Model Toolkit when a user creates a new model directory with the “mkmoddir” command.

### 3.2.1.2.7 Parameter Values

The XSPICE Requirements Document calls for named parameters that take any of the following types:

```
real
integer
string
Boolean
complex
```

```
real array
integer array
string array
Boolean array
complex array
```

Except for Boolean, these are the types which the SPICE 3C1 documentation claims are already handled by the parser. An examination of the code revealed that arrays of integers and strings were not implemented. These types were therefore added in the XSPICE development.

In the case of Boolean, the reserved words TRUE and FALSE are converted internally to integers with values of 1 or 0 respectively.

### 3.2.1.3 Interface Specification

This subsection provides notes on the design of the code model Interface Specification. This Interface Specification tells the simulator how to parse a code model reference in a SPICE deck, and how to setup data in structures passed to the code model C function. For additional information, refer to the Interface Design Document for the XSPICE Code Model Subsystem of the Automatic Test Equipment Software Support Environment (ATESSE).

A simple example of an Interface Specification for a summer model is shown below:

NAME\_TABLE:

C_Function_Name:	cm_summer
Spice_Model_Name:	summer
Description:	"A 2 to N input summer function"

POR T\_TABLE:

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

PARAMETER\_TABLE:

Parameter_Name:	in_offset	in_gain
Description:	"input offset"	"input gain"
Data_Type:	real	real

Default_Value:	0	1
Limits:	[-1e10, 1e10]	[-1e10, -]
Vector:	yes	yes
Vector_Bounds:	in	in
Null_Allowed:	yes	yes
Parameter_Name:	out_gain	out_offset
Description:	"output gain"	"output offset"
Data_Type:	real	real
Default_Value:	1.0	0.0
Limits:	[-, 1e10]	-
Vector:	no	no
Vector_Bounds:	-	-
Null_Allowed:	yes	yes

This example Interface Specification is divided into three tables. The first table provides a mapping between the name of the code model C function and the name as used on the .model card. The second table provides information on the inputs and outputs to the code model (called “ports”), while the third provides information on the code model parameters (used on .model cards).

An additional table is optional:

#### STATIC\_VAR\_TABLE:

Static_Var_Name:	scaled_input	summer_output
Description:	"Value of inputs just prior to summer"	"Value of summer prior to output gain/offset"
Data_Type:	real	real

This table defines per-instance variables that can be used to store data between iterations. In addition, the variables in this table can be output during a simulation through the SPICE3 “.save” mechanism. This mechanism allows models to compute and output data such as power or internal states, without the need to use a node.

#### 3.2.1.3.1 Name Table

This table tells the simulator what the name of the code model C function is and what the name of the model is on a .model line. It is recommended that the code model C function names be prefixed to help avoid name collisions at the linker level. It is also recommended that the names as they appear on the .model card be prefixed to avoid collisions with other models. It is permitted to use the same name for both the function and the .model name.

### 3.2.1.3.2 Port Table

The example above shows the types of information contained in the port table. This information includes:

- Port name
- Direction
- Default type
- Allowed types
- Whether or not the connection is a vector
- Constraints on the minimum and maximum vector size
- Whether or not null is allowed for the connection
- An ascii string describing the parameter

#### 3.2.1.3.2.1 Port Name

The port name is used to associate the nodes listed in a code model reference in a SPICE deck (e.g. “a1 1 2 gain”) with the data passed to the code model C function. The order in which port names appear in this table defines the order of port connections coded in an XSPICE input deck.

#### 3.2.1.3.2.2 Direction

The direction column of the Interface Specification table tells whether the port is an input, output, or input/output. The direction “input/output” is used for g, gd, h, and hd type connections.

This information is needed by the parser to identify outputs since outputs produce new equations in the matrix and new event lists in the digital node simulator. This information is also used by the parser and simulator to divide the connections into separate input and output structures fed to the code model C function. The inputs and outputs are also separated because outputs require partial derivatives, etc., whereas inputs do not.

#### 3.2.1.3.2.3 Default Type

As explained above, all ports must have a default type (e.g. v, vd, i, id, g, gd, ...). Typically, for analog code models the default type will be v, except in special cases such as the reluctance model which will be a gd type.

Use of a default type makes the code model reference in the SPICE decks look more like the standard SPICE elements. For example, a gain block can be coded as “a1 1 2 gain” instead of “a1 %v 1 %v 2 gain”.

### 3.2.1.3.2.4 Allowed Types

The allowed types specify what the default type can be overridden with. Typically the list of allowed types will include all types which make sense for the model, and which have the same data representation in the code model C function (e.g. v, vd, i, id are all real valued and can be treated identically in the code model C function).

Mixes of types that do not have the same underlying data representation are not allowed. To convert between types with different representations, a “node bridge” model must be used.

### 3.2.1.3.2.5 Vector

This column of the Interface Specification table specifies whether the connection is a vector (array) or not. This affects whether the parser expects to see a vector input (i.e. [1 2 3] ) or a scalar input. If vector is yes, then the parser will report an error if a list in brackets is not found.

The parser will determine the vector size dynamically by counting the number of connections in the brackets. This count is made available to the code model C function through the “PORT\_SIZE” macro.

Overriding of the default connection type for an array is allowed either for the entire array by placing the default override outside the brackets:

```
%i [1 2 3 4 5]
```

or on an element by element basis by placing the override inside the braces:

```
[1 2 %v 3 %i 4 %id (5 6)]
```

### 3.2.1.3.2.6 Vector Bounds

The number of elements allowed in a vector may be constrained by specifying minimum and/or maximum sizes or “bounds”. Either or both constraints may be omitted. Whenever practical, models should be created which work for arbitrary sizes.

### 3.2.1.3.2.7 Null Allowed

The “Null Allowed” column tells the parser if “null” can be used for the port’s connection. For example, for a single-input, single-output gain block, null would not be allowed for either the input or the output. For a flip-flop, null would be allowed for the set and clear inputs and for the q and not-q outputs. Complex checks such as whether or not both q and not-q can be null at the same time are not supported in the parser, but can be coded inside a model.

### 3.2.1.3.2.8 Description

This is an ASCII text string that describes the connection. It is used for documentation purposes only.

## 3.2.1.3.3 Parameter Table

The example above showed the types of information contained in the parameter table. This information includes:

- Parameter name
- Parameter type
- Default value
- Upper and lower limits for the default
- Whether or not the parameter is an array
- Constraints on minimum and maximum vector size
- Description of the parameter

### 3.2.1.3.3.1 Parameter Name

This is the name of the parameter as it would appear on the SPICE deck .model line. Since named parameters are used on .model lines exclusively, the order of the parameters in the Interface Specification table is unimportant.

### 3.2.1.3.3.2 Description

This is an ASCII text string that describes the parameter. It is used for documentation purposes only.

### 3.2.1.3.3.3 Data Type

The parameter type specifies the data type for the parameter from the following options:

integer	int
real	double
string	char *
boolean	int
complex	Complex_t

Integer parameters must have integer arguments. Default interpretation is base 10. Base 16 can be specified by using a leading "0x". Octal can be specified by using a leading "0".

Real parameters may have real parameters (e.g. 10.0) or base 10 integer parameters (e.g. 10). Either parameter is converted internally to type double.

String parameters may be arbitrary text. If embedded blanks or parenthesis are included, the string must be enclosed in double quotes.

Boolean parameters may take values of TRUE or FALSE, which are converted internally to 1 or 0 respectively.

Complex parameters are ordered pairs of reals or base 10 integers. They are converted internally to the type Complex\_t consisting of a structure with two doubles. Both the parameters and the internal representation are rectangular, with real followed by imaginary parts.

### 3.2.1.3.3.4 Default Value

Most parameters will have a default value that will be inserted if the user does not provide a value for the parameter on the .model card. A default for a vector applies to all elements of the vector. Defaults for individual elements of a vector are not supported.

### 3.2.1.3.3.5 Limits

In order to provide some built-in error checking on parameter values when the simulator is used stand-alone (i.e. without the ATESSE Simulator Interface process), upper and lower limits may be provided. Limits for a vector apply to all elements of the vector. Independent limits for individual elements of a vector are not supported.

### 3.2.1.3.3.6 Vector

This entry specifies if the parameter is a vector.

### 3.2.1.3.3.7 Vector Bounds

This entry specifies optional minimum and maximum sizes allowed for a vector parameter.

If desired, this entry may specify a port which the parameter is associated with. For example, for a summer an input offset vector parameter could be associated with the input port vector. The parser will report an error if the number of elements in the vector parameter differs from the number of elements in the vector port connection.

If this entry is left blank for a vector parameter, it is the responsibility of the code model to check for a valid number of elements.

### 3.2.1.3.3.8 Null Allowed

This entry specifies whether or not the parameter may be left unspecified on the .model card. If Null\_Allowed is true and the parameter is omitted from the model card, the default parameter will be used if one has been specified. If no default has been specified, the code model must assign its own default. The code model may check for this case using the “PARAM\_NULL” macro.

## 3.2.1.4 C Function and Accessor Macros

This subsection discusses the design for accessing data from the formal parameter(s) provided in the argument list of a code model C function. Data passed in the formal parameter list includes circuit data such as time, temperature, etc; values of model parameters, and values of inputs, outputs, partial derivatives, etc.

### 3.2.1.4.1 Design Approach

In considering possible designs, tradeoffs arise in the areas of simplicity of access to data (how complex a structure definition is), number of items in the formal parameter list, and flexibility/extensibility of the items in the parameter list.

After conducting a poll of representative users, it was decided that macros would be provided to access data in the argument list. This approach offers the best of many worlds. For example, access to the data can be provided with a simple and readable “PARAM(gain)” rather than the underlying structure reference:

```
private->param[1]->element[0].rvalue
```

In addition, the function argument definition can be maintained private and can be modified in the future without modifying existing model code (models will have to be recompiled however).

The use of various macros is illustrated in the following set of code fragments. Note that the argument to most macros is the name of a parameter or port as defined in the Interface Specification. Note also that all macros resolve to an lvalue (C terminology for something that can be assigned a value). Macros do not implement expressions or assignments.

```
void code_model(ARGS) /* private structure accessed by macros */
{
    /* Determining analysis type */
    if(ANALYSIS == AC) ...

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

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

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

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

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

    /* Accessing the value of a user defined node input */
    a_ptr = INPUT(a);
    x = a_ptr->component1;
    y = a_ptr->component2;

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

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

    /* Outputting the partial of output y wrt input a */
    PARTIAL(y,a) = PARAM(gain);

    /* Outputting the partial of output y(i) wrt input a(j) */
    PARTIAL(y[i],a[j]) = 0.0;

    /* Outputting gain from input a to output y in an AC analysis */
    ac_gain.real = 1.0;
    ac_gain.imag = 0.0;
```

```
AC_GAIN(y,a) = ac_gain;

/* Outputting a digital result */
OUTPUT_STATE(y) = ONE;
OUTPUT_STRENGTH(y) = STRONG;

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

/* Initializing and outputting a user defined node result */
if(INIT) {
    OUTPUT(y) = malloc(sizeof(user_defined_struct));
    y_ptr = OUTPUT(y);
}
else
    y_ptr = OUTPUT(y);
y_ptr->component1 = x1;
y_ptr->component2 = x2;

}
```

### 3.2.1.4.2 Access Macros

The set of macros provided for analog models is described below. The code model interface for analog was implemented first, before the event-driven algorithm, so this section covers macros applicable to analog models only. Macros for the event-driven algorithm are discussed in a separate section in this appendix covering the event-driven design.

This section provides overviews and design notes on macros. Refer to the Interface Design Document for the XSPICE Code Model Subsystem of the Automatic Test Equipment Software Support Environment (ATESSE) for additional information.

#### 3.2.1.4.2.1 Circuit Data Macros

The macros below are used to access general parameters of the circuit being analyzed.

```
INIT
ANALYSIS
NEW_TIMEPOINT
TIME
RAD_FREQ
TEMPERATURE
T(n)
```

INIT is an int that takes values of 1 or 0 depending on whether or not this is the first call to the code model for this instance respectively.

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

NEW\_TIMEPOINT is an int that takes values of 1 or 0 depending on whether or not this is the first call for this instance at the current analysis point.

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

RAD\_FREQ is a double representing the current analysis frequency in an AC analysis in radians per second.

TEMPERATURE is a double representing the current analysis temperature.

T(n) is a double array giving the analysis time for the last 8 timepoints in a transient analysis, where n takes on values of 0-7. T(0) is equal to TIME. T(1) is the last accepted timepoint. (T(0) - T(1)) is the timestep associated with the current time. In the current implementation of the simulator, only T(0) and T(1) are supported.

### 3.2.1.4.2.2 Parameter Data

The macros below are used to access information about parameters defined in the Interface Specification.

```
PARAM(gain)
PARAM(gain[i])
PARAM_SIZE(gain)
PARAM_NULL(gain)
```

PARAM(gain) resolves to the value of the scalar parameter “gain” which was defined in the Interface Specification tables. The type of “gain” is the type given in the Interface Specification. The same macro can be used regardless of type since it simply resolves to an lvalue.

PARAM(gain[i]) resolves to the value of the the i'th element of an array parameter “gain”.

PARAM\_SIZE(gain) resolves to an int representing the size of the “gain” vector parameter (which was dynamically determined when the SPICE deck was read). PARAM\_SIZE(gain) is undefined if gain is a scalar parameter.

PARAM\_NULL(gain) resolves to an int with value 0 or 1 depending on whether the user specified a value for gain, or whether the value is defaulted respectively.

### 3.2.1.4.2.3 Port Data Macros

The macros below are used to access information about the connections to ports of the model.

```
POR T_SIZE(a)  
POR T_NULL(a)
```

POR T\_SIZE(gain) resolves to an int representing the size of the “a” connection (which was dynamically determined when the SPICE deck was read). POR T\_SIZE(a) is undefined if gain is a scalar.

POR T\_NULL(a) resolves to an int with value 0 or 1 depending on whether the SPICE deck has a node specified for this connection, or has specified that the connection is null respectively.

Note that this data is kept separate from the actual input and output data since it is possible that a connection can be both an input and an output.

### 3.2.1.4.2.4 Input Data Macros

The macros below are used to access the value of model inputs.

```
INPUT(a)  
INPUT(a[i])
```

INPUT(a) resolves to the value of the scalar input “a” which was defined in the Interface Specification tables. The type of “a” is the type given in the Interface Specification. The same macro can be used regardless of type since it simply resolves to an lvalue.

INPUT(a[i]) resolves to the value of the the i'th element of an array input “a”.

### 3.2.1.4.2.5 Output Data Macros

The macros below are used to assign outputs computed by models.

```
OUTPUT(y)  
OUTPUT(y[i])
```

OUTPUT(y) resolves to the value of the scalar output “y” which was defined in the Interface Specification tables. The type of “y” is the type given in the Interface Specification. The same macro can be used regardless of type since it simply resolves to an lvalue.

OUTPUT(y[i]) resolves to the value of the the i'th element of an array output “y”.

### 3.2.1.4.2.6 Partial Derivative Macros

The macros below are used to assign partial derivatives computed by a model.

```
PARTIAL(y,a)
PARTIAL(y[i],a)
PARTIAL(y,a[j])
PARTIAL(y[i],a[j])
```

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

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

### 3.2.1.4.2.7 AC Gains

The macros below are used to assign AC gain values computed by a model.

```
AC_GAIN(y,a)
AC_GAIN(y[i],a)
AC_GAIN(y,a[j])
AC_GAIN(y[i],a[j])
```

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

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

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

### 3.2.1.4.2.8 Static Variable Macros

The macro below is used to access static variables declared in the Interface Specification.

```
STATIC_VAR(x)
```

STATIC\_VAR(x) resolves to an lvalue which can be assigned the value of some scalar code model result or state. The type of "x" is the type given in the Interface Specification. The same macro can be used regardless of type since it simply resolves to an lvalue. For static var vectors, use "pointer" as the type in the Interface Specification. Then, on the model initialization pass (INIT == TRUE), space for the vector can be malloc'ed, and the malloc'ed pointer can be assigned to the static var. On subsequent calls to the code model, the pointer will be accessible through the value of the static var.

### **3.2.1.5 Model Interface Implementation**

This section describes the modifications necessary in the SPICE 3C1 code to add the code model interface.

#### **3.2.1.5.1 Device Models**

The mechanism provided for adding devices in SPICE 3C1 is described in the report [Adding Devices to SPICE3](#). Adding new devices entails adding several new data structures and functions. For the new XSPICE simulator, a generic device is constructed in the SIM-MIF CSC and interfaced to the 3C1 simulator through these routines and data structures according to the following guidelines:

- The parsing code does not have to be written by the user
- The user is not required to understand or implement all of the required data structures/routines directly.
- It is relatively easy to repeat the implementation in future releases of SPICE3.

#### **3.2.1.5.2 Data Structures and Functions**

Data structures that must be implemented in the SPICE 3C1 mechanism include:

- A SPICEdev structure that contains information such as the number of terminals on the device, pointers to the device specific functions, and the size of the device's instance and model structures that need to be dynamically allocated. This structure is "the only externally visible" data about the device directly available to the rest of Nutmeg/SPICE. This structure is static.
- A model structure that contains link-list pointers to other models and to instances of this model, the model name, and any model specific data needed. This structure is dynamic.

- An instance structure that contains link-list pointers to other instances of this type, a back-pointer to the parent model structure, a list of external nodes, and any instance specific data needed. This structure is dynamic.
- A modelParms structure specifying the parameter names and types, and whether or not the parameters are input, output, or input/output with respect to the .save functionality. This structure is static.
- An instanceParms structure specifying the parameter names and types, and whether or not the parameters are input, output, or input/output with respect to the .save functionality. This structure is static.
- A set of #defines for any device specific constants.

Routines that must be implemented include a subset of the following, depending on what analysis modes are supported.

DEVparam()	Input a parameter to an instance data structure.
DEVmodParam()	Input a parameter to a model data structure.
DEVload()	Evaluate device equations and load matrix.
DEVsetup()	Preprocess device once after parsing.
DEVpzSetup()	Preprocess for Pole Zero analysis.
DEVtemperature()	Preprocess device after temp/etc. parm change.
DEVtrunc()	Perform truncation error calculations.
DEVfindBrance()	Search for device's branch equations.
DEVacLoad()	Evaluate equations and load AC analysis matrix.
DEVaccept()	Evaluate once-per-iteration after timepoint accept.
DEVdestroy()	Free all dynamically allocated device structs.
DEVmodDelete()	Free a particular model and it's instances.
DEVdelete()	Free a particular instance.
DEVsetic()	Get initial conditions from RHS vector.
DEVask()	Allow Nutmeg to ask about instance parameters/etc.
DEVmodAsk()	Allow Nutmeg to ask about model parameters/etc.
DEVpzLoad()	Evaluate equations and load PZ matrix?
DEVconvTest()	Test for convergence.
DEVsenSetup()	Preprocess for sensitivity analysis.
DEVsenLoad()	Evaluate equations and load sensitivity matrix.
DEVsenUpdate()	Update device sensitivity info.
DEVsenAcLoad()	Evaluate equations and load AC sensitivity matrix.
DEVsenPrint()	Print sensitivity info.
DEVsenTrunc()	Compute truncation error for sensitivity analysis.

Once these data structures and functions are implemented for the device, SPICE must be told about the existance of the new device by modifying CKT/SPIinit.c.

### 3.2.1.5.3 Syntax Parsing

In designing the SPICE deck syntax and code model interface, the intent is to remove as much work from the user as possible, requiring them to write only a specification for the model parameters, and the code that actually computes the input/output relationship and first partials for the particular model. The user should not have to know anything about the architecture of the simulator, or things such as how to interface with the matrix. Moreover, the syntax should be designed to work for any code model so that the user is not required to write parsing code. This is achieved by creating a new "A" device type and introducing new .model types for the "A" device for each new code model.

In general there are four areas of the SPICE 3C1 code that must be modified to achieve these goals:

- Modify the subcircuit expansion code to work properly with the new "A" device syntax.
- Modify the existing parsing code to work properly with the new syntax and implement the new parser for the general "A" type device.
- Create general purpose setup, load, ask, delete, etc. routines for the "A" type device that can work with all different models.
- Implement an automatic method for accepting a user's specification of parameters, defaults, etc. and mapping that specification into the data structures required by SPICE 3C1 to introduce a new model type for the "A" type device.

A careful study of the SPICE 3C1 program execution flow, functions, and their interaction with data structures was made. The intent of this study was to identify which routines must be modified/added, and the nature of the modifications required.

The following partial call-tree summarizes the major functions that require modifications. A brief description of the modification is provided to the right of the function name. Functions without descriptions are included for clarity only, and are not expected to need modification.

```
inp_spsource()
  inp_readall()
  inp_getopts()
  inp_subcktexpand()      Replace all '(' and ')' with white space
    do_it()
    modtranslate()
    devmodtranslate()      Add code to translate 'A' devices
    translate()            Modify to handle 'A' devices explicitly
    numnodes()
      inp_numnodes()      Return 0 for A devices
    numdevs()
```

```
inp_dodeck()
if_inpdeck()
INPtabInit()
INPpas1()
INPdomodel()    Change final else to look for code model
INPtypelook()
INPmakeMod()
INPpas2()        Add code to call INP2A for 'A' devices
MIF_INP2A()      New routine to parse general syntax
INPkillMod()
if_setndnames()  Add code to get names from A devices
```

### 3.2.1.5.4 Matrix Load Equations

This section provides a brief outline derivation of the equations involved in loading values into the SPICE matrix for the different analog input/output port types. Additional information on the matrix formulation can be found in SPICE2: A Computer Program to Simulate Semiconductor Circuits by Laurence W. Nagel.

#### 3.2.1.5.4.1 Newton-Raphson Iteration

SPICE solves simultaneous non-linear circuit equations using Newton-Raphson iteration. In this technique, each non-linear relation in the circuit is approximated by a Taylor series expansion truncated after the first derivative term. The result is a set of linear equations that can be solved using standard matrix-based techniques. The Newton-Raphson formulation begins with a guess as to the solution of the unknown circuit variables (usually zero is a good first guess), and then linearizes the equations about this operating point and solves the matrix. The solution of the linearized system then becomes the next guess and the process is repeated until the unknown circuit variables converge to a solution.

The linearized system of equations setup at each iteration can be expressed as:

$$AX = B$$

where  $A$  is the matrix,  $X$  is the vector of unknown circuit variables, and  $B$  is the "right-hand side" or forcing vector.

#### 3.2.1.5.4.2 Modified Nodal Analysis Formulation

There are various formulations that can be used to express circuit equations depending on which circuit variables are taken as the unknowns. The formulation used in SPICE is called "Modified Nodal Analysis" or MNA.

In the MNA formulation, the unknown circuit variables are taken to be the voltages at each node in the circuit and the currents through each voltage-defined branch (voltage source or inductor). These “node voltage” and “branch current” equations can be expressed in the form:

$$\begin{bmatrix} Y & B \\ C & D \end{bmatrix} \begin{bmatrix} V \\ I \end{bmatrix} = \begin{bmatrix} J \\ E \end{bmatrix}$$

where the matrix and vectors have been partitioned to show the distinction between the node voltage equations at the top of the matrix and the branch current equations at the bottom of the matrix. For each node in the circuit, there is one row (or equation) in the matrix expression. Likewise, for each voltage source in the circuit there is one equation (row). Hence, the matrix contains  $N_n + N_v$  rows where  $N_n$  is the number of nodes in the circuit and  $N_v$  is the number of voltage source branch currents.

The vector of unknowns is divided into the unknown node voltages  $V$  and the unknown branch currents  $I$ . The right hand side (RHS) of the matrix formulation is split into the current forcing vector  $J$  for the node voltage equations, and the voltage forcing vector  $E$  for the branch current equations. In the simplest case, these forcing vectors correspond to the independent current sources and independent voltage sources in the circuit respectively.

The rows corresponding to the node voltage equations must be setup to satisfy Kirchhoff's current law (KCL), and the rows that follow, corresponding to the branch current equations, are setup to satisfy Kirchhoff's voltage law (KVL). For the node voltage equations, the left hand side (LHS) of the matrix formulation expresses currents leaving the nodes, and the right hand side (RHS) expresses currents entering the nodes.

### 3.2.1.5.4.3 Matrix Load Functions

At each iteration during an analysis, SPICE calls a “load” function for each device in the circuit. This load function is responsible for looking in the solution vector (confusingly called “rhsOld” for historical reasons in the SPICE code) from the previous iteration, and using the voltages/currents in this solution vector to compute a new linearized model of the device. The linearized model, consisting of the model outputs and partial derivatives (Taylor expansion) is then used to build the matrix and RHS for the new iteration.

In XSPICE, the task of accessing inputs and building the matrix and RHS is performed by the MIFload() function for all code models. The following two subsections outline the derivation of the KCL and KVL equations expressed in MIFload().

### 3.2.1.5.4.4 Functions with Current Source Outputs

Consider a code model with a current source output. In general, this output may be controlled by a combination of zero or more currents and voltages in the circuit. For simplicity we derive only the equation for an output that depends on one differential voltage plus one branch current in the circuit.

The current source output for our example case draws current from node h and sources current to node i. It takes a differential controlling voltage input consisting of the voltage difference between nodes j and k, and a branch current input consisting of the current through some voltage source whose equation is at row l.

To develop the desired matrix load equations, we need some additional notation to apply to the matrix formulation defined above. Let superscripts correspond to the iteration being evaluated, and subscripts correspond to the row number (equation number) in the matrix. Then, writing the equation for the current source output as if it were an independent source being applied to the RHS, we have for equation i:

$$J_i^n = F(V_j^{n-1} - V_k^{n-1}, I_l^{n-1}) + p_1 [(V_j^n - V_k^n) - (V_j^{n-1} - V_k^{n-1})] + p_2 (I_l^n - I_l^{n-1})$$

where  $F$  is the code model's computation of the output at the current iteration  $n$  based on the solution vector at the previous iteration  $n - 1$ , and  $p_1$  and  $p_2$  are the partial derivatives from the voltage and current inputs respectively.

Rewriting this expression and grouping terms with like superscripts, we have:

$$J_i^n = [F(V_j^{n-1} - V_k^{n-1}, I_l^{n-1}) - p_1 (V_j^{n-1} - V_k^{n-1}) - p_2 I_l^{n-1}] + p_1 (V_j^n - V_k^n) + p_2 I_l^n$$

The term in brackets represents data that is constant at the current iteration and therefore must be placed in the RHS. The remaining terms are dependent on the unknown solution vector at the current iteration and go into the matrix (with signs appropriately reversed).

The same equation, with signs reversed, is used to describe the expression for row h, from which current is drawn.

### 3.2.1.5.4.5 Functions with Voltage Source Outputs

Now consider a controlled voltage source whose positive output is connected to node h and whose negative output is connected to node i. Let the equation for this voltage source reside at row m of the matrix. As before, the inputs will be a single differential voltage from node j to node k, and a current through some voltage source in the circuit whose equation is at row l of the matrix.

For this case, three matrix rows are affected. Rows corresponding to nodes h and i must be modified to take into account the fact that current now flows through the newly introduced voltage source. The equation for row m, corresponding to the new branch equation created by the voltage source, is also involved, of course.

The equations at rows h and i are modified by adding in +1 and -1 entries at the appropriate matrix row and column locations. The equation for row m can be derived through arguments similar to those used in the previous section. The result is:

$$V_k^n - V_i^n = [F(V_j^{n-1} - V_k^{n-1}, I_l^{n-1}) - p_1(V_j^{n-1} - V_k^{n-1}) - p_2 I_l^{n-1}] + p_1(V_j^n - V_k^n) + p_2 I_l^n$$

The term on the left side of the equality is implemented by placing +1 and -1 entries at columns k and i of row m in the matrix respectively. As before, the terms in brackets are dependent only on the previous iteration and go into the constant RHS vector. The remaining terms depend on the unknown vector at the current iteration and go into the matrix.

### 3.2.1.5.5 Analog Simulation States and Integration

This section documents the design for the State/Integration implementation.

The following functions are provided for code models to call:

```
void *cm_analog_alloc(int tag, int size)
void *cm_analog_get_ptr(int tag, int timepoint)
int cm_analog_integrate(double integrand, double *integral, *partial)
int cm_analog_converge(double *state)
```

With the exception of cm\_analog\_converge() these functions have been discussed in earlier sections.

The new cm\_analog\_converge() function is intended to tell the simulator that its argument should be subjected to the same convergence criteria as node voltages are. Like the cm\_analog\_integrate function's "integral" argument, the argument to cm\_analog\_converge() must be a "double" allocated using cm\_analog\_alloc(). Note that cm\_analog\_integrate() will automatically call cm\_analog\_converge() for its "integral" argument, so that the internal integral states of the S-Domain Transfer function will be properly converged.

The return values from cm\_analog\_integrate() and cm\_analog\_converge() are "int"s. They return 0 if they are successful, and non-zero if not. Failures would result from things such as supplying arguments ("integral" or "state") that were not allocated using cm\_analog\_alloc().

If there is an error, an error string address will be set in g\_mif\_info so that the caller can determine and print the cause of the error if desired. The error string address can be retrieved by the code model using function char \*cm\_message\_get\_errmsg().

The `cm_analog_alloc()` function allocates space in the same state vector that SPICE 3C1 uses for its capacitors and other devices. This decision was made because SPICE 3C1 already takes care of rotating the state vector as needed during DC Transfer (Swept DC) and Transient analysis.

An analysis of the 3C1 code indicates that state storage is allocated by `CKTsetup()` just after calling all of the device setup functions. The device setup functions increment `ckt->CKTnumStates` according to the number of doubles required. `CKTsetup()` then allocates `(ckt->CKTmaxOrder + 1)` state vectors required for rotation, each containing `ckt->CKTnumStates` doubles. The states are freed by `CKTdestroy()`.

The code model interface is defined such that the code models call `cm_analog_alloc()` on the first pass through the model (when INIT is true). This differs from the way SPICE 3C1 devices allocate states in a separate "setup" pass. Therefore, a decision was required on when the code model states will actually be allocated.

The only functions that manipulate the state vectors as a whole after they are allocated are `DCtrCurv()` and `DCtran()`. Both of these function's manipulations are limited to calling `bcopy()` one time to copy `CKTstate0` to `CKTstate1`, and rotating the `CKTstate[i]` pointers after each analysis point.

There are two choices for where to allocate states requested by the code models. One choice is to allocate them from within the `CKTsetup()` routine immediately after this routine allocates storage for SPICE states. This could be done by calling all the code models with dummy input arguments and INIT set to true, and ignoring the code model outputs and partials. The other choice is to wait until the first time `MIFload()` is called by `DCtran()` or `DCtrCurv()`.

The second option was selected because the arguments to the code model are already setup at that point. The manipulations of the states by the CKT routines are sufficiently benign to allow states to be incrementally REALLOC'ed for each code model instance during this first call to `MIFload()`. Some penalty for large circuits may result from realloc'ing large areas of memory, but the memory usage should be less than double.

Since `cm_analog_alloc()` does not require its caller to supply the `ckt` struct or the instance struct of the model currently being processed in its argument list, this data needs to be retrieved by the `cm_analog_alloc()` and other support functions through some other means. The `MIFload()` function is therefore modified to set these items in the `g_mif_info` global data structure before calling each user created code model C function. Use of this global simplifies the code model interface support functions (`cm_analog_alloc`, `cm_analog_integrate`, ...) by not requiring them to include `ckt` and the instance struct as arguments, and prevents the necessity to modify arguments of existing SPICE functions.

### 3.2.2 Code Model Support

The SIM-CM CSC contains support functions callable from code models. Several functions in this CSC (such as complex math support functions) have no interface to the simulator or to the other new CSCs. Others interface only to the new SIM-IPC CSC to allow code models to return data to the ATESSE Simulator Interface process. Still others (such as the state allocation and integration functions) interface directly with SPICE 3C1 code and/or data structures, and with new XSPICE SIM-MIF and SIM-EVT CSC functions to provide code modeling capabilities.

These code model support functions are grouped as shown below and are described in the following sections.

- Adding breakpoints
- Outputting error/warning messages to the Simulator Interface process
- Allocating local state storage
- Accessing local state storage
- Truncation error timestep control
- Performing numerical integration
- Performing complex number calculations
- Computing partial derivatives

#### 3.2.2.1 Adding Breakpoints

The XSPICE Requirements Specification calls for the ability to add temporary and permanent breakpoints from within a model. This functionality is provided by the following two functions.

```
void cm_analog_set_temp_bkpt (double time)
void cm_analog_set_perm_bkpt (double time)
```

Both functions take a breakpoint time as an argument.

#### 3.2.2.2 Output of Messages

The XSPICE Requirements Specification calls for the ability to do error reporting from within an executing code model. This functionality is provided by the following function.

```
void cm_message_send (char *msg)
```

This function takes an arbitrary text string as argument. The function then tags the message with the name of the instance in which the message originates, and writes the message to stdout.

### 3.2.2.3 Allocating Local Storage

The XSPICE Requirements Specification calls for a facility that allows each instance to have local storage in which state information can be saved between iterations and/or between analysis points. This storage is required since the code for different instances of the same model is shared.

The requirement for storage between successive iterations is satisfied by the optional Static Variable table in the Interface Specification. The functions defined below are designed to satisfy the need for storage of state information from which values at previous timepoints need to be retrieved.

There are a number of alternative ways of providing for this storage. For example, the SPICE 3C1 simulator provides a simple mechanism where a device model can request and access storage from a central pool organized as a large, single dimensional array of double precision values.

A better way to handle this problem is to provide an allocator similar to the ANSI library "malloc()" function. The advantage is that space may be allocated for arbitrary data types, and by defining appropriate variables and structures, can be accessed by name rather than an index into a non-descript array. With these thoughts in mind, the following local storage allocator is defined.

```
void *cm_analog_alloc (int tag; int size)
```

This allocator takes a tag which allows multiple allocations to be made for a single model, and the size in bytes of the storage to be allocated (computed by the C language "sizeof()" operator), and returns a generic pointer to space allocated.

As an example, assume that a code model needs space for an integer and a double. The code model developer could then define a structure such as:

```
typedef struct data_s {
    int i;
    double d;
} data_t;
```

and allocate and use space as:

```
#define MYSTATE 1
...
data_t *store;
...
if(INIT) {
    store = cm_analog_alloc(MY_STATE,sizeof(data_t));
    store->i = 0;
    store->d = 0.0;
}
...
store->i += n;
store->d += x;
```

As noted in the XSPICE Requirements Specification, the allocated storage space must be rotated through old timepoints as is done with space allocated in the SPICE 3C1 mechanism. This allows for numerical integration, accessing values at previous timepoints, etc. Hence, the underlying mechanism for the `cm_analog_alloc()` function allocates space in the existing SPICE 3C1 “double” valued state vectors.

Because of this rotation of state storage, it is important to note that a structure for which space is allocated should not contain other malloc’ed data. If multiple arrays with dynamically determined dimensions need to be allocated, each should be given a separate tag and allocated independently.

### 3.2.2.4 Accessing Local Storage

Once storage is allocated, a code model needs a way of retrieving the pointer to the allocated space on subsequent iterations and/or analysis points. This functionality is provided by the following function.

```
void *cm_analog_get_ptr(int tag; int n)
```

This function takes the integer tag used in allocating the space, and an integer from 0 to 1 which specifies the timepoint at which the state is to be retrieved, and returns a generic pointer to the previously allocated space. If n is 0, the structure is for the current analysis time. If n is 1, the structure contains data stored at the last accepted timepoint. An example of using this function is provided below.

```
typedef struct data_s {
    int     i;
    double d;
} data_t;
```

```
...
data_t *store;
data_t *oldstore;
double time_deriv;
...
if(INIT) {
    store = cm_analog_alloc(MY_STATE,sizeof(data_t));
    store->i = 0;
    store->d = 0.0;
}
else {
    store    = cm_analog_get_ptr(MY_STATE,0);
    oldstore = cm_analog_get_ptr(MY_STATE,1);
}
...
store->d = <some computation>;
time_deriv = (store->d - oldstore->d) / (T(0) - T(1));
```

### 3.2.2.5 Truncation Error Timestep Control

The XSPICE Requirements Specification calls for models that do numerical integration to have some way to control the timestepping process to limit truncation error. This requirement means that a model should have the ability to cut the timestep according to some estimate of truncation error in the integration. This requirement is satisfied by the `cm_analog_set_temp_bkpt()` function described above.

### 3.2.2.6 Integration

Although not specifically called out by the truncation error timestep control requirement, a function should be provided which makes numerical integration and truncation error estimation / timestep control as easy as possible. The following function is defined:

```
void cm_analog_integrate (double integrand;
                           double *integral; double *partial)
```

This function takes as input the integrand (the input to the integrator) and the current estimate of the integral (the output of the integrator), and produces as output a new integral value (new output of the integrator) and the partial of the integral with respect to the integrand. The integration itself is assumed to be with respect to time, which the function has access to from some global input.

The function `cm_analog_integrate()` is not a simple trapezoidal integrator. Instead, it performs integration according to the trapezoidal or gear method selected, and also takes care of computing and controlling allowed timestep size according to truncation error algorithms.

For this to occur, `cm_analog_integrate()` needs access to previous timepoint integral values. Hence, the formal argument “`*integral`” is assumed to point to a value in storage allocated by `cm_analog_alloc()`. This allows `cm_analog_integrate()` to access up to 7 previous states by computing pointers to the data in the rotated state vectors maintained by SPICE 3C1. For example, to access the value of `*integral` at the preceding timestep, `cm_analog_integrate()` would do:

```
*(ckt->CKTstate1 - ckt->CKTstate0 + integral)
```

The `cm_analog_integrate()` function has been found to work well for a simple integrator and for implementing the new capacitor/inductor models with initial conditions. Since the integrand is the value of the input at each iteration, the function has predictor/corrector properties similar to those used in the standard SPICE capacitor/inductor implicit integration method. However, further investigation is recommended to assure that the function will have good stability and convergence properties.

### 3.2.2.7 Complex Math Computations

To simplify coding of AC analysis models, several basic computational functions are required for complex number arithmetic. The basic functions are listed below.

```
Complex_t cm_complex_set (double real, double imag)  
  
Complex_t cm_complex_add (Complex_t x, Complex_t y)  
Complex_t cm_complex_sub (Complex_t x, Complex_t y)  
  
Complex_t cm_complex_mult (Complex_t x, Complex_t y)  
Complex_t cm_complex_div (Complex_t x, Complex_t y)
```

### 3.2.2.8 Automatic Partials

One of the problems with adding analog code models to SPICE-type simulators is the need for partial derivatives of outputs with respect to inputs. Partial derivatives are required by the simulator to solve non-linear simultaneous equations through Newton-Raphson interaction.

The XSPICE user has the option of coding partial derivatives directly, or of requesting XSPICE to automatically compute them through a call to `cm_analog_auto_partial()`. In the latter case, the function `MIFauto_partial()` (called from function `MIFload()`) will call the model N additional times, where N is the number of inputs. At each additional call, a single input to the model is varied by a small amount, and the partial of each output with respect to that input is approximated by divided differences.

In the implementation of this mechanism, function MIFauto\_partial() actually calls the code model N+1 additional times. At the end of varying the individual inputs, an additional call is made with all inputs restored to their nominal values. This is done to assure that if the model computes and stores internal states, etc., these states will be left at the values for the nominal case.

Note that the amount the inputs are varied must be small so that the divided difference computation will yield a good approximation to the true derivative. Otherwise, convergence problems may result. On the other hand, if the amount is too small, numerical truncation errors will result. The amount by which the input is varied is currently set equal to the convergence tolerance used by the analog simulation algorithm, and is different for current and voltage signals. For voltage signals, the amount is currently set to 1e-6 volts, and for currents, 1e-12 amps is used. Assuming that the target hardware platform on which the simulator is running uses 16 significant digits in its representation of double-precision floating point values, this allows for voltages and currents to go as high as 1e8 volts and 1e2 amps respectively while maintaining less than one percent error resulting from numerical truncation.

In future versions of the simulator, these fixed values may be modified so that they grow for large inputs. This is the approach used in SPICE in computing convergence tolerance, where "absolute" and "relative" tolerance values are used.

### 3.2.3 Event-Driven Simulation

The SIM-EVT CSC implements an embedded event-driven simulation capability that coordinates with the matrix-based, timestepped algorithm used by SPICE. This new event-driven simulation capability represents simulation data at a higher level of abstraction than normal SPICE analog nodes, allowing for orders-of-magnitude speed improvements over the standard SPICE algorithm for digital and other discrete time models. The algorithm is implemented separately from the representation of the data passed between code models so that multiple data types can be defined. This facility is called "user-defined nodes". New node types can be introduced by a user in a way analogous to the introduction of user-defined code models.

The timestepping of the SPICE analog simulation algorithm and the event timing of the event-driven simulation algorithm are coordinated through a set of support functions in the SIM-CM CSC. Special hybrid code models can be written to convert from the analog representation to an event-driven representation, or between event-driven representations. When such hybrid models are written expressly for the purpose of converting data types, they are referred to as "node bridges".

### 3.2.3.1 Circuit Parsing

Code models that function with event-driven node types use the same Code Model Toolset and circuit description syntax as those written for use with analog nodes. Therefore, the previously discussed Model Interface (SIM-MIF) CSC performs parsing of the circuit description instance and .model lines in the circuit description.

### 3.2.3.2 DC Convergence

The following algorithm is used in a DC operating point analysis or in the initial DC solution of a transient analysis:

- Initialize all event-driven nodes.
- Iterate all event-driven and hybrid (mixed analog and event-driven) models to solution. This allows event-driven models with initial conditions to post their outputs.
- Do analog DC convergence of analog/hybrid models. This allows the analog solution to create inputs to event-driven circuits.
- Iterate all event-driven/hybrid models to solution. This allows event-driven models to establish their outputs given the inputs established by the analog solution.
- If event-driven outputs changed, go to step 2

This algorithm has the advantage of keeping the two convergences separate, so that complexity is reduced. The cost is multiple iterations of the analog solution. A limit is placed on the number of times the algorithm can alternate between the event-driven and analog iterations. The normal default for this limit is the number of hybrid models in the circuit, since the worst case in the absense of loops would be for a sequence of analog circuitry followed by event-driven, followed by analog, etc. An even worse case would be for an analog/event-driven loop, but this will be reported as an error if it causes more than the allowed number of alternations.

Note that for the special (but common) case of

```
event-driven -> da_node_bridge -> ad_node_bridge -> event-driven -> etc.
```

the subsequent analog solution will normally require only two iterations for convergence since the only circuitry needing to converge is the feedback-free output/input connection of the two node bridges.

An alternative to the method outlined above would be to attempt to merge (or interleave) the analog continuation methods (GMIN stepping and/or source stepping) with the digital iteration. A difficulty is created by the abrupt steps that would occur when digital outputs change in the middle of the analog continuation method - making the method non-continuous. An exploration of solutions to this problem indicated the need for the hybrid models to perform their own continuation method solution of their outputs whenever their outputs change, making the hybrid models possibly unnecessarily complex to code. It is also not clear that this approach would result in better or faster convergence than the simpler algorithm above.

### 3.2.3.3 Transient Analysis Algorithm

The event-driven algorithm cuts down on simulation time by only evaluating models for which inputs have changed. Moreover, event times are independent from the analog timesteps, so that between two analog timesteps, several event times may occur.

A problem occurs when the analog simulation algorithm attempts a solution for a particular timestep and then decides it must decrease that timestep and attempt a new solution at an earlier time because the analog nodes/branches did not converge, or because local truncation error is too large for some integration. This backup and retry means that any events computed between the last accepted analog timepoint and the timepoint just rejected must be discarded. This implies that the event queues and states must be fully saved each time an analog timepoint is accepted, and that events computed after the accepted timepoint, but before the next analog timestep must not be output to Nutmeg or the ATESSE Simulator Interface process until the next timepoint is accepted.

Based on this analysis, the following algorithm is used for coordinating event times with analog timesteps.

```
while(not end of analysis) {  
  
    while (next event time <= next analog time) {  
  
        Do event solution, evaluating all event-only  
        and hybrid models. Analog outputs computed by  
        hybrid models are ignored - however, model may  
        force a new analog timestep by calling a  
        code model support function.  
  
    }  
  
    Evaluate all hybrid models. Event outputs are  
    recorded and may add events to the queue.  
  
    Do analog convergence iteration.
```

```
if(analog solution doesn't converge) {
    Discard intermediate events.
}
else {
    Output event results to Nutmeg or Simulator
    Interface process.
}
}
```

### 3.2.3.4 AC Analysis

Event-only models are not called in an AC analysis. Hybrid models will be called if they have both analog and event outputs on the same model, allowing AC gains to be defined for the analog paths.

### 3.2.3.5 Sources

Input of event data may be effected by a code model that reads a file of time/event pairs and uses the SIM-CM support functions to queue events as appropriate.

### 3.2.3.6 Display of Results

Event results (node states vs time) are output through a new "eprint" command added to the Nutmeg interface. The output format is shown below, and is designed to allow simple cut/paste operations to be performed to turn outputs into an equivalent input file. Notice that the output times are not equally spaced because of the event-driven nature of the simulation.

time	node_1	node_2	node_3
0.00000	Uu	Uu	Uu
1.234e-9	0s	1s	1s
1.376e-9	0s	0s	1s
2.5e-7	0s	0s	0s
2.5006e-9	Uz	0s	0s

### 3.2.3.7 Digital States and Strengths

The XSPICE simulator supports a pre-defined event driven node type that defines 12-state digital modeling.

Supported states include:

ZERO  
ONE  
UNKNOWN

Supported strengths include:

STRONG  
RESISTIVE  
HI\_IMPEDANCE  
UNDETERMINED

A node resolution algorithm is used to determine the final value obtained by a node when multiple models output to that node.

### 3.2.3.8 Access Macros

The following accessor macros may be used in digital models:

INPUT	OUTPUT
INPUT_STATE	OUTPUT_STATE
INPUT_STRENGTH	OUTPUT_STRENGTH
OUTPUT_DELAY	
LOAD	
TOTAL_LOAD	
MESSAGE	

#### 3.2.3.8.1 Input and Output Macros

```
void *INPUT(<input conn/port>)
void *OUTPUT(<output conn/port>)
```

The INPUT and OUTPUT macros provide the basic method of passing user-defined data to and from an event-driven or hybrid code model. The model will normally assign the returned pointers to local data structures for the particular node type under consideration and then proceed to access members of the structure as appropriate.

For the special case of digital nodes, the following special macros are also provided as a convenient alternative to use of INPUT and OUTPUT.

```
Digital_State_t    INPUT_STATE(<input conn/port>)
Digital_State_t    OUTPUT_STATE(<output conn/port>)
Digital_Strength_t INPUT_STRENGTH(<input conn/port>)
Digital_Strength_t OUTPUT_STRENGTH(<output conn/port>)
```

These macros provide direct access to the two members of the Digital\_t structure. For example, the macro INPUT\_STATE(x) is logically equivalent to:

```
((Digital_t *)(INPUT(x)))->state
```

### 3.2.3.8.2 Output Delay Macro

```
double OUTPUT_DELAY(<output conn/port>)
```

The OUTPUT\_DELAY macro is provided to assist models in creating simple delays. This macro is intended to allow basic gate models to be written without requiring use of “alloc” and “get\_ptr” functions and “breakpoint” setting functions. The macro should be used together with the OUTPUT macro (or OUTPUT\_STATE and OUTPUT\_STRENGTH macros, and causes the output data to be delayed from reaching the connected node by the value of time specified (in seconds).

If a new output is posted before a previous output has reached its connected node, the output transitions are determined as follows. If the new output transition would occur before the old transition, the old transition is removed from the event queue. If the new transition would occur after the old transition, both transitions will occur at the appropriate times.

### 3.2.3.8.3 Loading Macros

```
double LOAD(<any conn/port>)
double TOTAL_LOAD(<output conn/port>)
```

The LOAD and TOTAL\_LOAD macros are provided to allow modeling of changes in delay with fanout. The LOAD macro may be assigned a value by a model according to the load that the associated port places on the connected node. This assignment must be made during the model evaluation pass for which INIT is true.

The simulator will then compute the total loading on each node and assign the associated value to TOTAL\_LOAD. The model may then use the value of TOTAL\_LOAD at any pass after INIT to compute and output appropriate load-determined delays.

### 3.2.3.8.4 Output Message Macro

```
char *MESSAGE(<any conn/port>)
```

The MESSAGE macro is provided to allow models to output warnings and other informative messages. Such messages might include timing violations (e.g. insufficient setup time on an input), or watch events. The simulator copies the message to its own space and appends the name of the instance and the port. The messages become part of the output data for the timestep and may be viewed in output traces.

### 3.2.3.9 Support Functions

The event-driven simulation algorithm is essentially independent of the analog algorithm and takes its own independent timesteps. Therefore, many of the analog support functions (e.g. cm\_analog\_alloc, cm\_analog\_set\_temp\_bkpt, etc.) cannot be used with event models. The following support functions are provided for use by event models:

```
int      cm_event_queue(double time)
void*   cm_event_alloc(int tag, unsigned size)
void*   cm_event_get_ptr(int tag, int timepoint)
```

Note that since the algorithm is event-driven, there is no concept of separate temporary and permanent digital breakpoints. Instead, all event breakpoints are permanent and will cause the simulator to enter the calling model onto the event queue for the posted time. The time must be greater than the current event time.

### 3.2.3.10 Use of NULL in Vectors

Writing certain digital models is simplified by allowing the NULL keyword to be used on device connection lists inside of arrays. Initially NULL was allowed only on vector port connections as a whole, not on individual sub-ports. This restriction has been removed.

## 3.2.4 Enhancements

Several general enhancements are made to the SPICE 3C1 simulator to address board-level and system-level simulation requirements, and to provide improved error reporting, including:

- Arbitrary starting phase on time-varying independent sources in a transient analysis.

- Automatic translation of SPICE 2G6 style polynomial controlled source syntax to syntax for calling an internal polynomial controlled source code model.
- Support for dynamically introducing temporary and permanent breakpoints into the analog timestepping algorithm from analog/hybrid code models during a transient analysis.
- Supply ramping in a transient analysis to simulate the turnon of power supplies and other independent sources to the circuit.
- Convergence debugging support that helps identify the node/branch/device responsible for the convergence failure.
- Automatic “simple-limiting” applied to inputs of code models to assist convergence.
- Optional insertion of large valued resistors to ground at all nodes to prevent problems in circuits that have nodes with no DC paths to ground.

The addition of arbitrary phases to independent sources is implemented by modifications made solely to the SPICE 3C1 VSRC and ISRC model “load” functions. The remaining modifications are made through modifications to various locations in the SPICE 3C1 simulator code, and through the introduction of a new “enh” data structure as a sub-element of the SPICE 3C1 CKTcircuit data structure.

### 3.2.4.1 Adding new options

New options can be added by making appropriate modifications to CKT/CKTsetOpt.c and include/OPTdefs.h. CKTsetOpt.c should be modified to add an appropriate case statement to function CKTsetOpt() and to add a line in the data structure OPTtbl[]. OPTdefs.h should be modified to add a new #define “OPT\_...” for the new option’s internal identifier used in CKTsetOpt.

It should be noted that CKTsetOpt() normally assigns option data into a special “task” structure, and that the values placed there are later copied to the ckt structure in CKTdoJob(). This two-step process of assigning option data does not appear to serve any purpose in the simulator (see the call tree). Therefore, to simplify additions of new options, the new options will be assigned directly into the appropriate data structures without intermediate assignments to the task structure. This has the advantage of decreasing the number of modifications made to the SPICE 3C1 code. Otherwise, modifications would need to be made to TSKdefs.h, CKTnewTask.c, and CKTdoJob.c. If new releases of the simulator make use of the task structure, these additional file modifications can be made in the future.

### 3.2.4.2 Arbitrary Phase Sources

The SPICE 3C1 core is modified to support time-varying sources with non-zero phase in order to make the simulator more applicable to board-level and system-level work. This change can be meaningfully applied to the following three source types:

- SIN
- PULSE (when used to create periodic waveforms)
- SFFM

The SIN and PULSE sources are defined in SPICE with a “delay” parameter, but this parameter is not defined in a way that allows waveforms to begin at arbitrary phase. Instead, the delay simply keeps the source value at the “time-zero” value until the delay has elapsed.

To provide arbitrary phase, a new parameter is introduced. The new parameter specifies the initial phase of the waveform in degrees. For compatibility, this new parameter is added AFTER all existing parameters and will default to zero if not supplied.

The device types in SPICE 3C1 affected are V and I independent sources.

#### 3.2.4.2.1 Parsing Modifications

No modifications are required to the parsing code. Both the V and I sources take a single named parameter (e.g. SIN, PULSE, etc.) which is expected to be followed by a list of reals. This list of reals constitutes the coefficients for the source and is not named. Nor is its size specified in advance. The parsing code simply reads as many parameters as there are.

#### 3.2.4.2.2 Device Function Modifications

The device code in both DEV/VSRC and DEV/ISRC is modified to support the new functionality. For both of these devices, the functions affected include XXXload.c and XXXaccept.c. Function XXXload.c computes the value of the source at a given time. Function XXXaccept sets up the appropriate dynamic breakpoints (for the PULSE type only at version 3C1).

In the computation of waveform values for sin and pulse waveforms in SPICE 3C1, the code in these functions subtracts the time delay from the current time and outputs the time-zero value if the result is  $\leq 0.0$ . Otherwise, it computes the appropriate output based on the time ( $> 0.0$ ). This code is modified to the equivalent algorithm below:

```
subtract delay from time
if < 0
    set time = 0
compute waveform value based on time
```

The phase value is then added by inserting the following code after limiting the time to zero, but prior to the computation of the waveform value:

```
normalize phase to 0 - 2pi
convert phase to delta-time based on period of waveform
add delta-time to time
```

This algorithm gives results equivalent to the SPICE algorithm when PHASE is zero. When phase is non-zero, the time-zero value will be the value of the source at that phase. If there is a non-zero delay, the source will sit at the value given by the phase until the time has elapsed.

### 3.2.4.3 Supply Ramping

The XSPICE Requirements Document calls for the addition of supply ramping to the transient analysis mode to allow more accurate simulation of certain board-level circuits. The modifications made to the code are:

- Add a new option keyword “ramptime”. The new keyword sets the ramp time over which the supplies are brought from 0.0 to full voltage.
- Add new data item in ckt->enh.
- Modify CKTinit.c to initialize ramptime in ckt->enh to 0.0. This must be done explicitly (even though the memory is calloc’ed) because the zeroing of the memory produces a double which is not exactly 0.0 and compares will be made of the ramptime to 0.0.
- Modify DCtran.c to set a breakpoint at the end of the ramptime if ramptime != 0.0.
- Create a new cm\_analog\_ramp\_factor() function that returns a value from 0.0 to 1.0 and can be used by models that wish to participate in the ramping. This

function will set the factor to 1.0 if the analysis mode is not transient or if the ramptime was not set by a .options card (ramptime defaults to 0.0). Otherwise, the function will return 0.0 during MODETRANOP, a value between 0.0 and 1.0 during the ramptime, and 1.0 thereafter.

- Modify VSRC and ISRC to call cm\_analog\_ramp\_factor() and multiply their outputs by the factor.

### 3.2.4.4 Convergence Limiting

SPICE simulators use Newton-Raphson iteration to solve non-linear simultaneous equations. This process involves guessing the solution, linearizing the elements in the circuit, and constructing and solving a matrix expressing the linearized circuit approximation. The process is then repeated until the circuit variables stabilize or “converge”.

For certain types of transfer functions, this process can be dangerous unless some form of limiting is employed. For example, the simple case of a high voltage source in series with a resistor in series with a diode causes problems. At the first iteration, the guess is 0.0 volts across the diode. The linearized approximation for the diode at this operating point is a very high valued resistance. Hence, the solution of the linearized circuit will produce nearly the full value of the high-voltage source across the diode at the beginning of the next iteration. This can cause either an overflow error because of the exponential behavior of the current with respect to the applied voltage (the diode is modeled as a “g” type port), or a false convergence because of the comparison of the very large numbers that result. A similar but worse situation can occur for the case of a circuit consisting of a current source in parallel with a diode and a very high resistance.

In the development of SPICE, a number of methods were investigated for dealing with this problem, ranging from “simple limiting” where the voltage across the diode is prevented from changing too rapidly from iteration to iteration, to the “Colon” method, also known as “iteration on current/voltage” where the inverse of the transfer function is used.

In XSPICE, a method which can be automated is desired so that a user is not required to write code to perform the limiting. A simple-limiting method was therefore adopted. The method consists of few lines of code in the MIFload function implementing the following algorithm.

```
if(not first iteration) {  
    if(convergence limiting enabled) {  
        compute max allowable change as a fraction of the last  
        iteration's input value.  
        if computed maximum allowable change is less than
```

```
    some lower limit, reset it to this lower limit
    if the input changed by more than this amount,
    limit its change to this amount
    increment CKTnoncon to prevent false convergence in
    NIiter()
}
}
```

The initial defaults chosen for the max allowable change fraction and absolute limits are 0.1 and 0.25 (25%) respectively. For the case of the initial DC solution before gmin or source stepping, the limiting can reach 8.0E8 in the default maximum of 100 iterations, which should be sufficient for most circuit work.

For larger numbers, the user can increase option “itl1” to 300, which would allow the value to grow to 1.94E28. The default limiting fraction and absolute limit can also be changed by using new “convstep” and “convabsstep” options respectively:

```
.option convstep = <fraction>
.option convabsstep = <absolute limit>
```

In the current version of XSPICE, the same limits are applied to voltages and currents.

### 3.2.4.5 Convergence Debug Reporting

The XSPICE Requirements Specification calls for convergence debugging support to assist the user in determining which part of the circuit is failing to converge. This is provided in the form of messages sent to the Simulator Interface process or to stdout when the final iteration prior to run abort is made. The messages identify which node, branch current, or instance is failing to converge, and possibly what the last two values were.

To implement this functionality, a new data element is added to ckt->enh to hold data used in communicating between functions. Two flags are required:

- A flag “last\_NIiter\_call” to tell NIiter that this is the last time it is called before the run is aborted.
- A flag “report\_conv\_probs” set by NIiter to tell the convergence check functions that this is the last time they will be called, so they should output the debug info if any convergence check fails.

The debug info is output to a special function ENHconvdebug() which then writes it to stdout or to the IPC channel depending on the mode the simulator is in.

When writing to stdout, it outputs a message of the form

```
WARNING: Convergence problems at <type> <name>. <msg>
```

The last\_NIter\_call flag must be set by the analysis functions that control the overall iteration scheme. The top-level analysis functions are ACan.c DCop.c DCtrCurv.c and DCtran.c. All of these functions call CKTop.c or EVTop.c to perform the basic DC solution and it is sufficient to modify only CKTop.c and EVTiter.c (called from EVTop.c) to set last\_NIter\_call. For the special case of a transient analysis after the timestepping has begun, it is necessary to modify DCtran around line 655 to set and reset last\_NIter\_call appropriately depending on how close the delta is to delmin.

The report\_conv\_probs flag must be set inside NIter just before the iterlim given in the function's arg list is exceeded.

The convergence test functions that need to call ENHconvdebug() when report\_conv\_probs has been set include:

```
NIconvTest.c - report problems with voltages and currents.  
EVTiter.c - report problems with changed outputs  
CKTconvTest.c - report problems with device types that fail  
or  
XXXconvTest.c - report problems from individual instances that fail  
where XXX includes all device types that have a convTest function:
```

```
DEV/ASRC/ASRCconvTest.c DEV/BJT/BJTconvTest.c  
DEV/BSIM/BSIMconvTest.c DEV/DIO/DIOconvTest.c  
DEV/MOS1/MOS1convTest.c DEV/MOS2/MOS2convTest.c  
DEV/MOS3/MOS3convTest.c MIF/MIFconvTest.c
```

### 3.2.4.6 Resistors to Ground

A new option "rshunt" is added to circumvent the problem with "no DC path to ground" errors in a circuit description that cause the analysis to fail due to a singular matrix. If the following line is placed in a deck, then virtual resistors of the specified value will be placed from every analog node in the circuit to ground.

```
.option rshunt = 1e12
```

Using a value of 1e12 Ohms is generally sufficiently high that the behavior of the circuit is unaffected, but the matrix becomes well behaved.

This option is implemented in conjunction with the XSPICE enhancements (ENH) package and involves the introduction of a new data structure (rshunt\_data) under ckt->enh, plus the modification of four files in the CKT package:

CKTinit	Initialize the rshunt option
CKTsetOpt	Get the option and value from a .option(s) card
CKTsetup	Setup matrix pointers for the virtual resistors
CKTload	Perform the equivalent of RESload for the resistors
CKTacLoad	Perform the equivalent of RESload for the resistors

The “virtual” resistors are not actually present in the circuit description. Instead they are implemented by adding in the appropriate conductance values to the matrix as the simulation proceeds. This is performed by appropriate modifications to CKTload() and CKTacLoad() which are the driver functions for calling the individual device load functions (such as RESload).

To create the data needed by the modifications to CKTload() and CKTacLoad(), the CKTsetup() function is also modified to first count the number of nodes in the circuit (excluding node 0 (ground)), and then to call SMPmakeElt() to get the appropriate matrix pointer. To place a resistor to ground at a node, only one pointer is needed for the resistor - a pointer to the matrix diagonal for that node equation. Note that SMPmakeElt() must be called rather than making a more direct call to SMPfindElt() since the matrix is a sparse formulation and the deck read may not have actually created an entry in that matrix position.

### 3.2.5 Internal Code Models

The SIM-ICM CSC provides a special “poly” code model that is always included in the simulator. The “poly” code model is provided for compatibility to 2G6 SPICE decks that use a polynomial controlled source. 2G6 poly source syntax is translated automatically into the syntax used by this internal code model.

### 3.2.6 Internally Defined Nodes

The SIM-IDN CSC predefines a special “digital” node type for use with event-driven simulation. The “digital” type is included in the simulator core because special support for it is built into the Code Model Toolkit in the form of macros for input/output of digital states and strengths. This special support is not required for implementation of the digital node type, but it simplifies the writing of digital code models by better isolating the novice user from C language pointers and data structures.

All event-driven node types (user-defined nodes), including the special “digital” type, are treated as data in this document since they can be added by the user in a way similar to adding code models. Refer to chapters 4 and 5 for a definition of the functions involved in implementing node types.

### 3.2.7 Interprocess Communication

The SIM-IPC CSC is responsible for interprocess communication between the simulator and the ATESSE Simulator Interface and Batch Control processes. A compile-time option specifies which version of these ATESSE processes is used. For ATESSE version 1 processes, the communication is done using HP/Apollo's Aegis mailbox facility. For ATESSE version 2 processes, BSD Unix sockets are used.

The SIM-IPC CSC is enabled only if the “-ipc” command line switch is supplied when the simulator is invoked. Arguments to the -ipc switch specify the process with which the simulator will communicate (ATESSE Simulator Interface or Batch Control processes) and the pathname of the Aegis mailbox or BSD UNIX socket used. When the switch is used, the simulator bypasses the standard SPICE 3C1 input/output code and reads the input SPICE deck from the IPC channel, and returns results over the IPC channel rather than saving them in memory or writing them to a file.

The SIM-IPC CSC is implemented in two parts to facilitate porting the simulator to other Simulator Interface communication protocols. IPC protocol-specific calls are localized to the two files “IPCaegis.c” and “IPCsockets.c”. The selection of which IPC protocol is used in a particular environment is determined by a compile-time switch in the Makefile for the IPC subdirectory. Protocol-independent code is contained in files “IPC.c” and “IPCtiein.c”. Functions in these two files are called by various modifications made to the SPICE 3C1 core.

The principal modifications made within the SPICE 3C1 code for interprocess communication include:

- Check the command line arguments for the -ipc switch and initialize the g\_ipc global ipc information structure.
- Initialize interprocess communication from main().
- Reopen stdout and stderr as files in /usr/tmp to collect any warning or error messages written to these streams using printf functions.
- Modify the SPICE deck read function to read from the IPC channel.
- Modify various functions in the parsing and simulation code to return error-check status for compatibility with ATESSE version 1.
- Modify the BeginPlot functions in the OUTinterface.c CSC to setup the necessary structures to output the appropriate node voltages and instance currents during a simulation.
- Modify the different analysis mode functions (CKTop, DCtran, ACan, etc.) to call CKTdump() and EVTdump() to output data at the appropriate points, and to prefix and postfix the results from individual analysis points with delimiters.

- Modify the accounting information functions to record the CPU time used and to always output a listing of the input deck.
- Modify main() to send a completion status message with the CPU time used, and to cleanly terminate the IPC channel.

For further details regarding the IPC protocols and data formats used, refer to the Interface Design Document for the XSPICE Simulator of the Automatic Test Equipment Software Support Environment (ATESSE).

### 3.2.7.1 SPICE 3C1 Program Modifications

The version of XSPICE that works with IPC starts with the batch bspice.c main program rather than the interactive main() in nutmeg.c. To allow all affected files to work either in the standalone spice product or the IPC version, a new global data structure g\_ipc is defined in IPCtiein.c. g\_ipc.enabled is set to true if the batch simulator gets a command line argument:

```
-ipc [ INTERACTIVE | BATCH ] <ipc channel filename or port>
```

All modifications to the SPICE 3C1 program are written around "if" statements of the form if(g\_ipc.enabled) ...

The following modifications are made to the SPICE 3C1 core to add interprocess communications.

Near the top of main() in FTE/bspice.c, scan argv looking for -ipc. If found, get the mode and ipc channel information and set appropriate members in g\_ipc. Then code the processing that needs to be done immediately below this command line argument processing. This new code ends with "exit(0)" so that no other code in main() needs to be modified and so that bspice can be used normally if -ipc is not specified as a command line arg.

The function main() in bspice.c is modified as follows:

```
Create the ipc channel.  
Reopen stdout and stderr as files in /usr/tmp.  
Call inp_spsource(NULL, false, NULL) to read and process the deck.  
Call ft_dotsaves() to get .save cards.  
Call ft_dorun(NULL) to run the analysis.  
Call ft_cktcoms(false) to print accounting information and set g_ipc.runtime.  
Call ipc_send_std_files() to send across everything collected in stderr and stdout.  
Call ipc_send_endanal() to end >ABORTED or >ENDANAL with g_ipc.runtime.  
Close the ipc channel.  
Exit.
```

### 3.2.7.1.1 Modifications to inp\_spsource() Subtree

Modify FTE/inp.c/inp\_readall() to read from the channel if ipc is enabled.

Modify near line 664 of FTE/inp.c/inp\_dodeck() to set g\_ipc.syntax\_error if any of the li\_error lines are printed to stdout.

### 3.2.7.1.2 Modifications to ft\_dot saves() Subtree

None.

### 3.2.7.1.3 Modifications to ft\_dorun() Subtree

Modify CKT/CKTdoJob() to set g\_ipc.syntax\_error if error is returned from calls to CKTsetup() or CKTtemp(). Also, check if g\_ipc.syntax\_error is set after calling CKTtemp() and return error if so. Otherwise, it should call ipc\_send\_errchk() and proceed with calling the analysis functions to perform DC, AC, or TRANSIENT analysis.

Modify FTE/runcoms.c/dosim() to check for error on return from CKTdoJob(). If syntax\_error is true, then call ipc\_send\_errchk(). If any error, set g\_ipc.run\_error so that we can send ">ABORTED <time>" later.

### 3.2.7.1.4 Modifications to ft\_cktcoms() Subtree

Modify FTE/dotcards.c/ft\_cktcoms() so that it calls inp\_list with LS\_PHYSICAL as argument to print a deck listing.

Set g\_ipc.runtime in FTE/resource.c/printres() so that run time can be returned later.

## 3.2.7.2 Sending Results Delimiters

Modify CKT/DCop.c to send >DCOPB, >ENDDCOP delimiters before and after calling CKTdump().

Modify CKT/ACan.c and DCtran.c to call OUTpBeginPlot(), send >DATA, >ENDDATA delimiters, call CKTdump(), and call OUTendPlot to send operating point analysis data. Also, modify to send >DATA, >ENDDATA around the call to CKTdump() or CKTaccDump(). Make sure that the appropriate mode is communicated to beginPlot() before the call.

### 3.2.7.3 Selecting Data to Return

SPICE3 uses a data structure called “runDesc” in package FTE/OUTInterface.c to control what is saved during a simulation. This structure is loaded prior to each separate analysis type by a call to OUTpBeginPlot from each of the main analysis functions (ACan.c, DCtran.c, ...). Since ATESSE version 1 returned different data from DC/TRAN and AC analysis types, this is a good place to setup what to send to IPC based on analysis mode.

The work of creating “runDesc” is done by beginPlot() in FTE/OUTinterface.c. Therefore, selecting data to return over IPC is handled by modifying beginPlot() to load runDesc with only those items needed to be send over IPC when IPC is enabled. The somewhat involved modifications are described below. A complication arises from the way the version 1 ATESSE Simulator Interface process (MSPICE) expects the sign of currents from BJTs, JFETs, and MOSFETs to be modified by +1 or -1 depending on type (npn/pnp, etc.). This problem makes it necessary to add a “double” member (called ipc\_modtype) to the runDesc structure. Fortunately, the use of runDescr is limited to the OUTinterface package so that this does not cause much difficulty. Care must also be exercised in loading the runDesc “name” member with the name that the ATESSE version 1 Simulator Interface process expects.

### 3.2.7.4 Returning Results

With runDesc properly loaded, modifications for sending results are relatively easy through appropriate modifications to OUTpData(). Changes required to this function are restricted to the first major if clause and include:

- Not outputting the reference value (it will have been sent by modifications made to CKTdump()).
- Adding if statements of the form:

```
if(g_ipc.enabled)
    ipc_send_xxx() * run->data[i].ipc_modtype
else
    fileAddxxxValue()
```

Note that this will prevent data from being written to the rawfile so that this file will remain small. It has already been diverted to the /usr/tmp directory where it's creation should be harmless. The choice was made not to tell the simulator to use the “plot” structure instead of the rawfile output because supporting routines for the “plot” structure are absent in the bspice executable.

### 3.2.7.5 Modifications to BeginPlot

As mentioned above, modifications required to beginPlot to select the proper data for return and to build the runDesc structure accordingly are somewhat involved. The complexity is caused by the specific names required for use when returning data to Mspice, and the filtering of data to be returned. The names the data is tagged with and the filtering of what data to be returned must be compatible with the ATESSE version 1 system. Details can be found by looking at the "mcom.ftn" file in the ATESSE version 1 simulator process.

The runDesc structure is built in two major passes (and several minor ones) in the SPICE 3C1 simulator. The two major passes are:

- Specify items from the matrix solution vector to be returned by calling addDesc()
- Specify items from .save cards to be returned by calling addSpecialDesc().

The former can only return node voltages and branch currents of voltage sources. The latter must be used to return things such as currents in I, Q, D, J, M, C, and L type devices.

The standard SPICE 3C1 code used for building the runDesc structure is bypassed and new code is added to put only those items required by the ATESSE version 1 system into the list. The processing required is broken down into two sections below, one for voltages and V source currents and the second for currents from other devices.

#### 3.2.7.5.1 Voltages and V Source Currents

```
For(each equation in matrix)
    Get name
    If((name is numeric)
        and (name doesn't have a ':') /* indicating not part of a subckt */
        and (numeric value is < 100,000) /* indicating not an ms_server modification */
        then
            Call addDesc to add it to the send list
    Else /* must be a voltage source name or other device with current eqn */
        If((begins with 'v') and (doesn't have a ':')) then
            Eliminate the '#branch' suffix added by SPICE3
            If(Name matches syntax of Vtrans type source)
                Map to associated device name in g_ipc.vtrans table
            End If
            Call addDesc to add it to the send list
        End If
    End If
End For
```

### 3.2.7.5.2 Device Currents

Loop through ckt structs for each of the following device types in order: ICLDQJM, doing I sources always and other elements only if g\_ipc.returni is TRUE and CKTmode is not AC analysis.

```
If(name does not contain ':') then
    If(p-type Q, J, M)
        new modtype arg to addSpecialDesc() = -1.0
    Else
        new modtype arg to addSpecialDesc() = 1.0
    End If
    Call addSpecialDesc() with appropriate name, param, and new
    modtype args for as many currents as required by device (see below)
End If
```

Note: p vs n type is decided by looking at the int entry just after the GENmodel prefix in the instance's model. This entry is 1 if n type and -1 if ptype.

### 3.2.7.5.3 Parameter names for device types

The following table gives the parameter names for currents on different devices in SPICE 3C1. The names are ordered according to the output sequence required for ATESSE version 1 compatibility.

I	c
C	c
L	current
D	c
Q	cc cb
J	cd cg
M	cd cs cb

# 4

# Detailed Design

This chapter describes the detailed design of the functions that make up the following CSCs:

SIM-MIF	Model Interface
SIM-CM	Code Model support functions
SIM-EVT	Event-Driven simulation
SIM-ENH	Enhancements
SIM-ICM	Internal Code Models
SIM-IDN	Internally Defined Nodes
SIM-IPC	Interprocess Communications

For each function, a general description of the function and its return value (if any) is given, followed by Program Design Language (PDL) pseudo code.

## 4.1 Model Interface

The Model Interface (SIM-MIF) CSC provides support for code models developed through the Code Model Toolset. The SIM-MIF CSC introduces a new “A” type SPICE deck element card that is used by all pre-defined and user-written code models. The CSC consists of two major parts, a parser and a set of device functions analogous to those used by standard SPICE 3C1 device CSCs (e.g. RES, CAP, BJT, etc.), but shared by all code models.

The parser is composed of the following major functions:

```
MIF_INP2A
    MIFinit_inst
    MIFget_port_type
    MIFget_port
    MIFgetMod
    MIFgetValue
    MIFgettok
    MIFget_token
    MIFget_cntl_src_type
```

The MIF device functions are similar to those used by the standard SPICE 3C1 devices and documented in the Adding Devices to SPICE 3C1 reference. However, they are more generic in the sense that they work with all code model devices rather than with only a single device. These functions currently include support for DC, AC, and Transient simulation, parameter set and query operations, and structure deletion:

```
MIFmParam
MIFmAsk
MIFask
MIFsetup
MIFload
    MIFauto_partial
MIFconvTest
MIFtrunc
MIFmDelete
MIFdelete
MIFdestroy
```

#### 4.1.1 Function MIF\_INP2A

##### Summary

This function is called by INPpas2() in SPICE to parse the new "a" type element cards and build the required circuit structures for the associated code model device and instance. It first checks the model name at the end of the element card to be sure the model was found in pass 1 of the parser. If so, MIFgetMod is called to process the .model card, creating the necessary internal model structure and filling in the parameter value information. Next, the instance structure is allocated. Finally, the connections on the element card are scanned and the connection information is filled-in on the instance structure, and error checks are performed.

##### Called By

SPICE3C1/INPpas2()      INP/INPpas2.c

##### Returned Value

None.

##### Global Variables

SPICEdev \*DEVices[]; /\* IN - Information about models known to simulator. \*/

##### PDL Description

```
void MIF_INP2A(ckt,tab,current)

GENERIC    *ckt;      /* INOUT - Circuit structure to put mod/inst structs in. */
INPtables *tab;      /* INOUT - Symbol table for node names, etc.          */
card       *current; /* IN - The card we are to parse. Must be called 'current'
                     ...           for compatibility with SPICE3C1 macros.   */

{
    Get the instance card text from the card struct in 'current'.

    Get the name of the instance from the card and add it to
    symbol table 'tab'. [MIFgettok(), SPICE3C1/INPinsert()]
```

If this model's parameters have not already been processed,  
Create data structure for model in 'ckt'. [SPICE3C1/CKTmodCrt()]  
Allocate and initialize MIF specific model parameter data.  
Remaining initializations will be done by MIFmParam() and  
MIFsetup().  
Get parameter values from card and put them into the model structure.  
[MIFgetValue(), SPICE3C1/CKTmodParam()]  
Mark model entry in 'modtab' as having been processed.  
Return NULL indicating no error.  
Didn't find model, so return error string.  
}

#### 4.1.3 Function MIFgetValue

##### Summary

This function gets a parameter value from the .model text line into an IFvalue structure. The parameter type is specified in the argument list and is used to determine how to parse the text on the .model line. If the parameter is an array, the entire array is parsed and placed in the IFvalue structure along with the number of elements found.

##### Called By

MIFgetMod()            MIF/MIFgetMod.c

##### Returned Value

A SPICE3C1 IFvalue structure holding the value of the parameter, or NULL if error.

##### PDL Description

```
IFvalue *MIFgetValue(ckt,line,type,tab,err)

GENERIC  *ckt;      /* IN   - The circuit structure.          */
char     **line;    /* INOUT - The text line to read value from. */
int      type;     /* IN   - The type of data to read.        */
INPtables *tab;   /* IN   - Unused.                         */
char     **err;    /* OUT  - Error string text.            */
{
    If 'type' indicates value is an array,
        Get the array delimiter token. [MIFget_token()]
        Allocate space in the return value structure for array
        elements.

    Loop forever,
        Get the next token in 'line'. [MIFget_token()]
        Return with error if no more tokens.

        If array and token type is closing array delimiter,
            Return with error if number of values read so far is zero.

    Exit loop.
```

Process the token to extract value. Exit with  
error if token is not a value.

Exit loop after this single pass if not array.

Return the value read.

}

#### 4.1.4 Function MIFgettok

##### Summary

This function gets the next token from the input string supplied in its argument list. The input string pointer is advanced to the following token and the token from the input string is copied to storage allocated using the malloc() function. A pointer to this storage is returned. The original input string is left undisturbed.

##### Called By

MIF_INP2A()	MIF/MIF_INP2A.c
MIFget_token()	MIF/MIFutil.c
if_setndnames()	FTE/spiceif.c
translate()	FTE/subckt.c
devmodtranslate()	FTE/subckt.c
count_tokens()	ENH/ENHtranslate_poly.c
translate()	ENH/ENHtranslate_poly.c
get_poly_dimension()	ENH/ENHtranslate_poly.c

##### Returned Value

The string representing the token read.

##### PDL Description

```
char *MIFgettok(s)

char **s; /* INOUT - The text line to get the token from */
{

    Allocate space equal to size of 's' to guarantee space.

    Skip over any white space.

    Isolate the next token.

    Get token value.

    Skip over white space up to next token.

    Make a copy using only the space needed by the string length.

    Return new string holding token.
}
```

#### 4.1.5 Function MIFget\_token

##### Summary

This function gets the next token from the input string and returns the token and its type. Except for determining the type, it functions identically to MIFgettak. The type is returned as one of the following enumerated values:

MIF_LARRAY_TOK	-	A left array delimiter
MIF_RARRAY_TOK	-	A right array delimiter
MIF_LCOMPLEX_TOK	-	A left complex number delimiter
MIF_RCOMPLEX_TOK	-	A right complex number delimiter
MIF_PERCENT_TOK	-	A percent sign token
MIF_TILDE_TOK	-	A tilde sign token
MIF_NULL_TOK	-	No token found
MIF_STRING_TOK	-	A string other than the above

##### Called By

MIF_INP2A()	MIF/MIF_INP2A.c
MIFget_port_type()	MIF/MIF_INP2A.c
MIFget_port()	MIF/MIF_INP2A.c
MIFgetValue()	MIF/MIFgetValue.c

##### Returned Value

The string representing the token read.

##### PDL Description

```
char *MIFget_token(s, type)

char          **s;      /* INOUT - The text line to get the token from */
Mif_Token_Type_t *type; /* OUT   - The type of token found */

{
    Get the token from the input line 's'. [MIFgettak()]

    If no next token,
        Return NULL.
```

```
    Else,  
        Determine and set value for 'type'.  
        Return token string.  
    }
```

#### 4.1.6 Function MIFget\_cntl\_src\_type

##### Summary

This function takes an input connection/port type and an output connection/port type (MIF\_VOLTAGE, MIF\_CURRENT, etc.) and maps this pair to one of the four controlled source types used in SPICE (VCVS, VCIS, ICVS, ICIS).

##### Called By

MIFload()	MIF/MIFload.c
MIFsetup()	MIF/MIFsetup.c

##### Returned Value

The type of the controlled source.

##### PDL Description

```
Mif_Cntl_Src_Type_t MIFget_cntl_src_type(in_port_type, out_port_type)
Mif_Port_Type_t in_port_type; /* IN - The type of the input port */
Mif_Port_Type_t out_port_type; /* IN - The type of the output port */
{
    Determine and return the type from the respective input and output types.
}
```

#### 4.1.7 Function MIFmParam

##### Summary

This function is called by SPICE/Nutmeg to set the value of a parameter on a model according to information parsed from a .model card or information supplied interactively by a user. It takes the value of the parameter input in an IFvalue structure and sets the parameter on the specified model structure. Unlike the procedure for SPICE 3C1 devices, MIFmParam does not use enumerations for identifying the parameter to set. Instead, the parameter is identified directly by the index value of the parameter in the SPICEdev.DEVpublic.modelParms array.

##### Called By

CKTmodParam()                  CKT/CKTmodParam.c

##### Returned Value

Returns zero to indicate success. Otherwise returns a SPICE3C1 error message number.

##### Global Variables

```
SPICEdev *DEVices[]; /* IN - Information about models known to simulator. */
int        DEVmaxnum; /* IN - The number of elements in 'DEVices' */
```

##### PDL Description

```
int MIFmParam(param_index, value, inModel)

int        param_index; /* IN - The parameter to set */
IFvalue    *value;        /* IN - The value of the parameter */
GENmodel *inModel;       /* INOUT - The model structure on which to set the value */
{
    Get model type index from 'inModel'.

    Check parameter index 'param_index' for validity.

    Get type of value by looking up parameter in 'DEVices'.

    Initialize the parameter is_null and size elements in 'inModel'
    and allocate storage for the parameter.
```

Transfer the data from 'value' to the parameter storage in 'inModel'.

Return OK.

}

#### 4.1.8 Function MIFmAsk

##### Summary

This function is called by SPICE/Nutmeg to query the value of a parameter on a model. It is essentially the opposite of MIFmParam, taking the index of the parameter, locating the value of the parameter in the model structure, and converting that value into the IFvalue structure understood by Nutmeg.

##### Called By

CKTmodAsk()                  CKT/CKTmodAsk.c

##### Returned Value

Returns zero to indicate success. Otherwise returns a SPICE3C1 error message number.

##### Global Variables

```
SPICEdev *DEVices[]; /* IN - Information about models known to simulator. */
int        DEVmaxnum; /* IN - The number of elements in 'DEVices' */
```

##### PDL Description

```
int MIFmAsk(ckt, inModel, param_index, value)

CKTcircuit *ckt;            /* IN - The circuit structure */
GENmodel *inModel;          /* IN - The model to get the value from */
int        param_index;     /* IN - The parameter to get */
IFvalue    *value;          /* OUT - The value returned */

{
    Get model type index from 'inModel'.

    Check parameter index 'param_index' for validity.

    Get type of value by looking up parameter in 'DEVices'.

    Transfer the parameter data to 'value' from the parameter data
    in 'inModel'.
}
```

#### 4.1.9 Function MIFask

##### Summary

This function is called by SPICE/Nutmeg to query the value of a parameter on an instance. It locates the value of the parameter in the instance structure and converts the value into the IFvalue structure understood by Nutmeg. For code models, this function is provided to allow output of Instance Variables defined in the code model's Interface Specification.

##### Called By

CKTask()            CKT/CKTask.c

##### Returned Value

Returns zero to indicate success. Otherwise returns a SPICE3C1 error message number.

##### Global Variables

```
SPICEdev *DEVices[]; /* IN - Information about models known to simulator. */
int        DEVmaxnum; /* IN - The number of elements in 'DEVices' */
```

##### PDL Description

```
int MIFask(ckt, inInst, which, value, select)

CKTcircuit *ckt;        /* IN - The circuit structure */
GENinstance *inInst;     /* IN - The instance to get the value from */
int        which;        /* IN - The parameter to get */
IFvalue    *value;       /* OUT - The value returned */
IFvalue    *select;      /* IN - Unused */

{
    Get index of the model type associated with this instance from
    'inInst'.

    Check parameter index 'which' for validity.

    Get type of value by looking up parameter in 'DEVices' for
    the model.

    Transfer the parameter data to 'value' from the parameter data
    in the model.

}
```

#### 4.1.10 Function MIFsetup

##### Summary

This function is called by the CKTsetup() driver function to prepare all code model structures and all code model instance structures for simulation. It loops through all models of a particular code model type and provides defaults for any parameters not specified on a .model card. It loops through all instances of the model and prepares the instance structures for simulation. The most important setup task is the creation of entries in the SPICE matrix and the storage of pointers to locations of the matrix used by MIFload during a simulation.

##### Called By

CKTsetup()      CKT/CKTsetup.c

##### Returned Value

Returns zero to indicate success. Otherwise returns a SPICE3C1 error message number.

##### Global Variables

```
SPICEdev *DEVICES[]; /* IN - Information about models known to simulator. */
```

##### PDL Description

```
int MIFsetup(SMPmatrix, GENmodel, CKTcircuit, states)

SMPmatrix *matrix; /* INOUT - The analog simulation matrix structure */
GENmodel *inModel; /* INOUT - The head of the model list */
CKTcircuit *ckt; /* INOUT - The circuit structure */
int *states; /* INOUT - The states vector */

{
    Loop through all models in the linked list with head 'inModel'.

    For each parameter not given explicitly on the .model card,
        If parameter is null,
            Determine the size from information in global 'DEVICES'
            and allocate the parameter element(s).
```

Set the parameter element(s) to default value given in  
'DEVICES'.

Loop through all items in the linked list of instances attached  
to this element in the 'inModel' list.

Initialize items needed during simulation.

Loop through all items in the linked list of instances attached  
to this element in the 'inModel' list.

Allocate runtime structures for output connections/ports  
and grab a place in the state vector for all input connections/ports.

Loop through all items in the linked list of instances attached  
to this element in the 'inModel' list.

Loop through all connections on this instance and create equations  
needed for outputs and V sources associated with I inputs.

If the connection is null, skip to next connection.

Loop through all ports on this connection.

If port is null, skip to next.

Determine the type of this port.

Create a pointer to the smp data for quick access.

If it is a voltage source output,

Create a new entry in the 'ckt' node list for this  
voltage source current equation. [SPICE3C1/CKTmkCur()]

Create the new matrix equation entry needed for this  
voltage source. [SPICE3C1/SMPmakeElt()]

If it is a current input,

Create a new entry in the 'ckt' node list for the  
associated zero-valued voltage source.  
[SPICE3C1/CKTmkCur()]

Create the new matrix equation entry needed for this  
voltage source. [SPICE3C1/SMPmakeElt()]

If it is a vsource current input (refers to a vsource  
elsewhere in the circuit),

Locate the source and record its equation number,

creating a new entry in the 'ckt' node list if not  
already processed in another setup function.  
[SPICE3C1/CKTfndBranch()]

Now loop through all connections on the instance and create  
matrix data needed for loading partial derivatives of outputs  
into matrix.

If the connection is null or is not an output,  
skip to next connection.

Loop through all ports on this connection.

If port is null, skip to next.

For this port, loop through all connections  
and all ports to touch on each possible input.

If the connection is null or is not an input  
skip to next connection.

Loop through all the ports of this connection.

If port is null, skip to next.

Determine type of controlled source according  
to input and output types. [MIFget\_cntl\_src\_type()]

Get and record the matrix entry pointers associated  
with the partial derivative. [SPICE3C1/SMPmakeElt()]

}

#### 4.1.11 Function MIFload

##### Summary

This function is called by the CKTload() driver function to call the C function for each instance of a code model type. It loops through all models of that type and all instances of each model. For each instance, it prepares the structure that is passed to the code model by filling it with the input values for that instance. The code model's C function is then called, and the outputs and partial derivatives computed by the C function are used to fill the matrix for the next solution attempt.

##### Called By

CKTload()      CKT/CKTload.c

##### Returned Value

Returns zero to indicate success. Otherwise returns a SPICE3C1 error message number.

##### Global Variables

```
SPICEdev *DEVices[]; /* IN - Information about models known to simulator. */
Mif_Info_t g_mif_info; /* INOUT - Code model interface runtime data. */
```

##### PDL Description

```
int MIFload(inModel, ckt)

GENmodel *inModel; /* INOUT - The head of the model list */
CKTcircuit *ckt; /* INOUT - The circuit structure */

{
    Mif_Private_t cm_data; /* The data structure passed to a code model. */

    Setup the circuit data (non-model specific data) including initialization
    flags, analysis type, frequency, and time, in the 'cm_data' structure to
    be passed to the code models.

    Iterate over all models in the linked list pointed to by 'inModel',

```

If not an analog or hybrid model, continue to next in list.

Iterate over all instances in the linked list of instances on this model,

If not an analog or hybrid instance, continue to next instance.

```
/* ****
/* Setup the data to be passed to the user's code model function. */
/* ****
```

Record a pointer to the instance in 'g\_mif\_info' so that code model support library functions will have access to the instance being evaluated when they are called by the user's code model function.

If this is the first step in a transient analysis, copy state 1 data to state 0. Otherwise the data in state 0 would be invalid.

Loop through all connections on this instance,

If AC analysis, skip getting input values. The input values stay the same as they were at the last iteration of the operating point analysis.

If the connection is null, skip to next connection.

If this connection is not an input, skip to next connection.

Loop through all ports on this connection,

Skip to next port if this port is null.

If port type is Digital or User-Defined,

Get the total load on the connected node. (Note that we do not need to get the input values since a pointer to the input values has already been setup by EVTsetup() and does not change during the simulation.)

Else, it is an analog node and we get the input value as follows,

If this is the very first evaluation,

Set the input value to zero.

Else if this is the first evaluation at the current

```
analysis step (e.g. timepoint),  
  
Get the input value from the state storage.  
  
Else,  
  
    Get the input value from the matrix solution  
    at the previous iteration.  
  
    If convergence limiting enabled, limit maximum input change.  
  
    Compute the maximum the input is allowed to change  
    based on input type (voltage or current) and on  
    last value.  
  
    If input has changed too much, limit it and signal  
    not converged.  
  
Save value of input in state storage for use at next  
analysis step.  
  
Loop through all connections on this instance to zero out all  
outputs, partials, and AC gains for each output port of each  
connection,  
  
    If the connection is null or is not an output  
    skip to next connection.  
  
    Loop through all ports on this connection.  
  
        Skip if this port is null.  
  
        If port is not connected to an analog node,  
        continue to next port.  
  
        Initialize the output to zero.  
  
    Loop through all connections and ports that  
    could be inputs for this port and zero the partials.  
  
/* ***** */  
/* Call the user's code model function. */  
/* ***** */  
  
Setup the 'cm_data' structure to be passed to the code model  
function by setting its elements to point to the data in the  
instance prepared above.  
  
Initialize the auto_partial flag in the global 'g_mif_info'  
structure to FALSE.
```

Call the code model through the function pointer in the global 'DEVICES' device structure array.  
[<user code model C function identifier>]

If auto\_partial flag in 'g\_mif\_info' is TRUE, indicating that the code model function called cm\_analog\_auto\_partial(),

Call the code model additional times and compute the partials through divided differences. [MIFauto\_partial()]

```
/* **** */
/* Load the matrix with the output and partial derivative */
/* values computed by the code model. */
/* **** */
```

Loop through all connections to load the outputs computed by the code model (including the zero-valued V sources associated with current inputs),

If the connection is null, skip to next connection.

Loop through all ports on this connection.

Skip if this port is null.

If port is not connected to an analog node, continue to next port.

If port is a current input

Load the matrix with + and - i values as needed for the associated zero-valued V source.

If port is a voltage source output

Load the matrix with + and - i values.

If not AC analysis, load the right hand side vector with the V source output data.

If it has a current source output

If not AC analysis, load the right hand side with the I source output data.

Loop through all output connections to load the partials or AC gains into the matrix as appropriate.

If the connection is null or is not an output, skip to next connection.

Loop through all ports on this connection.

Skip if this port is null.

If port is not connected to an analog node,  
continue to next port.

For this port, loop through all connections  
to touch on each possible input,

If the connection is null or is not an input  
skip to next connection.

Loop through all the ports of this connection.

Skip if this port is null.

If not connected to an analog node, continue  
to next port.

Determine type of controlled source according  
to input and output types. [MIFget\_cntl\_src\_type()]

Load the matrix with the partials or AC gains  
as appropriate for port and analysis type.

}

#### 4.1.11.1 Static Function MIFauto\_partial

##### Summary

This function is called by MIFload() when a code model requests that partial derivatives be computed automatically. It calls the code model additional times with an individual input to the model varied by a small amount at each call. Partial derivatives of each output with respect to the varied input are then computed by divided differences.

##### Called By

MIFload()      MIF/MIFload.c

##### Returned Value

None.

##### Global Variables

```
Mif_Info_t g_mif_info; /* INOUT - Code model interface runtime data. */
```

##### PDL Description

```
static void MIFauto_partial(here, cm_func, cm_data)

MIFinstance *here; /* INOUT - The instance being evaluated */
void (*cm_func)(); /* IN - The code model function to be called */
Mif_Private_t *cm_data; /* INOUT - The data passed to the code model */

{
    Reset init and anal_init flags in global 'g_mif_info' structure
    and 'cm_data' structure before making additional calls to the model.

    /* **** */
    /* Save nominal analog outputs */
    /* **** */

    Loop through all connections on instance 'here',
    If the connection is null or is not an output
    skip to next connection.
```

Loop through all ports on this connection.

Skip if this port is null.

If not an analog port, continue to next port.

Save the nominal output for use in computing output deltas.

```
/* ****
/* Change each analog input by a small amount and call the model to */
/* compute new outputs. */
/* ****
```

Loop through all connections,

If the connection is null, skip to next connection.

If this connection is not an input, skip to next connection.

Loop through all ports on this connection.

Skip if this port is null.

Skip if port type is Digital or User-Defined type.

Compute the perturbation amount depending on type of input  
(voltage or current).

Save nominal input value and then perturb it.

Call model to compute new outputs.  
[<user code model C function identifier>]

```
/* ****
/* Compute the partials of each output with respect to the */
/* perturbed input by divided differences. */
/* ****
```

Loop through all analog output connections,

If the connection is null or is not an output  
skip to next connection.

Loop through all the ports of this connection.

Skip if this port is null.

If port type is Digital or User-Defined, skip it.

Compute partial by divided differences.

XSPICE Simulator  
Software Design Document

Detailed Design  
Model Interface

```
Zero the output in preparation for next call.  
  
Restore nominal input value saved earlier.  
  
/* ***** */  
/* Call model one last time to recompute nominal case. */  
/* ***** */  
  
Call the user's code model function through the function pointer  
'cm_func'. [<user code model C function identifier>]  
}
```

#### 4.1.12 Function MIFconvTest

##### Summary

This function is called by the CKTconvTest() driver function to check convergence of any states owned by instances of a particular code model type. It loops through all models of that type and all instances of each model. For each instance, it looks in the instance structure to determine if any variables allocated by cm\_analog\_alloc() have been registered by a call to cm\_analog\_converge() to have their convergence tested. If so, the value of the function at the last iteration is compared with the value at the current iteration to see if it has converged to within the same delta amount used in node convergence checks (as defined by SPICE 3C1).

##### Called By

CKTconvTest()            CKT/CKTconvTest.c

##### Returned Value

Returns zero to indicate success. Otherwise returns a SPICE3C1 error message number.

##### PDL Description

```
int MIFconvTest(inModel, ckt)

GENmodel *inModel; /* INOUT - The head of the model list */
CKTcircuit *ckt; /* INOUT - The circuit structure */

{
    Iterate through linked list of models pointed to by 'inModel',
    Iterate through list of all instances under this model,
    Loop through all states on the instance in the list
    of states that need to be converged.

    Get the current value and the value from the last
    iteration from the state vector.

    Check the change in the value against allowable
    changes CKTrelTol and CKTabsTol in 'ckt'.

    If change is too large,
        Report convergence problems if this is the last
```

XSPICE Simulator  
Software Design Document

Detailed Design  
Model Interface

```
iteration attempt before giving up.  
[ENHreport_conv_prob()]  
  
Increment CKTnoncon in 'ckt'.  
  
Rotate the current value to last value storage location  
on the instance in preparation for next call.  
  
return OK.  
}
```

#### 4.1.13 Function MIFtrunc

##### Summary

This function is called by the CKTtrunc() driver function to check numerical integration truncation error of any integrals associated with instances of a particular code model type. It traverses all models of that type and all instances of each model. For each instance, it looks in the instance structure to determine if any variables allocated by cm\_analog\_alloc() have been used in a call to cm\_analog\_integrate(). If so, the truncation error of that integration is computed and used to set the maximum delta allowed for the current timestep.

##### Called By

CKTtrunc()            CKT/CKTtrunc.c

##### Returned Value

Returns zero to indicate success. Otherwise returns a SPICE3C1 error message number.

##### PDL Description

```
int MIFtrunc(inModel, ckt, timeStep)

    GENmodel *inModel;        /* The head of the model list */
    CKTcircuit *ckt;          /* The circuit structure */
    double      *timeStep;     /* The timestep delta */

{
    Loop through all models in linked list pointed to by 'inModel',
        Loop through all instances of this model,
            Loop through all integration states on the instance,
                Limit timeStep according to truncation error.
                [MIFterr()]
}
```

#### 4.1.13.1 Static Function MIFterr

##### Summary

This is a modified version of the function SPICE3C1 CKTterr(). It limits the analog simulation timestep according to computed truncation error.

##### Called By

MIFtrunc()      MIF/MIFtrunc.c

##### Returned Value

None.

##### PDL Description

```
static void MIFterr(intgr, ckt, timeStep)

Mif_Intgr_t *intgr; /* IN - The state being integrated */
CKTCircuit *ckt; /* IN - The main circuit structure */
double *timeStep; /* INOUT - The next analog simulation timestep */

{
    Compute maximum allowable timestep based on estimate of integration
    truncation error of quantity in state vector pointed to by 'intgr'.

    Limit 'timeStep' to minimum of its current value and the timestep
    computed above.
}
```

#### 4.1.14 Function MIFmDelete

##### Summary

This function deletes a particular model defined by a .model card from the linked list of model structures of a particular code model type, freeing all dynamically allocated memory used by the model structure. It calls MIFdelete as needed to delete all instances of the specified model.

##### Called By

CKTdltMod()      CKT/CKTdltMod.c

##### Returned Value

Returns zero to indicate success. Otherwise returns a SPICE3C1 error message number.

##### PDL Description

```
int MIFmDelete(inModel, modname, kill)

GENmodel **inModel;    /* INOUT - The head of the model list */
IFuid modname;        /* IN     - The name of the model to delete */
GENmodel *kill;       /* IN     - The model structure to be deleted */

{
    Locate the model by name ('modname') or pointer ('kill')
    and cut it out of the list of models ('inModel')

    Free the instances under this model if any by removing from
    the head of the linked list until the head is null. [MIFdelete()]

    Free the model parameter data.

    Free the model and return OK.
}
```

#### 4.1.15 Function MIFdelete

##### Summary

This function deletes a particular instance from the linked list of instance structures, freeing all dynamically allocated memory used by the instance structure.

##### Called By

CKTdltInst()      CKT/CKTdltInst.c

##### Returned Value

Returns zero to indicate success. Otherwise returns a SPICE3C1 error message number.

##### PDL Description

```
int MIFdelete(inModel, name, inst)

GENmodel *inModel; /* INOUT - The head of the model list */
IFuid name; /* IN - The name of the instance to delete */
GENinstance **inst; /* IN - The instance structure to delete */

{
    Loop through all models in list pointed to by 'inModel',
    Loop through list of instances of this model,
    If 'name' or pointer 'inst' matches,
        Cut instance out of list.

    Return error if not found.

    Loop through all connections on the instance
    and dismantle the structures allocated during read-in/setup
    in MIFinit_inst(), MIFget_port(), and MIFsetup().

    Free the connection stuff allocated in MIFinit_inst, but
    don't free name/description. They are not owned by the instance.

    Loop through all instance variables on the instance and free data.

    Free the data used by the cm_... functions.

    Free the instance struct itself, and return OK.
}
```

#### 4.1.16 Function MIFdestroy

##### Summary

This function deletes all models and all instances of a specified device type. It traverses the linked list of model structures for that type and calls MIFmDelete on each model.

##### Called By

CKTdestroy()      CKT/CKTdestroy.c

##### Returned Value

None.

##### PDL Description

```
void MIFdestroy(inModel)
{
    GENmodel **inModel; /* INOUT - The head of the list of models to delete */
    {
        Free all models in list pointed to by 'inModel' by removing
        models from the head of the linked list until the head is null.
        [MIFmDelete()]
    }
}
```

## 4.2 Code Model Support

The Code Model (SIM-CM) CSC provides support functions callable from code models. Callable functions include:

```
cm_analog_alloc
cm_analog_get_ptr
cm_analog_integrate
cm_analog_converge
cm_analog_set_temp_bkpt
cm_analog_set_perm_bkpt
cm_analog_ramp_factor
cm_analog_not_converged
cm_analog_auto_partial

cm_message_get_errmsg
cm_message_send

cm_event_alloc
cm_event_get_ptr
cm_event_queue

cm_netlist_get_c
cm_netlist_get_l

cm_smooth_corner
cm_smooth_discontinuity
cm_smooth_pwl

cm_complex_set
cm_complex_add
cm_complex_subtract
cm_complex_multiply
cm_complex_divide
```

Support functions such as state allocation, integration, and convergence-test store information in the associated instance's MIFinstance structure. For analog models, state memory is allocated in the SPICE 3C1 circuit structure's CKTstates arrays and indices into these arrays are stored with the instance. Use of the CKTstates arrays allows rotating states through old timepoints to be done automatically by the existing SPICE 3C1 mechanism. The integration and convergence-test support functions are implemented using emulations of SPICE's Capacitor device algorithms and therefore require that the data being integrated or checked reside within the CKTstates arrays.

#### 4.2.1 Function cm\_analog\_alloc

##### Summary

This function is called from code model C functions to allocate state storage for a particular instance. It computes the number of doubles that need to be allocated in SPICE's state storage vectors from the number of bytes specified in it's argument and then allocates space for the states. An index into the SPICE state-vectors is stored in the instance's data structure along with a "tag" variable supplied by the caller so that the location of the state storage area can be found by cm\_analog\_get\_ptr().

##### Called By

Code model functions.

##### Returned Value

A pointer to the bytes allocated on success. A NULL pointer on failure.

##### Global Variables

```
Mif_Info_t g_mif_info; /* INOUT - Code model interface runtime data. */
```

##### PDL Description

```
void *cm_analog_alloc(tag, bytes)

int tag;      /* IN - The user-specified tag for this block of memory */
int bytes;    /* IN - The number of bytes to allocate */
{
    CKTcircuit *ckt; /* The main circuit structure */

    Get the address of the 'ckt' and instance structure from 'g_mif_info'.

    Scan states in instance struct and see if 'tag' has already been used.

    Compute number of doubles needed and allocate space in the 'ckt'
    states vector.

    Allocate space in instance structure for this state descriptor.

    Fill in the members of the state descriptor structure.
```

Add the states to the 'ckt' states vector.

Return pointer to the allocated space in state 0.

}

#### 4.2.2 Function cm\_analog\_get\_ptr

##### Summary

This function is called from code model C functions to return a pointer to state storage allocated with cm\_analog\_alloc(). A tag specified in its argument list is used to locate the state in question. A second argument specifies whether the desired state is for the current timestep or from a preceding timestep. The location of the state in memory is then computed and returned.

##### Called By

Code model functions.

##### Returned Value

A pointer to the specified data on success. A NULL pointer on failure.

##### Global Variables

```
Mif_Info_t g_mif_info; /* INOUT - Code model interface runtime data. */
```

##### PDL Description

```
void *cm_analog_get_ptr(tag, timepoint)

int tag;          /* IN - The user-specified tag for this block of memory */
int timepoint;    /* IN - The timepoint of interest - 0=current 1=previous */
{
    CKTcircuit *ckt; /* The main circuit structure */

    Get the address of the 'ckt' and instance structure from 'g_mif_info'.

    Scan states in instance structure and see if tag exists.

    Return NULL if tag not found.

    Return NULL if timepoint is not 0 or 1.

    Return address of requested state in 'ckt' states vector.
}
```

#### 4.2.3 Function cm\_analog\_integrate

##### Summary

This function performs a numerical integration on the state supplied in its argument list according to the integrand also supplied in the argument list. The next value of the integral and the partial derivative with respect to the integrand input is returned. The integral argument must be a pointer to memory previously allocated through a call to cm\_analog\_alloc(). If this is the first call to cm\_analog\_integrate, information is entered into the instance structure to mark that the integral should be processed by MIFtrunc and MIFconvTest.

##### Called By

Code model functions.

##### Returned Value

Zero if successful, or non-zero if error.

##### Global Variables

```
Mif_Info_t g_mif_info; /* INOUT - Code model interface runtime data. */
```

##### PDL Description

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

double integrand; /* IN - The integrand */
double *integral; /* INOUT - The current and returned value of integral */
double *partial; /* OUT - The partial derivative of integral wrt integrand */
{
    CKTCircuit *ckt; /* The main circuit structure */

    Get the address of the 'ckt' and instance structure from 'g_mif_info'.

    Check analysis type in 'g_mif_info' to be sure we're in transient analysis.

    Compute byte offset from start of state zero vector in 'ckt'.

    Check to be sure argument address is in range of state zero vector.

    Scan the instance structure to see if this byte offset has already been
```

encountered and is already setup for integration in the instance structure.

Report error if not found and this is not the first load pass in transient analysis.

If this is a new integral state,

Allocate space in instance structure for this integration descriptor and register it as needing to be subjected to convergence criteria during iteration. [cm\_analog\_converge()]

Compute the new integral and the partial. [cm\_static\_integrate()]

}

#### 4.2.3.1 Static Function cm\_static\_integrate

##### Summary

This function is a modified version of the function NIIntegrate(). It performs a trapezoidal integration of the specified quantity in the ckt state vector.

##### Called By

cm\_analog\_integrate() CM/CM.c

##### Returned Value

None.

##### Global Variables

Mif\_Info\_t g\_mif\_info; /\* INOUT - Code model interface runtime data. \*/

##### PDL Description

```
static void cm_static_integrate(byte_index, integrand, integral, partial)

int byte_index; /* IN - Byte offset into state vector */
double integrand; /* IN - The argument to the integral */
double *integral; /* OUT - The value of the integral */
double *partial; /* OUT - The partial derivative of integral wrt integrand */
{
    CKTcircuit *ckt; /* The main circuit structure */

    Get the address of the 'ckt' structure from 'g_mif_info'.

    Get pointers to integral values from current and previous timesteps in
    'ckt' state vector.

    Calculate the new integral value and partial derivative.
}
```

#### 4.2.4 Function cm\_analog\_converge

##### Summary

This function registers a state variable allocated with cm\_analog\_alloc() to be subjected to a convergence test at the end of each iteration. The state variable must be a double. Information is entered into the instance structure to mark that the state variable should be processed by MIFconvTest.

##### Called By

Code model functions.

`cm_analog_integrate()`      CM/CM.c

##### Returned Value

Zero if successful, or non-zero if error.

##### Global Variables

`Mif_Info_t g_mif_info; /* INOUT - Code model interface runtime data. */`

##### PDL Description

```
int cm_analog_converge(state)

double *state; /* IN - The state to be converged */
{
    CKTcircuit *ckt; /* The main circuit structure */

    Get the address of the 'ckt' and instance structure from 'g_mif_info'.

    Compute byte offset from start of state zero vector in 'ckt'.

    Check to be sure argument address is in range of state zero vector.

    Scan the convergence descriptor data in the instance structure to see
    if already registered. If so, do nothing, just return success.

    Allocate space in instance structure for this convergence descriptor.
```

Fill in the convergence descriptor data.

Return success.

}

#### 4.2.5 Function cm\_analog\_set\_temp\_bkpt

##### Summary

This function is called by a code model C function to set a temporary breakpoint. These temporary breakpoints remain in effect only until the next timestep is taken. A temporary breakpoint added with a time less than the current time, but greater than the last successful timestep causes the simulator to abandon the current timestep and decrease the timestep to hit the breakpoint. A temporary breakpoint with a time greater than the current time causes the simulator to make the breakpoint the next timepoint if the next timestep would produce a time greater than that of the breakpoint.

##### Called By

Code model functions.

##### Returned Value

Zero if successful, or non-zero if error.

##### Global Variables

```
Mif_Info_t g_mif_info; /* INOUT - Code model interface runtime data. */
```

##### PDL Description

```
int cm_analog_set_temp_bkpt(time)

double time; /* IN - The time of the breakpoint to be set */
{
    CKTcircuit *ckt; /* The main circuit structure */

    Get the address of the 'ckt' structure from 'g_mif_info'.

    Make sure breakpoint 'time' is not prior to last accepted timepoint
    in 'ckt'.

    If too close to a permanent breakpoint in 'ckt' or the current
    time, ignore it and return success.

    If less than current dynamic breakpoint in 'g_mif_info',
```

XSPICE Simulator  
Software Design Document

Detailed Design  
Code Model Support

    Make it the current breakpoint.

    Return success.

}

#### 4.2.6 Function cm\_analog\_set\_perm\_bkpt

##### Summary

This function is called by a code model C function to set a permanent breakpoint. These permanent breakpoints remain in effect from the time they are introduced until the simulation time equals or exceeds the breakpoint time. A permanent breakpoint added with a time less than the current time, but greater than the last successful timestep causes the simulator to abandon the current timestep and decrease the timestep to hit the breakpoint. A permanent breakpoint with a time greater than the current time causes the simulator to make the breakpoint the next timepoint if the next timestep would produce a time greater than that of the breakpoint.

##### Called By

Code model functions.

##### Returned Value

Zero if successful, or non-zero if error.

##### Global Variables

```
Mif_Info_t g_mif_info; /* INOUT - Code model interface runtime data. */
```

##### PDL Description

```
int cm_analog_set_perm_bkpt(time)

    double time;      /* The time of the breakpoint to be set */
{
    CKTcircuit *ckt; /* The main circuit structure */

    Get the address of the 'ckt' structure from 'g_mif_info'.

    If 'time' is less than current time in 'ckt',
        Force an immediate timestep cut. [cm_analog_set_temp_bkpt()]

    Else,
```

XSPICE Simulator  
Software Design Document

Detailed Design  
Code Model Support

Insert a new permanent breakpoint in the 'ckt' breakpoint list.  
[SPICE3C1/CKTsetBreak()]

Return success.

}

#### 4.2.7 Function cm\_ramp\_factor

##### Summary

This function returns the current value of the ramp factor associated with the “ramptime” option. For this option to work best, models with analog outputs that may be non-zero at time zero should call this function and scale their outputs and partials by the ramp factor.

##### Called By

Code model functions.

VSRCload()	DEV/VSRC/VSRCload.c
ISRCload()	DEV/ISRC/ISRCload.c

##### Returned Value

A value between 0.0 and 1.0 representing the ramp factor.

##### Global Variables

```
Mif_Info_t g_mif_info; /* INOUT - Code model interface runtime data. */
```

##### PDL Description

```
double cm_ramp_factor()

{
    CKTcircuit *ckt; /* The main circuit structure */

    Get the address of the 'ckt' structure from 'g_mif_info'.

    If ramptime given in enhancements structure in 'ckt' == 0.0,
        Return 1.0.

    Else if not transient analysis,
        Return 1.0.

    Else if time value in 'ckt' >= ramptime,
```

Return 1.0.

Else

Compute and return factor based on time value in 'ckt'.

}

#### 4.2.8 Function cm\_analog\_not\_converged

##### Summary

This function tells the simulator not to allow the current iteration to be the final iteration. It is called when a code model performs internal limiting on one or more of its inputs to assist convergence.

##### Called By

Code model functions.

##### Returned Value

None.

##### Global Variables

```
Mif_Info_t g_mif_info; /* IN - Code model interface runtime data. */
```

##### PDL Description

```
void cm_analog_not_converged()

{
    CKTcircuit *ckt; /* The main circuit structure */

    Get the address of the 'ckt' structure from 'g_mif_info'.

    Increment CKTnoncon member of 'ckt'.
}
```

#### 4.2.9 Function cm\_analog\_auto\_partial

##### Summary

This function tells the simulator to automatically compute approximations of partial derivatives of analog outputs with respect to analog inputs. When called from a code model, it sets a flag in the g\_mif\_info structure which tells function MIFload() and its associated MIFauto\_partial() function to perform the necessary calculations.

##### Called By

Code model functions.

##### Returned Value

None.

##### Global Variables

```
Mif_Info_t g_mif_info; /* IN - Code model interface runtime data. */
```

##### PDL Description

```
void cm_analog_auto_partial()  
{  
    Set the auto partial flag in 'g_mif_info' to true.  
}
```

#### 4.2.10 Function cm\_message\_get\_errmsg

##### Summary

This function returns the address of an error message string set by a call to some code model support function.

##### Called By

Code model functions.

##### Returned Value

A (possibly NULL) pointer to an error string set by some other code model library support function.

##### Global Variables

```
Mif_Info_t g_mif_info; /* IN - Code model interface runtime data. */
```

##### PDL Description

```
char *cm_message_get_errmsg()  
{  
    Return pointer to error message found in 'g_mif_info'.  
}
```

#### 4.2.11 Function cm\_message\_send

##### Summary

This function prints a message output from a code model, prepending the instance name.

##### Called By

Code model functions.

##### Returned Value

Zero if successful, or non-zero if error.

##### Global Variables

```
Mif_Info_t g_mif_info; /* IN - Code model interface runtime data. */
```

##### PDL Description

```
int cm_message_send(msg)
char *msg; /* IN - The message to output. */
{
    Get the address of the instance structure from 'g_mif_info'.
    Print the name of the instance and the message 'msg' to stdout.
    Return success.
}
```

#### 4.2.12 Function cm\_event\_alloc

##### Summary

This function is called from code model C functions to allocate state storage for a particular event-driven instance. It is similar to the function cm\_analog\_alloc() used by analog models, but allocates states that are rotated during event-driven timesteps instead of analog timesteps.

##### Called By

Code model functions.

##### Returned Value

A pointer to the bytes allocated on success. A NULL pointer on failure.

##### Global Variables

```
Mif_Info_t g_mif_info; /* IN - Code model interface runtime data. */
```

##### PDL Description

```
void *cm_event_alloc(tag, bytes)

int tag; /* IN - The user-specified tag for the memory block */
int bytes; /* IN - The number of bytes to be allocated */
{
    CKTcircuit *ckt; /* The main circuit structure */

    Get the address of 'ckt' and instance structure from 'g_mif_info'.

    If instance structure has already been initialized, return error.

    Scan state descriptor list on instance structure to determine
    if 'tag' is present. Report error if duplicate tag.

    Create a new state description structure in instance structure.

    Create a new state structure if list of states is currently null.
```

Create or enlarge the block of state memory.

Return pointer to allocated memory.

}

#### 4.2.13 Function cm\_event\_get\_ptr

##### Summary

This function is called from code model C functions to return a pointer to state storage allocated with cm\_event\_alloc(). A tag specified in its argument list is used to locate the state in question. A second argument specifies whether the desired state is for the current timestep or from a preceding timestep. The location of the state in memory is then computed and returned.

##### Called By

Code model functions.

##### Returned Value

A pointer to the requested data bytes on success. A NULL pointer on failure.

##### Global Variables

```
Mif_Info_t g_mif_info; /* IN - Code model interface runtime data. */
```

##### PDL Description

```
void *cm_event_get_ptr(tag, timepoint)

int tag; /* IN - The user-specified tag for the memory block */
int timepoint; /* IN - The timepoint - 0=current, 1=previous */
{
    CKTcircuit *ckt; /* The main circuit structure */

    Get the address of 'ckt' and instance structure from 'g_mif_info'.

    If instance structure has not been initialized, return error.

    Scan state descriptor list on instance structure to find the
    descriptor for 'tag'. Report error if tag not found.

    Get the state pointer from the tail of the list of states
    on the instance.
```

Backup the number of timesteps indicated by 'timepoint'.

Return pointer.

}

#### 4.2.14 Function cm\_event\_queue

##### Summary

This function queues an event for an instance participating in the event-driven algorithm.

##### Called By

Code model functions.

##### Returned Value

Zero on success. Non-zero on failure.

##### Global Variables

```
Mif_Info_t g_mif_info; /* IN - Code model interface runtime data. */
```

##### PDL Description

```
int cm_event_queue(time)

double time; /* IN - The time of the event to be queued */
{
    CKTcircuit *ckt; /* The main circuit structure */

    Get the address of 'ckt' and instance structure from 'g_mif_info'.

    If 'time' <= current event time in 'g_mif_info', return error.

    Queue an event for this instance at the indicated time.
    [EVTqueue_inst()]

    Return OK.
}
```

#### 4.2.15 Function cm\_netlist\_get\_c

##### Summary

This is a special function designed for use with the c\_meter code model. It returns the parallel combination of the capacitance connected to the first port on the instance.

##### Called By

Code model functions. Primarily model c\_meter.

##### Returned Value

The value of the capacitance attached to the first port on the calling instance.

##### Global Variables

```
.Mif_Info_t g_mif_info; /* IN - Code model interface runtime data. */
```

##### PDL Description

```
double cm_netlist_get_c()

{
    CKTcircuit *ckt; /* The main circuit structure */

    Get the address of 'ckt' and instance structure from 'g_mif_info'.

    Get index of node associated with first port on instance.

    Initialize total capacitance value to zero.

    /* **** */
    /* Look for capacitors connected directly to cmeter input. */
    /* **** */

    Get the index of SPICE capacitor models in the 'ckt' device
    model list array. [SPICE3C1/INPtypelook()]

    Get the head of the list of capacitor models in the circuit
    from 'ckt'.

    Scan through all capacitor instances and add in values
```

of any capacitors connected to cmeter input.

```
/* ****
/* Look for capacitors connected through zero-valued voltage sources. */
/* ****
```

Get the index of SPICE independent voltage source models in the  
'ckt' device model list array. [SPICE3C1/INPtypelook()]

Get the head of the list of voltage source models in the circuit  
from 'ckt'.

Scan through all voltage source instances and add in values  
of any capacitors connected to cmeter input through voltage source.

Skip to next source if current source is not a DC source  
with value 0.0.

See if voltage source is connected to cmeter input.  
If so, get other node voltage source is connected to.  
If not, skip to next source

Scan through all capacitor instances and add in values  
of any capacitors connected to the voltage source node.

Return the total capacitance value.

}

#### 4.2.16 Function cm\_netlist\_get\_l

##### Summary

This is a special function designed for use with the l\_meter model. It returns the equivalent value of inductance connected to the first port on the instance.

##### Called By

Code model functions. Primarily model l\_meter.

##### Returned Value

The value of the inductance attached to the first port on the calling instance.

##### Global Variables

```
Mif_Info_t g_mif_info; /* IN - Code model interface runtime data. */
```

##### PDL Description

```
double cm_netlist_get_l()

{
    CKTcircuit *ckt; /* The main circuit structure */

    Get the address of 'ckt' and instance structure from 'g_mif_info'.

    Get index of node associated with first port on instance.

    Initialize total inductance value to zero.

    /* **** */
    /* Look for inductors connected directly to cmeter input. */
    /* **** */

    Get the index of SPICE inductor models in the 'ckt' device
    model list array. [SPICE3C1/INPtypelook()]

    Get the head of the list of inductor models in the circuit
    from 'ckt'.

    Scan through all inductor instances and add in values
```

```
of any inductors connected to cmeter input.

/* ****
/* Look for inductors connected through zero-valued voltage sources. */
/* ****

Get the index of SPICE independent voltage source models in the
'ckt' device model list array. [SPICE3C1/INPtypelook()]

Get the head of the list of voltage source models in the circuit
from 'ckt'.

Scan through all voltage source instances and add in values
of any inductors connected to cmeter input through voltage source.

Skip to next source if current source is not a DC source
with value 0.0.

See if voltage source is connected to cmeter input.
If so, get other node voltage source is connected to.
If not, skip to next source

Scan through all inductor instances and add in values
of any inductors connected to the voltage source node.

Return the total inductance value.
```

}

#### 4.2.17 Function cm\_smooth\_corner

##### Summary

This function smooths the transition between two slopes into a quadratic (parabolic) curve.

##### Called By

Code model functions.

##### Returned Value

None.

##### PDL Description

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

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

{
    Calculate parabolic equation constants from 'x_center', 'y_center',
    'domain', 'lower_slope', and 'upper_slope'.

    Calculate 'y_output' and 'dy_dx' from parabolic equation
    and 'x_input'.
}
```

#### 4.2.18 Function cm\_smooth\_discontinuity

##### Summary

This function smooths the transition between two values using an  $x^2$  function.

##### Called By

Code model functions.

##### Returned Value

None.

##### PDL Description

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

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

{
    Calculate regions and parabolic equation constants from 'x_lower',
    'y_lower', 'x_upper', and 'y_upper'.

    Calculate 'y_output' and 'dy_dx' from parabolic equation
    and 'x_input'.
}
```

#### 4.2.19 Function cm\_climit\_fcn

##### Summary

This is a special function created for use with the CLIMIT controlled limiter model.

##### Called By

Code model functions.

##### Returned Value

None.

##### PDL Description

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

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

{
    Compute adjusted upper and lower limits from 'cntl_upper', 'upper_delta',
    'cntl_lower', and 'lower_delta'.

    Compute linear range from adjusted upper and lower limits and 'limit_range'.

    Compute un-limited output from 'in', 'gain', and 'in_offset'.

    Determine operating region and compute limited output values
    and partial derivatives. [cm_smooth_discontinuity()]
}
```

#### 4.2.20 Function cm\_smooth\_pwl

##### Summary

This is a transfer curve function which accepts as input an “x” value, and returns a “y” value.

##### Called By

Code model functions.

##### Returned Value

The smoothed output value.

##### PDL Description

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

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

{
    Determine segment boundaries within which 'x_input' resides.

    Compute output and partials.

    Return output value.
}
```

#### 4.2.21 Function cm\_complex\_set

##### Summary

This function takes two real numbers representing the real and imaginary parts of a complex number and returns the number in complex form.

##### Called By

Code model functions.

##### Returned Value

The complex result.

##### PDL Description

```
Complex_t cm_complex_set(real, imag)

double real; /* IN - The real value of the complex number */
double imag; /* IN - The imaginary value of the complex number */
{
    Create a complex number 'c' with the real and imaginary
    parts specified in the argument list, and return it.
}
```

#### 4.2.22 Function cm\_complex\_add

##### Summary

This function adds two complex numbers and returns the complex result.

##### Called By

Code model functions.

##### Returned Value

The complex result.

#### PDL Description

```
Complex_t cm_complex_add(x, y)

Complex_t x; /* IN - The first operand */
Complex_t y; /* IN - The second operand */
{
    Add the two complex numbers in rectangular form and return the result.
}
```

#### 4.2.23 Function cm\_complex\_subtract

##### Summary

This function subtracts two complex numbers and returns the complex result. The second operand is subtracted from the first.

##### Called By

Code model functions.

##### Returned Value

The complex result.

#### PDL Description

```
Complex_t cm_complex_subtract(x, y)

Complex_t x; /* IN - The first operand */
Complex_t y; /* IN - The second operand */
{
    Subtract the second operand from the first in and return the result.
}
```

#### 4.2.24 Function cm\_complex\_multiply

##### Summary

This function multiplies two complex numbers and returns the complex result.

##### Called By

Code model functions.

##### Returned Value

The complex result.

##### PDL Description

```
Complex_t cm_complex_multiply(x, y)

Complex_t x; /* IN - The first operand */
Complex_t y; /* IN - The second operand */
{
    Multiply the two complex numbers in rectangular form and return the result.
}
```

#### 4.2.25 Function cm\_complex\_divide

##### Summary

This function divides two complex numbers and returns the complex result. The first operand is divide by the second. The magnitude of the second operand is internally limited to prevent floating point computation errors.

##### Called By

Code model functions.

##### Returned Value

The complex result.

##### PDL Description

```
Complex_t cm_complex_divide(x, y)

Complex_t x;    /* IN - The first operand */
Complex_t y;    /* IN - The second operand */
{
    Compute the magnitude squared of the second operand.

    Print error message and limit to 1e-100 if less than 1e-100.

    Compute and return the result of dividing the first operand by
    the second, using the limited magnitude squared to prevent
    numerical overflow.
}
```

### 4.3 Event-Driven Simulation

The Event-Driven Simulation (SIM-EVT) CSC implements an embedded event-driven simulation capability that coordinates with the matrix-based, timestepped algorithm used by SPICE. The event-driven simulation capability allows for orders-of-magnitude speed improvements over the standard SPICE algorithm for digital and other discrete time models.

The algorithm is implemented separately from the representation of the data passed between models so that multiple data types can be defined. This capability is called “user-defined nodes”. New data types (node types) can be introduced by a user in a way analogous to the introduction of user-defined code models. A node resolution algorithm is used to determine the final value obtained by a node when multiple models output to that node.

```
EVTaccept
EVTbackup
    EVTbackup_node_data
    EVTbackup_state_data
    EVTbackup_msg_data
    EVTbackup_inst_queue
    EVTbackup_output_queue
EVTcall_hybrids
EVTdequeue
    EVTdequeue_output
    EVTdequeue_inst
    EVTprocess_output
EVTinit
    EVTcheck_nodes
    EVTcount_hybrids
    EVTinit_info
    EVTinit_queue
    EVTinit_limits
EVTiter
EVTload
    EVTcreate_state
    EVTadd_msg
    EVTcreate_output_event
    EVTprocess_output
EVTnext_time
EVTnode_copy
EVTop
    EVTnode_compare
EVTprint
    get_index
    print_data
EVTdump
    EVTsend_line
EVTqueue_output
EVTqueue_inst
EVTsetup
```

XSPICE Simulator  
Software Design Document

Detailed Design  
Event-Driven Simulation

```
EVTsetup_queues
EVTsetup_data
EVTsetup_jobs
EVTsetup_load_ptrs
EVTtermInsert
EVTinst_insert
EVTnode_insert
EVTport_insert
EVTooutput_insert
```

#### 4.3.1 Function EVTaccept

##### Summary

This function is called at the end of a successful (accepted) analog timepoint. It saves pointers to the states of the queues and data at this accepted time.

##### Called By

DCtran() CKT/DCtran.c

##### Returned Value

None.

##### PDL Description

```
void EVTaccept(ckt, time)

CKTcircuit *ckt; /* INOUT - The circuit structure */
double time; /* IN - The time at which analog solution was computed */
{
    Evt_Ckt_Data_t     *evt;          /* Main event-driven data structure */
    Evt_Inst_Queue_t   *inst_queue;   /* Substructure of 'evt' holding inst queue data */
    Evt_Output_Queue_t *output_queue; /* Substructure of 'evt' holding output queue data */
    Evt_Node_Data_t    *node_data;    /* Substructure of 'evt' holding node data */
    Evt_State_Data_t   *state_data;   /* Substructure of 'evt' holding state data */
    Evt_Msg_Data_t     *msg_data;     /* Substructure of 'evt' holding message data */
```

Exit if event structure 'evt' in 'ckt' indicates there are no event-driven instances.

Get the 'inst\_queue' structure from within 'evt'.

Process the 'inst\_queue' data as follows:

Loop through list of items modified since last time.

Get the index of the inst modified.

Update the 'last\_step' pointer of the queue for this index.

Reset the 'modified' flag in the queue for this index.

Set 'last\_time' in queue equal to 'time' and reset number modified to zero.

Get the 'output\_queue' structure from within 'evt'.

Process the 'output\_queue' data as follows:

Loop through list of items modified since last time.

Get the index of the inst modified.

Update the 'last\_step' pointer of the queue for this index.

Reset the 'modified' flag in the queue for this index.

Set 'last\_time' in queue equal to 'time' and reset number modified to zero.

Get the 'node\_data' structure from within 'evt'.

Process 'node\_data' as follows:

Loop through list of items modified since last time.

Get the index of the inst modified.

Update the 'last\_step' pointer of the queue for this index.

Reset the 'modified' flag in the queue for this index.

Reset number modified to zero.

Get the 'state\_data' structure from within 'evt'.

Process 'state\_data' as follows:

Loop through list of items modified since last time.

Get the index of the inst modified.

Update the 'last\_step' pointer of the queue for this index.

Reset the 'modified' flag in the queue for this index.

Reset number modified to zero.

Get the 'msg\_data' structure from within 'evt'.

Process 'msg\_data' as follows:

```
Loop through list of items modified since last time.  
Get the index of the inst modified.  
Update the 'last_step' pointer of the queue for this index.  
Reset the 'modified' flag in the queue for this index.  
Reset number modified to zero.  
}
```

#### 4.3.2 Function EVTbackup

##### Summary

This function resets the queues and data structures to their state at the new analog simulation time specified. The algorithms in this file assume the following timestep coordination between analog and event-driven algorithms:

```
while(not end of analysis) {  
  
    while (next event time <= next analog time) {  
        do event solution with call_type = event_driven  
        if any instance set analog breakpoint < next analog time  
            set next analog time to breakpoint  
    }  
  
    do analog timestep solution with call_type = analog  
    call all hybrid models with call_type = event_driven  
  
    if(analog solution doesn't converge)  
        Call EVTbackup  
    else  
        Call EVTaccept  
}
```

##### Called By

DCtran() CKT/DCtran.c

##### Returned Value

None.

##### PDL Description

```
void EVTbackup(ckt, new_time)  
  
CKTcircuit *ckt;          /* INOUT - The circuit structure */  
double     new_time;       /* IN - The time to backup to */  
{  
    Evt_Ckt_Data_t     *evt;      /* Main event-driven data structure */  
    Evt_Data_t         data;     /* Event-driven data computed during analysis */
```

Backup the node data. [EVTbackup\_node\_data()]  
Backup the state data. [EVTbackup\_state\_data()]  
Backup the msg data. [EVTbackup\_msg\_data()]  
Backup the inst queue. [EVTbackup\_inst\_queue()]  
Backup the output queue. [EVTbackup\_output\_queue()]  
  
Increment the count of transient timestep backups in the statistics member  
within the 'data' element of 'evt' in 'ckt'.  
}

#### 4.3.2.1 Static Function EVTbackup\_node\_data

##### Summary

This function resets the node data structure to the state it was in at the analog simulation time specified.

##### Called By

EVTbackup()                   EVT/EVTbackup.c

##### Returned Value

None.

##### PDL Description

```
static void EVTbackup_node_data(ckt, new_time)

CKTcircuit *ckt;               /* INOUT - The circuit structure */
double      new_time;           /* IN     - The time to backup to */
{
    Evt_Ckt_Data_t    *evt;       /* Main event-driven data structure */
    Evt_Node_Data_t    *node_data; /* Substructure of 'evt' holding node data */

    Access 'node_data' within the event structure 'evt' in 'ckt'.

    Loop through list of indexes modified since last accepted timepoint.

    Scan data for this node from 'last_step' forward up to 'new_time'
    to determine new setting for 'tail' pointer.

    Move later data to the 'free' list.

    Copy data from the location at 'tail' to 'rhs' and 'rhsold'
    members of the event node data structure. [EVTnode_copy()]

    Loop through the modified list and update and compact it as follows,

    If nothing after 'last_step',
        Remove this index from the list.
```

```
Else,  
    Keep the index in the list.  
}
```

#### 4.3.2.2 Static Function EVTbackup\_state\_data

## Summary

This function resets the state data structure to the state it was in at the analog simulation time specified.

### Called By

EVTbackup() EVT/EVTbackup.c

### Returned Value

None.

## PDL Description

```
static void EVTbackup_state_data(ckt, new_time)

CKTcircuit *ckt;           /* INOUT - The circuit structure */
double     new_time;        /* IN      - The time to backup to */
{

    Evt_Ckt_Data_t     *evt;           /* Main event-driven data structure */
    Evt_State_Data_t   *state_data;    /* Substructure of 'evt' holding state data */

    Access 'state_data' within the event structure 'evt' in 'ckt'.

    Loop through list of indexes modified since last accepted timepoint.

        Scan data for this instance from 'last_step' forward up to 'new_time'
        to determine new setting for 'tail' pointer.

        Move later data to the 'free' list.

    Loop through the modified list and update and compact it as follows,
    If nothing after 'last_step',
        Remove this index from the list.
    Else,
        Keep the index in the list.
}
```

#### 4.3.2.3 Static Function EVTbackup\_msg\_data

##### Summary

This function resets the message data structure to the state it was in at the analog simulation time specified.

##### Called By

EVTbackup()                   EVT/EVTbackup.c

##### Returned Value

None.

##### PDL Description

```
static void EVTbackup_msg_data(ckt, new_time)

CKTcircuit *ckt;               /* INOUT - The circuit structure */
double      new_time;           /* IN     - The time to backup to */
{
    Evt_Ckt_Data_t    *evt;       /* Main event-driven data structure */
    Evt_Msg_Data_t    *msg_data;   /* Substructure of 'evt' holding msg data */

    Access 'msg_data' within the event structure 'evt' in 'ckt'.

    Loop through list of indexes modified since last accepted timepoint.

        Scan data for this instance from 'last_step' forward up to 'new_time'
        to determine new setting for 'tail' pointer.

        Move later data to the 'free' list.

    Loop through the modified list and update and compact it as follows,

        If nothing after 'last_step',
            Remove this index from the list.

        Else,
            Keep the index in the list.
}
```

#### 4.3.2.4 Static Function EVTbackup\_inst\_queue

## Summary

This function resets the inst data structures to their state back to the new analog simulation time specified.

## Called By

## EVTbackup() EVT/EVTbackup.c

## Returned Value

None.

## PDL Description

```
static void EVTbackup_inst_queue(ckt, new_time)

CKTcircuit *ckt;           /* INOUT - The circuit structure */
double      new_time;       /* IN     - The time to backup to */

{
    Evt_Ckt_Data_t      *evt;          /* Main event-driven data structure */
    Evt_Inst_Queue_t    *inst_queue;   /* Substructure of 'evt' holding instance queue */

Access 'inst_queue' within the event structure 'evt' in 'ckt'.

Loop through list of indexes modified since last accepted timepoint
and remove selected events as follows:

Scan forward from 'last_step' and cut out data with posted time
> 'new_time' and add it to the 'free' list.

Scan forward from 'last_step' and set 'current' to first
event with event_time > 'new_time'.

Add set of items modified to set of items pending before updating the
pending list. This is needed because things may have been pulled from
the pending list in the course of queue processing since the last timestep.

Update the 'pending' list and 'next_time' as follows:

Loop through list of items in 'pending' list,
If nothing in queue at 'last_step', remove this index from the pending list.
```

Else, keep the index and update 'next\_time' to minimum of 'next\_time' and event time for this event.

Update the list of items modified since last accepted timepoint as follows:

Loop through list of items in 'modified' list,

If there are any items in list with posted time > 'last\_time',

Keep this index in the modified list.

Else

Remove the index from the modified list.

}

#### 4.3.2.5 Static Function EVTbackup\_output\_queue

## Summary

This function resets the output data structures to their state back to the new analog simulation time specified.

### Called By

## EVTbackup()

### Returned Value

None.

## PDL Description

```

static void EVTbackup_output_queue(ckt, new_time)

CKTcircuit *ckt;           /* INOUT - The circuit structure */
double      new_time;       /* IN     - The time to backup to */
{
    Evt_Ckt_Data_t   *evt;           /* Main event-driven data structure */
    Evt_Output_Queue_t *output_queue; /* Substructure of 'evt' holding output queue */

Access 'output_queue' within the event structure 'evt' in 'ckt'.

Loop through list of indexes modified since last accepted timepoint
and remove selected events as follows:

    Scan forward from 'last_step' and cut out data with posted time
    > 'new_time' and add it to the 'free' list.

    Scan forward from 'last_step' and set 'current' to first
    event with event_time > 'new_time'.

Add set of items modified to set of items pending before updating the
pending list. This is needed because things may have been pulled from
the pending list in the course of queue processing since the last timestep.

Update the 'pending' list and 'next_time' as follows:

    Loop through list of items in 'pending' list,
        If nothing in queue at 'last_step', remove this index from the pending list.

```

Else, keep the index and update 'next\_time' to minimum of 'next\_time' and event time for this event.

Update the list of items modified since last accepted timepoint as follows:

Loop through list of items in 'modified' list,

If there are any items in list with posted time > 'last\_time',

Keep this index in the modified list.

Else

Remove the index from the modified list.

}

#### 4.3.3 Function EVTcall\_hybrids

##### Summary

This function calls all models which have both analog and event-driven ports. It is called following successful evaluation of an analog iteration attempt to allow events to be scheduled by the hybrid models. The “CALL\_TYPE” is set to “EVENT\_DRIVEN” when the model is called from this function.

##### Called By

EVTTop()	EVT/EVTTop.c
DCtran()	CKT/DCtran.c
DCtrCurv()	CKT/DCtrCurv.c

##### Returned Value

None.

##### PDL Description

```
void EVTcall_hybrids(ckt)

CKTcircuit *ckt;           /* INOUT - The circuit structure */
{
    Evt_Ckt_Data_t     *evt;           /* Main event-driven data structure */
    Evt_Info_t          info;          /* Info substructure of 'evt' */

    Get the number of hybrid instances and the list of their instance indexes from
    'info' within 'evt' in the 'ckt' structure.

    Evaluate the code model for each instance with code model call type set to
    EVENT_DRIVEN.      [EVTload()]
}
```

#### 4.3.4 Function EVTdequeue

##### Summary

This function removes any items on the output and instance queues with event times matching the specified simulation time.

##### Called By

DCtran() CKT/DCtran.c

##### Returned Value

None.

##### PDL Description

```
void EVTdequeue(ckt, time)

CKTcircuit *ckt;      /* INOUT - The circuit structure */
double     time;       /* IN    - The event time of the events to dequeue */

{
    Take all items on output queue with matching time and set changed flags
    in output queue.  [EVTdequeue_output()]

    Take all items on inst queue with matching time and set to_call flags in
    inst queue.  [EVTdequeue_inst()]
}
```

#### 4.3.4.1 Static Function EVTdequeue\_output

##### Summary

This function de-queues output events with times matching the specified time.

##### Called By

EVTdequeue()                   EVT/EVTdequeue.c

##### Returned Value

None.

##### PDL Description

```
static void EVTdequeue_output(ckt, time)

CKTcircuit *ckt;         /* INOUT - The circuit structure */
double       time;        /* IN      - The event time of the events to dequeue */
{
    Evt_Ckt_Data_t        *evt;         /* Main event-driven data structure */
    Evt_Output_Queue_t    *output_queue; /* Substructure of 'evt' holding output queue */

    Access 'output_queue' within the event structure 'evt' in 'ckt'.

    Exit if nothing pending on 'output_queue' or if 'next_time' != 'time'.

    Scan the list of outputs pending.

    Get the index of the output.

    Get pointer to next event in queue at this index.

    If event time does not match 'time',
        Continue to next output.

    Else,
        Pull the event from the queue and process it.
        [EVTprocess_output()]

    Move 'current' to point to next non-removed item in list.

    Put this index into the 'modified' list if not already there.
```

Update the 'pending' list and 'next\_time' as follows:

Loop through list of items in 'pending' list,

If nothing in queue at 'last\_step', remove this index from the pending list.

Else, keep the index and update 'next\_time' to minimum of 'next\_time' and event time for this event.

}

#### 4.3.4.2 Static Function EVTdequeue\_inst

##### Summary

This function de-queues instance events with times matching the specified time.

##### Called By

EVTdequeue()                   EVT/EVTdequeue.c

##### Returned Value

None.

##### PDL Description

```
void EVTdequeue_inst(ckt, time)

CKTcircuit *ckt;         /* INOUT - The circuit structure */
double      time;         /* IN    - The event time of the events to dequeue */
{
    Evt_Ckt_Data_t    *evt;         /* Main event-driven data structure */
    Evt_Inst_Queue_t *inst_queue;    /* Substructure of 'evt' holding instance queue */

    Access 'inst_queue' within the event structure 'evt' in 'ckt'.

    Exit if nothing pending on 'inst_queue' or if 'next_time' != 'time'.

    Scan the list of instance events pending.

        Get the index of the instance.

        Get pointer to next event in queue at this index.

        If event time does not match 'time',

            Continue to next instance.

        Else,

            Pull the event from the queue and process it by adding it to
            the 'to_call' list.

            Move 'current' to point to next non-removed item in list.

            Put this index into the 'modified' list if not already there.
```

Update the 'pending' list and 'next\_time' as follows:

Loop through list of items in 'pending' list,

If nothing in queue at 'last\_step', remove this index from the pending list.

Else, keep the index and update 'next\_time' to minimum of 'next\_time' and event time for this event.

}

#### 4.3.4.3 Static Function EVTprocess\_output

##### Summary

This function processes a specified output after it is pulled from the queue.

##### Called By

EVTdequeue\_output()      EVT/EVTdequeue.c

##### Returned Value

None.

##### Global Variables

```
Evt_Udn_Info_t *g_evt_udn_info[]; /* IN - Function pointer data for user-defined nodes */
```

##### PDL Description

```
static void EVTprocess_output(ckt, output_index, value)

CKTcircuit *ckt;                /* INOUT - The circuit structure */
int         output_index;        /* IN - The index of the output to process */
void        *value;              /* IN - The output data to process */

{
    Evt_Ckt_Data_t        *evt;                /* Main event-driven data structure */
    Evt_Output_Queue_t *output_queue; /* Output queue data */
    Evt_Node_t          *rhs;                /* "Solution vector" where outputs are computed */
    Evt_Node_t          *rhsold;            /* "Solution vector" holding inputs to code models */

    Access 'output_queue' within the event structure 'evt' in 'ckt'.

    Access 'rhs' and 'rhsold' within the event structure 'evt' in 'ckt'.

    Get the index of the node this output is connected to from the 'evt' structure
    info data.

    Determine if output is different from current value in 'rhsold' at this node index
    and copy it to 'rhs' and 'rhsold' if so as follows:

        If number of outputs connected to the node is greater than one,
```

Get the subindex for this output on the node from the info structure in 'evt'.

Compare 'value' to the value in 'rhsold' at this node index and output subindex. [UDN 'compare' function from g\_evt\_udn\_info]

If not equal,

Copy 'value' to 'rhs' for this node and output subindex.  
[UDN 'copy' function from g\_evt\_udn\_info]

Copy 'value' to 'rhsold' for this node and output subindex.  
[UDN 'copy' function from g\_evt\_udn\_info]

Else,

Compare 'value' to the value in 'rhsold' at this node index.  
[UDN 'compare' function from g\_evt\_udn\_info]

If not equal,

Copy 'value' to 'rhs' for this node.  
[UDN 'copy' function from g\_evt\_udn\_info]

Copy 'value' to 'rhsold' for this node.  
[UDN 'copy' function from g\_evt\_udn\_info]

If copy operation was performed,

Mark this output as changed in 'output\_queue'.

}

#### 4.3.5 Function EVTinit

##### Summary

This function allocates and initializes evt structure elements after the number of instances, nodes, etc. have been determined in parsing during INPpas2. EVTinit also checks to be sure no nodes have been used for both analog and event-driven algorithms simultaneously.

##### Called By

if\_inpdeck() FTE/spiceif.c

##### Returned Value

Zero (OK) on success. Non-zero otherwise.

##### PDL Description

```
int EVTinit(ckt)

CKTcircuit *ckt;      /* INOUT - The circuit structure */
{
    Evt_Ckt_Data_t     *evt;      /* Main event-driven data structure in 'ckt' */

    Get 'evt' structure from 'ckt'.

    Check count of event-driven instances in 'evt'.

    Return OK immediately if there are no event-driven instances.

    Count the number of hybrids and hybrid outputs. [EVTcount_hybrids()]

    Exit with error if there are no hybrids in the circuit.

    Check that event nodes have not been used as analog nodes also.
    [EVTcheck_nodes()]

    Create info table arrays in 'evt' structure. [EVTinit_info()]

    Setup queue data structures in 'evt'. [EVTinit_queue()]

    Initialize limits in 'evt'. [EVTinit_limits()]

    Return OK.
}
```

#### 4.3.5.1 Static Function EVTcheck\_nodes

##### Summary

This function checks for illegal use of a single node by both the analog algorithm and the event algorithm.

##### Called By

EVTinit()                   EVT/EVTinit.c

##### Returned Value

Zero (OK) if there are no uses of a node by both algorithms. Non-zero otherwise.

##### PDL Description

```
static int EVTcheck_nodes(ckt)

CKTcircuit *ckt;         /* IN - The circuit structure */
{
    Evt_Ckt_Data_t    *evt;         /* Main event-driven data structure in 'ckt' */
    Evt_Node_Info_t   *event_node; /* Linked list of info about event-driven nodes */
    CKTnode           *analog_node; /* Linked list of info about SPICE3C1 analog nodes */

    Get pointer to event-driven node data 'event_node' from 'evt' structure
    in 'ckt'.

    Scan through 'event_node' list,
        Get pointer to analog node data list 'analog_node'.
        Scan through 'analog_node' list,
            If name of node in 'event_node' equals name of node in 'analog_node',
                Return error.

    Return OK.
}
```

#### 4.3.5.2 Static Function EVTcount\_hybrids

##### Summary

This function determines the number of hybrid (analog and event-driven) instances in the circuit.

##### Called By

EVTinit()                   EVT/EVTinit.c

##### Returned Value

Zero (OK) on success. Non-zero otherwise.

##### PDL Description

```
static int EVTcount_hybrids(ckt)

CKTcircuit *ckt;         /* INOUT - The circuit structure */
{
    Evt_Ckt_Data_t    *evt;    /* Main event-driven data structure in 'ckt' */
    Evt_Inst_Info_t   *inst;    /* Linked list of info about event-driven instances */
    MIFinstance       *fast;    /* Pointer to structure describing an instance */

    Get pointer to event-driven instance list ('inst') from 'evt' structure
    in 'ckt'.

    Scan through instances in 'inst',
        Get pointer to the structure ('fast') describing the instance.

        If 'fast' flags (set during parsing) indicate that instance is both
        analog and event-driven,
            Increment count of hybrid instances in 'evt'.

            Scan through list of ports on 'fast' and count number of
            event-driven outputs.

            Increment count of hybrid outputs in 'evt'.
}
```

#### 4.3.5.3 Static Function EVTinit\_info

##### Summary

This function creates information tables used to hold data on instances, ports, outputs, and nodes in the event-driven circuit representation. These arrays allow faster access to data associated with instances, nodes, ports, and outputs than that provided by the linked-list representations created during parsing.

##### Called By

EVTinit()                   EVT/EVTinit.c

##### Returned Value

Zero (OK) on success. Non-zero otherwise.

##### PDL Description

```
static int EVTinit_info(ckt)

CKTcircuit *ckt;         /* INOUT -> The circuit structure */
{
    Evt_Ckt_Data_t     *evt;         /* Main event-driven data structure in 'ckt' */

    Evt_Inst_Info_t    *inst;         /* Linked list of instance information */
    Evt_Node_Info_t    *node;         /* Linked list of node information */
    Evt_Port_Info_t    *port;         /* Linked list of port information */
    Evt_Output_Info_t  *output;       /* Linked list of output information */

    Evt_Inst_Info_t    **inst_table;    /* Table of instance information */
    Evt_Node_Info_t    **node_table;    /* Table of node information */
    Evt_Port_Info_t    **port_table;    /* Table of port information */
    Evt_Output_Info_t  **output_table;   /* Table of output information */
    int                *hybrid_index;    /* Table of hybrid instance indexes */
```

Get event-driven data 'evt' from 'ckt'.

Get count of instances from 'evt' and allocate space for the 'inst\_table' member of 'evt'.

Scan through elements in 'inst' linked-list member of 'evt' and assign pointers to instances into 'inst\_table' member of 'evt'.

Get count of nodes from 'evt' and allocate space for the 'node\_table' member of 'evt'.

Scan through elements in 'node' linked-list member of 'evt' and assign pointers to nodes into 'node\_table' member of 'evt'.

Get count of ports from 'evt' and allocate space for the 'port\_table' member of 'evt'.

Scan through elements in 'port' linked-list member of 'evt' and assign pointers to ports into 'port\_table' member of 'evt'.

Get count of outputs from 'evt' and allocate space for the 'output\_table' member of 'evt'.

Scan through elements in 'output' linked-list member of 'evt' and assign pointers to outputs into 'output\_table' member of 'evt'.

Get count of hybrids from 'evt' and allocate space for the 'hybrid\_index' array member of 'evt'.

Scan through elements in 'inst\_table' member of 'evt',

If instance is hybrid,

Assign index of instance into 'hybrid\_index' member of 'evt'.

Return OK.

}

#### 4.3.5.4 Static Function EVTinit\_queue

##### Summary

This function prepares the event-driven queues for simulation.

##### Called By

EVTinit()                   EVT/EVTinit.c

##### Returned Value

Zero (OK) on success. Non-zero otherwise.

##### PDL Description

```
static int EVTinit_queue(ckt)

CKTcircuit *ckt;         /* INOUT - The circuit structure */
{
    Evt_Ckt_Data_t       *evt;       /* Main event-driven data structure in 'ckt' */

    Evt_Inst_Queue_t     *inst_queue;   /* Instance queue structure in 'evt' */
    Evt_Node_Queue_t     *node_queue;   /* Node queue structure in 'evt' */
    Evt_Output_Queue_t   *output_queue; /* Output queue structure in 'evt' */

    Get event-driven data 'evt' from 'ckt'.

    Get count of instances from 'evt' and allocate space for the 'inst_queue'
    member of 'evt'.

    Get count of nodes from 'evt' and allocate space for the 'node_queue'
    member of 'evt'.

    Get count of outputs from 'evt' and allocate space for the 'output_queue'
    member of 'evt'.
}
```

#### 4.3.5.5 Static Function EVTinit\_limits

##### Summary

This function initializes the iteration limits applicable to the event-driven algorithm.

##### Called By

EVTinit()                   EVT/EVTinit.c

##### Returned Value

Zero (OK) on success. Non-zero otherwise.

##### PDL Description

```
static int EVTinit_limits(ckt)

CKTcircuit *ckt;         /* INOUT - The circuit structure */
{
    Evt_Ckt_Data_t       *evt;       /* Main event-driven data structure in 'ckt' */

    Get event-driven data 'evt' from 'ckt'.

    Set maximum number of event load calls for a single event iteration
    to the number of event outputs. This allows for the maximum possible
    number of events that can trickle through any circuit that does not
    contain loops.

    Set maximum number of alternations between analog and event-driven iterations
    to the number of event outputs on hybrids.
}
```

#### 4.3.6 Function EVTiter

##### Summary

This function iterates through event-driven outputs and instances until the outputs no longer change. The general algorithm used is:

Do:

```
Scan list of changed outputs from instance evaluations
  Put entry in the node queue 'to_eval' list for
    each node associated with a changed output.

Scan node 'to_eval' list
  Resolve nodes with multiple outputs posted.
  Create inverted state on nodes with attached
    inverted inputs.
  Put items on the instance queue 'to_call' list
    for each changed node.
  If transient analysis, put state of the node
    into the node data structure.

Scan instance 'to_call' list
  Call EVTload() for each instance on list to
    evaluate each instance's response to its new
    inputs.

While there are changed outputs.
```

##### Called By

EVTtop()	EVT/EVTtop.c
DCtran()	CKT/DCtran.c

##### Returned Value

Zero on success. Non-zero otherwise.

##### Global Variables

```
Evt_Udn_Info_t *g_evt_udn_info[]; /* IN - Function pointer data for user-defined nodes */
```

## PDL Description

```
int EVTiter(ckt)

CKTcircuit *ckt;      /* INOUT - The circuit structure */
{
    Evt_Ckt_Data_t     *evt;      /* Main event-driven data structure in 'ckt' */

    Evt_Output_Queue_t *output_queue; /* Output queue data structure in 'evt' */
    Evt_Node_Queue_t  *node_queue;  /* Node queue data structure in 'evt' */
    Evt_Inst_Queue_t  *inst_queue;  /* Instance queue data structure in 'evt' */

    Mif_Boolean_t      *to_eval;    /* Flags in 'node_queue' */
    Mif_Boolean_t      *to_call;    /* Flags in 'inst_queue' */

    Get event-driven data 'evt' from 'ckt'.

    Get 'output_queue', 'node_queue', and 'inst_queue' structures from 'evt'.

    Loop until return statement reached inside loop when no more output change,
    or until limits data in 'evt' indicates too many passes through loop.

    Loop through changed items in 'output_queue',
        Get index of node that output is connected to from output table info
        in 'evt'.
        If this node is not already on node queue's 'to_eval' list, add it.
        Reset the changed flag on the 'output_queue' for this output.

    Loop through items in 'to_eval' list of 'node_queue',
        If number of outputs on node is > 1,
            Resolve the node value if there are multiple outputs on it,
            and test if new node value is different than old value.
            [UDN 'resolve' function from g_evt_udn_info]
            [UDN 'compare' function from g_evt_udn_info]

        Else,
            EVTload() function has already determined that they were
            not equal so no need to do test here.

        If not equal
            Make inverted copy in 'evt' node data structure if needed,
            [UDN 'copy' function from g_evt_udn_info]
            [UDN 'invert' function from g_evt_udn_info]
```

Place indexes of instances with inputs connected  
to the node in the 'to\_call' list of the instance queue.

If transient analysis mode,

Save the node data onto the node results structure in 'evt'  
and mark that it has been modified, even if the resolved node  
value has not changed (since outputs have). [EVTnode\_copy()]

Clear the 'to\_eval' flag on the node queue.

Loop through items in 'to\_call' list of 'inst\_queue',

Call the instance's code model function to compute new outputs.  
[EVTload()]

If analysis type is DC,

Increment count of operating point passes in 'evt' statistics  
data.

If no outputs changed,

Iteration is over, so return with success.

Too many passes through loop, return with error.

}

#### 4.3.7 Function EVTload

##### Summary

This function is used to call a specified event-driven or hybrid code model during an event-driven iteration. The “CALL\_TYPE” is set to “EVENT\_DRIVEN” when the model is called from this function.

##### Called By

EVTtop()	EVT/EVTtop.c
DCtran()	CKT/DCtran.c

##### Returned Value

Zero on success. Non-zero otherwise.

##### Global Variables

```
Mif_Info_t      g_mif_info;    /* INOUT - Code model interface runtime data. */
SPICEdev        *DEVICES[];   /* IN      - Information about models known to simulator. */
```

##### PDL Description

```
int EVTload(ckt, inst_index)

CKTcircuit *ckt;          /* INOUT - The circuit structure */
int       inst_index;     /* IN      - The instance to call code model for */
{
    Evt_Ckt_Data_t    *evt;      /* Main event-driven data structure in 'ckt' */
    MIFinstance        *inst;     /* The instance to be evaluated */
    Mif_Private_t      cm_data;   /* Data structure passed to code model */

    Get event-driven data 'evt' from 'ckt'.

    Use 'inst_index' to locate 'inst' in instance table info structure
    of 'evt'.

    Setup circuit data in 'cm_data' structure to be passed to code model function.

    Assign data needed by cm_... functions into 'g_mif_info' (e.g. 'inst' pointer,
    call type, etc.)
```

If 'inst' structure indicates this is not the initialization pass and 'g\_mif\_info' structure indicates we are in transient analysis mode,

Create a new state for the instance in the state data structure in 'evt'.  
[EVTcreate\_state()]

Loop through all connections on the instance 'inst' and setup 'load', 'total\_load', and 'msg' on all ports, and the changed flags and output pointers on all outputs as follows,

If connection is null,

Continue to next.

Loop through each port on the connection.

Skip if port is null.

If port type is digital or User-Defined type,

Initialize the 'msg' pointer on the port to NULL, initialize the 'load' value to zero, and get the total load from the 'node\_data' member of 'evt'.

If connection is an output, initialize the changed flag to true and create a new output event object in the free list if transient analysis mode. [EVTcreate\_output\_event()]

Else,

Get the analog input value. All we need to do is set it to zero if 'CKTmode' in 'ckt' indicates that this is the first evaluation. Otherwise, the value should still be around from last successful analog call.

Finish preparation of 'cm\_data' by setting members to point to the data in 'inst'.

Get the code model type from 'inst' and call the code model function through the function pointer in the SPICE3 'DEVICES' data structure.  
[<code model function>]

Loop through all connections on 'inst' and process the messages and event outputs as follows,

Loop through each port,

Skip if port is null.

```
Process the message if any. [EVTadd_msg()]

If this is the initialization pass, add the load value output
by the model into the total load on the connected node.

If connection is not an event output, continue to next port.

Process the output. [EVTprocess_output()]

Increment counts of operating point load calls and transient analysis
load calls in statistics member of 'evt'.

Mark 'inst' as having been initialized.

}
```

#### 4.3.7.1 Static Function EVTcreate\_state

##### Summary

This function creates a new state storage area for a particular instance during an event-driven simulation. New states must be created so that old states are saved and can be accessed by code models in the future. The new state is initialized to the previous state value.

##### Called By

EVTload()                   EVT/EVTload.c

##### Returned Value

None.

##### Global Variables

```
Mif_Info_t       g_mif_info;       /* INOUT - Code model interface runtime data. */
```

##### PDL Description

```
static void EVTcreate_state(ckt, inst_index)

CKTcircuit *ckt;       /* INOUT - The circuit structure */
int       inst_index;   /* IN     - The instance to create state for */
{
    Evt_Ckt_Data_t    *evt;       /* Main event-driven data structure in 'ckt' */
    Evt_State_Data_t   *state_data; /* State data member of 'evt' */

    Get event-driven data 'evt' from 'ckt'.

    Get 'state_data' from 'evt'.

    Return immediately if 'state_data' indicates there are no states needed
    on this instance.

    Get size of state data to be allocated from 'state_data'.

    Allocate a new state for the instance. Try to get the memory from
    the free list on 'state_data' if possible. Otherwise, call the
```

storage allocator.

Append the new state into the 'state\_data' linked list at the 'tail'  
and update the 'tail' pointer.

Copy the previous state in the list to the new state and set the step  
to the current event step as found in 'g\_mif\_info'.

Mark that the 'state\_data' for this instance has been modified.

}

#### 4.3.7.2 Static Function EVTadd\_msg

##### Summary

This function records a message output by a code model into the message results data structure.

##### Called By

EVTload()                   EVT/EVTload.c

##### Returned Value

None.

##### Global Variables

```
Mif_Info_t       g_mif_info;       /* INOUT - Code model interface runtime data. */
```

##### PDL Description

```
static void EVTadd_msg(ckt, port_index, msg_text)

CKTcircuit *ckt;       /* INOUT - The circuit structure */
int       port_index;   /* IN     - The port to add message to */
char      *msg_text;    /* IN     - The message text */

{
    Evt_Ckt_Data_t    *evt;       /* Main event-driven data structure in 'ckt' */
    Evt_Msg_Data_t    *msg_data;   /* Message data member of 'evt' */

    Get event-driven data 'evt' from 'ckt'.

    Get 'msg_data' from 'evt'.

    Allocate a new message structure for the instance. Try to get the memory from
    the free list on 'msg_data' if possible. Otherwise, call the storage allocator.

    Append the new message structure into the 'msg_data' linked list at the 'tail'
    and update the 'tail' pointer.

    Copy the text in 'msg_text' to this new message structure. [MIFcopy()]
```

XSPICE Simulator  
Software Design Document

Detailed Design  
Event-Driven Simulation

Set the step on the new message to the current event step as found in 'g\_mif\_info'.

Mark that the 'msg\_data' for this instance has been modified.

}

#### 4.3.7.3 Static Function EVTcreate\_output\_event

## Summary

This function creates a new output event. The output event is created at the head of the free list in the output queue and a pointer to it is returned. It is assumed that the caller will move the event to a new location if needed after the code model is called.

### Called By

## EVTload() EVT/EVTload.c

## Returned Value

None.

## Global Variables

```
Evt_Udn_Info_t *g_evt_udn_info[]; /* IN - Function pointer data for user-defined nodes */
```

## PDL Description

```
static void EVTcreate_output_event(ckt, node_index, output_index, value_ptr)

CKTcircuit *ckt;           /* INOUT - The circuit structure */
int      node_index;       /* IN    - The node type the port is connected to */
int      output_index;     /* IN    - The output index for this port */
void    **value_ptr;       /* OUT   - The event created */

{
    Evt_Ckt_Data_t      *evt;           /* Main event-driven data structure in 'ckt' */
    Evt_Output_Queue_t  *output_queue; /* Output queue structure in 'evt' */

    Get the 'evt' structure from 'ckt'.

    Get the 'output_queue' structure from 'evt'.

    If free list in 'output_queue' is not empty,
        Assign 'value_ptr' to point to the head of the free list.

    Else,
```

Allocate a new event structure.

Call the user-defined node 'create' function to allocate space for  
the value associated with the event.  
[UDN 'create' function through g\_eve\_udn\_info].

Splice the new event into the head of the free list in 'output\_queue' and set  
'value\_ptr' to point to the head of the free list.

}

#### 4.3.7.4 Static Function EVTprocess\_output

## Summary

This function processes an event-driven output produced by a code model. If transient analysis mode, the event is placed into the output queue according to its (non-zero) delay. If DC analysis, the event is processed immediately.

### Called By

## EVTload() EVT/EVTload.c

#### Returned Value

None.

## Global Variables

```
Mif_Info_t      g_mif_info;          /* INOUT - Code model interface runtime data. */
Evt_Udn_Info_t *g_evt_udn_info[]; /* IN - Function pointer data for user-defined nodes */
```

## PDL Description

```

static void EVTprocess_output(ckt, changed, output_index, invert, delay)

CKTcircuit      *ckt;           /* INOUT - The circuit structure */
Mif_Boolean_t    changed;       /* IN   - Has output changed? */
int             output_index;  /* IN   - The output of interest */
Mif_Boolean_t    invert;        /* IN   - Does output need to be inverted? */
double          delay;         /* IN   - The output delay in transient analysis */
{
    Evt_Ckt_Data_t     *evt;           /* Main event-driven data structure in 'ckt' */
    Evt_Output_Queue_t *output_queue; /* Output queue structure in 'evt' */
    Evt_Node_t         *rhs;           /* "Solution vector" where outputs are computed */
    Evt_Node_t         *rhsold;        /* "Solution vector" holding inputs to code models */

Access 'output_queue' within the event structure 'evt' in 'ckt'.
Access 'rhs' and 'rhsold' within the event structure 'evt' in 'ckt'.

If 'g_mif_info' indicates analysis mode is transient, just put the output
event on the queue to be processed at a later time as follows

```

If model signaled that output was not posted ('changed' = false),

Leave the event structure on the free list and return.

Remove the (now used) event structure from the head of the free list.

Invert the output value in the event if 'invert' is true.  
[UDN 'invert' function through g\_eve\_udn\_info].

Add it to the queue. [EVTqueue\_output()]

Return.

If not transient analysis, process immediately as follows,

If model signaled that output was not posted ('changed' = false),  
just return.

If number of outputs connected to the node is greater than one,

Get the subindex for this output on the node from the info structure  
in 'evt'.

Invert the output value in 'rhs' if 'invert' is true.  
[UDN 'invert' function through g\_eve\_udn\_info].

Compare output value in 'rhs' to the value in 'rhsold' at this node index and  
output subindex. [UDN 'compare' function from g\_evt\_udn\_info]

If not equal,

Copy value to 'rhsold' for this node and output subindex.  
[UDN 'copy' function from g\_evt\_udn\_info]

Else,

Invert the output value in 'rhs' if 'invert' is true.  
[UDN 'invert' function through g\_eve\_udn\_info].

Compare output value in 'rhs' to the value in 'rhsold' at this node index.  
[UDN 'compare' function from g\_evt\_udn\_info]

If not equal,

Copy value to 'rhsold' for this node.  
[UDN 'copy' function from g\_evt\_udn\_info]

If copy operation was performed,

Mark this output as changed in 'output\_queue'.

}

#### 4.3.8 Function EVTnext\_time

##### Summary

This function determines and returns the time of the next scheduled event on the inst and output queues.

##### Called By

DCtran() CKT/DCtran.c

##### Returned Value

The next event time, or 1e30 if no events pending.

##### PDL Description

```
double EVTnext_time(ckt)

CKTcircuit *ckt;           /* IN - The circuit structure */
{
    Evt_Ckt_Data_t     *evt;          /* Main event-driven data structure in 'ckt' */
    double              next_time;    /* The return value */
    Evt_Inst_Queue_t   *inst_queue;   /* The instance queue data in 'evt' */
    Evt_Output_Queue_t *output_queue; /* The output queue data in 'evt' */

    Initialize 'next_time' to machine infinity.

    Get 'inst_queue' and 'output_queue' from 'evt' member of 'ckt'.

    If 'inst_queue' has any pending events, set 'next_time' to minimum of
    its current value and the 'next_time' listed in 'inst_queue'.

    If 'output_queue' has any pending events, set 'next_time' to minimum of
    its current value and the 'next_time' listed in 'output_queue'.

    Return 'next_time'.
}
```

#### 4.3.9 Function EVTnode\_copy

##### Summary

This function copies the state of a node structure.

##### Called By

EVTbackup_node_data()	EVT/EVTbackup.c
EVTiter()	EVT/EVTiter.c
EVTtop_save()	EVT/EVTtop.c

##### Returned Value

None.

##### Global Variables

```
Evt_Udn_Info_t *g_evt_udn_info[]; /* IN - Function pointer data for user-defined nodes */
```

##### PDL Description

```
void EVTnode_copy(ckt, node_index, from, to)

CKTcircuit *ckt;           /* IN - The circuit structure */
int      node_index;       /* IN - The node to copy */
Evt_Node_t *from;          /* IN - Location to copy from */
Evt_Node_t **to;           /* OUT - Location to copy to */

{
    Evt_Ckt_Data_t   *evt;        /* Main event-driven data structure in 'ckt' */
    Evt_Node_Data_t  *node_data;  /* Node data structure in 'evt' */

    Get 'evt' structure from 'ckt'.

    Get 'node_data' from 'evt'.

    If destination 'to' is currently NULL,
        If free list in 'node_data' is not empty,
            Pull node structure from free list and set 'to' to point to it.
```

```
Else,  
  
    Allocate a new node structure and point 'to' to it.  
  
    Create storage for node value and outputs in node structure.  
    [UDN 'create' function from g_evt_udn_info]  
  
    If node table info in 'evt' indicates that this node requires an  
    inverted copy of the value,  
  
        Create storage for inverted node value in node structure.  
        [UDN 'create' function from g_evt_udn_info]  
  
        Copy the node data in 'from' to 'to'.  
        [UDN 'copy' function from g_evt_udn_info]  
}
```

#### 4.3.10 Function EVTop

##### Summary

This function is used to perform an operating point analysis in place of CKTop when there are event-driven instances in the circuit. It alternates between doing event-driven iterations with EVTiter and doing analog iterations with NLiter and CKTop until no more event-driven outputs change.

##### Called By

ACan()	CKT/ACan.c
DCop()	CKT/DCop.c
DCtrCurv()	CKT/DCtrCurv.c
DCtran()	CKT/DCtran.c

##### Returned Value

Zero if successful. Non-zero otherwise.

##### PDL Description

```
int EVTop(ckt, firstmode, continuemode, max_iter, first_call)

CKTcircuit    *ckt;          /* INOUT - The circuit structure */
long           firstmode;    /* IN   - The SPICE 3C1 CKTop() firstmode parameter */
long           continuemode; /* IN   - The SPICE 3C1 CKTop() continuemode paramter */
int            max_iter;    /* IN   - The SPICE 3C1 CKTop() max iteration parameter */
Mif_Boolean_t  first_call;  /* IN   - Is this the first time through? */

{
    Evt_Ckt_Data_t    *evt;        /* Main event-driven data structure in 'ckt' */
    Evt_Inst_Queue_t  *inst_queue; /* Instance queue structure in 'evt' */

    Get the 'inst_queue' member of the 'evt' structure in 'ckt'.

    Initialize 'to_call' entries in 'inst_queue' to force calling all event/hybrid
    instance the first time through the queue in the simulation.

    Alternate between event-driven and analog solutions until
    there are no changed event-driven outputs as follows,

    Call EVTiter() to establish initial outputs from event/hybrid instances
    with states (e.g. flip-flops). [EVTiter()]

    Now do analog solution for current state of hybrid outputs as follows,
```

If first analog solution, call CKTop() to start the convergence from initial conditions (zero). [CKTop()]

Otherwise attempt to converge with NIiter() from current solution, and call CKTop() only if solution not reached. [NIiter(), CKTop()]

Call all hybrids to allow new event outputs to be posted. [EVTcall\_hybrids()]

Increment count of successful alternations in statistics member of 'evt'.

If options member of 'evt' indicates that a .option card in the circuit description specified not to alternate algorithms to a final solution, exit immediately with this first pass solution.

If no hybrid instances produced different event outputs, alternation is completed, so exit with success.

If number of analog/event-driven alternations exceeds bounds given in limits member of 'evt', report convergence problem areas and exit with error. [ENHreport\_conv\_prob()]

}

#### 4.3.11 Function EVTop\_save

##### Summary

This function saves results from the operating point analysis into the node data area.

##### Called By

ACan()	CKT/ACan.c
DCop()	CKT/DCop.c
DCtrCurv()	CKT/DCtrCurv.c
DCtran()	CKT/DCtran.c

##### Returned Value

None.

##### PDL Description

```
void EVTop_save(ckt, op, step)

CKTcircuit    *ckt; /* INOUT - The circuit structure */
Mif_Boolean_t op;   /* IN - True if from a DCOP analysis, false if TRANOP, etc. */
double        step; /* IN - Step in DC transfer function analysis */

{
    Evt_Ckt_Data_t    *evt;      /* Main event-driven data structure in 'ckt' */
    Evt_Node_Data_t   *node_data; /* Node data structure in 'evt' */
    Evt_Node_t         *rhsold;   /* 'Solution vector' of event-driven nodes */

    Get 'evt' structure from 'ckt'.

    Get 'node_data' and 'rhsold' elements of 'evt'.

    Loop through all event driven nodes in circuit,
        If 'node_data' linked list of results is null,
            Create a node data item at the head of the list and copy the node
            value in 'rhsold' to it, filling in 'step' and 'op' from argument
            list. [EVTnode_copy()]

        Else,
            Go to the end of the 'node_data' linked list and compare the value
            in 'rhsold' with the value there. [EVTnode_compare()]
}
```

If not equal,

Create a node data item at the end of the list and copy the node value in 'rhsold' to it, filling in 'step' and 'op' from argument list. [EVTnode\_copy()]

}

#### 4.3.11.1 Static Function EVTnode\_compare

## Summary

This function compares the resolved values of the old and new states on a node. The actual comparison is done by calling the appropriate user-defined node compare function.

### Called By

EVTTop\_save() EVT/EVTTop.c

## Returned Value

None.

## Global Variables

```
Evt_Udn_Info_t *g_evt_udn_info[]; /* IN - Function pointer data for user-defined nodes */
```

## PDL Description

```

static void EVTnode_compare(ckt, node_index, node1, node2, equal)
{
    CKTcircuit      *ckt;          /* IN - The circuit structure */
    int              node_index;   /* IN - The index for the node in question */
    Evt_Node_t       *node1;       /* IN - The first value */
    Evt_Node_t       *node2;       /* IN - The second value */
    Mif_Boolean_t   *equal;       /* OUT - The computed result */
{
    Evt_Ckt_Data_t  *evt;         /* Main event-driven data structure in 'ckt' */

    Get 'evt' structure from 'ckt'.

    Lookup the UDN node type index in the node table information structure in
    'evt'.

    Compare 'node1' and 'node2' values and set 'equal' accordingly.
    [UDN 'compare' function from g_evt_udn_info]
}

```

#### 4.3.12 Function EVTprint

##### Summary

This function is used to provide a simple tabular output of event-driven node data. This printout is invoked through a new Nutmeg command called "eprint" which takes event-driven node names as argument.

##### Called By

doCommand() CP/front.c (through function pointer in global cp\_coms[])

##### Returned Value

None.

##### Global Variables

```
Mif_Info_t      g_mif_info;      /* IN - Code model interface runtime data. */
Evt_Udn_Info_t *g_evt_udn_info[]; /* IN - Function pointer data for user-defined nodes */
```

##### PDL Description

```
void EVTprint(wl)

wordlist *wl;      /* IN - The command line entered by user */
{
    CKTcircuit      *ckt;      /* The circuit structure */
    Evt_Ckt_Data_t   *evt;      /* Main event-driven data structure in 'ckt' */
    Evt_Node_t       *node_data; /* Node data element of 'evt' */
    Evt_Msg_t        *msg_data; /* Message data element of 'evt' */

    Get the circuit structure 'ckt' from 'g_mif_info'.

    Get the event-driven data structure 'evt' from 'ckt'.

    Count the number of node name arguments to the command by counting
    the elements in the 'wl' linked list.

    Get data for each node as follows,
        For each node name in 'wl',
```

```
Get the index that corresponds to this node. [get_index()]

Get the user-defined node index from the node table information
in 'evt'.

For each node name in 'wl',
    Print the column node name identifiers.

For each node name in 'wl',
    Get the print string for the node value retrieved from 'node_data'.
    [UDN 'print_val' function in g_evt_udn_info]

    Print the data for the set of nodes at the current step. [print_data()]

    While there is more data for any node,
        For each node name in 'wl',
            Get the print string for the node value in 'node_data'.
            [UDN 'print_val' function in g_evt_udn_info]

            Print the data for the set of nodes at the current step. [print_data()]

Print messages for all ports as follows,
    Loop through all event-driven data ports in circuit,
        Get messages for this port from 'msg_data'.

        If no messages on this port, skip.

        Print the port description, including the node name, instance name,
        connection name, and port name.

        Print the messages on this port contained in 'msg_data'.

    Print statistics data from 'evt'.
}
```

#### 4.3.12.1 Static Function get\_index

##### Summary

This function determines the index of a specified event-driven node.

##### Called By

EVTprint()                   EVT/EVTprint.c

##### Returned Value

The index of the node, or -1 if not found.

##### Global Variables

```
Mif_Info_t       g_mif_info;       /* IN - Code model interface runtime data. */
```

##### PDL Description

```
static int get_index(node_name)

char *node_name;       /* IN - The name of the node to search for */
{
    CKTcircuit       *ckt;       /* The circuit structure */
    Evt_Ckt_Data_t   *evt;       /* Main event-driven data structure in 'ckt' */

    Get the circuit structure 'ckt' from 'g_mif_info'.

    Get the event-driven data structure 'evt' from 'ckt'.

    Scan through all the event-driven nodes in the 'evt' info data,
        If found, return index.

    Return -1.
}
```

#### 4.3.12.2 Static Function print\_data

##### Summary

This function prints the values of one or more nodes to standard output.

##### Called By

EVTprint()                   EVT/EVTprint.c

##### Returned Value

None.

##### PDL Description

```
static void print_data(dcop, step, node_value, nargs)

Mif_Boolean_t dcop;               /* IN - Is this the operating point data */
double         step;              /* IN - The analysis step if dcop */
char          **node_value;     /* IN - The array of values to be printed */
int           nargs;             /* IN - The size of the value array */
{
     Format and print a line of data for number of args specified.
}
```

#### 4.3.13 Function EVTdump

##### Summary

This function is called to send event-driven node data to the IPC channel. A “mode” argument determines how the data is located in the event data structure and what data is sent.

If the mode is DCOP, then this is necessarily the first call to the function. In this case, the set of event-driven nodes is scanned to determine which should be sent. Only nodes that are not inside subcircuits are sent. Next, the function sends a “dictionary” of node names/types vs. node indexes. Finally, the function sends the DC operating point solutions for the event-driven nodes in the dictionary.

If the mode is DCTRCURVE, it is assumed that the function has already been called with mode = DCOP. The function scans the solution vector and sends data for any nodes that have changed.

If the mode is TRAN, it is assumed that the function has already been called once with mode = DCOP. The function scans the event data for nodes that have changed since the last accepted analog timepoint and sends the new data.

Note: This function must be called BEFORE calling EVTop\_save or EVTaccept() so that the state of the node data structure will allow it to determine what has changed.

##### Called By

ACan()	CKT/ACan.c
DCop()	CKT/DCop.c
DCtrCurv()	CKT/DCtrCurv.c
DCtran()	CKT/DCtran.c

##### Returned Value

None.

##### Global Variables

```
Ipc_Tiein_t      g_ipc;           /* IN - Interprocess communication data */
Evt_Udn_Info_t   *g_evt_udn_info; /* IN - Function pointer data for user-defined nodes */
```

## Local Data Types

```
typedef struct evtdump_s {
    Mif_Boolean_t send;          /* True if this node should be sent */
    int ipc_index;              /* Index for this node in dict sent to CAE system */
    char *node_name_str;        /* Node name */
    char *udn_type_str;         /* UDN type */
} evtdump_dict_t;
```

## PDL Description

```
void EVTdump(ckt, mode, step)

CKTcircuit *ckt; /* IN - The circuit structure */
Ipc_Anal_t mode; /* IN - The analysis mode for this call */
double step; /* IN - Sweep step for a DCTRCURVE analysis. 0.0 for DCOP, TRAN */
{
    static evtdump_dict_t *node_dict; /* Dictionary of node data used for IPC */

    Evt_Ckt_Data_t *evt;           /* Main event-driven data structure in 'ckt' */
    Evt_Node_Data_t *node_data;   /* Node data structure in 'evt' */
    Evt_Node_t *rhsold;          /* 'Solution vector' of event-driven nodes */

    Get 'evt' structure from 'ckt'.

    Get 'node_data' and 'rhsold' elements of 'evt'.

    Return immediately if 'g_ipc' indicates IPC is not enabled.

    Get number of event-driven nodes from 'evt' count structure.

    Exit immediately if no event-driven nodes in circuit.

    If this is the first call, create the dictionary info in 'node_dict' as follows,
        Loop through all nodes to determine which nodes should be sent,
            Get the name of the node.
                If name is in a subcircuit, mark that node should not be sent
                and continue to next node.
                Otherwise, fill in elements of dictionary structure for this node.
            Increment count of nodes to be sent.
    Exit if there are no nodes to be sent.
```

If this is the first call, send the dictionary information over the IPC channel.  
[ipc\_send\_evtdict\_prefix(), ipc\_send\_line(buff), ipc\_send\_evtdict\_suffix()]

If this is the first call, send the operating point solution and return.  
[ipc\_send\_evtdata\_prefix(), EVTsend\_line(), ipc\_send\_evtdata\_suffix()]

Otherwise, this must be DCTRCURVE or TRAN mode and we need to send only data  
for nodes that has changed since the last call. The determination of what to  
send is modeled after code in EVTop\_save() for DCTRCURVE and EVTaccept() for TRAN.

If mode is DCTRCURVE,

Send data prefix. [ipc\_send\_evtdata\_prefix()]

Loop through event nodes.

If dictionary indicates this node should be sent,

Locate end of linked-list in 'node\_data'.

Compare entry at end of list to value in 'rhsold'.  
[UDN 'compare' function in g\_evt\_udn\_info]

If value in 'rhsold' is different, send it. [IPCsend\_line()]

Send data suffix and return. [ipc\_send\_evtdata\_suffix()]

If mode is TRAN,

Send data prefix. [ipc\_send\_evtdata\_prefix()]

Loop through list of nodes in 'node\_data' modified,

Get the index of the node modified.

If dictionary indicates this node should be sent,

Scan through new events in 'node\_data' starting at 'last\_step'  
and send the data for each event. [IPCsend\_line()]

Send data suffix and return. [ipc\_send\_evtdata\_suffix()]

}

#### 4.3.13.1 Static Function EVTsend\_line

## Summary

This function formats the event node data and sends it to the IPC channel.

### Called By

## EVTdump() EVT/EVTdump.c

## Returned Value

None.

## Global Variables

```
Evt Udn Info t *g_evt_udn_info[]; /* IN - Function pointer data for user-defined nodes */
```

## PDL Description

```

static void EVTsend_line(ipc_index, step, node_value, udn_index)

int          ipc_index;           /* IN - The index used in the dictionary */
double       step;                /* IN - The analysis step */
void         *node_value;        /* IN - The node value */
int          udn_index;          /* IN - The user-defined node index */

{
    Get the data to send by calling the appropriate UDN functions.
    [UDN 'plot_val' function in g_evt_udn_info]
    [UDN 'print_val' function in g_evt_udn_info]
    [UDN 'ipc_val' function in g_evt_udn_info]

    Send it to the IPC channel.  [ipc_send_event()]

}

```

#### 4.3.14 Function EVTqueue\_output

##### Summary

This function places the specified output event onto the output queue. It is called during a transient analysis.

The linked list in the queue for the specified output is searched beginning at the current head of the pending events to find the location at which to insert the new event. The events are ordered in the list by event\_time. If the event is placed before the end of the list, subsequent events are removed from the list by marking them as "removed" and recording the time of removal. This allows efficient backup of the state of the queue if a subsequent analog timestep fails.

##### Called By

EVTprocess\_output()                   EVT/EVTload.c

##### Returned Value

None.

##### PDL Description

```
void EVTqueue_output(ckt, output_index, udn_index, new_event,
                     posted_time, event_time)

CKTcircuit      *ckt;           /* INOUT - The circuit structure */
int             output_index;  /* IN   - The output in question */
int             udn_index;     /* IN   - The associated user-defined node type */
Evt_Output_Event_t *new_event; /* IN   - The event to queue */
double          posted_time;  /* IN   - The current time */
double          event_time;   /* IN   - The time the event should happen */
{
    Evt_Output_Queue_t *output_queue; /* Output queue data in 'evt' member of 'ckt' */

    Put the times into the event structure 'new_event'.

    Update 'next_time' member in 'output_queue'.

    Find location in 'output_queue' at which to insert event starting at
    'current'.

    If new event needs to be spliced into middle of existing list,
```

Splice it in.

Mark later events as removed.

Else,

Put new event at the end of list.

Add to the list of outputs modified on 'output\_queue' since last accepted timestep.

Add to the list of outputs with events pending on 'output\_queue'.

}

#### 4.3.15 Function EVTqueue\_inst

## Summary

This function places the specified instance event onto the inst queue.

The linked list in the queue for the specified inst is searched beginning at the current head of the pending events to find the location at which to insert the new event. The events are ordered in the list by event\_time.

### Called By

~~cm\_event\_queue()~~ CM/CMevt.c

#### Returned Value

None.

## PDL Description

```
void EVTqueue_inst(ckt, inst_index, posted_time, event_time)

CKTcircuit *ckt          /* INOUT - The circuit structure */
int      inst_index;    /* IN     - The instance in question */
double   posted_time;  /* IN     - The current time */
double   event_time;   /* IN     - The time the event should happen */
{
    Evt_Inst_Queue_t *inst_queue; /* Instance queue data in 'evt' member of 'ckt' */

    Update 'next_time' member in 'inst_queue'.

    Create a new event, or get one from the free list in 'inst_queue' if
    available.

    Find location in 'inst_queue' at which to insert event starting at
    'current'.

    If new event needs to be spliced into middle of existing list,
        Splice it in.

        Mark later events as removed.

    Else.
```

XSPICE Simulator  
Software Design Document

Detailed Design  
Event-Driven Simulation

Put new event at the end of list.

Add to the list of insts modified on 'inst\_queue' since last accepted timestep.

Add to the list of insts with events pending on 'inst\_queue'.  
}

#### 4.3.16 Function EVTsetup

##### Summary

This function clears/allocates the event-driven queues and data structures immediately prior to a new analysis. In addition, it places entries in the job list so that results data from multiple analysis can be retrieved similar to SPICE 3C1 saving multiple "plots".

##### Called By

CKTdoJob()                    CKT/CKTdoJob.c

##### Returned Value

Zero if success, non-zero otherwise.

##### Global Variables

```
Mif_Info_t        g_mif_info;        /* IN - Code model interface runtime data. */
```

##### PDL Description

```
int EVTsetup(ckt)
{
    CKTcircuit *ckt;        /* INOUT - The circuit structure */
    Evt_Ckt_Data_t *evt;     /* Main event-driven data structure in 'ckt' */

    Get 'evt' structure from 'ckt'.

    Exit immediately if counts in 'evt' indicate there are no event-driven instances
    in the circuit.

    Clear the instance, node, and output queues, and initialize the 'to_call'
    elements in the instance queue to call all event-driven instances.
    [EVTsetup_queues()]

    Allocate and initialize the node, state, message, and statistics data.
    [EVTsetup_data()]
```

Set the job pointers to the allocated results, states, messages,  
and statistics so that data will be accessible after run.  
[EVTsetup\_jobs()]

Setup the pointers in the MIFinstance structure for inputs, outputs,  
and total loads. [EVTsetup\_load\_ptrs()]

Initialize event-driven analysis step in 'evt' to zero.

}

#### 4.3.16.1 Static Function EVTsetup\_queues

##### Summary

This function clears the event-driven queues in preparation for a new simulation.

##### Called By

EVTsetup()                   EVT/EVTsetup.c

##### Returned Value

Zero if success, non-zero otherwise.

##### PDL Description

```
static int EVTsetup_queues(ckt)
{
    CKTcircuit *ckt;         /* INOUT - The circuit structure */
    Evt_Ckt_Data_t    *evt;         /* Main event-driven data structure in 'ckt' */

    Get 'evt' structure from 'ckt'.

    Clear the instance queue in 'evt'.

    Clear the node queue in 'evt'.

    Clear the output queue in 'evt'.
}
```

#### 4.3.16.2 Static Function EVTsetup\_data

##### Summary

This function sets up the event-driven node, state, and message data runtime structures in preparation for a new simulation.

##### Called By

EVTsetup()                   EVT/EVTsetup.c

##### Returned Value

Zero if success, non-zero otherwise.

##### Global Variables

```
Evt_Udn_Info_t *g_evt_udn_info[]; /* IN - Function pointer data for user-defined nodes */
```

##### PDL Description

```
static int EVTsetup_data(ckt)
{
    CKTcircuit *ckt;         /* INOUT - The circuit structure */
    Evt_Ckt_Data_t *evt;         /* Main event-driven data structure in 'ckt' */

    Get 'evt' structure from 'ckt'.

    Allocate main substructures of data. Note that we don't free any old structures
    since they are pointed to by jobs and need to be maintained so that results from
    multiple jobs are kept around like SPICE does.

    Allocate and initialize node data in 'evt' as follows,
        For number of event-driven nodes in circuit,
            Create and initialize 'rhs' and 'rhsold' members.
            [UDN 'create' function in g_evt_udn_info]
            [UDN 'initialize' function in g_evt_udn_info]

            Initialize the 'total_load' value to zero.
```

Allocate and initialize state data.

Allocate and initialize msg data.

}

#### 4.3.16.3 Static Function EVTsetup\_jobs

##### Summary

This function prepares the jobs data for a new simulation.

##### Called By

EVTsetup()                   EVT/EVTsetup.c

##### Returned Value

Zero if success, non-zero otherwise.

##### PDL Description

```
static int EVTsetup_jobs(ckt)

CKTcircuit *ckt;         /* INOUT - The circuit structure */
{
    Evt_Ckt_Data_t     *evt;         /* Main event-driven data structure in 'ckt' */
    Get 'evt' structure from 'ckt'.

    Increment the number of jobs in 'evt'.

    Allocate/reallocate necessary data in 'jobs' member of 'evt'.
}
```

#### 4.3.16.4 Static Function EVTsetup\_load\_ptrs

##### Summary

This function sets up the required data in the MIFinstance structure of event-driven and hybrid instances.

##### Called By

EVTsetup()                   EVT/EVTsetup.c

##### Returned Value

Zero if success, non-zero otherwise.

##### PDL Description

```
static int EVTsetup_load_ptrs(ckt)
{
    CKTcircuit *ckt;         /* INOUT - The circuit structure */
    Evt_Ckt_Data_t    *evt;         /* Main event-driven data structure in 'ckt' */

    Get 'evt' structure from 'ckt'.

    Loop through all event-driven and hybrid instances,
        Loop through all connections,
            Skip if connection is null.
            Loop through all ports,
                Skip if port is not digital or user-defined type,
                Set input pointer to point to 'rhsold' node data in 'evt' or to
                    the 'rhsold' inverted value as appropriate.

                Set output pointer to point to 'rhs' node value or 'rhs' output
                    value at index i, where i is given by the output_subindex in the
                    'evt' node info structure.
                Note that this is only for the DCOP analysis. During a transient
                    analysis, new structures will be created and the pointers will
                    be set by EVTload.
}
```

#### 4.3.17 Function EVTtermInsert

## Summary

This function is called by MIF\_INP2A during the parsing of the input deck. EVTtermInsert is similar to SPICE 3C1's INPtermInsert except that it is used when the node type is event-driven. Calls to this function build the info lists for instances, nodes, outputs, and ports. The completion of the info struct is carried out by EVTinit following the parsing of all instances in the deck.

### Called By

MIFget\_port() MIF/MIF\_INP2A.c

## Returned Value

None.

## PDL Description

```

void EVTtermInsert(ckt, fast, node_name, type_name, conn_num, port_num, err_msg)

CKTcircuit    *ckt;           /* INOUT - The circuit structure */
MIFinstance   *fast;          /* INOUT - The instance being parsed */
char          *node_name;     /* IN    - The node name */
char          *type_name;     /* IN    - The type of node */
int           conn_num;       /* IN    - The port connection number */
int           port_num;       /* IN    - The sub-port number - 0 for scalar ports */
char          **err_msg;      /* OUT   - Returned error message if any */

{
    Get the instance index and create new entry in inst
    info list if this is a new instance.  [EVTinst_insert()]

    Get the node index and create new entry in node info
    list if this is a new node.  [EVTnode_insert()]

    Create new entry in port info list and return port index.
    [EVTport_insert()]

    Create new entry in output info list if appropriate.
    [EVToutput_insert()]
}

```

#### 4.3.17.1 Static Function EVTinst\_insert

##### Summary

This function locates or creates a new entry for the specified instance in the event-driven “info” structures during parsing.

##### Called By

EVTtermInsert()                   EVT/EVTtermInsert.c

##### Returned Value

None.

##### PDL Description

```
static void EVTinst_insert(ckt, fast, inst_index, err_msg)

CKTcircuit   *ckt;               /* INOUT - The circuit structure */
MIFinstance   *fast;              /* INOUT - The instance being parsed */
int           *inst_index;        /* OUT - The index found or added */
char          **err_msg;          /* OUT - Error message if any */

{
    Evt_Ckt_Data_t   *evt;       /* Main event-driven data structure in 'ckt' */
    Evt_Inst_Info_t  *inst;       /* Instance information in 'evt' member of 'ckt' */

    Get 'evt' structure and 'inst' substructure from 'ckt'.

    Scan list of instances in 'inst' linked list to see if instance is already there
    and get its index.

    If not found, create a new entry at end of 'inst' list and increment the
    instance count in 'evt' counts structure.

    Record the inst index in instance structure 'fast' and return it.

}
```

#### 4.3.17.2 Static Function EVTnode\_insert

##### Summary

This function locates or creates a new entry for the specified node in the event-driven "info" structures during parsing.

##### Called By

EVTtermInsert()      EVT/EVTtermInsert.c

##### Returned Value

None.

##### Global Variables

```
int                g_evt_num_udn_types; /* IN - Number of user-defined node types */
Evt_Udn_Info_t *g_evt_udn_info[];    /* IN - Function pointer data for user-defined nodes */
```

##### PDL Description

```
static void EVTnode_insert(ckt, fast, inst_index, node_name, type_name,
                          conn_num, port_num, node_index, output_subindex, err_msg)

CKTcircuit *ckt;                    /* INOUT - The circuit structure */
MIFinstance *fast;                /* INOUT - The instance being parsed */
int            inst_index;         /* IN - The index of inst in evt structures */
char           node_name;          /* IN - The node name */
char           type_name;          /* IN - The node type specified */
int           conn_num;            /* IN - The port connection number */
int           port_num;            /* IN - The sub-port number - 0 if scalar port */
int           *node_index;        /* OUT - The node index found or added */
int           *output_subindex;    /* OUT - The output number on this node */
char           **err_msg;         /* OUT - Error message text if any */
{
    Evt_Ckt_Data_t *evt;        /* Main event-driven data structure in 'ckt' */

    Get 'evt' structure from 'ckt'.

    Scan the list of user-defined node types in 'g_evt_udn_info' and get the index
```

of the type whose name matches 'type\_name'.

Report error if not found.

If connection data in 'fast' indicates that port is inverted,

Check to be sure invert function exists in 'g\_evt\_udm\_info' for this type  
and report error if not.

Scan list of nodes in event structure 'evt' to see if already there by  
looking for 'node\_name'.

If found, verify that user-defined node type is same as type for existing  
node entry.

If not found, create a new entry in list of nodes on 'evt' and increment the  
node count.

Update flag on node that indicates if inversion is used by any  
instance inputs.

Increment counts of ports, outputs connected to node.

If this is an input, add instance to list if not already there.

Record the node index in 'fast'.

Return the node index.

}

#### 4.3.17.3 Static Function EVTport\_insert

##### Summary

This function locates or creates a new entry for the specified port in the event-driven “info” structures during parsing.

##### Called By

EVTtermInsert()                   EVT/EVTtermInsert.c

##### Returned Value

None.

##### PDL Description

```
static void EVTport_insert(ckt, fast, inst_index, node_index, node_name,
                           conn_num, port_num, port_index, err_msg)

CKTcircuit *ckt;                 /* INOUT - The circuit structure */
MIFinstance *fast;               /* INOUT - The instance being parsed */
int       inst_index;            /* IN    - The index of inst in evt structures */
int       node_index;            /* IN    - The index of the node in evt structures */
char      *node_name;            /* IN    - The node name */
int       conn_num;              /* IN    - The port connection number */
int       port_num;              /* IN    - The sub-port number - 0 if scalar port */
int       *port_index;           /* OUT - The port index found or added */
char      **err_msg;             /* OUT - Error message text if any */

{
    Evt_Ckt_Data_t    *evt;    /* Main event-driven data structure in 'ckt' */
    Evt_Port_Info_t   *port;    /* Port information in 'evt' */

    Get 'evt' structure and 'port' substructure from 'ckt'.

    Scan list of instances in 'port' linked list to find the end of the list.

    Increment the count of ports in the counts member of 'evt'.

    Allocate a new member at the end of the 'port' list and fill in the elements.

    Record the port index in instance structure 'fast'.

    Return the port index.
}
```

#### 4.3.17.4 Static Function EVToutput\_insert

##### Summary

This function locates or creates a new entry for the specified output in the event-driven “info” structures during parsing.

##### Called By

EVTtermInsert()      EVT/EVTtermInsert.c

##### Returned Value

None.

##### PDL Description

```
static void EVToutput_insert(ckt, fast, inst_index, node_index, port_index,
                            output_subindex, conn_num, port_num, err_msg)

CKTcircuit *ckt;                /* INOUT - The circuit structure */
MIFinstance *fast;             /* INOUT - The instance being parsed */
int        inst_index;          /* IN - The index of inst in evt structures */
int        node_index;          /* IN - The index of the node in evt structures */
int        port_index;          /* IN - The index of the port in the evt structures */
int        output_subindex;     /* IN - The output on this node */
int        conn_num;            /* IN - The port connection number */
int        port_num;            /* IN - The sub-port number - 0 if scalar port */
char      **err_msg;           /* OUT - Error message text if any */

{
    Evt_Ckt_Data_t    *evt;     /* Main event-driven data structure in 'ckt' */
    Evt_Output_Info_t *output; /* Output information structure in 'evt' */

    Get 'evt' structure and 'output' substructure from 'ckt'.

    Scan list of entries in 'output' linked list to find the end of the list.

    Increment the count of outputs in the counts member of 'evt'.

    Allocate a new member at the end of the 'output' list and fill in the elements.

    Record the output index in instance structure 'fast'.

    Return the output index.
}
```

#### 4.4 Enhancements

The Enhancements (SIM-ENH) CSC implements a variety of general enhancements made to the SPICE 3C1 simulator to make it more effective for board-level and system-level simulation.

```
ENHreport_conv_prob
ENHtranslate_poly
needs_translating
count_tokens
translate
get_poly_dimension
```

#### 4.4.1 Function ENHreport\_conv\_prob

##### Summary

This function reports convergence problem messages from nodes, branch currents, or instances. This function is setup to allow providing the SI with information identifying the type of convergence problem. For now, it simply writes to stdout.

##### Called By

NIconvTest()	NI/NIconvTest.c
MIFconvTest()	MIF/MIFconvTest.c
MIFload()	MIF/MIFload.c
EVTiter()	EVT/EVTiter.c
EVTop()	EVT/EVTop.c

##### Returned Value

None.

##### PDL Description

```
void ENHreport_conv_prob(type, name, msg)

Enh_Conv_Source_t type; /* IN - Node, branch, or instance */
char *name; /* IN - The name of the node/branch/instance */
char *msg; /* IN - An optional message */
{
    Convert the enum 'type' to a string for printing.

    Check for 'msg' == NULL and turn into null string if found.

    Print the convergence problem information to stdout, including
    the type string, 'name', and optional message string 'msg'.
}
```

#### 4.4.2 Function ENHtranslate\_poly

##### Summary

This function translates all SPICE2G6 style polynomial controlled sources in the deck to new polynomial controlled source code model syntax.

##### Called By

inp\_spsource()                  FTE/inp.c

##### Returned Value

The (possibly modified) linked list of lines in the input deck.

##### PDL Description

```
struct line *ENHtranslate_poly(deck)

struct line *deck; /* INOUT - Linked list of lines in input deck */
{
    Iterate through each card in linked list 'deck',
    If doesn't need to be translated, continue to next card.
    [needs_translating()]

    Create two new line structures and splice into 'deck'.

    Create the translated cards. [translate()]

    Comment out the original line by inserting a '*' at the beginning.

    Return head of (possibly modified) 'deck' linked list.
}
```

#### 4.4.2.1 Static Function needs\_translating

##### Summary

This function determines if a controlled source card represents a non-linear controlled source, and therefore needs to be translated into the internal “poly” code model syntax. It checks the card to see if it has too many tokens to be a simple linear dependent source.

##### Called By

ENHtranslate\_poly()                  ENH/ENHtranslate\_poly.c

##### Returned Value

One if card needs translating, or zero if not.

##### PDL Description

```
static int needs_translating(card)

char *card;                        /* IN - The input deck card text */
{
    If source type (first character of 'card') is 'e' or 'g',
        If number of tokens is <= 6, [count_tokens()]
            Return 0.

        Else
            Return 1.

    Else if source type is 'f' or 'h',
        If number of tokens is <= 5, [count_tokens()]
            Return 0.

        Else
            Return 1.

    Else
        Return 0.
}
```

#### 4.4.2.2 Static Function count\_tokens

##### Summary

This function counts the number of tokens on a card.

##### Called By

needs_translating()	ENH/ENHtranslate_poly.c
translate()	ENH/ENHtranslate_poly.c

##### Returned Value

Number of tokens on card.

##### PDL Description

```
static int count_tokens(card)

char *card;           /* IN - The input deck card text */
{
    Get and count tokens until end of line reached. [MIFgettok()]
}
```

#### 4.4.2.3 Static Function translate

##### Summary

This function translates the syntax of a SPICE 2G6 polynomial controlled source into the syntax used by the internal XSPICE “poly” code model.

##### Called By

ENHtranslate\_poly() ENH/ENHtranslate\_poly.c

##### Returned Value

A NULL pointer on success, or an error string on failure.

##### PDL Description

```
static char *translate(orig_card, inst_card, mod_card)

char *orig_card; /* IN - The original untranslated card */
char **inst_card; /* OUT - The instance card created by the translation */
char **mod_card; /* OUT - The model card created by the translation */

{
    Count the number of tokens on 'orig_card' for use in parsing.
    [count_tokens()]

    Determine the dimension of the poly source.  [get_poly_dimension()]

    Compute number of input connection tokens based on type (first character
    of 'orig_card') and dimension of poly source.

    Compute number of coefficients based on number of tokens on card and
    number of connection tokens expected.  Return error if less than one.

    Split card into name, output connections, input connections,
    and coefficients.  [MIFgettok()]

    Compute the size needed for the new cards to be created.

    Allocate space for the instance and model cards ('inst_card' and 'mod_card')
    and build the instance card text and model card text.

    Return NULL to indicate no error.
}
```

#### 4.4.2.4 Static Function get\_poly\_dimension

##### Summary

This function determines the dimension (number of inputs) of a polynomial controlled source that needs translating.

##### Called By

translate()      ENH/ENHtranslate\_poly.c

##### Returned Value

The dimension of the poly source, or 0 if failure.

##### PDL Description

```
static int get_poly_dimension(card)

char *card;           /* IN - The card text */
{
    Skip over name and output connections. [MIFgettok()]

    Check the next token to see if it is the keyword "poly".
    If not, return a dimension of 1.

    Must have been "poly", so next line must be a number
    Convert it to an integer. If successful, return the number,
    else, return 0 to indicate an error.
}
```

## 4.5 Internal Code Models

The Internal Code Models (SIM-ICM) CSC implements a special “poly” code model that is included in the simulator core.

The code model functions included are:

```
icm_poly
evterm
nxtwr
```

#### 4.5.1 Function icm\_poly

##### Summary

This code model implements the non-linear polynomial controlled sources available in SPICE 2G6. An automatic translator added into the simulator front end is used to map 2G6 syntax into a call to this model in the required syntax.

This model may also be called directly as follows:

```
a1 [ <input(s)> ] <output> xxx
.model xxx poly ( coef = [ <list of 2G6 compatible coefficients> ] )
```

Refer to the SPICE 2G6 User Guide for an explanation of the coefficients.

This model is patterned after the FORTRAN code used in the SPICE 2G6 simulator. Function cm\_poly() below performs the functions of subroutines NLCSRC and EVPOLY. Function evterm() performs the function of subroutine EVTERM, and function nxtpwr() performs the function of subroutine NXTPWR.

##### Called By

CKTload()	CKT/CKTload.c
CKTacLoad()	CKT/CKTacLoad.c

##### Returned Value

None.

##### PDL Description

```
void icm_poly (ARGS)
ARGS = Mif_Private_t *private; /* INOUT - Code model inputs, outputs, and parameters */
{
    If ARGS indicates analysis type is AC, just return. Poly source is
    non-linear and doesn't make sense for AC analysis.

    Get input values and coefficients from ARGS.

    Compute the output of the source by summing the required products as follows:

        Get the list of powers for the product terms in this term of the sum.
        [nxtpwr()]
```

Form the product of the inputs taken to the required powers.  
[evterm()]

Add the product times the appropriate coefficient into the sum.

Compute and output the partials for each input as follows:

For each input,

Get the list of powers for the product terms in this term of  
the sum. [nxtpwr()]

If power for input for which partial is being evaluated  
is zero, the term is a constant, so the partial is zero,  
so go to next input.

Form the product of the inputs taken to the required powers as  
follows:

If input is not the one for which the partial is being taken  
take the term to the specified exponent. [evterm()]

Else, take the derivative of this term as  $n*x^{n-1}$ .  
[evterm()]

Add the product times the appropriate coefficient into the sum.

}

#### 4.5.1.1 Static Function evterm

##### Summary

This function raises a value x to a power n.

##### Called By

icm\_poly            ICM/POLY/cfunc.mod

##### Returned Value

The result of raising x to the n'th power.

##### PDL Description

```
static double evterm(x, n)

double x; /* IN - The value to be raised to exponent n */
int n; /* IN - The exponent */
{
    Compute and return the value of x**n.
}
```

#### 4.5.1.2 Static Function nxtpwr

##### Summary

This function is a literal translation of subroutine NXTPWR in SPICE 2G6. This was done to guarantee compatibility with the ordering of coefficients used by 2G6. The 2G6 User Guide does not completely define the algorithm used and the GOTO loaded FORTRAN code is difficult to unravel. Therefore, a one-to-one translation was deemed the safest approach.

No attempt is made to document the function statements since no documentation is available in the 2G6 code. However, it can be noted that the code appears to generate the exponents of the product terms in the sum-of-products produced by the following expansion for two and three dimensional polynomials:

2D:

$$(a + b)^n$$

3D:

$$(a + (b + c))^n$$

where n begins at 1 and increments as needed for as many terms as there are coefficients on the polynomial source SPICE deck card, and where terms that are identical under the laws of associativity are dropped. Thus, for example, the exponents for the following sums are produced:

2D:

$$a + b + a^2 + ab + b^2 + c^3 + \dots$$

3D:

$$a + b + c + a^2 + ab + ac + b^2 + bc + c^2 + a^3 + \dots$$

##### Called By

icm\_poly ICM/POLY/cfunc.mod

Returned Value

None.

PDL Description

```
static void nxtpwr(pwrseq, pdim)

int *pwrseq; /* INOUT - Array of exponents */
int pdim;      /* IN      - The dimension */
{
    int i;
    int k;
    int km1;
    int psum;

    if(pdim == 1) goto stmt80;
    k = pdim;
stmt10: if(PWRSEQ(k) != 0) goto stmt20;
    k = k - 1;
    if(k != 0) goto stmt10;
    goto stmt80;
stmt20: if(k == pdim) goto stmt30;
    PWRSEQ(k) = PWRSEQ(k) - 1;
    PWRSEQ(k+1) = PWRSEQ(k+1) + 1;
    goto stmt100;
stmt30: km1 = k - 1;
    for(i = 1; i <= km1; i++)
        if(PWRSEQ(i) != 0) goto stmt50;
stmt40: PWRSEQ(1) = PWRSEQ(pdim) + 1;
    PWRSEQ(pdim) = 0;
    goto stmt100;
stmt50: psum = 1;
    k = pdim;
stmt60: if(PWRSEQ(k-1) >= 1) goto stmt70;
    psum = psum + PWRSEQ(k);
    PWRSEQ(k) = 0;
    k = k - 1;
    goto stmt60;
stmt70: PWRSEQ(k) = PWRSEQ(k) + psum;
    PWRSEQ(k-1) = PWRSEQ(k-1) - 1;
    goto stmt100;
stmt80: PWRSEQ(1) = PWRSEQ(1) + 1;

stmt100: return;
}
```

## 4.6 Internally Defined Nodes

The Internally Defined Nodes (SIM-IDN) CSC implements a special “digital” event-driven node type which is included in the simulator core.

This node type is made accessable to the simulator through a structure of type Evt\_Udn\_Info\_t, which contains pointers to functions that implement actions such as data allocation, copying, comparing, etc. The function pointer data structure for the “digital” type is described below together with the functions it references.

#### 4.6.1 Info Structure idn\_digital\_info

##### Summary

This structure defines pointers used to describe and implement the digital data type used in event-driven simulation.

##### PDL Description

```
Evt_Udn_Info_t idn_digital_info = {  
    "d",  
    "12 state digital data",  
    idn_digital_create,  
    idn_digital_dismantle,  
    idn_digital_initialize,  
    idn_digital_invert,  
    idn_digital_copy,  
    idn_digital_resolve,  
    idn_digital_compare,  
    idn_digital_plot_val,  
    idn_digital_print_val,  
    idn_digital_ipc_val  
};
```

#### 4.6.1.1 Function idn\_digital\_create

##### Summary

This function performs the create operation for the digital 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 idn_digital_create(evt_struct)
{
    void **evt_struct; /* OUT - The allocated structure memory */
    Allocate space for a structure of type "Digital_t".
}
```

#### 4.6.1.2 Function idn\_digital\_dismantle

##### Summary

This function performs the dismantle operation for the digital 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 idn_digital_dismantle(evt_struct)
void *evt_struct; /* INOUT - The structure to dismantle */
{
    Do nothing. There are no internally allocated things to dismantle.
}
```

#### 4.6.1.3 Function idn\_digital\_initialize

##### Summary

This function performs the initialize operation for the digital node type.

##### Called By

EVTsetup\_data()                   EVT/EVTsetup.c

##### Returned Value

None.

##### PDL Description

```
void idn_digital_initialize(evt_struct)
void *evt_struct; /* INOUT - The structure to initialize */
{
    Initialize to state member of 'evt_struct' to ZERO
    and strength member to UNDETERMINED.
}
```

#### 4.6.1.4 Function idn\_digital\_invert

##### Summary

This function performs the invert operation for the digital node type.

##### Called By

EVTiter()	EVT/EVTiter.c
EVTprocess_output()	EVT/EVTload.c
EVTnode_insert()	EVT/EVTtermInsert.c

##### Returned Value

None.

##### PDL Description

```
void idn_digital_invert(evt_struct)

void *evt_struct; /* INOUT - The structure to invert */
{
    Invert the state member of 'evt_struct'.
}
```

## 4.7 Interprocess Communication

The Interprocess Communication (SIM-IPC) CSC provides functions called to receive SPICE decks from the ATESSE Simulator Interface or Batch Control processes, and to return results to those processes. These functions include:

```
ipc_initialize_server
ipc_terminate_server
ipc_get_line
ipc_send_line
ipc_send_data_prefix
ipc_send_data_suffix
ipc_send_dcop_prefix
ipc_send_dcop_suffix
ipc_send_evtdict_prefix
ipc_send_evtdict_suffix
ipc_send_evtdata_prefix
ipc_send_evtdata_suffix
ipc_send_errchk
ipc_send_end
ipc_send_double
ipc_send_complex
ipc_send_event
ipc_flush
ipc_handle_stop
ipc_handle_returni
ipc_handle_mintime
ipc_handle_vtrans
ipc_send_std_files
    ipc_send_stdout
    ipc_send_stderr
ipc_screen_name
ipc_get_devices
ipc_free_devices
ipc_check_pause_stop
```

These functions communicate with a set of transport-level functions that implement the interprocess communication under one of the following protocol types:

BSD UNIX Sockets  
HP/Apollo Mailboxes

For each transport protocol, the following functions are written:

```
ipc_transport_initialize_server
ipc_transport_get_line
ipc_transport_terminate_server
ipc_transport_send_line
```

#### 4.7.1 Function ipc\_initialize\_server

##### Summary

This function creates the interprocess communication channel server mailbox or socket.

##### Called By

main() FTE/bspice.c

##### Returned Value

Completion status.

##### PDL Description

```
Ipc_Status_t ipc_initialize_server (server_name, m, p)

char          *server_name; /* IN - Mailbox path or port number */
Ipc_Mode_t      m;           /* IN - Interactive or batch mode flag */
Ipc_Protocol_t  p;           /* IN - Type of IPC protocol */
{
    Call transport-level initialize function to create/open IPC channel
    specified in 'server_name'.  [ipc_transport_initialize()]

    If error,
        Return error.

    If mode flag 'm' indicates batch mode,
        Open batch results .log file using name returned from
        ipc_transport_initialize().
}
```

#### 4.7.2 Function ipc\_terminate\_server

##### Summary

This function deallocates the interprocess communication channel mailbox or socket.

##### Called By

main() FTE/bspice.c

##### Returned Value

Completion status.

##### PDL Description

```
Ipc_Status_t ipc_terminate_server ()  
{  
    Call transport level terminate function to close IPC channel.  
    [ipc_transport_terminate()]  
}
```

#### 4.7.3 Function ipc\_get\_line

##### Summary

This function gets a SPICE deck input line from the interprocess communication channel. Any special control commands in the deck beginning with a ">" or "#" character are processed internally by this function and not returned to SPICE.

The function intercepts and processes the following commands: #RETURNI, #MINTIME, #VTRANS, >PAUSE, >CONT, >STOP, >INQCON, >NETLIST, >ENDNET. Other > records are silently ignored. The function also intercepts and ignores old-style .TEMP cards generated by MSPICE

##### Called By

inp_readall()	FTE/inp.c
ipc_transport_initialize_server()	IPC/IPCaegis.c
ipc_transport_initialize_server()	IPC/IPCsockets.c
ipc_check_pause_stop()	IPC/IPCTiein.c

##### Returned Value

Completion status.

IPC_STATUS_OK	- for successful reads
IPC_STATUS_NO_DATA	- when NO_WAIT and no data available
IPC_STATUS_END_OF_DECK	- at end of deck (>ENDNET seen)
IPC_STATUS_ERROR	- otherwise

##### PDL Description

```
Ipc_Status_t ipc_get_line (str, len, wait)

char      *str;      /* OUT - Text retrieved from IPC channel */
int       *len;      /* OUT - Length of text string */
Ipc_Wait_t  wait;    /* IN  - Blocking mode flag */

{
    Loop,
    Read one SPICE line into 'str' from the IPC channel, setting 'len'
    as appropriate using the transport-level get_line function.
    [ipc_transport_get_line()]

    If return status indicates no data available, or error, return status.
```

```
If return status indicates end of deck, return error.  
  
If status indicates data successfully read from IPC channel,  
  
    Trap any control lines which cannot be processed by the simulator  
(e.g. >INQCON) and process them here. Then continue to top of  
loop to read another card. [ipc_handle_stop(), ipc_handle_returni(),  
ipc_handle_mintime(), ipc_handle_vtrans(), ipc_flush()]  
  
Otherwise, return to caller with OK status.  
}
```

#### 4.7.4 Function ipc\_send\_line

##### Summary

This function sends a line of text over the interprocess communication channel.

##### Called By

ipc_get_line()	IPC/IPC.c
ipc_send_data_prefix()	IPC/IPC.c
ipc_send_data_suffix()	IPC/IPC.c
ipc_send_dcop_prefix()	IPC/IPC.c
ipc_send_dcop_suffix()	IPC/IPC.c
ipc_send_evtdict_prefix()	IPC/IPC.c
ipc_send_evtdict_suffix()	IPC/IPC.c
ipc_send_evtdata_prefix()	IPC/IPC.c
ipc_send_evtdata_suffix()	IPC/IPC.c
ipc_send_errchk()	IPC/IPC.c
ipc_send_end()	IPC/IPC.c
ipc_send_stdout()	IPC/IPCtiein.c
ipc_send_stderr()	IPC/IPCtiein.c
EVTdump()	EVT/EVTdump.c

##### Returned Value

Completion status.

##### PDL Description

```
Ipc_Status_t ipc_send_line (str)

    char          *str;      /* The text to send */
{
    If short string, send it immediately. [ipc_send_line_binary()]

    Otherwise, send it as multiple strings because Mspice cannot handle
    things longer than 80 characters. [ipc_send_line_binary()]
}
```

#### 4.7.5 Function ipc\_send\_line\_binary

##### Summary

This function sends a logical line of data over the interprocess communication channel. It is similar to ipc\_send\_line() except does not expect the data to be null terminated. Use this for binary data strings that may have embedded nulls. A newline is appended to the data sent.

##### Called By

ipc_send_line()	IPC/IPC.c
ipc_send_double()	IPC/IPC.c
ipc_send_complex()	IPC/IPC.c
ipc_send_event()	IPC/IPC.c

##### Returned Value

Completion status.

##### PDL Description

```
Ipc_Status_t ipc_send_line (str)
{
    char          *str;      /* The text to send */
    If short string, send it immediately. [ipc_send_line_binary()]
    Otherwise, send it as multiple strings because Mspice cannot handle
    things longer than 80 characters. [ipc_send_line_binary()]
}
```

#### 4.7.6 Function ipc\_send\_data\_prefix

##### Summary

This function sends a ">DATA" line over the interprocess communication channel to signal that this is the beginning of a results dump for the current analysis point.

##### Called By

ACan()	CKT/ACan.c
DCtrCurv()	CKT/DCtrCurv.c
DCtran()	CKT/DCtran.c

##### Returned Value

Completion status.

##### PDL Description

```
Ipc_Status_t ipc_send_data_prefix (time)

double    time;      /* IN - The analysis point for this data set */
{
    Send >DATA line.  [ipc_send_line()]
}
```

#### 4.7.7 Function ipc\_send\_data\_suffix

##### Summary

This function sends a “>ENDDATA” line over the interprocess communication channel to signal that this is the end of a results dump from a particular analysis point.

##### Called By

ACan()	CKT/ACan.c
DCtrCurv()	CKT/DCtrCurv.c
DCtran()	CKT/DCtran.c

##### Returned Value

Completion status.

##### PDL Description

```
Ipc_Status_t ipc_send_data_suffix ()  
{  
    Send Mspice >ENDDATA line.  [ipc_send_line()]  
  
    Flush the results.  [ipc_flush()]  
}
```

#### 4.7.8 Function ipc\_send\_dcop\_prefix

##### Summary

This function sends a ">DCOPB" line over the interprocess communication channel to signal that this is the beginning of a results dump from a DC operating point analysis.

##### Called By

DCop()	CKT/DCop.c
ACan()	CKT/ACan.c
DCtrCurv()	CKT/DCtrCurv.c
DCtran()	CKT/DCtran.c

##### Returned Value

Completion status.

##### PDL Description

```
Ipc_Status_t ipc_send_dcop_prefix ()  
{  
    Send >DCOPB line.  [ipc_send_line()]  
}
```

#### 4.7.9 Function ipc\_send\_dcop\_suffix

##### Summary

This function sends a ">ENDDATA" line over the interprocess communication channel to signal that this is the end of a results dump from a particular analysis point.

##### Called By

DCop()	CKT/DCop.c
ACan()	CKT/ACan.c
DCtrCurv()	CKT/DCtrCurv.c
DCtran()	CKT/DCtran.c

##### Returned Value

Completion status.

##### PDL Description

```
Ipc_Status_t ipc_send_dcop_suffix ()
{
    Send Mspice >ENDDCOP line.  [ipc_send_line()]

    Flush the results.  [ipc_flush()]
}
```

#### 4.7.10 Function ipc\_send\_evtdict\_prefix

##### Summary

This function sends a ">EVTDICT" line over the interprocess communication channel to signal that this is the beginning of an event-driven node dictionary.

The line is sent only if the IPC is configured for UNIX sockets, indicating use with the V2 ATESSE SI process.

##### Called By

EVTdump()                   EVT/EVTdump.c

##### Returned Value

Completion status.

##### PDL Description

```
Ipc_Status_t ipc_send_evtdict_prefix ()
{
    If not using sockets,
        Return.

    Else,
        Send an >EVTDICT line.  [ipc_send_line()]
}
```

#### 4.7.11 Function ipc\_send\_evtdict\_suffix

##### Summary

This function sends a ">ENDDICT" line over the interprocess communication channel to signal that this is the end of an event-driven node dictionary.

The line is sent only if the IPC is configured for UNIX sockets, indicating use with the V2 ATESSE SI process.

##### Called By

EVTdump()                           EVT/EVTdump.c

##### Returned Value

Completion status.

##### PDL Description

```
Ipc_Status_t ipc_send_evtdict_suffix ()
{
    If not using sockets,
        Return.

    Else,
        Send an >ENDDICT line.  [ipc_send_line()]
}
```

#### 4.7.12 Function ipc\_send\_evtdata\_prefix

##### Summary

This function sends a ">EVTDATA" line over the interprocess communication channel to signal that this is the beginning of an event-driven data set.

The line is sent only if the IPC is configured for UNIX sockets, indicating use with the V2 ATESSE SI process.

##### Called By

EVTdump()                   EVT/EVTdump.c

##### Returned Value

Completion status.

##### PDL Description

```
Ipc_Status_t ipc_send_evtdata_prefix ()  
{  
    If not using sockets,  
        Return.  
    Else,  
        Send an >EVTDATA line.  [ipc_send_line()]  
}
```

#### 4.7.13 Function ipc\_send\_evtdata\_suffix

##### Summary

This function sends a “>ENDDATA” line over the interprocess communication channel to signal that this is the end of an event-driven node dictionary.

The line is sent only if the IPC is configured for UNIX sockets, indicating use with the V2 ATESSE SI process.

##### Called By

EVTdump()                           EVT/EVTdump.c

##### Returned Value

Completion status.

##### PDL Description

```
Ipc_Status_t ipc_send_evtdata_suffix ()
{
    If not using sockets,
        Return.

    Else,
        Send an >ENDDATA line.  [ipc_send_line()]
}
```

#### 4.7.14 Function ipc\_send\_errchk

##### Summary

This function sends a “#ERRCHK [GO—NOGO]” message over the interprocess communication channel to signal that the initial parsing of the input deck has been completed and to indicate whether or not errors were detected.

##### Called By

CKTdoJob()                    CKT/CKTdoJob.c

##### Returned Value

Completion status.

##### Global Variables

```
Ipc_Tiein_t        g_ipc;        /* INOUT - Data shared by IPC package functions */
```

##### PDL Description

```
Ipc_Status_t ipc_send_errchk()
{
    If 'g_ipc' indicates error check already sent, return.

    If syntax error has occurred,
        Send #ERRCHK NOGO.  [ipc_send_line()]

    Else,
        Send #ERRCHK GO.  [ipc_send_line()]

    Flush the results.  [ipc_flush()]
}
```

#### 4.7.15 Function ipc\_send\_end

##### Summary

This function sends either a ">ENDANAL" or a ">ABORTED" message over the inter-process communication channel together with the total CPU time used to indicate whether or not the simulation completed normally.

##### Called By

main() FTE/bspice.c

##### Returned Value

Completion status.

##### Global Variables

```
Ipc_Tiein_t      g_ipc;      /* INOUT - Data shared by IPC package functions */
```

##### PDL Description

```
Ipc_Status_t ipc_send_endanal()
{
    Send >ABORTED or >ENDANAL with cpu time used according to
    data in 'g_ipc'.  [ipc_send_line()]

    Flush the results.  [ipc_flush()]
}
```

#### 4.7.16 Function ipc\_send\_double

##### Summary

This function sends a double data value over the interprocess communication channel preceded by a character string that identifies the simulation variable.

##### Called By

OUTpData()                  FTE/OUTinterface.c

##### Returned Value

Completion status.

##### PDL Description

```
Ipc_Status_t ipc_send_double (tag, value)
{
    char      *tag;      /* IN - The node or instance */
    double    value;     /* IN - The data value to send */
    {
        Format value into string and send the data. [ipc_send_line_binary()]
    }
}
```

#### 4.7.17 Function ipc\_send\_complex

##### Summary

This function sends a complex data value over the interprocess communication channel preceded by a character string that identifies the simulation variable.

##### Called By

OUTpData()                          FTE/OUTinterface.c

##### Returned Value

Completion status.

##### PDL Description

```
Ipc_Status_t ipc_send_complex (tag, value)

char          *tag;      /* IN - The node or instance */
Mif_Complex_t  value;    /* IN - The data value to send */
{
    Format value into string and send it. [ipc_send_line_binary()]
}
```

#### 4.7.18 Function ipc\_send\_event

##### Summary

This function sends data from an event-driven node over the interprocess communication channel. The data is sent only if the IPC is configured for UNIX sockets, indicating use with the version 2 ATESSE Simulator Interface process.

##### Called By

EVTsend\_line()                   EVT/EVTdump.c

##### Returned Value

Completion status.

##### PDL Description

```
Ipc_Status_t ipc_send_event (ipc_index, step, plot_val, print_val, ipc_val, len)

int      ipc_index;    /* IN - Index used in EVTDICT */
double   step;         /* IN - Analysis point or timestep (0.0 for DC) */
double   plot_val;    /* IN - The value for plotting purposes */
char    *print_val;   /* IN - The value for printing purposes */
void    *ipc_val;    /* IN - The binary representation of the node data */
int      len;          /* IN - The length of the binary representation */

{
    If not using sockets,
        Return.

    Else,
        Report error if size of data 'len' is too big for IPC channel
        block size (currently 1000 bytes).

        Place the 'ipc_index' into a string buffer with a trailing space.

        Put the 'step' bytes in.

        Put 'plot_val' in.

        Put 'print_val' in.

        Put the length 'len' of the binary representation in.
```

Put the binary representation bytes 'ipc\_val' in.

Send the data to the IPC channel. [ipc\_send\_line\_binary()]

}

#### 4.7.19 Function ipc\_flush

##### Summary

This function flushes the interprocess communication channel buffer contents.

##### Called By

ipc_get_line()	IPC/IPC.c
ipc_send_line_binary()	IPC/IPC.c
ipc_send_data_suffix()	IPC/IPC.c
ipc_send_dcop_suffix()	IPC/IPC.c
ipc_send_evtdict_suffix()	IPC/IPC.c
ipc_send_errchk()	IPC/IPC.c
ipc_send_end()	IPC/IPC.c
ipc_send_std_files()	IPC/IPCtiein.c

##### Returned Value

Completion status.

##### PDL Description

```
Ipc_Status_t ipc_flush ()
{
    If batch mode,
        Send contents of buffer to .log file.
    Else,
        Call transport level function to send strings.
}
```

#### 4.7.20 Function ipc\_handle\_stop

##### Summary

This function sets a flag in the g\_ipc variable to signal that a stop message has been received over the IPC channel. This flag will cause the simulator to abort the current simulation immediately following the current iteration.

##### Called By

ipc\_get\_line()           IPC/IPC.c

##### Returned Value

None.

##### Global Variables

Ipc\_Tiein\_t           g\_ipc;   /\* OUT - IPC global data flags, etc. \*/

##### PDL Description

```
void ipc_handle_stop()
{
    Set 'stop_analysis' member of 'g_ipc' to true.
}
```

#### 4.7.21 Function ipc\_handle\_returni

##### Summary

This function sets a flag in the g\_ipc variable to signal that a message has been received over the IPC channel specifying that analog current values are to be returned in the results data sets. This will cause the simulator to return currents for the standard discrete components of SPICE3C1, and for backwards compatibility with the ATESSE version 1 system, will cause the simulator to process #VTRANS control cards and return currents from ATESSE version 1 format zero-valued voltage source monitor elements.

##### Called By

ipc\_get\_line()           IPC/IPC.c

##### Returned Value

None.

##### Global Variables

Ipc\_Tiein\_t           g\_ipc; /\* OUT - IPC global data flags, etc. \*/

##### PDL Description

```
void ipc_handle_returni()
{
    Set 'returni' member of 'g_ipc' to true.
}
```

#### 4.7.22 Function ipc\_handle\_mintime

##### Summary

This function sets a value in the g\_ipc variable that specifies how often data is to be returned as it is computed. If the simulator takes timestep backups, data may still be returned more often than that specified by 'mintime' so that glitches are not missed.

##### Called By

ipc\_get\_line()           IPC/IPC.c

##### Returned Value

None.

##### Global Variables

Ipc\_Tiein\_t           g\_ipc; /\* OUT - IPC global data flags, etc. \*/

##### PDL Description

```
void ipc_handle_mintime(time)

double time; /* IN - The minimum time between data sets returned */
{
    Set 'mintime' member of 'g_ipc' to 'time'.
}
```

#### 4.7.23 Function ipc\_handle\_vtrans

##### Summary

This function processes arguments from a #VTRANS card received over the IPC channel. The data on the card specifies that a particular zero-valued voltage source name should be translated to the specified instance name when returning the current for that source over the IPC channel.

##### Called By

ipc\_get\_line()           IPC/IPC.c

##### Returned Value

None.

##### Global Variables

Ipc\_Tiein\_t           g\_ipc; /\* OUT - IPC global data flags, etc. \*/

##### PDL Description

```
void ipc_handle_vtrans(vsrc, dev)

char *vsrc; /* IN - The name of the voltage source to be translated */
char *dev; /* IN - The device name the vsource name should be translated to */
{
    Add the voltage source name ('vsrc') and instance name ('dev')
    pair to the 'vtrans' member of 'g_ipc'.
}
```

#### 4.7.24 Function ipc\_send\_std\_files

##### Summary

This function sends the data written to stdout and stderr over the IPC channel. These streams were previously redirected to temporary files in routine main() before the simulation began so that error messages and other informative messages could be collected.

##### Called By

main() FTE/bspice.c

##### Returned Value

The success or failure status of sending the data.

##### PDL Description

```
Ipc_Status_t ipc_send_std_files()
{
    Send the data collected from writes to the stdout stream.
    [ipc_send_stdout()]

    Send the data collected from writes to the stderr stream.
    [ipc_send_stderr()]

    Flush the data to the IPC channel.  [ipc_flush()]
}
```

#### 4.7.24.1 Static Function ipc\_send\_stdout

##### Summary

This function sends the data written to stdout over the IPC channel. This stream was redirected in routine main() to a temporary file so that information written to this file descriptor could be collected and sent over the IPC channel at the end of the simulation.

##### Called By

ipc\_send\_std\_files()      IPC/IPCtiein.c

##### Returned Value

None.

##### PDL Description

```
static void ipc_send_stdout()
{
    Rewind the redirected stdout stream.

    Begin reading from the top of file and send lines over the IPC channel.
    Don't send newlines, since they will be appended by ipc_send_line().
    [ipc_send_line()]

    Finally, rewind file again to discard the data already sent.
}
```

#### 4.7.24.2 Static Function ipc\_send\_stderr

##### Summary

This function sends the data written to stderr over the IPC channel. This stream was redirected in routine main() to a temporary file so that information written to this file descriptor could be collected and sent over the IPC channel at the end of the simulation.

##### Called By

ipc\_send\_std\_files()      IPC/IPCtiein.c

##### Returned Value

None.

##### PDL Description

```
static void ipc_send_stderr()
{
    Rewind the redirected stderr stream.

    Begin reading from the top of file and send lines over the IPC channel.
    Don't send newlines, since they will be appended by ipc_send_line().
    [ipc_send_line()]

    Finally, rewind file again to discard the data already sent.
}
```

#### 4.7.25 Function ipc\_screen\_name

##### Summary

This function screens names of instances and nodes to limit the data returned over the IPC channel.

##### Called By

beginPlot()                    FTE/OUTinterface.c

##### Returned Value

True if the name corresponds to a data item that should be returned over the IPC channel during the simulation.

##### Global Variables

Ipc\_Tiein\_t                g\_ipc; /\* IN - IPC global data flags, etc. \*/

##### PDL Description

```
Ipc_Boolean_t ipc_screen_name(name, mapped_name)

char *name;                /* IN - Original name */
char *mapped_name;        /* OUT - New name (possibly translated) */
{
    If name contains a ':' character, assume it is a subcircuit and
    return false.

    If name is numeric,

        Return false if numeric value is > 100,000 which indicates it was
        added by the ms_server process in the ATESSE version 1 system.

        Otherwise, copy 'name' to 'mapped_name' and return true.

    If name contains a '#' character and the characters after the '#' are
    'branch',

        This is a voltage source name, so strip the "#branch" suffix added
        by the simulator, copy the stripped name to 'mapped_name', and
```

```
    return true.

Else

    Copy 'name' to 'mapped_name' unaltered.

If length of 'mapped_name' is 8 characters and the 6'th character is
a '$',

    This is a source added by the ATESSE 1 system for monitoring currents
    on a device/subcircuit and needs translating. Scan the 'vtrans' data
    in 'g_ipc' to find the matching name and copy the new name from the
    'vtrans' table to 'mapped_name'. Return true.

Else

    Return true.
}
```

#### 4.7.26 Function ipc\_get\_devices

##### Summary

This function is used by the OUTinterface package to determine what instances will have data returned over the IPC channel during the simulation. It creates two arrays, one holding the names of the instance to be returned, and the other holding a multiplication factor of +1 or -1 for currents.

##### Called By

addIspecials()	FTE/OUTinterface.c
addCLDQJMspecials()	FTE/OUTinterface.c

##### Returned Value

Number of instances placed into output arrays.

##### PDL Description

```
int ipc_get_devices(circuit, device, names, modtypes)

void    *circuit;      /* IN - The CKTcircuit circuit structure */
char    *device;       /* IN - The device name as it appears in the info struct */
char    ***names;      /* OUT - Array of name strings to be built */
double  **modtypes;   /* OUT - Array of types to be built */
{
    Get the index of model 'device' in the linked list of models in 'circuit'.
    [INPtypelook()]

    Iterate through all models of this type in 'circuit',
    Iterate through all instance of this model,
    Get the name of the instance from the instance structure.

    Continue to next instance if name has an embedded ':' character
    indicating it is in a subcircuit.

    Otherwise, add the name to the 'names' array.

    If the device is a BJT, JFET, MOS1, MOS2, or MOS3 transistor,
    Look up the type in the model structure and assign
    the value to found to 'modtypes' for this instance.
```

```
Else
    Assign +1 to 'modtypes' for this instance.
    Return the number of instances placed in the 'names' and 'modtypes' arrays.
}
```

#### 4.7.27 Function ipc\_free\_devices

##### Summary

This function frees temporary data created by ipc\_get\_devices().

##### Called By

addIspecials()	FTE/OUTinterface.c
addCLDQJMspecials()	FTE/OUTinterface.c

##### Returned Value

None.

##### PDL Description

```
void ipc_free_devices(num_items, names, modtypes)

int      num_items;    /* IN    - Number of things to free */
char    **names;       /* INOUT - Array of name strings to be freed */
double   *modtypes;    /* INOUT - Array of types to be freed */
{
    For the number of items specified,
        Free the space used by the string in the 'names' array.
    Free the 'names' and 'modtypes' arrays.
}
```

#### 4.7.28 Function ipc\_check\_pause\_stop

##### Summary

This function is called at various times during a simulation to check for incoming messages of the form >STOP or >PAUSE signaling that simulation should be stopped or paused. Processing of the messages is handled by ipc\_get\_line().

##### Called By

NIiter()	NI/NIiter.c
NIacIter()	NI/NIacIter.c

##### Returned Value

None.

##### Global Variables

```
Ipc_Tiein_t      g_ipc; /* IN - IPC global data flags, etc. */
```

##### PDL Description

```
void ipc_check_pause_stop()
{
    If 'g_ipc' 'stop_analysis' flag indicates that we have already seen
    stop analysis command, don't call ipc_get_line(), just return.
    This is provided so that the function can be called multiple times
    during the process of stopping.

    Otherwise do a non-blocking call to ipc_get_line() to check for messages.
    We assume that the only possible messages at this point are >PAUSE
    and >STOP, so we don't do anything with the returned text.
    [ipc_get_line()]
}
```

#### 4.7.29 BSD UNIX Sockets Transport Layer

The following functions provide the transport-level specific routines for implementation of interprocess communication using BSD UNIX Sockets.

```
ipc_transport_initialize_server  
ipc_transport_get_line  
ipc_transport_send_line  
ipc_transport_terminate_server
```

#### 4.7.29.1 Function ipc\_transport\_initialize\_server

## Summary

This function creates and opens a BSD internet stream type socket for communicating with client processes. If the simulator is operating in batch mode, the function also reads the first message from the IPC channel which is expected to be the name of the file to write result to.

A simple format is established for messages exchanged on the socket. The format of a message line is:

bytes	description
0	recognition character for beginning of line; value is BOL_CHAR.
1-4	message length (not including bytes 0-4); 32 bits in htonl format; if value = -1, then EOF and socket should be closed.
5-N+5	message body of length specified in bytes 1-4.

### Called By

ipc\_initialize\_server() IPC/IPC.c

## Returned Value

Success or failure status of creating the socket.

## Global Variables

```
Ipc_Tiein_t      g_ipc;        /* IN - IPC global data flags, etc. */
Ipc_Sock_State_t sock_state; /* INOUT - State of socket */
```

## PDL Description

```
Ipc_Status_t ipc_transport_initialize_server (server_name, mode, protocol,  
                                         batch_filename)  
  
char           *server_name;      /* IN - The port number for the socket */  
Ipc_Mode_t     mode;            /* IN - Not used */
```

XSPICE Simulator  
Software Design Document

Detailed Design  
Interprocess Communication

```
Ipc_Protocol_t protocol; /* IN - Not used */  
char *batch_filename; /* OUT - Batch .log file pathname */  
{  
    Create an internet stream type socket.  
  
    Bind port number specified in 'server_name' to socket.  
  
    Listen on the socket for open calls from client.  
  
    Set 'sock_state' to initialized.  
  
    If 'g_ipc' indicates that we are in batch mode,  
        Read the first line from the IPC channel. [ipc_get_line()]  
        Copy the line text into 'batch_filename'.  
}
```

#### 4.7.29.2 Function ipc\_transport\_get\_line

##### Summary

This function reads one logical line (data packet) from the IPC channel.

##### Called By

ipc_get_line()	IPC/IPC.c
ipc_transport_terminate_server ()	IPCsockets.c

##### Returned Value

Status of reading from the socket. Returns IPC\_STATUS\_NO\_DATA if there is no data available on the socket. Otherwise, returns IPC\_STATUS\_OK, or IPC\_STATUS\_ERROR.

##### Global Variables

```
Ipc_Sock_State_t    sock_state; /* INOUT - State of socket */
```

##### PDL Description

```
Ipc_Status_t ipc_transport_get_line (str, len, wait)

char        *str;    /* OUT - Returns the result, null terminated */
int         *len;    /* OUT - Length of str */
Ipc_Wait_t   wait;   /* IN - Wait or don't wait on incoming msg */
{
    Return error if 'sock_state' indicates socket port not yet initialized
    or connected to client.

    If 'sock_state' indicates socket not yet connected to client,
        Accept a connection request.

        Mark 'sock_state' as connected to client.

    Attempt to read 5 character header in blocking or non-blocking mode,
    depending on 'wait'.

    If end of file,
```

Close the stream and set the socket to uninitialized.

If first char is not BOL\_CHAR, return error.

Interpret message length (N) from bytes 2 - 5 of header.

Read N characters to get the message body into 'str'. This call always blocks because we expect the message body to immediately follow its header.

If end of file,

Close the stream and set the socket to uninitialized.

Record length of message in 'len' and add a null terminator to 'str'.

Return OK.

}

#### 4.7.29.3 Function ipc\_transport\_send\_line

##### Summary

This function writes a logical line of information to the IPC channel. First it sends a message header and then the actual message body.

##### Called By

ipc\_send\_line()           IPC/IPC.c

##### Returned Value

Status of writing to the socket. Returns IPC\_STATUS\_OK if data successfully written, or IPC\_STATUS\_ERROR otherwise.

##### Global Variables

Ipc\_Sock\_State\_t    sock\_state; /\* IN - State of socket \*/

##### PDL Description

```
Ipc_Status_t ipc_transport_send_line (str, len)

char *str; /* IN - String to write */
int len; /* IN - Number of characters out of STR to write */
{
    Return error if 'sock_state' indicates socket port not yet connected
    to client.

    Write message 5 byte header to socket stream, beginning with BOL_CHAR,
    and followed by 4-byte message body length 'len'.

    Write message body ('len' bytes from 'str').
}
```

#### 4.7.29.4 Function ipc\_transport\_terminate\_server

##### Summary

This function reads all pending incoming messages and discards them. Reading continues until a read error occurs or EOF is reached, at which time the socket is closed. Note that this function does not actually close the socket. This is done in ipc\_transport\_get\_line, which is called in this function.

##### Called By

ipc\_terminate\_server()                   IPC/IPC.c

##### Returned Value

Returns IPC\_STATUS\_EOF if successfully terminated, or IPC\_STATUS\_ERROR otherwise.

##### PDL Description

```
Ipc_Status_t ipc_transport_terminate_server()
{
    Continue to read lines (with ipc_transport_get_line()) and ignore
    them until socket EOF is reached and ipc_transport_get_line() closes
    the socket. [ipc_transport_get_line()]
}
```

#### 4.7.30 HP/Apollo Mailbox Transport Layer

The following functions provide the transport-level specific routines for implementation of interprocess communication using HP/Apollo Mailboxes.

```
ipc_transport_initialize_server
ipc_transport_get_line
ipc_transport_send_line
ipc_transport_terminate_server
```

#### 4.7.30.1 Function ipc\_transport\_initialize\_server

##### Summary

This function creates an Aegis Mailbox IPC channel for communicating with client processes. It also reads the first message from the IPC channel, which is expected to be the name of the file to write result to if operating in batch mode. For backwards compatibility with the ATESSE version 1.0 system, this first message is also read if operating in interactive mode.

##### Called By

ipc\_initialize\_server()                   IPC/IPC.c

##### Returned Value

Returns IPC\_STATUS\_OK if successful, or IPC\_STATUS\_ERROR otherwise.

##### Global Variables

```
Ipc_Mbx_State_t mbx_state; /* INOUT - State of mailbox */
```

##### PDL Description

```
Ipc_Status_t ipc_transport_initialize_server (server_name, m, p, batch_filename)  
  
char                 *server_name;       /* IN - The mailbox pathname */  
Ipc_Mode_t          m;                    /* Not used */  
Ipc_Protocol_t      p;                    /* Not used */  
char                 *batch_filename;     /* OUT - Batch filename returned */  
{  
    Create the mailbox.  
  
    If not successful,  
  
        Mark 'mbx_state' as uninitialized.  
  
    Return error.  
  
Else
```

Get the first message and save it in batch\_filename. [ipc\_get\_line()]

}

#### 4.7.30.2 Function ipc\_transport\_get\_line

##### Summary

This function reads data sent by a client over the mailbox channel. It also handles the initial opening of the mailbox channel.

##### Called By

ipc_get_line()	IPC/IPC.c
ipc_transport_terminate_server ()	IPC/IPCaegis.c

##### Returned Value

Status of reading from the socket. Returns IPC\_STATUS\_NO\_DATA if there is no data available on the socket. Otherwise, returns IPC\_STATUS\_OK, or IPC\_STATUS\_ERROR, or IPC\_STATUS\_EOF.

##### Global Variables

```
static Ipc_Mbx_State_t mbx_state; /* INOUT - State of mailbox */
```

##### PDL Description

```
Ipc_Status_t ipc_transport_get_line (str, len, wait)

    char          *str;    /* The string text read from IPC channel */
    int           *len;    /* The length of str */
    Ipc_Wait_t    wait;   /* Blocking or non-blocking */

{
    If 'mbx_state' indicates that mailbox is not initialized, return error.

    Loop forever.

    Read a message from the IPC channel in blocking or non-blocking
    mode according to the value of 'wait'.

    Return no-data if channel is empty and call was in non-blocking mode.

    If message is an open request,
        If 'mbx_state' indicates we are already connected to a client,
```

```
    return error.  
  
Otherwise, Acknowledge the request and mark 'mbx_state' as  
connected to client.  
  
Return any data included with the open request.  
  
Else if message is EOF  
  
    Deallocate the mailbox, mark 'mbx_state' as uninitialized, and  
    return end-of-file.  
  
Else if message is data  
  
    Get the message data into 'str' and 'len' and return OK.  
}
```

#### 4.7.30.3 Function ipc\_transport\_send\_line

##### Summary

This function sends a message to the current client through the mailbox channel.

##### Called By

ipc\_send\_line()                           IPC/IPC.c

##### Returned Value

IPC\_STATUS\_OK if successful. Otherwise, IPC\_STATUS\_ERROR.

##### Global Variables

```
static Ipc_Mbx_State_t mbx_state; /* IN - State of mailbox */
```

##### PDL Description

```
Ipc_Status_t ipc_transport_send_line (str, len)

char *str; /* IN - The bytes to send */
int len; /* IN - The number of bytes from str to send */
{
    If 'mbx_state' indicates we are not currently connected to a client,
    return error.

    Pack message into mailbox data structure.

    Put the message into the mailbox.
}
```

#### 4.7.30.4 Function ipc\_transport\_terminate\_server

##### Summary

This function reads all pending incoming messages and discards them. Reading continues until a read error occurs or EOF is reached, at which time the socket is closed. Note that this function does not actually close the socket. This is done in ipc\_transport\_get\_line, which is called in this function.

##### Called By

ipc\_terminate\_server()           IPC/IPC.c

##### Returned Value

Returns IPC\_STATUS\_EOF if successfully terminated, or IPC\_STATUS\_ERROR otherwise.

##### PDL Description

```
Ipc_Status_t ipc_transport_terminate_server()
{
    Continue to read lines (with ipc_transport_get_line()) and ignore
    them until EOF is reached and ipc_transport_get_line() closes
    the mailbox. [ipc_transport_get_line()]
}
```

# 5

# CSCI Data

This section describes the major new data structures added to SPICE 3C1 in implementing the XSPICE simulator. Refer to Appendix A and the references for information on the data structures defined in the SPICE 3C1 core.

## 5.1 Interface to Code Model Libraries

New data structures are added to the simulator at link time that describe the set of code models that will be known to the simulator at runtime. These data structures and associated C functions are created using the Code Model Toolset and involve the following files for each individual code model:

ifspec.ifs

ifspec.c

ifspec.o

cfunc.mod

cfunc.c

cfunc.o

In general, these files will appear in a separate “model directory” for each code model.

In addition, the Code Model Toolset generates the following files immediately prior to linking a set of specified code models with the simulator core.

CMextrn.h  
CMinfo.h

There will be one copy of these files for each simulator executable. They are created in a “simulator directory” and contain the data structure identifiers associated with each model linked with the XSPICE simulator core.

### 5.1.1 Interface Specification Data

The ifspec.o object file contains the information required by the SIM-MIF parser functions to parse and error check the XSPICE input deck, as well as the information required to call appropriate functions during a simulation. This file is a compiled version of ifspec.c, which is generated automatically by the Code Model Preprocessor from a user-written ifspec.ifs Interface Specification file.

The ifspec.c file which is compiled to create ifspec.o contains an XXX\_info data structure, where XXX is the name of the code model C function. This structure is of type SPICEdev and is initialized with an appropriate IFdevice data structure containing the data required by the parser, as well as the set of function pointers used during simulation and the sizes of the MIFmodel and MIFinstance structures that will be dynamically allocated during deck parsing.

The ifspec.c file contains all information needed by the MIF parser to interpret and error check a code model referenced in an XSPICE input deck, as well as to call appropriate functions during a simulation. This information includes:

- A table of model parameters in standard SPICE 3C1 IFparm format.
- A function pointer to the code model function.
- A table describing the expected connections on the device, including default type, direction, allowed types, etc.
- A table describing model parameters in more detail than the SPICE 3C1 IFparm format. This table defines default values, types, limits, etc.
- A table describing static variables.
- Function pointers to the SIM-MIF CSC functions called during parsing and simulation.

- The size of dynamic MIFinstance and MIFmodel structures that are allocated in the circuit structure linked lists to hold runtime information about the models and instances.

The definition of the major structures used in ifsproc.c to define the model are shown below.

```

typedef struct SPICEdev {
    IFdevice DEVpublic;

    int (*DEVparam)();          /* routine to input a parameter to a device instance */
    int (*DEVmodParam)();       /* routine to input a parameter to a model */
    int (*DEVload)();           /* routine to load the device into the matrix */
    int (*DEVsetup)();          /* setup routine to preprocess devices once before solution
                                begins */
    int (*DEVpzSetup)();        /* setup routine to process devices specially for pz
                                analysis */
    int (*DEVtemperature)();    /* subroutine to do temperature dependent setup processing */
    int (*DEVtrunc)();          /* subroutine to perform truncation error calc. */
    int (*DEVfindBranch)();     /* subroutine to search for device branch eq.s */
    int (*DEVacLoad)();          /* ac analysis loading function */
    int (*DEVaccept)();         /* subroutine to call on acceptance of a timepoint */
    void (*DEVdestroy)();        /* subroutine to destroy all models and instances */
    int (*DEVmodDelete)();      /* subroutine to delete a model and all instances */
    int (*DEVdelete)();          /* subroutine to delete an instance */
    int (*DEVsetic)();          /* routine to pick up device initconds from rhs */
    int (*DEVask)();             /* routine to ask about device details */
    int (*DEVmodAsk)();          /* routine to ask about model details */
    int (*DEVpzLoad)();          /* routine to load for pole-zero analysis */
    int (*DEVconvTest)();        /* convergence test function */
    int (*DEVsenSetup)();        /* routine to setup the device sensitivity info */
    int (*DEVsenLoad)();          /* routine to load the device sensitivity info */
    int (*DEVsenUpdate)();       /* routine to update the device sensitivity info */
    int (*DEVsenAcLoad)();       /* routine to load the device ac sensitivity info */
    void (*DEVsenPrint)();       /* subroutine to print out sensitivity info */
    int (*DEVsenTrunc)();        /* subroutine to print out sensitivity info */

    int DEVinstSize;            /* size of an instance */
    int DEVmodSize;              /* size of a model */

} SPICEdev;

typedef struct sIFdevice {

    /* SPICE 3C1 members */

    char *name;                  /* name of this type of device */
    char *description;            /* description of this type of device */

    int terms;                   /* number of terminals on this device */
    int numNames;                 /* number of names in termNames */
}

```

```

char **termNames;           /* pointer to array of pointers to names */
                           /* array contains 'terms' pointers */

int numInstanceParms;      /* number of instance parameter descriptors */
IFparm *instanceParms;     /* array of instance parameter descriptors */

int numModelParms;         /* number of model parameter descriptors */
IFparm *modelParms;        /* array of model parameter descriptors */

/* New members */

void (*cm_func)();          /* pointer to code model function */

int num_conn;               /* number of code model connections */
Mif_Conn_Info_t *conn;       /* array of connection info for mif parser */

int num_param;              /* number of parameters = numModelParms */
Mif_Param_Info_t *param;    /* array of parameter info for mif parser */

int num_inst_var;           /* number of instance vars = numInstanceParms */
Mif_Inst_Var_Info_t *inst_var; /* array of instance var info for mif parser */

} IFdevice;

typedef struct sIFparm {
    char *keyword;
    int id;
    int dataType;
    char *description;
} IFparm;

typedef struct Mif_Conn_Info_s {

    char      *name;           /* Name of this connection */
    char      *description;    /* Description of this connection */
    Mif_Dir_t direction;      /* Is this connection an input, output, or both? */
    Mif_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 */
    Mif_Port_Type_t *allowed_type; /* The allowed types */
    char      **allowed_type_str; /* The allowed types in string form */
    Mif_Boolean_t is_array;    /* True if connection is an array */
    Mif_Boolean_t has_lower_bound; /* True if there is an array size lower bound */
    int       lower_bound;     /* Array size lower bound */
    Mif_Boolean_t has_upper_bound; /* True if there is an array size upper bound */
    int       upper_bound;     /* Array size upper bound */
    Mif_Boolean_t null_allowed; /* True if null is allowed for this connection */

} Mif_Conn_Info_t;

```

```

typedef struct Mif_Param_Info_s {

    char          *name;           /* Name of this parameter */
    char          *description;   /* Description of this parameter */
    Mif_Data_Type_t type;         /* Is this a real, boolean, string, ... */
    Mif_Boolean_t has_default;   /* True if there is a default value */
    Mif_Parse_Value_t default_value; /* The default value */
    Mif_Boolean_t has_lower_limit; /* True if there is a lower limit */
    Mif_Parse_Value_t lower_limit; /* The lower limit for this parameter */
    Mif_Boolean_t has_upper_limit; /* True if there is a upper limit */
    Mif_Parse_Value_t upper_limit; /* The upper limit for this parameter */
    Mif_Boolean_t is_array;       /* True if parameter is an array */
    Mif_Boolean_t has_conn_ref;   /* True if parameter is associated with a
                                  connector */

    int           conn_ref;        /* The subscript of the associated connector */
    Mif_Boolean_t has_lower_bound; /* True if there is an array size lower bound */
    int           lower_bound;     /* Array size lower bound */
    Mif_Boolean_t has_upper_bound; /* True if there is an array size upper bound */
    int           upper_bound;     /* Array size upper bound */
    Mif_Boolean_t null_allowed;   /* True if null is allowed for this parameter */

} Mif_Param_Info_t;

typedef struct Mif_Inst_Var_Info_s {

    char          *name;           /* Name of this instance var */
    char          *description;   /* Description of this instance var */
    Mif_Data_Type_t type;         /* Is this a real, boolean, string, ... */
    Mif_Boolean_t is_array;       /* True if instance var is an array */

} Mif_Inst_Var_Info_t;

```

### 5.1.2 C Function Data

The cfunc.o file contains the user written code model C function in compiled form. In a manner similar to the ifspec file, cfunc.o is produced from cfunc.c which is automatically generated by the Code Model Toolkit from a user supplied cfunc.mod file.

All data passed to and from the code model function in cfunc.mod is passed in a single structure of type "Mif\_Private\_t". This type is isolated from the code model developer during the writing of the cfunc.mod file by the Code Model Toolset. The Code Model Toolset places a special "macro" called "ARGS" in the code model function argument list. The Code Model Toolset's preprocessor translates this macro into the following argument list:

```
(Mif_Private_t *private)
```

References to items in this structure are also made with “macros” translated by the Code Model Toolset. For example, a statement in a code model of the form:

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

(where “a” is a port name defined in the Interface Specification file) is translated by the Code Model Toolset to:

```
in = private->conn[0]->port[0]->input.rvalue;
```

In all cases except the macro ARGS, the translation of a macro is to a C language lvalue.

The approach of having the developer code his or her function with macros rather than direct references to the structure in the function argument list serves two purposes:

- The developer is insulated from the detailed syntax of the structure and can express items in terms of the port and parameter names defined in the Interface Specification.
- The private structure can be modified or extended in the future with the worst effect on existing code models being a requirement to recompile them.

The contents of the Mif\_Private\_t structure are defined by the available macros recognized by the Code Model Toolset. There are four major parts to the structure:

- Data about the circuit such as the analysis mode, time, frequency, etc.
- Data about ports in which inputs, outputs, and partials are accessed.
- Data about parameters in which parameter values and other information can be found.
- Data about static variables.

Refer to Appendix B and the Interface Design Document for the XSPICE Simulator of the Automatic Test Equipment Software Support Environment (ATESSE) for more information on these macros. The basic typedefs for data passed to and from a code model through Mif\_Private\_t are shown below. Refer to include/MIFcmdat.h and related include files for further definition.

```

/*
 * The top level data structure passed to code models.
 */

typedef struct Mif_Private_s {

    Mif_Circ_Data_t      circuit;      /* Information about the circuit */
    int                  num_conn;     /* Number of connections on this model */
    Mif_Conn_Data_t      **conn;       /* Information about each connection */
    int                  num_param;    /* Number of parameters on this model */
    Mif_Param_Data_t     **param;      /* Information about each parameter */
    int                  num_inst_var; /* Number of instance variables */
    Mif_Inst_Var_Data_t  **inst_var;   /* Information about each inst variable */

} Mif_Private_t;

typedef struct Mif_Circ_Data_s {

    Mif_Boolean_t         init;        /* True if first call to model - a setup pass */
    Mif_Analysis_t        anal_type;   /* Current analysis type */
    Mif_Boolean_t         anal_init;   /* True if first call in this analysis type */
    Mif_Call_Type_t       call_type;   /* Analog or event type call */
    double                time;        /* Current analysis time */
    double                frequency;   /* Current analysis frequency */
    double                temperature; /* Current analysis temperature */
    double                t[8];        /* History of last 8 analysis times t[0]=time */

} Mif_Circ_Data_t;

typedef struct Mif_Conn_Data_s {

    char                 *name;       /* Name of this connection - currently unused */
    char                 *description; /* Description of this connection - unused */
    Mif_Boolean_t         is_null;    /* Set to true if null in SPICE deck */
    Mif_Boolean_t         is_input;   /* Set to true if connection is an input */
    Mif_Boolean_t         is_output;  /* Set to true if connection is an output */
    int                  size;       /* The size of an array (1 if scalar) */
    Mif_Port_Data_t      **port;     /* Pointer(s) to port(s) for this connection */

} Mif_Conn_Data_t;

typedef struct Mif_Param_Data_s {

    Mif_Boolean_t         is_null;    /* True if no value given on .model card */
    int                  size;       /* Size of array (1 if scalar) */
    Mif_Value_t           *element;   /* Value of parameter(s) */

} Mif_Param_Data_t;

```

```
typedef struct Mif_Inst_Var_Data_s {

    int          size;           /* Size of array (1 if scalar)      */
    Mif_Value_t *element;        /* Value of instance variables(s)   */

} Mif_Inst_Var_Data_t;
```

### 5.1.3 Linker Include Data

The Interface Specification and code model C function described above are integrated into the simulator using the CMextrn.h and CMinfo.h files generated by the Code Model Toolset. These two files are included during the compilation of CKT/SPIinit.c which defines the DEVICES[] array that holds all information on all models known to the simulator.

Each time a new simulator is made, CKT/SPIinit.c is compiled into the local “simulator directory” created by “mksimdir” and compiled immediately prior to linking. For further description of these files, refer to the Interface Design Document for the Simulator of the Automatic Test Equipment Software Support Environment (ATESSE).

## 5.2 Interface to User Defined Node Libraries

New data structures and functions can also be added to the simulator at compilation time that describe event-driven node types that will be known to the simulator at runtime.

### 5.2.1 C Function Data

To add a new user-defined node, several functions must be written which perform basic manipulations on the event-driven node data. The functions (required and optional) are listed below. For optional functions, the function can be left undefined and the pointer placed into the Evt\_Udn\_Info\_t structure shown below can be specified as NULL.

**Required functions:**

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

**Optional functions:**

- |           |  |
|-----------|--|
| dismantle | Free allocations inside structure (but not structure itself).                |
| invert    | Invert logical value of structure.   |
| resolve   | Determine the resultant when multiple outputs are connected to a node.       |
| plot_val  | Output a real value for specified structure component for plotting purposes. |
| print_val | Output a string value for specified structure component for printing.        |
| ipc_val   | Output a binary representation of the data and an int giving its size.       |

These functions are linked into the simulator and are accessed using a function pointer technique similar to that used by SPICE for accessing device model specific functions. The data structure used to hold the function pointers is automatically placed at the bottom of the “udnfunc.c” file created by the “mkudndir” utility in the Code Model Toolkit. Its definition is:

```
typedef struct {
    char    *name;
    char    *description;
    void    ((*create))(void **evt_struct);
    void    ((*dismantle))(void *evt_struct);
    void    ((*initialize))(void *evt_struct);
    void    ((*invert))(void *evt_struct);
    void    ((*copy))(void *evt_from_struct, void *evt_to_struct);
    void    ((*resolve))(int num_struct, void **evt_struct_arrayt, void *evt_struct);
    void    ((*compare))(void *evt_struct1, void *evt_struct2, Mif_Boolean_t *equal));
    void    ((*plot_val))(void *evt_struct, char *member, double *val);
    void    ((*print_val))(void *evt_struct, char *member, char **val);
    void    ((*ipc_val))(void *evt_struct, void **ipc_val, int *ipc_val_size));
} Evt_Udn_Info_t;
```

### 5.2.2 Linker Include Data

The following two include files are used to specify the data to be linked into the simulator. These files are similar to the Linker Include Data used for adding code models and are placed in a “simulator directory” and included by SPInit.c when it is compiled immediately prior to linking the simulator.

UDNextrn.h  
UDNinfo.h

## 5.3 Model and Instance Data Structures

The MIF model and instance structures are patterned after the XXXmodel and XXXinstance dynamically allocated structures used by standard SPICE 3C1 devices. These structures form the basis for the linked lists of models and instances that are traversed during a simulation to process the entire circuit.

The model structure “MIFmodel” obeys the SPICE 3C1 requirement that its first elements match the “prefix” typedef GENmodel used by all devices. The additional information specific to type MIFmodel is added after the prefix and holds the values of parameters read from the SPICE deck.

The instance structure “MIFinstance” partially violates the SPICE 3C1 requirement that its first elements match the “prefix” type GENinstance. This violation is required in order to allow code models to have an arbitrary number of ports. Areas of the SPICE 3C1 code that accessed the areas of the GENinstance structure that are changed in the ATESSE simulator have been modified. The number of such code modifications required in SPICE 3C1 is minimal. New releases should be carefully checked to determine if additional modifications are needed.

The information specific to MIFinstance (not part of GENinstance) includes the following:

- Data on ports used in determining the input values to be given to code models and in filling the matrix according to the outputs computed by the code models.
- The values of parameters. This is a pointer set to the structure in the MIFmodel structure parameter data since the current implementation of the XSPICE simulator does not support per-instance parameters.
- The values of static variables used by the code models to store data between calls. These static variables are needed since the code model function is shared by multiple instances. They substitute for “static” class data in a C function.

- A table of state information for the instance. This area of the structure is used by the cm\_analog\_alloc() and cm\_analog\_get\_ptr() code model support functions.
- A table of integration information for the instance. This area of the structure is used by the cm\_analog\_integrate() code model support function.
- A table of convergence check information for the instances. This area of the structure is used by cm\_analog\_converge() and indirectly by cm\_analog\_integrate().

The definition of MIFmodel and MIFinstance typedefs are shown below.

```
typedef struct sMIFinstance {

    struct sMIFmodel     *MIFmodPtr;          /* backpointer to model */
    struct sMIFinstance *MIFnextInstance; /* pointer to next instance of current model */
    IFuid                MIFname;            /* pointer to character string naming this
                                                instance */

    int                  num_conn;           /* number of connections on the code model */
    Mif_Conn_Data_t     **conn;              /* array of data structures for each
                                                connection */

    int                  num_inst_var;        /* number of instance variables on the code
                                                model */
    Mif_Inst_Var_Data_t **inst_var;         /* array of structs for each instance var */

    int                  num_param;          /* number of parameters on the code model */
    Mif_Param_Data_t    **param;             /* array of structs for each parameter */

    int                  num_state;          /* Number of state tags used for this inst */
    Mif_State_t          *state;              /* Info about states */

    int                  num_intgr;          /* Number of integrals */
    Mif_Intgr_t          *intgr;              /* Info for integrals */

    int                  num_conv;            /* Number of things to be converged */
    Mif_Conv_t            *conv;               /* Info for convergence things */

    Mif_Boolean_t         initialized;        /* True if model called once already */

    Mif_Boolean_t         analog;              /* true if this inst is analog or hybrid
                                                type */

    Mif_Boolean_t         event_driven;       /* true if this inst is event-driven or hybrid
                                                type */

    int                  inst_index;         /* Index into inst_table in evt struct in
                                                ckt */
}

} MIFinstance ;
```

```
typedef struct sMIFmodel {

    int          MIFmodType;      /* type index of this device type */
    struct sMIFmodel *MIFnextModel; /* pointer to next possible model in linked list */
    MIFinstance   *MIFinstances;  /* pointer to list of instances that have this
                                    model */
    IFuid        MIFmodName;     /* pointer to character string naming this model */

    int          num_param;      /* number of parameters on the code model */
    Mif_Param_Data_t **param;   /* array of structs for each parameter */

    Mif_Boolean_t analog;       /* true if this model is analog or hybrid type */
    Mif_Boolean_t event_driven; /* true if this model is event-driven or hybrid
                                type */

} MIFmodel;
```

## 5.4 Event-Driven Simulation Data

All data for the event-driven simulation algorithm except that stored by the MIFmodel and MIFinstance structures is stored in a new element added to the main CKTcircuit structure. The element is named “evt” and is typically referenced as ckt->evt->... The typedef used to declare “evt” is shown below:

```
typedef struct {
    Evt_Count_t  counts;        /* Number of insts, nodes, etc. */
    Evt_Info_t   info;         /* Static info about insts, etc. */
    Evt_Queue_t  queue;        /* Dynamic queued events */
    Evt_Data_t   data;         /* Results and state data */
    Evt_Limit_t  limits;       /* Iteration limits, etc. */
    Evt_Job_t    jobs;         /* Data held from multiple job runs */
    Evt_Option_t options;     /* Data input on .options cards */
} Evt_Ckt_Data_t;
```

### 5.4.1 Evt\_Count\_t

The “counts” element of “evt” is used to store counts of the number of instances, nodes, outputs, etc. involved in the event-driven portion of the circuit. This data is collected as the circuit description deck is read and parsed. The associated typedef is shown below:

```
typedef struct {
    int          num_insts;     /* number of event/hybrid instances parsed */
    int          num_hybrids;    /* number of hybrids parsed */
    int          num_hybrid_outputs; /* number of outputs on all hybrids parsed */
    int          num_nodes;      /* number of event nodes parsed */
}
```

```

    int      num_ports;          /* number of event ports parsed */
    int      num_outputs;        /* number of event outputs parsed */
} Evt_Count_t;

```

The primary purpose of the “counts” information is to allow the simulator to allocate arrays for efficient access to data after initial parsing.

#### 5.4.2 Evt\_Info\_t

The “info” element of “evt” holds static data about instances, nodes, ports, outputs and hybrids (models with both analog and event-driven inputs/outputs). This data constitutes the simulator’s internal representation of the event-driven circuits, and is used during simulation to locate data to be accessed and manipulated. The main structure definition is shown below:

```

typedef struct {
    Evt_Inst_Info_t *inst_list;           /* static info about event/hybrid instances */
    Evt_Node_Info_t *node_list;           /* static info about event nodes */
    Evt_Port_Info_t *port_list;           /* static info about event ports */
    Evt_Output_Info_t *output_list;        /* static info about event outputs */
    int             *hybrid_index;         /* vector of inst indexes for hybrids */
    Evt_Inst_Info_t **inst_table;          /* vector of pointers to elements in inst_list */
    Evt_Node_Info_t **node_table;          /* vector of pointers to elements in node_list */
    Evt_Port_Info_t **port_table;          /* vector of pointers to elements in port_list */
    Evt_Output_Info_t **output_table;       /* vector of pointers to elements in
                                              output_list */
} Evt_Info_t;

```

The four XXX\_list elements are built during parsing in linked list form to avoid continual “realloc” calls. After the full circuit description has been parsed and full counts are available of the number of nodes, outputs, etc., arrays of pointers (XXX\_table) are created to allow fast access to individual items during simulation. The array of hybrid indexes is created as an efficient means to call all hybrid models immediately following a successful analog timepoint solution.

The definition of the four types of info substructures used is given below:

```

typedef struct Evt_Output_Info_s {
    struct Evt_Output_Info_s *next; /* the next in the linked list */
    int node_index;                /* index into node info struct for this output */
    int output_subindex;           /* index into output data in node data struct */
    int inst_index;                /* Index of instance the port is on */
    int port_index;                /* Index of port the output corresponds to */
} Evt_Output_Info_t;

```

```

typedef struct Evt_Port_Info_s {
    struct Evt_Port_Info_s *next;      /* the next in the linked list of node info */
    int                  inst_index;   /* Index of instance the port is on */
    int                  node_index;   /* index of node the port is connected to */
    char                *node_name;    /* name of node port is connected to */
    char                *inst_name;    /* instance name */
    char                *conn_name;    /* connection name on instance */
    int                  port_num;     /* port number of instance connector */
} Evt_Port_Info_t;

typedef struct Evt_Node_Info_s {
    struct Evt_Node_Info_s *next;      /* the next in the linked list */
    char                *name;         /* Name of node in deck */
    int                  udn_index;    /* Index of the node type */
    Mif_Boolean_t        invert;       /* True if need to make inverted copy */
    int                  num_ports;    /* Number of ports connected to this node */
    int                  num_outputs;  /* Number of outputs connected to this node */
    int                  num_insts;    /* The number of insts receiving node as input */
    Evt_Inst_Index_t    *inst_list;   /* Linked list of indexes of these instances */
} Evt_Node_Info_t;

typedef struct Evt_Inst_Info_s {
    struct Evt_Inst_Info_s *next;      /* the next in the linked list */
    MIFinstance          *inst_ptr;    /* Pointer to MIFinstance struct for this instance */
} Evt_Inst_Info_t;

```

#### 5.4.3 Evt\_Queue\_t

The “queue” element of “evt” holds data about events queued during a transient analysis. In addition, this element is used during a DCop analysis for bookkeeping on what outputs and nodes are changing value as states propagate through the circuit, and what instances need to be called because their inputs are attached to nodes that have changed.

The main queue structure definition is:

```

typedef struct {
    Evt_Inst_Queue_t    inst;          /* dynamic queue for instances */
    Evt_Node_Queue_t    node;          /* dynamic queue of changing nodes */
    Evt_Output_Queue_t  output;        /* dynamic queue of delayed outputs */
} Evt_Queue_t;

```

The three subelements included are “inst”, “node”, and “output”. The “inst” and “output” elements include both bookkeeping information and queues of pending events (in a transient analysis). The “node” element is used solely for bookkeeping.

At the beginning of an analysis, all instances are marked as needing to be called. During the analysis, the code models post outputs. Since there can be multiple outputs posted to a given node, the outputs must be stored and used as input to a node-resolution algorithm

that determines the resulting node value. If the resulting node value changes from its previous value, then all instances with inputs on the node are then marked as needing to be called again. The basic sequence during analysis is shown in Figure 5.1.

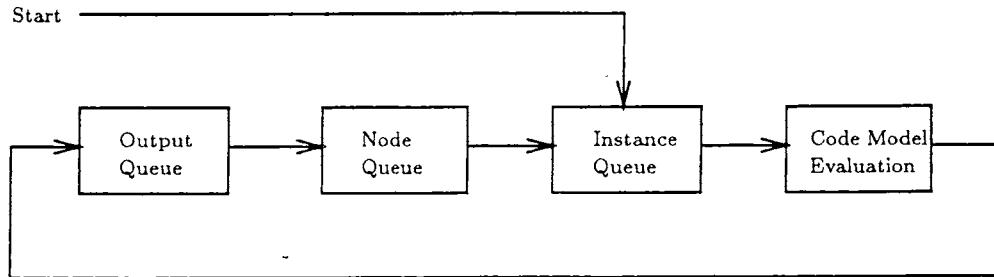


Figure 5.1 Event Evaluation Sequence.

The queues contain data necessary to implement this sequence of events in a time efficient manner. Much of the bookkeeping information included in the queues is required to support the need to "backup" during a transient analysis when the analog algorithm rejects an attempted timepoint and cuts the timestep back to a previous time. The data involved is shown below and explained in the following paragraphs.

```

/* Output queue and event structure */

typedef struct {
    Evt_Output_Event_t **head;           /* Beginning of linked lists */
    Evt_Output_Event_t ***current;       /* Beginning of pending events */
    Evt_Output_Event_t ***last_step;     /* Values of 'current' at last accepted
                                         timepoint */
    Evt_Output_Event_t **free;          /* Linked lists of items freed by backups */
    double last_time;                  /* Time at which last_step was set */
    double next_time;                  /* Earliest next event time in queue */
    int num_modified;                  /* Number modified since last accepted
                                         timepoint */
    int *modified_index;               /* Indexes of modified outputs */
    Mif_Boolean_t *modified;           /* Flags used to prevent multiple entries */
    int num_pending;                  /* Count of number of pending events in lists */
    int *pending_index;                /* Indexes of pending events */
    Mif_Boolean_t *pending;             /* Flags used to prevent multiple entries */
    int num_changed;                  /* Count of number of outputs that changed */
    int *changed_index;                /* Indexes of outputs that changed */
    Mif_Boolean_t *changed;             /* Flags used to prevent multiple entries */
} Evt_Output_Queue_t;

```

```

typedef struct Evt_Output_Event_s {
    struct Evt_Output_Event_s *next; /* the next in the linked list */
    double event_time; /* Time for this event to happen */
    double posted_time; /* Time at which event was entered in queue */
    Mif_Boolean_t removed; /* True if event has been deactivated */
    double removed_time; /* Time at which event was deactivated */
    void *value; /* The delayed value sent to this output */
} Evt_Output_Event_t;

/* Node queue */

typedef struct {
    int num_to_eval; /* Count of number of nodes that need to be
                      evaluated */
    int *to_eval_index; /* Indexes of nodes to be evaluated */
    Mif_Boolean_t *to_eval; /* Flags used to prevent multiple entries */
    int num_changed; /* Count of number of nodes that changed */
    int *changed_index; /* Indexes of nodes that changed */
    Mif_Boolean_t *changed; /* Flags used to prevent multiple entries */
} Evt_Node_Queue_t;

/* Instance queue and event structure */

typedef struct {
    Evt_Inst_Event_t **head; /* Beginning of linked lists */
    Evt_Inst_Event_t ***current; /* Beginning of pending events */
    Evt_Inst_Event_t ***last_step; /* Values of 'current' at last accepted timepoint */
    Evt_Inst_Event_t **free; /* Linked lists of items freed by backups */
    double last_time; /* Time at which last_step was set */
    double next_time; /* Earliest next event time in queue */
    int num_modified; /* Number modified since last accepted timepoint */
    int *modified_index; /* Indexes of modified instances */
    Mif_Boolean_t *modified; /* Flags used to prevent multiple entries */
    int num_pending; /* Count of number of pending events in lists */
    int *pending_index; /* Indexes of pending events */
    Mif_Boolean_t *pending; /* Flags used to prevent multiple entries */
    int num_to_call; /* Count of number of instances that need to be
                      called */
    int *to_call_index; /* Indexes of instances to be called */
    Mif_Boolean_t *to_call; /* Flags used to prevent multiple entries */
} Evt_Inst_Queue_t;

typedef struct Evt_Inst_Event_s {
    struct Evt_Inst_Event_s *next; /* the next in the linked list */
    double event_time; /* Time for this event to happen */
    double posted_time; /* Time at which event was entered in queue */
} Evt_Inst_Event_t;

```

The bookkeeping strategy adopted is based on the assumption that the circuit may contain a relatively large number of event-driven instances, but that only a small number at any given time will have pending events or will have changed or been modified since the last accepted analog timepoint. Under this assumption, it is inefficient to scan all elements to determine which need to be manipulated. Therefore, lists of the active elements are kept. Each list includes the following three types of information:

- The number of elements in the list.
- An array of indexes of the elements in the list.
- An array of flags that mark whether or not the element is already in the list, to prevent it from being put in the list multiple times.

The two arrays used in these lists are allocated to a size equal to the full count of possible instances, nodes, outputs, etc. respectively. This is done to avoid the use of the relatively slow “malloc” and “free” operations during the analysis.

During a transient analysis, events are stored in time ordered form in linked lists in the output and inst queues as they are generated. Each queue contains four arrays of linked lists:

head	Array of pointers to first element in list. NULL if no elements in list.
current	Array of pointers to pointers to pending events. Points to a NULL pointer if no pending events for that index.
last_step	Array of pointers to pointers to state of pending events at last accepted analog timepoint.
free	Array of pointers to event structures removed from the event queue during an analog analysis timepoint backup.

The array subscript corresponds to the index of the instance or output in the “info” structure of “evt”. The double indirection in the “last\_step” and “current” arrays is needed to allow for efficient splicing of data into the lists and for the possibility that there may be no pending events. A diagram of this setup is shown in Figure 5.2.

When an analog timepoint is accepted, the “last\_step” pointers are set equal to the “current” pointers. When events are pulled from the queue (at the location pointed to by “current”), “current” is advanced. When events are added to the queue, the list is scanned forward from the “current” location and the new event is spliced into the list at the appropriate point to maintain the list in a time-ordered format. If a timestep backup occurs in the analog algorithm, the list is scanned from “last\_step” and “current” is reset to the first

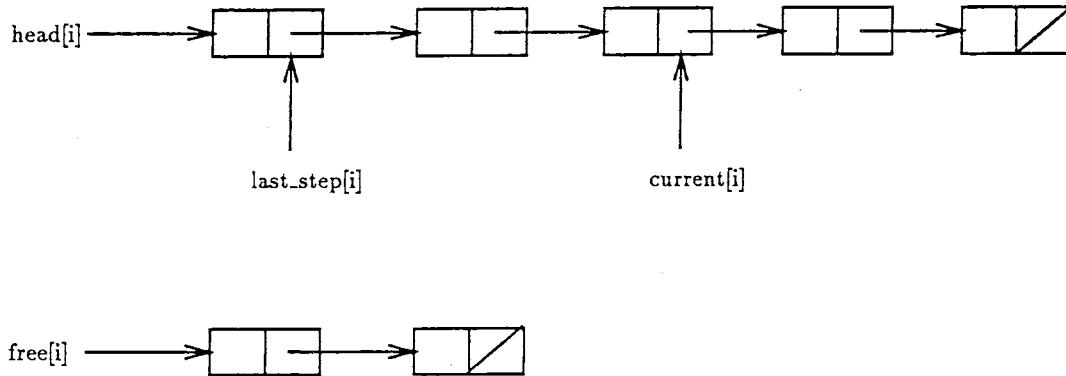


Figure 5.2 Event Queue Structure.

event with event time greater than the new analog timepoint to be attempted. Events with a posted time greater than this new analog time are pulled from the queue and put in the free list. Whenever new event structures are needed, this free list is checked first to see if any are already available before the more expensive “malloc” operation is performed.

The use of separate free lists for each index is somewhat inefficient and may be modified in a future release (Their use in the “data” element of “evt” described below is required however). In addition, the current release does not free events in the queue that are older than the last accepted timepoint, even though this data is not needed. This may also be changed in a future release to provide more efficient use of system memory during simulation.

#### 5.4.4 Evt\_Data\_t

The “data” element of “evt” holds results data generated during an analysis. The main data structure definition is:

```
typedef struct {
    Evt_Node_Data_t    *node;           /* dynamic event solution vector */
    Evt_State_Data_t   *state;          /* dynamic event instance state data */
    Evt_Msg_Data_t     *msg;            /* dynamic event message data */
    Evt_Statistic_t    *statistics;     /* Statistics for events, etc. */
} Evt_Data_t;
```

The “node”, “state”, and “msg” elements of this structure are constructed similar to the construction of the “inst” and “output” queues described above, with the exception that the “current” array used in the queues is replaced by a “tail” element which always points to the NULL pointer at the end of the list so that elements can be efficiently added to the end of the list.

The “statistics” element of this structure is used to record various data used primarily in debugging the simulator during development.

All of the elements in the “data” structure are declared as pointers and dynamically allocated. This allows the data to be efficiently moved to the “jobs” element of “evt” at the end of an analysis, so that data from the simulation run is not lost when a new run is performed. This is similar to SPICE’s capability of keeping multiple “plots” from multiple runs. However, a mechanism for accessing old job data is not currently incorporated into the simulator.

#### 5.4.4.1 Evt\_Node\_Data\_t

The definition of the elements in ckt->evt->data.node is:

```
typedef struct {
    Evt_Node_t    **head;          /* Beginning of linked lists */
    Evt_Node_t    ***tail;         /* Location of last item added to list */
    Evt_Node_t    ***last_step;    /* 'tail' at last accepted timepoint */
    Evt_Node_t    **free;          /* Linked lists of items freed by backups */
    int           num_modified;   /* Number modified since last accepted timepoint */
    int           *modified_index; /* Indexes of modified nodes */
    Mif_Boolean_t *modified;      /* Flags used to prevent multiple entries */
    Evt_Node_t    *rhs;            /* Location where model outputs are placed */
    Evt_Node_t    *rhsold;         /* Location where model inputs are retrieved */
    double        *total_load;    /* Location where total load inputs are retrieved */
} Evt_Node_Data_t;

typedef struct Evt_Node_s {
    struct Evt_Node_s *next;       /* pointer to next in linked list */
    Mif_Boolean_t    op;           /* true if computed from op analysis */
    double          step;          /* DC step or time at which data was computed */
    void            **output_value; /* Array of outputs posted to this node */
    void            *node_value;   /* Resultant computed from output values */
    void            *inverted_value; /* Inverted copy of node_value */
} Evt_Node_t;
```

The “node” element of ckt->data holds both the values of outputs and the resolved node values for all event-driven instances/nodes in the circuit. This element contains “rhs” and “rhsold” vectors similar to the vectors of SPICE of the same name.

During a DC analysis, inputs to models are taken from “rhsold” and new outputs from models are placed in “rhs”. The outputs are then subjected to the node resolution algorithm (specific to each node type), to produce new node values. The data is then copied to “rhsold” for use in future comparisons. If there are any instances with inputs on the node that are inverted, the inverted value is also computed and stored in “rhsold” before the next call to models. At the end of the DC analysis, values for all nodes are copied to “head[i]”.

During a transient analysis, outputs are delayed by a strictly non-zero value. This delay is accomplished by placing the outputs into the output queue, rather than immediately into "rhs". When the event time is reached for the output to take effect, the output is pulled from the queue and copied to "rhs". The node resolution algorithm is then performed and the resultant copied to "rhsold" as in the DC analysis. The inverted value is also computed if needed in "rhsold". A copy of this new value for the node is then created and placed in the ckt->data->node linked list at the location pointed to by "tail[i]".

The "total\_load" element of ckt->evt->data holds fanout information posted by models during the initialization pass. This data can be accessed by model outputs to simulate changes in output delay with different loads.

#### 5.4.4.2 Evt\_State\_Data\_t

The definition of the elements in ckt->evt->data.state is:

```

typedef struct {
    Evt_State_t    **head;           /* Beginning of linked lists */
    Evt_State_t    ***tail;          /* Location of last item added to list */
    Evt_State_t    ***last_step;     /* 'tail' at last accepted timepoint */
    Evt_State_t    **free;           /* Linked lists of items freed by backups */
    int            num_modified;     /* Number modified since last accepted timepoint */
    int            *modified_index;   /* List of indexes modified */
    Mif_Boolean_t  *modified;        /* Flags used to prevent multiple entries */
    int            *total_size;       /* Total bytes for all states allocated */
    Evt_State_Desc_t **desc;         /* Lists of description structures */
} Evt_State_Data_t;

typedef struct Evt_State_s {
    struct Evt_State_s  *next;        /* Pointer to next state */
    struct Evt_State_s  *prev;        /* Pointer to previous state */
    double             step;          /* Time at which state was assigned (0 for DC) */
    void               *block;         /* Block of memory holding all states on inst */
} Evt_State_t;

typedef struct Evt_State_Desc_s {
    struct Evt_State_Desc_s *next;    /* Pointer to next description */
    int                  tag;          /* Tag for this state */
    int                  size;          /* Size of this state */
    int                  offset;        /* Offset of this state into the state block */
} Evt_State_Desc_t;

```

This data is used to allow event-driven models to store data between calls and access the value the data held at the previous event time. New states are created immediately prior to calling a model during a transient analysis, and the new state is initialized to be equal to the previous state by copying the block of data held in the Evt\_State\_t structure.

The current version of the simulator does not free state data that are older than the last accepted timepoint, even though this data is not needed. This may be changed in a future release to provide more efficient use of system memory during simulation.

#### 5.4.4.3 Evt\_Msg\_Data\_t

The definition of the elements in ckt->evt->data.msg is:

```
typedef struct {
    Evt_Msg_t    **head;           /* Beginning of linked lists */
    Evt_Msg_t    ***tail;          /* Location of last item added to list */
    Evt_Msg_t    ***last_step;     /* 'tail' at last accepted timepoint */
    Evt_Msg_t    **free;           /* Linked lists of items freed by backups */
    int          num_modified;    /* Number modified since last accepted timepoint */
    int          *modified_index;  /* List of indexes modified */
    Mif_Boolean_t *modified;      /* Flags used to prevent multiple entries */
} Evt_Msg_Data_t;

typedef struct Evt_Msg_s {
    struct Evt_Msg_s    *next;      /* Pointer to next state */
    Mif_Boolean_t        op;         /* true if output from op analysis */
    double               step;       /* DC step or time at which message was output */
    char                 *text;      /* The value of the message text */
    int                  port_index; /* The index of the port from which the message came */
} Evt_Msg_t;
```

Message data is created when a model uses the "MESSAGE" macro. These messages are saved with the event-driven results data, and allow models to announce general warnings or other information such as insufficient setup time at a port, etc.

#### 5.4.4.4 Evt\_Statistic\_Data\_t

The definition of the elements in ckt->evt->data.statistics is:

```
typedef struct {
    int          op_alternations;   /* Total alternations between event and analog */
    int          op_load_calls;     /* Total load calls in DCUP analysis */
    int          op_event_passes;   /* Total passes through event iteration loop */
    int          tran_load_calls;   /* Total inst calls in transient analysis */
    int          tran_time_backups; /* Number of transient timestep cuts */
} Evt_Statistic_t;
```

This statistics data is used primarily for debugging the simulator development. This data is currently printed by the new "eprint" command added to Nutmeg that is used to display results of an event-driven simulation in tabular format.

#### 5.4.5 Evt\_Limit\_t

The “limits” element of “evt” is used to hold iteration limits applicable to the event-driven simulation algorithm. The definition of this structure is:

```
typedef struct {
    int      max_event_passes; /* max loops allowed in convergence of event nodes */
    int      max_op_alternations; /* max loops through event/analog alternation */
} Evt_Limit_t;
```

#### 5.4.6 Evt\_Job\_t

The “jobs” element of “evt” is used to hold results of multiple simulation runs. The definition of this structure is:

```
typedef struct {
    int          num_jobs;           /* Number of jobs run */
    char        **job_name;         /* Names of different jobs */
    Evt_Node_Data_t **node_data;   /* node_data for different jobs */
    Evt_State_Data_t **state_data; /* state_data for different jobs */
    Evt_Msg_Data_t **msg_data;     /* messages for different jobs */
    Evt_Statistic_t **statistics;  /* Statistics for different jobs */
} Evt_Job_t;
```

#### 5.4.7 Evt\_Option\_t

The “options” element of “evt” is used to hold options for the event-driven simulation algorithm set through a .options card. The definition of this structure is:

```
typedef struct {
    Mif_Boolean_t op_alternate;    /* Alternate analog/event solutions in OP
                                    analysis */
} Evt_Option_t;
```

### 5.5 General Enhancements Data

A general enhancements (“enh”) structure is added to the SPICE 3C1 CKTcircuit structure. The data in the “enh” structure is used for various purposes including:

- Convergence error reporting to assist debugging.
- Adding temporary breakpoints in a transient analysis.

- Adding supply ramping in a transient analysis
- Adding simple limiting to inputs of code models to assist in convergence.
- Adding resistors to ground at all nodes to prevent singular matrix problems in circuits with no DC path to ground.

The definition of the structure holding the enhancements data inside CKTcircuit is shown below.

```

typedef struct {
    Enh_Bkpt_t      breakpoint; /* Data used by dynamic breakpoints */
    Enh_Ramp_t       ramp;      /* New options added to simulator */
    Enh_Conv_Debug_t conv_debug; /* Convergence debug info dumping data */
    Enh_Conv_Limit_t conv_limit; /* Convergence limiting info */
    Enh_Rshunt_t     rshunt_data; /* Shunt conductance from nodes to ground */
} Enh_Ckt_Data_t;

typedef struct {
    double          current;   /* The current dynamic breakpoint time */
    double          last;      /* The last used dynamic breakpoint time */
} Enh_Bkpt_t;

typedef struct {
    double          ramptime;  /* supply ramping time specified on .options */
} Enh_Ramp_t;

typedef struct {
    Mif_Boolean_t   last_NIiter_call; /* True if this is the last call to NIiter() */
    Mif_Boolean_t   report_conv_probs; /* True if conv test functions should send debug
                                         info */
} Enh_Conv_Debug_t;

typedef struct {
    Mif_Boolean_t   enabled;    /* True if convergence limiting enabled on code
                                 models */
    double          abs_step;   /* Minimum limiting step size */
    double          step;       /* Fractional step amount */
} Enh_Conv_Limit_t;

typedef struct {
    Mif_Boolean_t   enabled;    /* True if rshunt option used */
    double          gshunt;     /* 1.0 / rshunt */
    int             num_nodes;  /* Number of nodes in matrix */
    double          **diag;     /* Pointers to matrix diagonals */
} Enh_Rshunt_t;

```

## 5.6 Global Data Structures

The following global data are added to SPICE 3C1. Global data is used primarily to avoid the need to change function argument lists in the SPICE 3C1 core.

### 5.6.1 User-Defined Node Data

Two new global variables are introduced by the SIM-EVT CSC to allow access to functions that define user-defined node data manipulations. These variables are:

```
int g_evt_num_udn_types;  
Evt_Udn_Info_t *g_evt_udn_info[];
```

The first variable gives the number of user-defined node types known to the simulator. The second is an array of pointers to structures holding user-defined node function pointers. These variables are declared and initialized by code in SPIinit.c when a new simulator is built, similar to the declaration and initialization of the SPICE “DEVices” array and “DEVmaxnum” variables that define what models are known to the simulator.

The definition of Evt\_Udn\_Info\_t is given in a previous section.

### 5.6.2 Model Interface Data

A new global data structure named “g\_mif\_info” is introduced by the SIM-MIF CSC to allow data to be passed to and from modified functions of the SPICE 3C1 simulator, without modifying their argument lists. This structure holds information such as the current state of the simulation, pointers to the current instance being evaluated and the circuit structure, an error message returned from a code model support function, and the current dynamic breakpoint time.

The definition of the typedef for g\_mif\_info is shown below.

```
typedef struct {  
    Mif_Circuit_Info_t circuit; /* Circuit data that will be needed by MIFload */  
    MIFinstance *instance; /* Current instance struct */  
    CKTcircuit *ckt; /* The ckt struct for the circuit */  
    char *errormsg; /* An error msg from a cm.... function */  
    Mif_Bkpt_Info_t breakpoint; /* Data used by dynamic breakpoints */  
} Mif_Info_t;  
  
typedef struct {  
    Mif_Boolean_t init; /* TRUE if first call to model */  
    Mif_Boolean_t anal_init; /* TRUE if first call for this analysis type */  
    Mif_Analysis_t anal_type; /* The type of analysis being performed */  
}
```

```

    Mif_Call_Type_t call_type; /* Type of call to code model - analog or event-driven */
    double          evt_step;   /* The current DC step or time in event analysis */
} Mif_Circuit_Info_t;

typedef struct {
    double          current;  /* The current dynamic breakpoint time */
    double          last;     /* The last used dynamic breakpoint time */
} Mif_Bkpt_Info_t;

```

### 5.6.3 Interprocess Communication Data

A new global data structure named “g\_ipc” is introduced by the SIM-IPC CSC to allow data to be passed to and from modified functions of the SPICE 3C1 simulator, without modifying their argument lists. This structure holds information such as whether interprocess communication is enabled by the -ipc command-line switch, the interprocess communication mode (interactive or batch), and configuration information stripped from the SPICE deck received by the SIM-IPC CSC functions.

The definition of the typedef for g\_ipc is shown below.

```

typedef struct {
    Ipc_Boolean_t  enabled;      /* True if we are using IPC */
    Ipc_Mode_t     mode;        /* INTERACTIVE or BATCH mode */
    Ipc_Anal_t    anal_type;   /* DCOP, AC, ... mode */
    Ipc_Boolean_t  syntax_error; /* True if error occurred during parsing */
    Ipc_Boolean_t  run_error;   /* True if error occurred during simulation */
    Ipc_Boolean_t  errchk_sent; /* True if #ERRCHK has been sent */
    Ipc_Boolean_t  returni;     /* True if simulator should return currents */
    double         mintime;     /* Minimum time between timepoints returned */
    double         last_time;   /* Last timepoint returned */
    double         cpu_time;    /* CPU time used during simulation */
    Ipc_Boolean_t  *send;       /* Used by OUTinterface to determine what to send */
    char           *log_file;   /* Path to write log file */
    Ipc_Vtrans_t   vtrans;      /* Used by OUTinterface to translate v sources */
    Ipc_Boolean_t  stop_analysis; /* True if analysis should be terminated */
} Ipc_Tiein_t;

```

# 6

# CSCI Data Files

## 6.1 Data File to CSC/CSU Cross Reference

Table 6.1 below provides a summary of the data files and the CSCs/CSUs which access them:

## 6.2 Standard Streams

The standard error stream (stderr) and standard output stream (stdout) are redirected to files:

```
/usr/tmp/atesse_xspice.out  
/usr/tmp/atesse_xspice.err
```

File	CSC/CSU
Standard Streams	INP FTE All with printf()
SPICE Rawfile	FTE/runcoms.c/dosim() FTE/OUTinterface.c
IPC Channel	IPC/IPC.c
Batch Results	IPC/IPC.c

Table 6.1 XSPICE Files.

when interprocess communication is enabled by the -ipc command line option. This redirection is required to support sending of error messages and other data normally written to a user terminal to the ATESSE Simulator Interface process. There are a significantly large number of places in the SPICE 3C1 simulator where information is written directly with printf() statements. If the error and output streams are not redirected, these statements would require source code modifications.

The SIM-IPC CSC rewinds the redirected streams and sends their contents over the interprocess communication channel. This is done through calls to function ipc\_send\_std\_files().

### **6.3 SPICE Rawfile**

SPICE 3C1 normally writes results to a “rawfile” named rawspice when running in batch mode. When interprocess communication is enabled, this rawfile is not created since the simulator sends results over the interprocess communication channel.

### **6.4 Interprocess Communication Channel File**

The interprocess communication mechanism requires either an HP/Apollo Aegis “mailbox” file, or a BSD UNIX “socket” file to send and receive data from other processes. The pathname to use for this file is determined at runtime according to the second argument to the -ipc command-line switch.

### **6.5 Batch Results File**

The interprocess communication mechanism writes results to a file instead of sending them over the interprocess communication channel when the simulator is running in BATCH mode (specified as the first argument to the -ipc command-line switch). The pathname to use for this file is determined at runtime from the first message received over the interprocess communication channel.

## 7

# Requirements Traceability

Requirements governing the design of the ATESSE XSPICE simulator 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 XSPICE Simulator to requirements. Note that requirements under sections 3.2.2.4.3.3 and 3.2.2.4.4 are covered by the XSPICE Code Model Subsystem as documented in the Software Design Document for the XSPICE Code Model Subsystem of the Automatic Test Equipment Software Support Environment (ATESSE).

Project Unique Identifier	SRS Requirements Section
SPICE3C1 Core	3.2.1
SIM-MIF	3.1.7 3.2.2 3.2.2.4.3 3.4.1 3.4.4
SIM-CM	3.2.2 3.2.2.4.3 3.4.7
SIM-EVT	3.2.2 3.2.2.4.2.6 3.2.2.4.2.7
SIM-ENH	3.2.2 3.2.2.4.1 3.2.2.4.2
SIM-ICM	3.2.2 3.2.2.4.1 3.2.2.4.1.3
SIM-IDN	3.2.2 3.2.2.4.2.6
SIM-IPC	3.1 3.2.2 3.2.2.4.1 3.2.2.4.1.1 3.2.2.4.1.2 3.4

Table 7.1 XSPICE Requirements Cross-Reference.

# 8

# Notes

## 8.1 Glossary

<b>ATESSE</b>	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.
<b>Code Model Library</b>	The set of code models supplied with the ATESSE simulator. This library is a component of the Code Model Subsystem CSCI.
<b>Code Model Preprocessor</b>	A code model development tool that assists in adding new code models to the ATESSE 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.
<b>Code Model Subsystem</b>	A set of tools and predefined code models included with the ATESSE simulator.
<b>Code Model Toolset</b>	A set of tools that assist in adding new code models to the ATESSE simulator.
<b>Interprocess Communication</b>	Communication between two separate processes running concurrently on one or more computers.
<b>lvalue</b>	C Language terminology for something that can be assigned a value.
<b>Make</b>	A UNIX software development tool that automates the compilation and linking of a program.

<b>Makefile</b>	A file that instructs the UNIX make utility how to compile and link a program.
<b>malloc</b>	A standard library function for dynamically allocating memory in a program.
<b>Parser</b>	A program or procedure that reads a description in a programming language and interprets the meaning of the description.
<b>Rawfile</b>	The standard output file from the SPICE 3C1 simulator which holds results computed during a simulation.
<b>SPICE</b>	An analog simulation program developed at the University of California at Berkeley.
<b>XXXinfo</b>	A structure holding parsing information for a device or model known to the simulator.
<b>XXXinstance</b>	A dynamically allocated structure holding information about an instance of a circuit.
<b>XXXmodel</b>	A dynamically allocated structure holding information about a model of a circuit.

## 8.2 Acronyms

<b>ANSI</b>	American National Standards Institute
<b>ATESSE</b>	Automatic Test Equipment Software Support Environment
<b>CDRL</b>	Contract Deliverable Requirement List
<b>CM</b>	Code Model
<b>CSC</b>	Computer Software Component
<b>CSCI</b>	Computer Software Configuration Item
<b>CSU</b>	Computer Software Unit
<b>ENH</b>	Enhancements CSC
<b>EVT</b>	Event-Driven Simulation CSC
<b>ICM</b>	Internal Code Models CSC
<b>IPC</b>	Interprocess Communication CSC
<b>MIF</b>	Model Interface CSC
<b>MNA</b>	Modified Nodal Analysis
<b>SPICE</b>	Simulation Program with Integrated Circuit Emphasis
<b>SPICE2G6</b>	Version 2G6 of the Berkeley SPICE simulator program
<b>SPICE3C1</b>	Version 3C1 of the Berkeley SPICE simulator program
<b>SRS</b>	Software Requirement Specification
<b>UNIX</b>	UNIX Operating System
<b>X11</b>	X Window System Version 11

### 8.3 Project Unique Identifiers

<b>BCP</b>	ATESSE Version 2 Batch Control Process
<b>BCP1</b>	ATESSE Version 1 Batch Control Process
<b>BCPDB</b>	ATESSE Version 2 Batch Control Process Data Base
<b>BCPDB1</b>	ATESSE Version 1 Batch Control Process Data Base
<b>CM</b>	ATESSE Code Model Subsystem CSC
<b>SI</b>	ATESSE Version 2 Simulator Interface process
<b>SI-SIM-IN</b>	Output to the ATESSE Version 2 Simulator Interface process
<b>SI-SIM-OUT</b>	Input from the ATESSE Version 2 Simulator Interface process
<b>SI1</b>	ATESSE Version 1 Simulator Interface process
<b>SI1-SIM-IN</b>	Output to the ATESSE Version 1 Simulator Interface process
<b>SI1-SIM-OUT</b>	Input from the ATESSE Version 1 Simulator Interface process
<b>SIM</b>	ATESSE Simulator CSC
<b>SIM-BCP-IN</b>	Input from the ATESSE Version 2 Batch Control Process
<b>SIM-BCP-OUT</b>	Output to the ATESSE Version 2 Batch Control Process
<b>SIM-BCP1-IN</b>	Input from the ATESSE Version 1 Batch Control Process
<b>SIM-BCP1-OUT</b>	Output to the ATESSE Version 1 Batch Control Process
<b>SIM-BCPDB-OUT</b>	Output to the ATESSE Version 2 Batch Control Process Database
<b>SIM-BCPDB1-OUT</b>	Output to the ATESSE Version 1 Batch Control Process Database
<b>SIM-CM-IN</b>	Input from the Code Model Subsystem
<b>SIM-CM</b>	Code Model support CSC
<b>SIM-EVT</b>	Event-Driven Simulator CSC
<b>SIM-ENH</b>	Enhancements CSC
<b>SIM-ICM</b>	Internal Code Models CSC
<b>SIM-IDN</b>	Internally Defined Nodes CSC

XSPICE Simulator  
Software Design Document

Notes  
Project Unique Identifiers

SIM-IPC	Interprocess Communication CSC
SIM-MIF	Model Interface CSC

## APPENDICES

# A

## SPICE 3C1 Annotated Call Tree

The documentation available for the 3C1 level of SPICE is relatively high level and incomplete. In addition, the code contains very few comments. Therefore, an analysis of the simulator code was done prior to beginning modifications. The major architecture of the simulator as determined from this analysis has been discussed in Section 3 of this document, including a simplified call tree of the code.

This appendix and the two that follow it cover some of the analysis in greater depth. Appendix A gives a more detailed call tree of the simulator together with annotations and the names of the source files in which the functions are found. Appendix B gives an overview of some of the most important type definitions used. Appendix C provides notes on the actual variables that instantiate these type definitions.

To get maximum benefit from this material, the reader should have the source code available. This documentation covers the SPICE 3C1 release only.

The annotated call tree appears below.

```
main() @ FTE/nutmeg.c or FTE/bspice.c
SIMinit() @ FTE/spiceif.c : just call SPIinit()
process command line args
DEVInit() @ FTE/display.c : get attributes/fcn ptrs for different graphics
initialize menu display
open input file
inp_spsource() @ FTE/inp.c : read entire deck and parse it
read spiceinit file, news file, etc.
switch(mode)
batch
    ft_dotsave() @ FTE/dotcards.c : gets .save cards and does ? with them
    ft_dorun()   @ FTE/runcoms.c : do the simulation
    ft_cktcoms() @ FTE/dotcards.c : print listing, output, and accounting
```

```

menu
  calls routines like initscanner(), parse(), driver(), etc.
  some of which do not seem to be in the spice directories
interactive
  cp_evloop() @ CP/front.c : command line interpreter

SIMinit() @ FTE/spiceif.c : just call SPIinit()
SPIinit() @ CKT/SPIinit.c : Nutmeg and spice exchange fcn ptrs

inp_spsource() @ FTE/inp.c : read entire deck and parse it
  inp_readall() @ FTE/inp.c : read lines of input into deck structure
  inp_getopts() @ FTE/options.c : separate out options cards
  separate out control cards
  inp_subcktexpand() @ FTE/subckt.c : expand subcircuits
  inp_dodeck() @ FTE/inp.c : parse deck and setup circuit data structs
  cp_evloop() @ CP/front.c : process the .plot, etc. control commands

inp_dodeck() @ FTE/inp.c : parse deck and setup circuit data structs
  discard old error messages in deck
  if_inpdeck() @ FTE/spiceif.c : parse deck and setup circuit data structs
  out_init() @ CP/output.c : initialize 'more' type output ???
  print error messages

if_inpdeck() @ FTE/spiceif.c : parse deck and setup circuit data structs
  CKTinit() @ CKT/CKTinit.c : malloc and initialize 'CKTcircuit' structure
  CKTnewTask() @ CKT/CKTnewTask.c : malloc and init new task structure
  CKTnewAnal() @ CKT/CKTnewAnal.c : malloc and init new analysis struct
  INPpas1() @ INP/INPpas1.c : process .model cards and create structs
  INPpas2() @ INP/INPpas2.c : parse rest of deck and create structs
  INPkillMods() @ INP/INPkillMods.c : free temporary model structs

INPpas1() @ INP/INPpas1.c : process .model cards and create structs
  INPdomodel() @ INP/INPdomodel.c : parse .model card and make temp struct

INPpas2() @ INP/INPpas2.c : parse rest of deck and create structs
  INP2R() @ INP/INP2R.c : parse R card and make instance struct
  INP2C() @ INP/INP2C.c : parse C card and make instance struct
  .

ft_dorun() @ FTE/runcoms.c : just call dosim("run")
  dosim() @ FTE/runcoms.c : open rawfile, do simulation, and close rawfile

```

```
dosim() @ FTE/runcoms.c : open rawfile, do simulation, and close rawfile
    open rawfile for output
    if_run() @ FTE/spiceif.c : do the simulation
    close rawfile

if_run() @ FTE/spiceif.c : do the simulation by calling spice function
    (*(ft_sim->doAnalyses)) : ft_sim == SIMinfo @ CKT/SPIinit.c ==
        CKTdoJob() @ CKT/CKTdoJob.c : run requested analysis types

CKTdoJob() @ CKT/CKTdoJob.c : run requested analysis types
    initialize simulation data
    CKTsetup() @ CKT/CKTsetup.c : call device setup routines
    CKTtemp() @ CKT/CKTtemp.c : call device temp routines
    CKTic() @ CKT/CKTic.c : do .nodeset initializations
    DCop() @ CKT/DCop.c : run DC operating point analysis
    DCtrCurv() @ CKT/DCtrCurv.c : run DC operating point analysis
    ACan() @ CKT/ACan.c : run Swept AC analysis
    DCtran() @ CKT/DCtran.c : run transient analysis

DCop @ CKT/DCop.c : run DC operating point analysis
    OUTpBeginPlot @ FTE/OUTinterface.c : initialize saving of results
    CKTnames() @ CKT/CKTnames.c : malloc and init node/equation name list
    CKTop() @ CKT/CKTop.c : run an operating point analysis
    CKTdump() @ CKT/CKTdump.c : output results to rawfile or to memory
    OUTendPlot @ FTE/OUTinterface.c : terminate saving of results

CKTop() @ CKT/CKTop.c : run an operating point analysis
    NIiter() @ NI/NIiter.c : iterate to a solution
    if not converged, try gmin stepping
    else try source stepping

CKTdump() @ CKT/CKTdump.c : output results to rawfile or to memory
    OUTpData() @ FTE/OUTinterface.c : output a vector at current anal point

    OUTpData() @ FTE/OUTinterface.c : output a vector at current anal point
    fileStartPoint() @ FTE/OUTinterface.c : output anal point info
    fileAddRealValue() @ FTE/OUTinterface.c : output a single number
    fileEndPoint() @ FTE/OUTinterface.c : nop

cp_evloop() @ CP/front.c : command line interpreter
    doblock() @ CP/front.c : process a block of statements
    docommand() @ CP/front.c : process a single statement (e.g. run)
        (*cp_coms[i].cofunc) @ FTE/cmtdtab.c : do a command ==
            com_run() @ FTE/runcoms.c : run a simulation
            com_plot() @ FTE/doplot.c : plot results
            com_print() @ FTE/postcoms.c : plot results
            ...

```

# B

## SPICE 3C1 Structure Typedefs

This appendix describes a number of important SPICE 3C1 type definitions. The definitions are grouped according to the following areas:

- Nutmeg
- SPICE
- Model/Instance
- Input Parser

### B.1 Nutmeg Type Definitions

See also the SPICE 3C1 document titled Front End to Simulator Interface.

IFfrontEnd @ include/IFsim.h

Pointers to front end functions used by simulator.

IFsimulator @ include/IFsim.h (See also: CKT/SPIinit.c )

Name and description of simulator.

Pointers to simulator functions used by front end.

IFdevice Pointers to data about devices supported.

IFanalysis Pointers to data about analysis supported.

IFparm Pointers to data about node parameters supported.

IFdevice    @ include/IFsim.h

Name and description of device.  
Number of terminals and names of terminals.  
IFparm   Instance parameters applicable to this device.  
IFparm   Model parameters applicable to this device.

IFanalysis @ include/IFsim.h

Name and description of analysis type.  
IFparm   Analysis parameters applicable to this analysis.

IFparm      @ include/IFsim.h

Keyword string used by user to refer to parameter.  
Unique integer identifier used by simulator.  
Parameter operations allowed (e.g. set, query, etc.)  
Text description of parameter used for help.

IFvalue     @ include/IFsim.h

A union of different value types, including vectors.

IFuid       @ include/IFsim.h

A GENERIC pointer used for quick equality checks to  
avoid doing string compares.

circ   @ include/FTEdefs.h

Pointer to CKTcircuit structure  
Pointer to INP symbol table  
Pointer to deck structures  
Pointer to options cards  
Pointer to various commands for the circuit  
Pointer to circuit nodes??  
Pointer to devices in circuit??  
Pointer to task for this circuit  
Pointers to other stuff...

struct comm   @ include/CPdefs.h

Data used by docommand() to lookup command  
strings and call an appropriate function.

## B.2 SPICE Specific Type Definitions

See also the SPICE 3C1 document titled The Spice3 Implementation Guide.

CKTcircuit @ include/CKTdefs.h

GENmodel \*CKTHead Beginning of linked list circuit structure.  
STATistics \*CKTstat Statistics on time spent in various functions.  
Arrays holding states for individual instances.  
Current analysis time, delta, etc.  
Pointer to sparse matrix structure  
Arrays of CKTRhs vectors  
CKTnode Pointer to linked list of structs describing equations.  
Numerous constants, etc.  
JOB \*CKTcurJob

CKTnode @ include/CKTdefs.h

IFuid Name of this equation (node/source)  
Type of equation (NODE\_VOLTAGE or NODE\_CURRENT)  
Equation number  
Initialization data (nodeset and ic)  
Pointer to next entry in linked list.

STATistics @ include/OPTdefs.h

Various counts and time statistics on simulation execution.

SPICEanalysis @ include/JOBdefs.h

IFanalysis Parameters applicable to this type of analysis.  
Parameter set and query function pointers for use by front end.

TSKtask @ include/TSKdefs.h

JOB \*jobs List of analysis to run.  
Various data common to all jobs.

JOB @ include/JOBdefs.h

A standard prefix for different analysis job structures including:

ACAN @ include/ACdefs.h  
OP @ include/OPdefs.h  
PZAN @ include/PZdefs.h  
TFan @ include/TFdefs.h  
TRANan @ include/TRANdefs.h

```
TRCV  @ include/TRCVdefs.h
```

### B.3 Model/Instance Type Definitions

See also the SPICE 3C1 document titled [The Spice3 Implementation Guide.](#)

```
GENmodel  @ include/GENdefs.h
```

A standard prefix for different device model structures. Items in this prefix are the only items visible to the front end and to simulator functions that are not specific to this device.

See:

```
include/ASRCdefs.h
include/BJTdefs.h
include/BSIMdefs.h
include/CAPdefs.h
include/CCCSdefs.h
include/CCVSdefs.h
include/CSWdefs.h
include/DEVdefs.h
include/DIOdefs.h
include/INDdefs.h
include/ISRCdefs.h
include/JFETdefs.h
include/MESdefs.h
include/MOS1defs.h
include/MOS2defs.h
include/MOS3defs.h
include/RESdefs.h
include/SWdefs.h
include/TRAdefs.h
include/URCdefs.h
include/VCCSdefs.h
include/VCVSdefs.h
include/VSRCdefs.h
```

```
GENinstance  @ include/GENdefs.h
```

A standard prefix for different device instance structures. Items in this prefix are the only items visible to the front end and to simulator functions that are not specific to this device.

See:

```
include/ASRCdefs.h
include/BJTdefs.h
include/BSIMdefs.h
include/CAPdefs.h
include/CCCSdefs.h
include/CCVSdefs.h
include/CSWdefs.h
include/DEVdefs.h
include/DIOdefs.h
include/INDdefs.h
include/ISRCdefs.h
include/JFETdefs.h
include/MESdefs.h
include/MOS1defs.h
include/MOS2defs.h
include/MOS3defs.h
include/RESdefs.h
include/SWdefs.h
include/TRAdefs.h
include/URCdefs.h
include/VCCSdefs.h
include/VCVSdefs.h
include/VSRCdefs.h
```

SPICEdev @ include/DEVdefs.h

IFdevice DEVpublic Tells front end about parms, etc. for this device.  
Pointers to functions that manipulate this device's data.  
The sizes of dynamically allocated instance and model data structures.

See:

```
DEV/ASRC/ASRC.c
DEV/BJT/BJT.c
DEV/BSIM/BSIM.c
DEV/CAP/CAP.c
DEV/CCCS/CCCS.c
DEV/CCVS/CCVS.c
DEV/CSW/CSW.c
DEV/DIO/DIO.c
DEV/IND/IND.c
DEV/ISRC/ISRC.c
DEV/JFET/JFET.c
DEV/MES/MES.c
DEV/MOS1/MOS1.c
DEV/MOS2/MOS2.c
DEV/MOS3/MOS3.c
DEV/RES/RES.c
DEV/SW/SW.c
DEV/TRA/TRA.c
DEV/URC/URC.c
```

DEV/VCCS/VCCS.c  
DEV/VCVS/VCVS.c  
DEV/VSRC/VSRC.c  
DEV/VSRC/VSRCtemp.c

#### B.4 Input Parser Type Definitions

See also the SPICE 3C1 document titled The Spice3 Implementation Guide.

card \* include/INPdefs.h (Same as line \* include/FTEinp.h )

Line number.  
Strings holding line text in logical and/or physical format.  
Pointer to next card in list.

INPtables \* include/INPdefs.h

Hash tables of names.  
Pointers to default model structures.

INPmodel \* include/INPdefs.h

Used to save model data during parsing.

INPparseTree \* include/INPparseTree.h

Used to hold parse tree for expressions.

# C

## SPICE 3C1 Structure Declarations

This appendix describes a number of important SPICE 3C1 structure definitions. These definitions are grouped according to the following areas:

- Nutmeg
- SPICE

### C.1 Nutmeg Structure Declarations

```
IFsimulator *ft_sim  @ FTE/bspice.c or FTE/nutmeg.c
```

The main simulator data structure visible to the front end. Set to SIMinfo by a call to SIMinit() in main().

```
static IFsimulator SIMinfo  @ CKT/SPIinit.c

CKTxxxx() function pointers
(IFdevice **) DEVices  @ CKT/SPIinit.c
SPICEdev RESinfo  @ DEV/RES/RES.c
    char *RESnames[]  @ DEV/RES/RES.c
    IFparm RESpTable[] @ DEV/RES/RES.c
    IFparm RESmPTable[] @ DEV/RES/RES.c
    RESxxxx function pointers
(IFanalysis **) analInfo  @ CKT/SPIinit.c
    SPICEanalysis TRANinfo @ CKT/TRANsetParm.c
        IFparm TRANparms @ CKT/TRANsetParm.c
    IFparm nodeParms  @ CKT/SPIinit.c
    char *specSigList[] @ CKT/SPIinit.c
```

```
struct comm cp_coms[] @ FTE/cmdtab.c
```

An array of commands accepted by the Nutmeg user interface and pointers to their functions.

```
struct circ *ft_curckt @ FTE/circuits.c or FTE/bspice.c
```

The currently active circuit. Allocated in inp\_dodeck(). This structure is global and is manipulated by many functions in FTE. At the point of function if\_run() in the execution tree, pointers to the CKTcircuit and TSKtask data structures in circ are broken out and passed to the simulator via a call to (\*(ft\_sim->doAnalyses)) == CKTdoJob(). The subtree CKTdoJob() does not see the circ structure, only the CKTcircuit and TSKtask structures pointed to by the arguments to CKTdoJob().

```
struct circ *ft_circuits @ FTE/circuits.c or FTE/bspice.c
```

A global linked list of multiple circ structures maintained by the front end.

## C.2 SPICE Structure Declarations

```
IFfrontEnd *SPfrontEnd @ CKT/SPIinit.c
```

The main front end data structure containing pointers to front end functions that must be called by the simulator as it runs. Set to Nutmeginfo by a call to SIMinit() in main().

```
static IFfrontEnd Nutmeginfo @ FTE/bspice.c or FTE/nutmeg.c
```

IFnewUid() function pointer.  
seconds() function pointer.  
OUTxxxx() function pointers.

```
CKTcircuit *ckt @ most functions in CKT and DEV packages
```

The current circuit begin simulated. The ckt pointer is passed up and down the subtree of simulator functions. It behaves as a ‘global’ data structure for exchanging information between the various simulator modules.

```
GENmodel *inModel  @  most functions in DEV
```

The head of a linked list of model structures in the current circuit. This pointer has its origin in the ckt structure. Each model structure in the list points to linked lists of instances of the corresponding model type. When various CKT package routines operate on the circuit, they loop through all the model structures in the inModel list, and all of the instance structures inside each model structure in order to operate on all the components in the circuit. Routines such as XXXload() that operate on specific device types will cast the inModel pointer into a pointer for the particular devices structure (e.g. RESmodel) allowing access to information in the model structure that is private to that device type.