THE SPICE3 IMPLEMENTATION GUIDE

by

Thomas L. Quarles

Memorandum No. UCB/ERL M89/44

24 April 1989

THE SPICE3 IMPLEMENTATION GUIDE

by

Thomas L. Quarles

Memorandum No. UCB/ERL M89/44

24 April 1989

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

Preface

This memo is one of six containing the text of the Ph.D. dissertation *Analysis of Performance and Convergence Issues for Circuit Simulation.* The dissertation itself is available as UCB/ERL Memorandum M89/42. The other appendices are available as:

Memo number	Title
UCB/ERL M89/43	The Front End to Simulator Interface
UCB/ERL M89/45	Adding Devices to SPICE3
UCB/ERL M89/46	SPICE3 Version 3C1 Users Guide
UCB/ERL M89/47	Benchmark Circuits: Results for SPICE3

This memo was originally Appendices B and C of the dissertation and provides a detailed description of the data structures and subroutines which make up the SPICE3 program. This version describes the internal details of SPICE3, version 3C1.

Table of Contents

CHAPTER 1

Data structures

This section provides a detailed look at the data structures used throughout SPICE3. Knowledge of these structures at this level of detail should be necessary only when making changes to the packages using them. For all other purposes, the overview provided in the program architecture chapter should be sufficient. This section assumes a familiarity with that overview and presents only the details.

1.1. Interface structures

The structures used in the interface between SPICE3 and its front end, all those whose name starts with IF, are described in the front end to simulator interface specification appendix rather than in this appendix. These structures and the interface they are a part of apply to more than just SPICE3, and thus their description was separated out into a separate document.

1.2. Circuit structures

The circuit package contains structures which describe a circuit as a whole, but not specifics of any given device or analysis.

1.2.1. The CKTcircuit structure

The CKTcircuit structure is the closest thing SPICE3 has to global data. Due to the decision to make SPICE3 handle multiple circuits completely independently, actual global data would provide an unwanted connection between circuits, so all such data has been moved into this structure. As such, this structure has become a sort of "catch-all" for global data values as well as data for algorithms or packages which don't have enough data to have been worth generating a private data structure for them yet. Thus, this structure is both the most widely known and one of the most frequently chang-

```
typedef struct {                    /* data structure to describe an entire circuit */
    GENmodel *CKThead[MAXNUMDEVS];
    STATistics *CKTstat;
    double *(CKTstates[8]);
#define CKTstate0 CKTstates[0]
#define CKTstate1 CKTstates[1]
#define CKTstate2 CKTstates[2]
#define CKTstate3 CKTstates[3]
#define CKTstate4 CKTstates[4]
#define CKTstate5 CKTstates[5]
#define CKTstate6 CKTstates[6]
#define CKTstate7 CKTstates[7]
    double CKTtime;
    double CKTdelta;
    double CKTdeltaOld[7];
    double CKTtemp;
    double CKTnomTemp;
    double CKTvt;
    double CKTag[7];                /* the gear variable timestep matrix */
    int CKTorder;                   /* the integration method order */
    int CKTmaxOrder;                /* maximum integration method order */

    SMPmatrix *CKTmatrix;           /* pointer to sparse matrix */
    int CKTniState;                 /* internal state */
    double *CKTrhs;                 /* current rhs value - being loaded */
    double *CKTrhsOld;              /* previous rhs value for convergence testing */
    double *CKTrhsSpare;            /* spare rhs value for reordering */
    double *CKTirhs;                /* current rhs value - being loaded (imag) */
    double *CKTirhsOld;             /* previous rhs value (imaginary)*/
    double *CKTirhsSpare;           /* spare rhs value (imaginary)*/

    double *CKTrhsOp;               /* operating point values */
    double *CKTsenRhs;              /* current sensitivity rhs  values */
    double *CKTseniRhs;             /* current sensitivity rhs  values (imag)*/


/*
 *  symbolic constants for CKTniState
 *      Note that they are bitwise disjoint
 */

#define NISHOULDREORDER 0x1
#define NIREORDERED 0x2
#define NIUNINITIALIZED 0x4
#define NIACSHOULDREORDER 0x10
#define NIACREORDERED 0x20
#define NIACUNINITIALIZED 0x40
#define NIDIDPREORDER 0x100
#define NIPZSHOULDREORDER 0x200
```

```
    int CKTintegrateMethod;         /* the integration method to be used */

/* known integration methods */
#define TRAPEZOIDAL 1
#define GEAR 2

    char *CKTtitle;                 /* the job title */
    int CKTmaxEqNum;                /* the largest circuit equation defined */
    int CKTcurrentAnalysis;         /* the analysis in progress (if any) */

/* defines for the value of  CKTcurrentAnalysis */
#define DOING_DCOP 1
#define DOING_TRCV 2
#define DOING_AC   4
#define DOING_TRAN 8

    CKTnode *CKTnodes;
    CKTnode *CKTlastNode;
#define NODENAME(ckt, nodenum) CKTnodName(ckt, nodenum)
    int CKTnumStates;
    long CKTmode;

/* defines for CKTmode */

/* old 'mode' parameters */
#define MODE 0x3
#define MODETRAN 0x1
#define MODEAC 0x2

/* old 'modedc' parameters */
#define MODEDC 0x70
#define MODEDCOP 0x10
#define MODETRANOP 0x20
#define MODEDCTRANCURVE 0x40

/* old 'initf' parameters */
#define INITF 0x3f00
#define MODEINITFLOAT 0x100
#define MODEINITJCT 0x200
#define MODEINITFIX 0x400
#define MODEINITSMSIG 0x800
#define MODEINITTRAN 0x1000
#define MODEINITPRED 0x2000

/* old 'nosolv' parameter */
#define MODEUIC 0x10000l

    int CKTbypass;
    int CKTdcMaxIter;               /* iteration limit for dc op.  (itl1) */
    int CKTdcTrcvMaxIter;           /* iteration limit for dc tran. curv (itl2) */
    int CKTtranMaxIter;             /* iteration limit for each timepoint for tran*/
                                    /* (itl4) */
    int CKTbreakSize;
```

```
  int CKTbreak;
  double CKTsaveDelta;
  double CKTminBreak;
  double *CKTbreaks;
  double CKTabstol;
  double CKTpivotAbsTol;
  double CKTpivotRelTol;
  double CKTreltol;
  double CKTchgtol;
  double CKTvoltTol;
  double CKTgmin;
  double CKTdelmin;
  double CKTtrtol;
  double CKTfinalTime;
  double CKTstep;
  double CKTmaxStep;
  double CKTinitTime;
  double CKTomega;
  double CKTsrcFact;
  double CKTdiagGmin;
  int CKTnumSrcSteps;
  int CKTnumGminSteps;
  int CKTnoncon;
  int CKTwantOP;
  double CKTdefaultMosL;
  double CKTdefaultMosW;
  double CKTdefaultMosAD;
  double CKTdefaultMosAS;
  unsigned int CKThadNodeset:1;
  unsigned int CKTfixLimit:1;    /* flag to indicate that the limiting of*/
                                 /* MOSFETs should be done as in SPICE2 */
  JOB *CKTcurJob;

  SENstruct *CKTsenInfo;         /* the sensitivity information */

}CKTcircuit;
```

Figure 1.1
Definition of the CKTcircuit structure

---

ing structures when non device-specific changes are made.

CKThead

This array provides pointers to the first model of each of the device types. Each of these pointers thus points to the head of a linked list of a different type of device model. The definition of this array is the only place in SPICE3 where the number of device types supported is limited, and that only by the dimension (MAXNUMDEVS) of the array.

CKTstat

This is the accounting structure used to accumulate statistics about the simulation. Statistics accumulated include timing, iteration counts, and timepoint counts. Additional statistics, if collected, should be added to this structure.

CKTstates

CKTstates and the macros CKTstate*i* for all integer values of *i* are used to store the per timepoint data. Each device will reserve portions of each of these vectors in its DEVsetup routine, and the higher level code will allocate them and manipulate them between timepoints so that vector *i* always contains data from the *i*th previous timepoint, where the 0th previous timepoint is defined as the current point.

CKTtime

CKTtime stores the current simulation time during transient analysis.

CKTdelta, CKTdeltaOld

CKTdelta contains the current timestep, while the array CKTdeltaOld contains the current timestep as its zeroth element along the 6 previous timesteps in order.

CKTtemp, CKTnomTemp

CKTtemp is the default temperature for devices in the current simulation. Any device which does not have an overriding temperature specification will operate at the temperature CKTtemp. CKTnomTemp is the default parameter measurement temperature for models, and all model parameters will be assumed to be measured at this temperature unless overridden by a Tnom specification on the model.

CKTvt

This is an obsolete value, retained here for backward compatibility and conversion purposes. This is the value of the thermal voltage at the temperature CKTtemp. In previous versions of SPICE3 which did not have per instance temperatures, this was used for all Vt calculations.

CKTag, CKTagp

These arrays contains the coefficients for the corrector and predictor, respectively, of the variable timestep integration algorithm.

CKTorder, CKTmaxOrder

CKTorder is the current integration order, and varies as the simulation progresses. CKTmaxOrder is the maximum value CKTorder may take on, either because the user or program has limited it.

CKTmatrix

This is the sparse matrix for the current circuit. Details of this structure are in the sparse matrix package structure description.

CKTniState

This is a bit vector containing status information on the numeric package. Various bits indicate such information as whether the numeric package needs to reorder the matrix, whether the package has been initialized completely or whether additional data needs to be allocated, and similar data for ac, pole-zero, and other algorithms of the numeric package.

CKTrhs, CKTrhsOld, CKTrhsSpare, CKTirhs, CKTirshOld, CKTirhsSpare

These vectors form a two by three grid of possible right hand side vectors for the matrix package. The three with names having the prefix CKTi are the imaginary parts of the right hand sides used in complex calculations, while those without the *i* are the real parts. The vectors ending in rhs are the normal right hand side to be loaded by device routines. The vectors ending in rhsOld are the right hand side vector from the previous iteration which now contain the solution for that iteration. The vectors ending in rhsSpare are extra vectors of the same size required by the SMP package for its own internal use.

CKTrhsOp, CKTsenRhs, CKTseniRhs

These values were added to the structure to support sensitivity analysis and are described in the sensitivity report[Chou88a].

CKTintegrateMethod

This indicates which integration method is to be used by SPICE3. Constants have been defined for the trapezoidal and gear methods, all other values are currently unassigned and illegal.

CKTmaxEqNum

This is the number of equations currently defined. Since the ground node is equation zero, this is always one greater than the largest equation number.

CKTcurrentAnalysis

This is an obsolete value used by the restart code to find the correct analysis to resume simulation with. Since the code has been enhanced to support more than one analysis of a given type, this variable no longer contains sufficient information, but has not yet been removed for reasons of backward compatibility.

CKTnodes, CKTlastNode

CKTnodes points to the first element in the linked list of node structures which describe the equations of the circuit. CKTlastNode points to the last node structure on the list headed by CKTnodes and is used for fast addition on the end of the list, since the list is kept sorted by node number and full list searches are not performed for reasons of efficiency.

CKTnumStates

CKTnumStates is the size of the CKTstates vectors in units of the size of a double precision variable.

CKTmode

CKTmode is a bit vector containing data on the state machine used to control analyses. The low order four bits are used to determine what type of analysis is being performed, ac, dc, or Transient. The next four bits enumerate the types of dc analyses that may be in progress, operating point, transfer curve, or transient operating point. The next 8 are the state of the NIiter state machine, and control the source of the voltages seen by the devices in the load routines. Finally, the next bit indicates the presence of the UIC flag on the transient command and

the forced use of the initial conditions given without a full operating point analysis.

CKTbypass

This is a flag which has a non-zero value if inactive device bypass is to be permitted to occur in devices if the code to do so has been compiled in.

CKTdcMaxIter, CKTtrcvMaxIter, CKTtranMaxIter

These three values determine the maximum number of iterations to be expended respectively on the computation of an operating point before giving up, the computation of any one point in a dc transfer curve analysis before giving up, and the computation of a single timepoint in a transient analysis before giving up and reducing the timestep.

CKTbreakSize, CKTbreaks

CKTbreaks is the actual breakpoint table, and consists of an allocated array of doubles containing all of the current breakpoints, including at least one and possibly two previous points. CKTbreakSize is the actual size of the table in units of the size of a double precision value.

CKTsaveDelta

This is the saved timestep from before the step was cut to hit a breakpoint. An effort is made to return the analysis to this stepsize immediately after the breakpoint to minimize the disruption caused by the breakpoint.

CKTminBreak

This is the minimum amount of time by which two potential breakpoints must be separated in order to be considered distinct. Whenever two breakpoints are closer together than this amount the later of the two is discarded.

CKTabstol, CKTchgtol, CKTvoltTol, CKTpivotAbsTol

These four tolerances are the absolute error tolerances on, respectively, current, charge, voltage, and the minimum acceptable pivot element in the matrix.

CKTreltol, CKTpivotRelTol

These are the relative error tolerance, and the factor by which a pivot may be the numerically

non-optimal value and still be chosen for its sparseness preserving properties.

CKTgmin

This is a small "minimal" conductance which is used in several device models to ensure that they do not create zero conductance branches. This is NOT to be confused with the G_{min} used in the Gmin stepping algorithm.

CKTdelMin

This is the minimum timestep which is allowed. Timesteps below this value will result in "Timestep too small" errors. It is currently set unconditionally to 1E-9 times the largest timestep allowed by the user.

CKTtrtol

This is a factor which is used to account for the fact that the truncation error estimation algorithm tends to be too conservative by a significant factor. The value is currently defaulted to the value of 7 as chosen by Nagel[Nage75a].

CKTinitTime, CKTstep, CKTmaxStep, CKTfinalTime

These four variables control a transient simulation, and give the time at which output should start, the user requested timestep, the upper limit on internal timesteps, and the ending time of the simulation. These are copied into these variables from the transient analysis specific structure at the start of the transient simulation since too many references to them are made in too much of the code to make the proper checks that the structure pointed to by CKTcurJob is of the correct type and reference through it to these parameters. At some point in the future, the necessary checks should be made unnecessary by a thorough testing of the code to eliminate other places which find these good approximate values and use them for other purposes, and them replace these with macros which reference the proper field in the substructure.

CKTomega

This is the frequency of the current ac analysis. This should probably be replaced by a macro referencing the ac analysis structure directly, but has been left here for backward compatibility.

CKTsrcFact

This is the factor by which all independent source values must be multipied before loading the right hand side vector to implement the source stepping method.

CKTdiagGmin

This is the G_{min} of the Gmin stepping method, not to be confused with the small conductance CKTgmin described above.

CKTnumSrcSteps, CKTnumGminSteps

These are the number of steps to be taken in the source stepping and Gmin stepping methods, respectively.

CKTnoncon

This is a flag which is incremented by device code when it is determined that the current iteration has not or will not produce a correct solution or that the iteration has not converged, and thus another iteration will be forced.

CKTdefaultMosL, CKTdefaultMosW, CKTdefaultMosAD, CKTdefaultMosAS

These four variables contain the user specified or system provided defaults for the length, width, drain area, and source area, respectively, of mosfets. They don't really fit in well with the structure of SPICE3, but are needed for backward compatibility with SPICE2, and this structure is the only place they can be put where all the MOS models will have access to them.

CKThadNodeset

This is a one bit flag to indicate that .nodeset lines have occurred in the current circuit, and therefore a second convergence iteration will be required in the dc analysis.

CKTfixLimit

This is a simple flag to indicate that the MOSfet limiting should be performed in the same way as in SPICE2, which included an error making it unsymmetrical, but inadvertently helping the convergence of many circuits. Setting this flag to 1 will emulate the old performance.

CKTcurJob

This is a pointer to the JOB structure for the analysis currently being performed. This will be set in CKTdoJob before the analysis specific subroutine is called.

CKTsenInfo

This is a pointer to the structure providing information about any sensitivity analysis being performed concurrently with the current analysis. This pointer will be set in CKTdoJob and be used only by code related to sensitivity as described in the separate sensitivity documentation.

1.2.2. The CKTnode structure

```
typedef struct sCKTnode {     /* data structure to describe an equation */
    IFuid name;
    int type;
    int number;
    double ic;
    double nodeset;
    double *ptr;
    struct sCKTnode *next;
    unsigned int icGiven:1;
    unsigned int nsGiven:1;
} CKTnode;
```

Figure 1.2
Definition of the CKTnode structure

The CKTnode structure is used to describe an equation in the circuit. Generally, a node will correspond to a node in the circuit, but equations added for voltage sources will also generate equations and thus a CKTnode structure.

name

This is the name of the node, and will be a unique identifier as supplied by the front end.

type

This is an indication of the type of data represented by the equation. Current legal values are NODE_VOLTAGE and NODE_CURRENT for equations representing the voltage at a node and

equations representing the current through a device respectively.

number

This is the equation number assigned to the equation.

ic

This is the initial condition specification provided by the user for the node in question.

nodeset

This is the suggested initial voltage for a node as provided by the user in a ".nodeset" statement.

ptr If either a ".nodeset" or ".ic" statement applies to this node, this is a pointer to the numeric value field of the sparse matrix corresponding to the diagonal element at location (*number*, *number*).

next

This provides a pointer to the node structure representing the next higher numbered equation.

icGiven, nsGiven

These are one bit flags which indicate whether this equation has been the subject of a ".ic" or ".nodeset" statement.

1.3. Analysis structures

These structures are used by SPICE3 to maintain data about analyses that are to be performed. Each specific analysis to be performed by SPICE3 is called a "job" and has a data structure of its own. All jobs have a standard prefix on their data structure so that higher level software can examine the jobs and organize them into a group of jobs to be performed all at once and called a "task." In addition, each type of analysis has a single structure which describes it to the higher level code.

1.3.1. SPICEanalysis structure

The SPICEanalysis structure provides information to the higher level routines about the analyses that SPICE3 is capable of performing.

public

This is the simulator to front end interface specified description of the analysis parameters, and is described in detail in the documentation on that interface.

size

This is the size in bytes of the private data structure used by the analysis to store its data.

setParm

This is a pointer to a function which accepts values of parameters and sets them in the device specific data structure.

askQuest

This is a pointer to a function which accesses values in the device specific data structure and makes them available to another routine which does not have any knowledge of the device specific data structures.

1.3.2. TSKtask structure

The TSKtask structure defines a set of analyses to be performed together. The analyses will share such things as the option values set for them by the SPICE2 ''.options'' statement. The task will

---

```
typedef struct {
    IFanalysis public;
    int size;
    int (*(setParm))();
    int (*(askQuest))();
}SPICEanalysis;
```

Figure 1.3
Declaration of the SPICEanalysis structure

```
typedef struct {
    JOB taskOptions;            /* job structure at the front to hold options */
    JOB *jobs;
    char *TSKname;
    double TSKtemp;
    double TSKnomTemp;
    int TSKmaxOrder;            /* maximum integration method order */
    int TSKintegrateMethod;     /* the integration method to be used */
    int TSKcurrentAnalysis;     /* the analysis in progress (if any) */
    int TSKbypass;              /* allow bypass? */
    int TSKdcMaxIter;           /* iteration limit for dc op.  (itl1) */
    int TSKdcTrcvMaxIter;       /* iteration limit for dc tran. curv (itl2) */
    int TSKtranMaxIter;         /* iteration limit for each timepoint for tran*/
                                /* (itl4) */
    int TSKnumSrcSteps;         /* number of steps for source stepping */
    int TSKnumGminSteps;        /* number of steps for Gmin stepping */
    double TSKminBreak;
    double TSKabstol;
    double TSKpivotAbsTol;
    double TSKpivotRelTol;
    double TSKreltol;
    double TSKchgtol;
    double TSKvoltTol;
    double TSKgmin;
    double TSKdelmin;
    double TSKtrtol;
    double TSKdefaultMosL;
    double TSKdefaultMosW;
    double TSKdefaultMosAD;
    double TSKdefaultMosAS;
    unsigned int TSKfixLimit:1;
}TSKtask;
```

Figure 1.4
Declaration of the TSKtask structure

consist of a set of values for task-wide data and a linked list of the individual jobs to be performed as a part of the task. For convenience and to present a more uniform interface to the front end, the ".options" statement has been presented has an analysis itself, and thus would normally be in the linked list of jobs to perform, but has been special cased. There must be only one ".options" job in the task, and the values from all ".options" statements must be merged to produce that single job. These options must be readily accessible at the start of the task, and thus the ".options" analysis has

been singled out and placed directly in the TSKtask structure. The ''.options'' analysis then uses the TSKtask structure directly as its own job data structure rather than having a separate one. The elements of the TSKtask structure are:

taskOptions

The standard job header for the ''.options'' job that applies to the entire task.

jobs

A linked list of the jobs structures of all of the analyses to be performed as part of this task.

TSKname

The name of the task as assigned by the front end.

TSKtemp

The default temperature at which instances will operate in this task.

TSKnomTemp

The default temperature at which model parameters will be considered to have been measured.

TSKmaxOrder

The maximum integration order permitted during the transient simulations of this task.

TSKintegrateMethod

The integration method to be used during the transient simulations of this task.

TSKcurrentAnalysis

The type of analysis currently in progress in this task. This field is not currently used and should be replaced by a JOB pointer in a future version.

TSKbypass

A flag to indicate whether the user wishes inactive device bypass to be enabled or disabled assuming it was not disabled at compiled time.

TSKdcMaxIter, TSKdcTrcvMacIter, TSKtranMaxIter

The maximum number of Newton-Rapheson iterations that will be taken before giving up in a

dc operating point analysis, a dc transfer curve analysis, and a single transient timepoint.

TSKnumSrcSteps, TSKnumGminSteps

The number of steps the source stepping or gmin stepping method should take in attempting to find an operating point.

TSKabstol, TSKpivotAbsTol, TSKpivotRelTol, TSKreltol

TSKchgtol, TSKvoltTol, TSKtrtol

The per task tolerances to be used in the correspondingly named fields of the CKTcircuit structure during the analysis.

TSKgmin

A small user-specifyable minimal conductance used in several models when a zero conductance is not acceptable.

TSKdelmin

The minimum timestep permitted during transient analysis before the program gives up. This is currently overwritten by the transient analysis code because of problems with users setting unrealisticly small values and causing simulations to run forever.

TSKdefaultMosL, TSKdefaultMosW, TSKdefaultMosAD, TSKdefaultMosAS

The default values for MOSfet length, width, drain area, and source area for this circuit.

TSKfixLimit

A flag to indicate whether the unsymmetric, but sometimes effective limiting of SPICE2 should be used or whether the correct symetric limiting of SPICE3 should be used.

1.3.3. JOB structure

The job structure is used to contain the data needed to run a single analysis. This is actually a structure prefix, and is included as the first element of every analysis specific data structure. This allows common code to manipulate the structures without knowing the details of each analysis type's specific structure. The individual fields are:

```
typedef struct sJOB{
    int JOBtype;                /* type of job */
    struct sJOB *JOBnextJob;    /* next job in list */
    IFuid JOBname;              /* name of this job */
} JOB;
```

Figure 1.5
Declaration of the JOB structure

JOBtype

An integer that identifies the type of analysis the JOB structure provides more details of. The integer is the index in the analInfo array to the SPICEanalysis structure which provides a description of the analysis along with pointers to the functions which implement it.

JOBnextJob

A pointer to the next JOB structure in the linked list of structures that makes up a TASK.

JOBname

A character string used to identify the analysis during output.

1.3.4. ACAN structure

The ACAN structure contains the data necessary to control an ac analysis. The first three fields of the structure are those required to match the JOB structure. The remaining fields are specific to the ac analysis.

ACstartFreq

The lowest frequency at which the ac analysis is to be performed.

ACstopFreq

The highest frequency at which the ac analysis is to be performed.

ACfreqDelta

The frequency change factor. If the frequency is being stepped linearly, this is the size of the

```
typedef struct {
    int JOBtype;
    JOB *JOBnextJob;        /* pointer to next thing to do */
    char *JOBname;          /* name of this job */
    double ACstartFreq;
    double ACstopFreq;
    double ACfreqDelta;     /* multiplier for decade/octave stepping, */
                            /* step for linear steps. */
    double ACsaveFreq;      /* frequency at which we left off last time*/
    int ACstepType;         /* values described below */
    int ACnumberSteps;
} ACAN;
```

Figure 1.6
Declaration of the ACAN structure

step in *Hz*, if the frequency is being stepped with either the octave or decade stepping option, this is the factor by which each frequency is multiplied to obtain the next frequency.

ACsaveFreq

If the analysis is interrupted, this is the last frequency at which the analysis was complete, thus allowing the analysis to continue from where it left off.

ACstepTyep

The type of step. This takes on one of three symbolic constant values. When set to LINEAR, linear stepping is done with the number of steps interpreted as the total number of frequency points. When set to OCTAVE or DECADE, logarithmic stepping is done with the number of steps interpreted as the number of frequency points per octave or decade.

ACnumberSteps

ACnumberSteps is the number of steps to be computed and output, interpreted as described under ACstepType above.

1.3.5. OP structure

The OP stucture simply provides an indication that an operating point is to be done and provides a name for it. Since the operating point is very simple and has no additional parameters, only the minimal structure as required by the JOB prefix is needed.

1.3.6. PZAN structure

The PZAN structure describes a pole-zero analysis to be performed. The first three fields are the standard JOB structure prefix.

PZnodeI

The positive input node.

PZnodeG

The negative input node.

PZnodeJ

The positive output node.

PZnodeK

The negative output node.

PZdiffJK

The equation number of an additional equation added to the matrix for the output port.

```
typedef struct {
    int JOBtype;
    JOB *JOBnextJob;
    char *JOBname;
} OP;
```

Figure 1.7
Declaration of the OP structure

```
typedef struct {
    int JOBtype;
    JOB *JOBnextJob;
    IFuid JOBname;
    int PZnodeI ;
    int PZnodeG ;
    int PZnodeJ ;
    int PZnodeK ;
    int PZdiffJK ;
    int PZdiffIG ;
    int PZflagVI ;
    int PZflagPZ ;
    int PZnumswaps ;
    root* PZpoleList;
    root* PZzeroList;
    double *PZJK_Jptr;
    double *PZJK_Kptr;
    double *PZJK_JKptr;
    double *PZIG_Iptr;
    double *PZIG_Gptr;
    double *PZIG_IGptr;
} PZAN;
```

Figure 1.8
Declaration of the PZAN structure

PZdiffIG

The equation number of an additional equation added to the matrix for the input port.

PZflagVI

This flag indicates the type of transfer function to analize. A value of zero indicates an $\frac{\text{outputvoltage}}{\text{inputvoltage}}$ transfer function, while a value of one indicates an $\frac{\text{outputvoltage}}{\text{inputcurrent}}$ transfer function.

PZflagPZ

A flag to indicate the exact type of analysis to be perfored. Setting the low order bit calls for a pole analysis, setting the next bit calls for a zero analysis, setting both bits calls for both analyses.

PZnumswaps

The is a count of the number of row and column swaps have been performed in the decomposition of the matrix.

PZpoleList

The pointer to the beginning of a linked list of poles already found in this analysis.

PZzeroList

The pointer to the beginning of a linked list of zeroes already found in this analysis.

PZJK_Jptr, PZJK_Kptr, PZJK_JKptr

These are pointers into the sparse matrix at locations in the additional row PZdiffJK at columns corresponding to NODEJ, NODEK, and PZdiffJK respectively.

PZIG_Iptr, PZIG_Gptr, PZIG_IGptr

These are pointer into the sparse matrix at locations in the additional row PZdiffIG at columns corresponding to NODEJ, NODEK, and PZdiffIG respectively.

1.3.6.1. root structure

This is a simple structure used by the pole-zero code to collect the poles and zeros into linked lists.

real, imag

The real and imaginary parts of the value of the pole or zero.

```
typedef struct sroot {
   double real;
   double imag;
   struct sroot *next;
} root;
```

Figure 1.9
Declaration of the root structure

---

next

The pointer to the next pole or zero in the linked list.

1.3.7. TFan structure

The TFan structure describes a transfer function analysis to be performed. The first three elements of the structure are the standard JOB head to link the structure into the TSKtask linked list. The remaining fields are:

TFoutPos, TFoutNeg

These are the positive and negative nodes of the output port if the output port signal is a voltage, otherwise they are ignored.

TFoutSrc

This is the identifying name of the output source if the output variable is the current throught a source.

TFinSrc

This is the identifying name of the input source, either a voltage source or a current source.

```
typedef struct {
    int JOBtype;
    JOB *JOBnextJob;
    IFuid JOBname;
    CKTnode *TFoutPos;
    CKTnode *TFoutNeg;
    IFuid TFoutSrc;
    IFuid TFinSrc;
    unsigned int TFoutIsV :1;
    unsigned int TFoutIsI :1;
    unsigned int TFinIsV :1;
    unsigned int TFinIsI :1;
} TFan;
```

Figure 1.10
Declaration of the TFan structure

TFoutIsV, TFoutIsI

A pair of flags to indicate whether the output signal is to be the voltage between two nodes or the current throught a voltage source. If neither of these flags has been set, then no output signal has been specified which is an error.

TFinIsV, TFinIsI

A pair of flags to indicate whether the input signal is from a voltage source or a current source. These flags are currently mutually exclusive, but both exist for historical reasons.

1.3.8. TRANan structure

The TRANan structure contains the data necessary to control a transient analysis. The first three fields are the standard JOB header. The remaining fields are:

TRANfinalTime

The time at which the transient simulation should end.

TRANstep

The user suggested time step for the transient simulation.

```
typedef struct {
    int JOBtype;
    JOB *JOBnextJob;
    char *JOBname;
    double TRANfinalTime;
    double TRANstep;
    double TRANmaxStep;
    double TRANinitTime;
    long TRANmode;
    GENERIC * TRANplot;
} TRANan;
```

Figure 1.11
Declaration of the TRANan structure

TRANmaxStep

The user specified or program default maximum internal time step.

TRANinitTime

The first time at which output data should be stored.

TRANmode

A word that may contain the "UIC" flag to indicate that the initial conditions supplied in the specification are to replace an operating point analysis before the start of the transient analysis.

TRANplot

The plot structure returned by OUTpBeginPlot which needs to be stored in the event the analysis is paused and resumed.

1.3.9. TRCV structure

The TRCV structure contains the data necessary to control a sensitivity analysis. The first three elements of this stucture are the standard JOB header.

```
typedef struct {
    int JOBtype;
    JOB *JOBnextJob;
    char *JOBname;
    double TRCVvStart[TRCVNESTLEVEL];       /* starting voltage/current */
    double TRCVvStop[TRCVNESTLEVEL];        /* ending voltage/current */
    double TRCVvStep[TRCVNESTLEVEL];        /* voltage/current step */
    double TRCVvSave[TRCVNESTLEVEL];        /* voltage of this source BEFORE */
                                            /* analysis-to restore when done */
    IFuid TRCVvName[TRCVNESTLEVEL];         /* source being varied */
    GENinstance *TRCVvElt[TRCVNESTLEVEL];   /* pointer to source */
    int TRCVvType[TRCVNESTLEVEL];           /* type of element being varied */
    int TRCVset[TRCVNESTLEVEL];             /* flag to indicate this nest level used */
    int TRCVnestLevel;                      /* number of levels of nesting called for */
    int TRCVnestState;                      /* iteration state during pause */
} TRCV;
```

Figure 1.12
Declaration of the TRCV structure

TRCVvStart

This array contains the starting voltage or current values for each of the sources to be swept.

TRCVvStop

This array contains the final voltage or current values for each of the sources to be swept

TRCVvStep

This array contains the step to be added to the value of each of the sources as it progresses from TRCVvStart to TRCVvStop.

TRCVvSave

This array is used to save the original dc values of the sources while transfer curve values are being put into their places during the course of the analysis. When the analysis is over, these values must be restored to their original places.

TRCVvName

An array containing the names of the sources to be swept.

TRCVvElt

A pointer to the GENinstance structure for the individual sources to be swept.

TRCVvType

The type of each element.

TRCVset

An array of flags to indicate to the transfer curve analysis code which nesting levels actually have data strored in them and which have been left empty so that the empty nesting levels can be properly skipped over without generating errors.

TRCVnestLevel

The highest nesting level that has been used so far in the analysis.

TRCVnestState

This is the nesting level of the voltage or current that changed most recently which needs to be

saved so that the analysis can be resumed correctly after a pause.

1.3.10. SENstruct structure

The sensitivity structures are described seprately in the documentation on SPICE3 sensitivity analysis[Chou88a].

1.4. Device structures

There are three categories of data structures used for the devices in SPICE3. The data structures which describe the devices at the level needed by the front end are covered in detail the the front end to simulator interface appendix. The data structures which describe the devices at the level of detail needed by the rest of the simulator to successfully use them is described in the appendices describing the addition of a device and the device to simulator interface. The only structures left are the internal structures of the devices themselves. These structures are quite variable, containing lots of device dependent data, but they all contain a standard prefix.

1.4.1. GENmodel structure

The GENmodel structure defines the standard prefix which must appear at the beginning of all device models. This structure allows code in SPICE3 to search through device models without any knowledge of the devices themselves, or even of the type of the devices. The fields of this structure

```
typedef struct sGENmodel {          /* model structure for a device */
    int GENmodType;                 /* type index of this device type */
    struct sGENmodel *GENnextModel; /* pointer to next possible model in
                                     * linked list */
    GENinstance * GENinstances;     /* pointer to list of instances that have this
                                     * model */
    IFuid GENmodName;               /* pointer to character string naming this model */
} GENmodel;
```

Figure 1.13
Declaration of the GENmodel structure

are:

GENmodType

The integer subscript of the descriptor for the type of device this model represents. This subscript allows code to access the correct routines to process the model by referencing the SPICEdev structure with that subscript in the DEVices array and thereby accessing all of the data on the model and the routines to process it.

GENnextModel

A pointer to the next model in the linked list structure. This model will have the same value for GENmodType and a distinct name, but no other information about it is known.

GENinstances

A pointer to the first element of the linked list of instances of this model. The instance linked list also has a standardized structure so that traversal of that list can also be done without knowledge of the device.

GENmodName

The name of the model.

1.4.2. GENinstance structure

The GENinstance structure is used to traverse the device specific per instance data structures without any knowledge of the devices. All devices can be examined with this structure if appropriate care is used not to reference a node with a number greater than the number of nodes defined for a device of that type in the corresponding SPICEdev structure. The fields are:

GENmodPtr

A backpointer to the model this is an instance of. The model contains the type of the device if this is needed and only an instance pointer is available. This also provides access to model parameters when only an instance pointer is given.

```
typedef struct sGENinstance {
    struct sGENmodel *GENmodPtr;          /* backpointer to model */
    struct sGENinstance *GENnextInstance; /* pointer to next instance of
                                           * current model*/
    IFuid GENname;                        /* pointer to character string naming this instance */
    int GENnode1;                         /* appropriate node numbers */
    int GENnode2;                         /* appropriate node numbers */
    int GENnode3;                         /* appropriate node numbers */
    int GENnode4;                         /* appropriate node numbers */
    int GENnode5;                         /* appropriate node numbers */
} GENinstance ;
```

Figure 1.14
Declaration of the GENinstance structure

GENnextInstance

This is a pointer to the next instance in the linked list of device instances.

GENname

The name of the instance.

GENnode*i*

These are node numbers for the nodes the instance is attached to. Not all of these are always valid, reference to a terminal GENnode*i* for *i* greater than (DEVices[inst->GENmodPtr->GENmodType])->DEVpublic.terms is an error and will produce unpredictable results.

1.5. Sparse matrix structures

These structures are used by the sparse matrix package described elsewhere to manage the sparse matrices used in the simulation. These structures are all *private* to the sparse matrix package. Other parts of SPICE3 are allowed to have pointers to these structures, but use of any of the fields in the structures is restricted to the matrix package, thus changes to the structure only affect the matrix package.

1.5.1. SMPmatrix structure A *client* in this description refers to a program using the sparse matrix package as opposed to the package itself. All arrays are of size one greater than the largest node number, and are numbered from 0 to the largest node number. Row and column zero are used in establishing the matrix structure and in providing entires corresponding to connections to the ground node, but are ignored in most operations on the matrix since the voltage of the ground node is fixed. This removes the extra equation produced by the Modified Nodal Analysis formulation used in SPICE3.

SMProwHead, SMPcolHead

These arrays contain pointers to the first SMPelement structure representing an element in each row or column of the matrix.

```
typedef struct {
    SMPelement **SMProwHead;    /* pointer to first in row */
    SMPelement **SMPcolHead;    /* pointer to first in col */
    int *SMProwCount;           /* number in row excluding diag -1 */
    int *SMPcolCount;           /* number in column excluding diag -1 */
    int *SMProwMapIn;           /* mapping array from external to
                                /* internal row numbers */
    int *SMPcolMapIn;           /* mapping array from external to
                                /* internal column numbers */
    int *SMProwMapOut;          /* mapping array from internal to
                                /* external row numbers */
    int *SMPcolMapOut;          /* mapping array from internal to
                                /* external column numbers */
    int SMPsize;                /* number of rows and columns */
    int SMPallocSize;           /* number of rows and columns allocated */
    int SMPnonZero;             /* number of non-zero terms in matrix */
    int SMPoldNonZ;             /* number of non-zero terms in
                                /* matrix before reordering */
    int SMPbadi;                /* row of last troublesome entry */
    int SMPbadj;                /* column of last troublesome entry*/
}SMPmatrix;
```

Figure 1.15
Declaration of the SMPmatrix structure

SMProwCount, SMPcolCount

During matrix build contain a count of the number of non-zero off diagonal entries in each row and column. Destroyed during reordering, these are recomputed as necessary during later reordering.

SMProwMapIn, SMPcolMapIn

Arrays used to translate client row and column numbers to internal row and column numbers. Indexed by client row and column numbers, the entries give corresponding internal row and column numbers.

SMProwMapOut, SMPcolMapOut

Arrays used to translate internal row and column numbers to client row numbers. Indexed by internal row and column numbers, the entries give corresponding client row and column numbers

SMPsize

This integer gives the size of the active part of the sparse matrix structure. The actual structure allocated may be larger due to the need to efficiently expand the structure, but this is the effective size of the matrix.

SMPallocSize

This is the actual allocated size of the sparse matrix structure arrays, and will usually be larger than SMPsize since actually growing the arrays is very time consuming. Whenever the matrix must be grown, its arrays are grown by at least SMPALLOCINCREMENT (100) with the extra space reflected only in SMPallocSize and not in SMPsize.

SMPnonZero

The number of non-zero entries in the matrix.

SMPldNonZ

For statistics gathering, this is the value of SMPnonZero saved just before a reordering takes place.

SMPbadi, SMPbadj

The saved numbers of the internal row and column at which the matrix package had trouble during a reordering.

1.5.2. SMPelement structure

```
typedef struct sSMPelement {
    double SMPvalue;                /* matrix entry for this point */
    double SMPiValue;               /* imaginary matrix entry for this point */
    struct sSMPelement *SMProwNext; /* pointer to next in row */
    struct sSMPelement *SMPcolNext; /* pointer to next in column */
    int SMProwNumber;               /* internal row number */
    int SMPcolNumber;               /* internal column number */
} SMPelement;
```

Figure 1.16
Declaration of the SMPelement structure

SMPvalue

The actual value of the entry in the matrix.

SMPiValue

If the matrix is complex instead of a real, this field contains the imaginary part of the value. This field is ignored by all operations that do not specify that they operate on complex values. Note that this field immediately follows SMPvalue without any intervening space, thus making it possible to access the imaginary field using a pointer to the real field by incrementing the pointer by one.

SMProwNext, SMPcolNext

The pointers used to link to the next entry in the row or column.

SMProwNumber, SMPcolNumber

The row and column numbers of this element.

1.6. Input parser structures

These structures are used within the input parser to maintain data about the current circuit being parsed.

1.6.1. INPtab structure

This structure is used to contain data on all names defined in the front end which are not equation names. The actual data structure is a hash table made up of linked lists of INPtab structures with matching hash values. The fields of the structure are:

t_ent

This is the character string which defines the name stored in this instance of the INPtab.

t_next

This contains a pointer to the next entry in the hash chain.

1.6.2. INPnTab structure

```
struct INPtab {
   char *t_ent;
   struct INPtab *t_next;
};
```

Figure 1.17
Declaration of the INPtab structure

```
struct INPnTab {
   char *t_ent;
   GENERIC* t_node;
   struct INPnTab *t_next;
};
```

Figure 1.18
Declaration of the INPnTab structure

This structure is almost identical to the INPtab structure described above, but is used for names corresponding to equations in the circuit such as node names or current equation names. The basic structure is the same, with a hash table of linked lists. The only difference is the data placed in the table, and the presence of one additional field in the structure.

t_node

This element contains the pointer to the simulator's internal structure corresponding to this name.

1.6.3. INPtables This is the overall input package data structure. One instance of this structure is allocated for each circuit to be parsed and passed to every input package routine to help it in parsing.

Most of the fields of this structure are of the form defXmod. Each of these fields points to a SPICE3 internal model structure which represents the default model used for the device of type *X*. The remaining fields are:

INPsymtab, INPsize

INPsymtab is the hash table for non-equation names. This is an allocated array of pointers to the beginning of hash chains. The allocated array of pointers contains INPsize pointers.

INPtermsymtab, INPtermsize

INPtermsymtab is the hash table for equation or terminal names. This is an allocated array of pointers to the beginning of INPtermsize hash chains.

1.6.4. card structure

The card structure is used to hold the image of an input line and all the data associated with it during parsing. Lines are read and broken into physical and logical lines. Physical lines are delimited by newline characters at their end. Logical lines are what results from concatenating continuation lines onto the lines they continue. The fields are:

linenum

The line number associated with this card image. It is the line number associated with the

```
typedef struct sINPtables{
    struct INPtab **INPsymtab;
    struct INPnTab **INPtermsymtab;
    int INPsize;
    int INPtermsize;
    GENERIC *defAmod;          /* unused */
    GENERIC *defBmod;          /* arbitrary source */
    GENERIC *defCmod;          /* capacitor */
    GENERIC *defDmod;          /* diode */
    GENERIC *defEmod;          /* vcvs */
    GENERIC *defFmod;          /* cccs */
    GENERIC *defGmod;          /* vccs */
    GENERIC *defHmod;          /* ccvs */
    GENERIC *defImod;          /* indep. current source */
    GENERIC *defJmod;          /* jfet */
    GENERIC *defKmod;          /* mutual inductor */
    GENERIC *defLmod;          /* inductor */
    GENERIC *defMmod;          /* mosfet */
    GENERIC *defNmod;          /* unused */
    GENERIC *defOmod;          /* unused */
    GENERIC *defPmod;          /* unused */
    GENERIC *defQmod;          /* bjt */
    GENERIC *defRmod;          /* resistor */
    GENERIC *defSmod;          /* voltage controlled switch */
    GENERIC *defTmod;          /* transmission line */
    GENERIC *defUmod;          /* uniform RC line */
    GENERIC *defVmod;          /* indep. voltage source */
    GENERIC *defWmod;          /* current controlled switch */
    GENERIC *defYmod;          /* unused */
    GENERIC *defZmod;          /* mesfet */
} INPtables;
```

Figure 1.19
Declaration of the INPtables structure

```
typedef struct card{
   int linenum;
   char *line;
   char *error;
   struct card *nextcard;
   struct card *actualLine;
} card;
```

Figure 1.20
Declaration of the card structure

physical line that the first character of the logical line came from.

line

The logical text of the line. If this is a single logical line that was read as a single physical line, this will be the exact line. If the logical line consists of multiple physical lines marked with continuation characters in column one, this will be the logical line after the continuation lines have been concatenated onto the lines they continue with the continuation character replaced by a single space.

error

This is the text of any error messages that are associated with this logical line.

nextcard

A pointer to next card structure representing the next logical line in the input.

actualLine

If this pointer is not NULL, then this logical line is actually made up of several physical lines. This pointer will point to a linked list of the physical lines that make up the logical line. Each element of the linked list is a card structure as well, but in that list the *line* field is always a physical line, never a logical line, *nextcard* points to the next physical line comprising the logical line, and the *actualLine* pointer is always NULL.

1.6.5. INPmodel structure

The INPmodel structure is used by the input parser package to hold information about a model which occurs in the input. During pass one pointers to the lines containing ".MODEL" cards are saved in a linked list of these structures. During pass two of the parser, these lines are actually parsed as needed and pointers to the models created are stored in the structure for fast access. Currently, the head of the list of INPmodel structures is stored in a global variable, thus limiting the parser to having one circuit in progress at a time. This restriction should be removed in a future version of the parser.

1.6.6. INPparseTree structure

The INPparseTree structure contains data on a parse tree the front end has read and will later interpretively evaluate for the simulator. The parse tree structure encapsulates the standard structure

```
typedef struct sINPmodel{
    IFuid INPmodName;   /* uid of model */
    int INPmodType;     /* type index of device type */
    struct sINPmodel *INPnextModel;  /* link to next model */
    int INPmodUsed;     /* flag to indicate it has already been used */
    card *INPmodLine;   /* pointer to line describing model */
    GENERIC *INPmodfast;   /* high speed pointer to model for access */
} INPmodel;
```

Figure 1.21
Declaration of the INPmodel structure

```
typedef struct INPparseTree {
    IFparseTree p;
    struct INPparseNode *tree;
    struct INPparseNode **derivs;
} INPparseTree;
```

Figure 1.22
Declaration of the INPparseTree structure

defined by the front end interface, appending additional fields to it for internal use in the parser. The fields are:

p

The parse tree structure required by the front end.

tree

The actual internal parse tree structure for the equation actual provided by the user.

derivs

An allocated array of pointers to internal parse tree structures for trees representing the partial derivatives of the original parse tree with respect to every variable used in the expression.

1.6.7. INPparseNode

The INPparseNode structure is the structure which is used to build parse trees. These structures are built up into a tree arrangement with each entry either representing a variable or constant, or a operator with one or two operands which are themselves INPparseNode structures.

type

This indicates the type of node the instance is, a simple operator such as +, −, or ×, a constant, or a function call.

```
typedef struct INPparseNode {
    int type;                   /* One of PT_*, below. */
    struct INPparseNode *left;  /* Left operand, or single operand. */
    struct INPparseNode *right; /* Right operand, if there is one. */
    double constant;            /* If INP_CONSTANT. */
    int valueIndex;             /* If INP_VAR, index into vars array. */
    char *funcname;             /* If INP_FUNCTION, name of function, */
    int funcnum;                /* ... one of PTF_*, */
    double (*function)();       /* ... and pointer to the function. */
} INPparseNode;
```

Figure 1.23
Declaration of the INPparseNode structure

left

The left operand of the operator in this node if it is a two operand operator, the only operand if it is a single operand operator, and null if it is not an operator.

right

The right operand of the operator in this node if it is a two operand operator, otherwise null.

constant

The numeric value of a constant stored in the node.

valueIndex

The index of the variable being placed in the tree at this point. The index is into the array of variable names in the IFparseTree structure in the INPparse tree structure this node is descended from.

funcname

The name of the function called from this node.

funcnum

The number of the function from an internal table. This is used to more rapidly process the functions using case statement instead of string comparison operations.

function

A pointer to the internal function which actually implements the parse tree operation.

1.6.8. PTelement structure

The PTelement structure is used to hold temporary data during the parsing of an expression into a parse tree. These are pushed onto a parse stack as parsing progresses and popped off as operations consume them. The component parts of this structure are:

token

The lexical type of the token the lexical analyzer has found. This can be an operator, the end of the input, parentheses, or a value.

```
typedef struct PTelement {
    int token;
    int type;
    union {
        char *string;
        double real;
        INPparseNode *pnode;
    } value;
} PTelement ;
```

Figure 1.24
Declaration of the PTelement structure

type

If token indicates a value, this will indicate the type of the value, and will be one of TYP_NUM indicating a number, TYP_STRING indicating a character string, or TYP_PNODE indicating a parse tree node. The lexical analyzer will not generate TYP_PNODE, but the parser will generate them internally and push them onto the stack.

value

This is a union used to hold the value of the token. Depending on the value of type, one of the three fields will be filled in.

string

The character string found as a token.

real

The real valued number found by the lexical analyzer.

pnode

A parse tree internal node structure resulting from the combination of previously parsed elements.

CHAPTER 2

Packages

This chapter describes in much more detail the packages internal to SPICE3 which are used to construct the entire program. Each of these packages is independent and operates without details of the internals of other packages. These descriptions should be adequate for anyone using the package, but do not provide details of the algorithms used. Those algorithms that are particularly new or interesting have been described earlier, and most routines are relatively short and simple to understand. Further details of the basic algorithms can also be found in previous reports on SPICE[Cohe76a, Nage75a] and in detailed comments in the code.

Each of the packages used will require that one or more header files be included. The structure of the SPICE3 header files is such that it is always safe to include an additional header file, thus header files should include those files they need, but they should also be explicitly included if their contents are needed directly and not depend on their inclusion by another header file.

2.1. CKT

The CKT or circuit package is not really a single independent package, but is a set of very closely related packages and interface routines. This package is the only block of code in SPICE3 which needs to know the entire structure of the SPICE3 data structures and is the interface point for all of the other packages. The header file "CKTdefs.h" is assumed to be included before any references are made to any of these routines. Many of these routines should never be called by name since they are the routines to be called from the front end through the front end to simulator interface, and thus are usually called through function pointers. They remain available so that calls to them may be made internally as well as through the interface.

Due to the size of this package, the description of it has been broken down into subpackages of more closely related routines.

2.1.1. Simulator interface routines

These routines implement the interface specified in the front end to simulator standard. Detailed specifications of these routines are not provided here since it would be a duplication of the specification in the interface standard, but a general statement of purpose and additional notes where there are any special considerations for SPICE3 in the implementation of the routines is provided.

2.1.1.1. SPIinit

```
int SPIinit(ckt, restart)
    GENERIC *ckt;     /* circuit to operate on */
    int restart;      /* start over or resume? */
```

This subroutine is, in many ways, the central part of SPICE3. All operations in SPICE3 must be preceded by a call to this routine, and it is this routine that determines the capabilities of the simulator. The data structure initializations contained in this file define the analyses and devices supported by SPICE3, both for SPICE3's internal structures and for the front end interface. Additionally, all global variables in SPICE3 are declared and initialized in this file.

2.1.1.2. CKTinit

```
int CKTinit(ckt)
    GENERIC **ckt;    /* new circuit to create */
```

This routine performs all the necessary operations to initialize a circuit to be processed later. The circuit structure is allocated and all default values filled in, as well as performing initialization of substructures as needed.

2.1.1.3. CKTdestroy

```
int CKTdestroy(ckt)
    GENERIC *ckt;    /* circuit to destroy */
```

CKTdestroy dismantles all the data structures associated with the given circuit. All structures and substructures are freed. On return, *ckt* is no longer a valid pointer to a circuit, and must not be used as such.

2.1.1.4. CKTnewNode

```
int CKTnewNode(ckt, node, name)
    GENERIC *ckt;        /* circuit node is to be in */
    GENERIC **node;      /* node pointer to return */
    IFuid name;          /* name of the node to be created */
```

This routine creates a new node in the circuit. Creating the node involves allocating a node structure, linking that structure into the node list, allocating a new equation number to the node, storing the IFuid in the node structure, and filling in the rest of the structure with appropriate defaults. Note that if used inside of SPICE3, this requires a prior call to the front end function IFnewUid to create the unique id to be stored in the node structure.

2.1.1.5. CKTground

```
int CKTground(ckt, node, name)
    GENERIC *ckt;        /* circuit ground node is in */
    GENERIC **node;      /* ground node pointer to return */
    IFuid name;          /* name of the node to be created */
```

This is the special routine to add the ground node to the circuit. This node is distinguished in that it must be given equation number zero, and thus a special location in the linked list of nodes, otherwise this routines operates as CKTnewNode does. Note that the interface specification calls for only a single call to this subroutine succeeding, but for future calls to return the error E_EXISTS and ignore the new name passed in, but return the old one instead.

2.1.1.6. CKTbindNode

```
int CKTbindNode(ckt, inst, term, node)
    GENERIC *ckt;       /* circuit to operate on */
    GENERIC *inst;      /* pointer to instance */
    int term;           /* terminal number */
    GENERIC *node;      /* node structure */
```

CKTbindNode connects a node in the circuit with a specific terminal of a device instance. For this to work, both the node and device instance must already exist, and the terminal number specified must be less than or equal to the maximum number of terminals supported by the device. Note that terminals are numbered from one, not zero.

2.1.1.7. CKTfndNode

```
int CKTfndNode(ckt, node, name)
    GENERIC *ckt;       /* circuit to operate on */
    GENERIC **node;     /* node structure */
    IFuid name;         /* uid of node to find */
```

This function performs a search through the node structures in the specified circuit to find the one with the given unique identifier associated with it. If the identifier is found, *node is set to point to the node found, otherwise E_NOTFOUND is returned. This function is obviously not a fast one, since it must search a linked list, but is provided as a backstop for simple front ends which do not wish to maintain a complete symbol table and are willing to pay the costs of doing so.

2.1.1.8. CKTinst2Node

```
int CKTbindNode(ckt, instPtr, terminal, node, nodeName
    GENERIC *ckt;       /* circuit to operate on */
    GENERIC *instPtr;   /* pointer to instance */
    int terminal;       /* terminal number */
    GENERIC **node;     /* node structure */
    IFuid *nodeName;    /* node name */
```

This function, again provided for front ends which do not maintain full symbol tables, provides both the name (IFuid) and node pointer of a node given an instance and terminal number it is

attached to.

2.1.1.9. CKTsetNodPm

```
int CKTsetNodPm(ckt, node, parm, value, selector)
    GENERIC *ckt;       /* circuit to operate on */
    GENERIC *node;      /* node structure */
    int parm;           /* parameter number */
    IFvalue *value;     /* parameter value */
    IFvalue *selector;  /* sub parameter selector */
```

CKTsetNodPm is used to set parameters on circuit nodes. This is intended to generalize the capabilities of the ''.nodeset'' and ''.ic'' commands of SPICE2 by allowing any parameters to be set on nodes. The selector argument is currently unused, but is specified by the front end interface.

2.1.1.10. CKTaskNodQst

```
int CKTaskNodQst(ckt, node, parm, value, selector)
    GENERIC *ckt;       /* circuit to operate on */
    GENERIC *node;      /* node structure */
    int parm;           /* parameter number */
    IFvalue *value;     /* parameter value */
    IFvalue *selector;  /* sub parameter selector */
```

This function provides the inverse of CKTsetNodPm by allowing the front end to find out the value of any of the node parameters.

2.1.1.11. CKTdltNod

```
int CKTdltNod(ckt, node)
    GENERIC *ckt;       /* circuit to operate on */
    GENERIC *node;      /* node structure */
```

This function is required by the front end interface and should delete a node from the circuit. Since SPICE3 does not yet have all the facilities required to perform this operation, it simply returns the E_UNSUPP error code. Before this function is implemented, it will be necessary to institute node reference counts to ensure that the node is no longer referenced before deletion, to provide a way to renumber circuit equations if this is not the highest numbered equation, and to delete rows and

columns from the matrix.

2.1.1.12. CKTcrtElt

```
int CKTcrtElt(ckt, modPtr, instPtr, name)
    GENERIC *ckt;          /* circuit to operate on */
    GENERIC *modPtr;       /* model to create instance of */
    GENERIC **instPtr;     /* returned new instance */
    IFuid name;            /* name of new instance */
```

CKTcrtElt creates circuit elements by instantiating existing models. A new instance will be created with all default parameters and with the name given. The returned instPtr will point directly to the created instance and can be used by subsequent calls to set parameters, bind nodes, and query parameters.

2.1.1.13. CKTparam

```
int CKTparam(ckt, inst, parm, value, selector)
    GENERIC *ckt;          /* circuit to operate on */
    GENERIC *inst;         /* instance being modified */
    int parm;              /* parameter number */
    IFvalue *value;        /* parameter value */
    IFvalue *selector;     /* sub parameter selector */
```

CKTparam modifies existing elements in the circuit by changing the values of various parameters associated with them. Since the code at this level is not permitted to know any of the details of the device implementations, a lower level routine is provided by each device to handle its own parameters and this routine simply passes the problem off to the correct device specific routine after determining the type of device involved. The *selector* parameter is currently unused, but can be used to select a specific value out of an array of similar values to be changed.

2.1.1.14. CKTask

```
int CKTask(ckt, inst, parm, value, selector)
    GENERIC *ckt;       /* circuit to operate on */
    GENERIC *inst;      /* instance to query */
    int parm;           /* parameter number */
    IFvalue *value;     /* parameter value */
    IFvalue *selector;  /* sub parameter selector */
```

CKTask is the inverse of CKTparam, providing the ability to query parameters of an instance, as well as providing a means for more complete output by querying many other variables associated with an instance, such as internal capacitances, equivalent conductances and similar internal state information. The choice of the parameters available is up to the device implementor, since a device specific subroutine is used to access the actual data structures. The *selector* parameter is used to index into an array of similar parameters to select a single one to output. For example, the circuit sensitivity outputs are vectors of the same size as the node voltage/source current vectors, but a simple output system may prefer to have SPICE3 index into this vector to pick out a specific sensitivity instead of getting the entire vector and having to pick the value out itself.

2.1.1.15. CKTfndDev

```
int CKTfndDev(ckt, type, inst, name, modPtr, modName)
    GENERIC *ckt;       /* circuit to operate on */
    int *type;          /* type of the device */
    GENERIC **inst;     /* found instance */
    IFuid name;         /* name of instance to find */
    GENERIC  *modPtr;   /* pointer to model of instance found*/
    IFuid modName;      /* name of model if known */
```

CKTfndDev is used to try to find a specific instance pointer from as much or as little information as the front end has available about it. Any of the variables may be used to help identify the instance except *inst*, which is the primary output variable from this routine. Any of the variables may be specified as NULL without affecting the success of CKTfndDev except *ckt* and *name* which **must** be specified.

2.1.1.16. CKTdltInst

```
int CKTdltInst(ckt, inst)
    GENERIC *ckt;     /* circuit to operate on */
    GENERIC *inst;    /* instance being deleted */
```

This routine is intended to allow the deletion of instances from the circuit. All of the bookkeeping necessary to permit this has not been completed, so this routine returns the error code E_UNSUPP to the front end to indicate that SPICE3 does not yet support the deletion of devices.

2.1.1.17. CKTmodCrt

```
int CKTmodCrt(ckt, inst, parm, value, selector)
    GENERIC *ckt;          /* circuit to operate on */
    int type;              /* type of model to create */
    GENERIC **model;       /* model being created */
    IFuid name;            /* name of new model */
```

CKTmodCrt creates a new model in the circuit. The model is initialized with only a name and a default set of parameters which may be overridden with CKTmodParam. The *type* of the model is the identification of which of the SPICE3 device implementations this model is to use. The types are identified by their zero based index in the *devices* array of the *IFsimulator* structure, where they should be located by name. The model structure is initialized to an all zero bit pattern with a size given by the DEVmodSize field of the SPICEdev structure for the specified device type. This implementation checks for model name duplications even through the front end interface does not call for this level of checking.

2.1.1.18. CKTmodParam

```
int CKTmodParam(ckt, model, parm, value, selector)
    GENERIC *ckt;          /* circuit to operate on */
    GENERIC *model;        /* model being modified */
    int parm;              /* parameter number */
    IFvalue *value;        /* parameter value */
    IFvalue *selector;     /* sub parameter selector */
```

This function is equivalent to CKTparam, but operates on models instead of instances. The specified parameter is updated to have to the specified value by a device type specific subroutine provided by the device implementor. As with CKTparam, the *selector* parameter is not yet used, but is provided to comply with the front end interface specification and for possible future needs.

2.1.1.19. CKTmodAsk

```
int CKTmodAsk(ckt, model, parm, value, selector)
    GENERIC *ckt;       /* circuit to operate on */
    GENERIC *model;     /* model being queried */
    int parm;           /* parameter number */
    IFvalue *value;     /* parameter value */
    IFvalue *selector;  /* sub parameter selector */
```

This function performs the equivalent function to CKTask but for models instead of instances. Unlike CKTask though, the selector parameter is not yet used by any device, although it is passed through for possible future use.

2.1.1.20. CKTfndMod

```
int CKTfndMod(ckt, type, model, modName)
    GENERIC *ckt;       /* circuit to operate on */
    int *type;          /* type of the model */
    GENERIC **model;    /* model pointer on return */
    IFuid modName;      /* name of model to find */
```

This routine is similar to CKTfndDev in that it is used to find a model pointer given a minimal amount of information about the model itself. An efficient front end will maintain its own data structures and not depend on this function, since CKTfndMod must perform a relatively inefficient search through the model structures.

2.1.1.21. CKTdltMod

```
int CKTdltMod(ckt, model)
    GENERIC *ckt;       /* circuit to operate on */
    GENERIC *model;     /* model being deleted */
```

This function is provided for compatibility with the front end interface and future expansion, but is not implemented yet. The current action is to return E_UNSUPP as permitted by the front end to indicate that SPICE3 does not yet have the capability of deleting a model from the circuit. The function should not be hard to implement given the existing structure and the constraints on the function required by the front end, but it is not useful until the CKTdltInst function has been implemented.

2.1.1.22. CKTnewTask

```
int CKTnewTask(ckt, task, taskName)
    GENERIC *ckt;       /* circuit to operate on */
    GENERIC **task;     /* task structure to be created */
    IFuid taskName;     /* name to give the task */
```

CKTnewTask allows the front end to create a task structure which can describe a set of analyses to be performed by the simulator as a single step. This function allocates the necessary structure and initializes it to default values, but does not insert any analysis requests in the new task, although it does create the "options" analysis with default values.

2.1.1.23. CKTnewAnal

```
int CKTnewAnal(ckt, type, name, anal, task)
    GENERIC *ckt;       /* circuit to operate on */
    int type;           /* type of analysis being requested */
    IFuid name;         /* name of analysis */
    GENERIC **anal;     /* returned analysis pointer */
    GENERIC *task;      /* task this analysis should be part of */
```

CKTnewAnal adds a new analysis to an existing task. The new analysis type is determined within the front end by searching through the *analyses* array within the IFsimulator structure and using the index of the desired analysis as the *type* value passed to CKTnewAnal. The *anal* pointer returned can then be used to add parameters to further describe the characteristics of the analysis.

2.1.1.24. CKTsetAnalPm

```
int CKTsetAnalPm(ckt, anal, parm, value, selector)
    GENERIC *ckt;       /* circuit to operate on */
    GENERIC *anal;      /* analysis being modified */
    int parm;           /* parameter number */
    IFvalue *value;     /* parameter value */
    IFvalue *selector;  /* sub parameter selector */
```

CKTsetAnalPm is used to set parameters further detailing an analysis to be performed in the same way as CKTparam further describes an instance. Again, since the code does not have any knowledge of the analysis data structures at this level, a lower level routine is called.

2.1.1.25. CKTaskAnalQ

```
int CKTaskAnalQ(ckt, anal, parm, value, selector)
    GENERIC *ckt;       /* circuit to operate on */
    GENERIC *anal;      /* analysis being queried */
    int parm;           /* parameter number */
    IFvalue *value;     /* parameter value */
    IFvalue *selector;  /* sub parameter selector */
```

CKTaskAnalQ allows the front end to query the simulator for details of an analysis in the same manner as CKTask querys an instance.

2.1.1.26. CKTfndAnal

```
int CKTfndAnal(ckt, analIndex, anal, name, task, taskName)
    GENERIC *ckt;       /* circuit to operate on */
    int *analIndex;     /* Type of analysis */
    GENERIC **anal;     /* analysis pointer to return */
    IFuid name;         /* name of analysis */
    GENERIC *task;      /* pointer to task containing analysis*/
    IFuid taskName;     /* name of task */
```

CKTfndAnal is provided for a simple front end to search through an existing task to find a specific analysis request given the name of the analysis. This routine is not overly efficient, but since most tasks will be small, and an intelligent front end should never need to use this routine anyway, the cost of such an implementation is negligible.

2.1.1.27. CKTfndTask

```
int CKTfndTask(ckt, task, taskName)
    GENERIC *ckt;       /* circuit to operate on */
    GENERIC **task;     /* task pointer to return */
    IFuid taskName;     /* name of task */
```

CKTfndTask is a procedure to be used by a very simple front end to find a pointer to a task given nothing but the name of the task and the circuit it is a task in. This function is not implemented in SPICE3, since SPICE3 does not maintain a list of the tasks associated with a circuit, and thus always returns E_UNSUPP.

2.1.1.28. CKTdelTask

```
int CKTdelTask(ckt, task)
    GENERIC *ckt;       /* circuit to operate on */
    GENERIC *task;      /* task being deleted */
```

CKTdelTask is used to delete a task and all of the analyses it is composed of, thus freeing up the memory its structures occupy. Note that these are only the task structures that are freed not the circuit structures since more than one task may reference the same circuit.

2.1.1.29. CKTdoJob

```
int CKTdoJob(ckt, reset, task)
    GENERIC *ckt;       /* circuit to operate on */
    int reset;          /* restart or continue? */
    GENERIC *task;      /* task to perform */
```

This is the master routine for actually running SPICE jobs. CKTdoJob takes a task and performs all of the analyses described in it. CKTdoJob is free to re-arrange the analyses within the task at its discretion, combining them, sorting them for reduced computation time, or other optimizations desired. In the case of this routine, it extracts the values set in the "options" analysis and sets them in the circuit structure. Then, depending on the value of the restart parameter and the state saved in the task structure, it either jumps to a specific analysis or steps through the analyses in its preferred

order. Note that many of the analyses are special cased in this routine, such as extracting any sensitivity analysis and setting it up to be performed during the other analyses as appropriate.

2.1.2. General glue routines

These routines are used to hold the entire package together. They lean heavily toward routines which primarily step through the corresponding device dependent subroutines, calling those for which instances have been defined or the specific one needed for the operation in question. In many cases, some of the higher level operations have been added to them as a convenience, such as having the routine which loads the matrix clear it first.

2.1.2.1. CKTacDump

```
int CKTacDump(ckt, freq, plot)
    CKTcircuit *ckt;    /* circuit to operate on */
    double freq;        /* Frequency at which analysis was done */
    GENERIC *plot;      /* plot pointer returned by OUTpBeginPlot */
```

This routine is used by the ac analysis phase of SPICE3 to output the computed results. CKTacDump will properly format the contents of the CKTrhsOld and CKTirhsOld vectors and pass them to the output routine along with the current frequency as the real-valued independent variable.

2.1.2.2. CKTacLoad

```
int CKTacLoad(ckt)
    CKTcircuit *ckt;    /* circuit to operate on */
```

This is a simple control routine which calls the DEVacLoad subroutine for every device type for which the subroutine exists and instances of the device type exist. This should perform the specified ac loading operation on all devices in the circuit for which such an operation has been defined. As a convenience, before the matrix is loaded, it will first be cleared.

2.1.2.3. CKTaccept

```
int CKTaccept(ckt)
    CKTcircuit *ckt;    /* circuit to operate on */
```

This routine is used by the transient code when it has been determined that a particular timepoint is to be accepted, thus allowing each device to perform any once per timepoint cleanup or preparation for the next timepoint. This is primarily intended for the generation of breakpoints by sources and transmission lines which should be done exactly once at a timepoint, and only if the timepoint is to be accepted, but can be used for anything else. As with all other routines that iterate through all devices, this will call the corresponding device type specific routine, DEVaccept, for each device type for which that routine has been defined and of which there are instances in the current circuit.

2.1.2.4. CKTconvTest

```
int CKTconvTest(ckt)
    CKTcircuit *ckt;    /* circuit to operate on */
```

CKTconvTest loops through the device specific convergence test routines until one of them signals non-convergence by incrementing the member CKTnoncon in the CKTcircuit structure, or all of them have been called.

2.1.2.5. CKTfndBr

```
int CKTfndBranch(ckt, name)
    CKTcircuit *ckt;    /* circuit to operate on */
    IFuid name;         /* name of device to find branch of */
```

CKTfndBranch is used by SPICE3 to find the branch equations associated with other devices to allow them to act as controls. The CKTfndBranch routine will call every device type specific DEVfindBranch routine for which there are corresponding devices in existence, each of which will check for the existence of a device with the given IFuid as a name and return the equation number of

the corresponding current.

2.1.2.6. CKTic

```
int CKTic(ckt)
    CKTcircuit *ckt;    /* circuit to operate on */
```

CKTic is used to establish the initial conditions before performing a dc analysis. The circuit's right hand side vector is cleared and then loaded with the nodeset and initial condition values given for nodes, with initial condition values taking precedence over nodeset values. If the ''uic'' keyword has been given, each device's DEVsetic function is then called to allow devices which did not have an instance specific initial condition given to pick up the node initial conditions. The right hand side vector is left loaded with the .nodeset and .ic values.

2.1.2.7. CKTload

```
int CKTload(ckt, restart)
    GENERIC *ckt;    /* circuit to operate on */
```

In many ways, CKTload is the heart of SPICE3. At each iteration of the dc, transient, and operating point analyses, CKTload is called on to clear the matrix and right hand side, then evaluate every device in the circuit. The devices are evaluated by calls to the device type specific DEVload functions, in which each device is expected to be evaluated and the proper additions made to the right hand side vector and the circuit matrix. After all of the devices have been loaded, the nodeset and initial condition additions may be set, depending on the mode in effect at the time, by adding a 1OMEGA resistance from each node to ground and connecting a constant current source of 1OMEGA times the desired voltage from ground to the node. Finally, cumulative timing statistics are maintained for the load operation.

2.1.2.8. CKTpzLoad

```
int CKTpzLoad(ckt, s, type)
    CKTcircuit *ckt;    /* circuit to operate on */
    SPcomplex *s;       /* complex multiplier */
    int type;           /* type of load */
```

CKTpzLoad is the primary driving routine for the pole zero analysis device evaluation. Each device's pole zero evaluation routine will be called to evaluate the device, thus loading the complex sparse matrix and right hand side. Following this, depending on *type*, column operations are performed in the resulting matrix structure before the matrix is solved. The possible values of type and their meaning are:

Type	operation
1	clear row and column I
2	clear row and column I-G
3	clear row and column I
4	clear row and column J-K
5	clear row and column I-G
6	no-op
7	no-op
8	illegal type value
9	illegal type value
10	illegal type value
11	clear row and column J
12	clear row and column J
13	clear row and column J-K
14	clear row and column J-K
15	clear row and column J-K
16	clear row and column J-K
17	clear row and column J

2.1.2.9. CKTsenLoad

```
int CKTsenLoad(ckt)
    GENERIC *ckt;    /* circuit to operate on */
```

This is the equivalent of both the CKTload and CKTacLoad functions for sensitivity analysis. CKTsenLoad clears the sensitivity right hand side vectors and then calls all of the appropriate DEVsenLoad or DEVsenAcLoad functions.

2.1.2.10. CKTsenPrint

```
void CKTsenPrint(ckt)
   CKTcircuit *ckt;    /* circuit to operate on */
```

CKTsenPrint is a diagnostic routine used for calling all of the device type specific sensitivity diagnostic printing routines to print debugging information to the standard output.

2.1.2.11. CKTsenSetup

```
int CKTsenSetup(ckt)
   CKTcircuit *ckt;    /* circuit to operate on */
```

CKTsenSetup simply iterates through the device dependent sensitivity setup routines.

2.1.2.12. CKTsenUpdate

```
int CKTsenUpdate(ckt)
   GENERIC *ckt;    /* circuit to operate on */
```

CKTsenUpdate is used to update sensitivity information on the devices, and simply iterates through all of the device dependent sensitivity update routines.

2.1.2.13. CKTsetup

```
int CKTsetup(ckt)
   CKTcircuit *ckt;    /* circuit to operate on */
```

CKTsetup iterates through the device specific setup routines, ensures that the subsidiary setup routines such as SENsetup and NIreinit are called correctly, and allocates the CKTstate vectors to ensure that the circuit is properly prepared for simulation. The setup routine should not be called repeatedly, since it allocates all of the memory needed, but should be called exactly once after the circuit has been defined. Repeatable portions of the initialization are called from the CKTtemp routine, which must be called after CKTsetup.

2.1.2.14. CKTsetup

```
int CKTpzSetup(ckt)
    CKTcircuit *ckt;    /* circuit to operate on */
```

CKTpzSetup iterates through the device specific pole-zero setup routines and allocates the CKTstate vectors to ensure that the circuit is properly prepared for simulation. The pzSetup routine is designed to be called specifically to set up for a pole-zero analysis, which requires that voltage sources be treated specially. Before this routine is called, NIdestroy must first be called to free up the structures this routine will re-allocate. For the most part, the actions performed are identical to those performed by CKTsetup, with the exception of leaving out sensitivity setup operations, and calling a different set of device specific setup operations to ensure that input voltage sources are left out of the circuit.

2.1.2.15. CKTtemp

```
int CKTtemp(ckt)
    CKTcircuit *ckt;    /* circuit to operate on */
```

CKTtemp is used to iterate through the DEVtemp routines which are used for final parameter preprocessing before the simulation. CKTtemp may be called as many times as desired, and neither it nor the device routines it calls will allocate additional memory, except that used for error messages, nor will the circuit structure be modified. Any time a parameter to a device or a major circuit parameter, such as temperature, is modified, CKTtemp should be called to update all the calculations that are dependent on that parameter.

2.1.2.16. CKTtrunc

```
int CKTtrunc(ckt, timeStep)
    CKTcircuit *ckt;        /* circuit to operate on */
    double *timeStep;       /* returned new timestep */
```

CKTtrunc is the main calling point for the truncation error calculation. CKTtrunc will successively call the truncation error functions for every device which has one and set *timeStep* to the minimum timestep required by any of the devices to meed the truncation error requirement.

2.1.3. Analysis packages

Each of these sub-packages implements a single type of analysis to be performed by the overall system.

The analysis packages all have a very similar form, consisting of three routines and a table of data. The routines are those for setting parameters, querying parameters, and actually performing the analysis. This section only provides an overview and highlights of the individual routines. Further details of the structure can be found in the front end interface description. Additional work on this structure to make this the only thing to change when adding analyses is still needed, currently the analysis routine itself must be called explicitly from CKTdoJob.

2.1.3.1. ac analysis

The ac analysis computes a dc operating point and all of the necessary small signal parameters, then sweeps all ac sources through a set of frequencies, computing the ac small signal response at each of those frequencies.

2.1.3.1.1. ACsetParm

```
int ACsetParm(ckt, anal, which, value)
    CKTcircuit *ckt;    /* circuit to operate on */
    GENERIC *anal;      /* Analysis to modify parameter of */
    int which;          /* parameter to modify */
    IFvalue *value;     /* new value of parameter */
```

ACsetParm stores the parameter values as described. Note that the *dec*, *oct*, and *lin* keywords are mutually exclusive, with the last one specified overriding the others. The source file containing this file also contains the definition of the ACinfo structure and its components describing the ac analysis itself.

2.1.3.1.2. ACaskQuest

```
int ACaskQuest(ckt, anal, which, value)
    CKTcircuit *ckt;    /* circuit to operate on */
    GENERIC *anal;      /* Analysis to query parameter of */
    int which;          /* parameter to query */
    IFvalue *value;     /* value of parameter to return */
```

Since *dec*, *oct*, and *lin* are mutually exclusive, querying for them returns one for whichever one is set, zero for the others, and zero for all of them if none have been set.

2.1.3.1.3. ACan

```
int ACan(ckt, restart)
    CKTcircuit *ckt;    /* circuit to operate on */
    int restart;        /* start over or resume? */
```

ACan is a very simple routine, most of the complexity is for handling the boundary cases such as a linear step of zero or a logarithmic step over a one point range. A tolerance on the stopping criteria is required due to rounding error or a point can be missed or an extra point accidentally added. In the case of stopping the analysis, which can be done after the analysis at any frequency is completed, the most recently completed frequency is saved in the ACsaveFreq field of the ac analysis structure, and a non-zero value in this location during startup will trigger a resume from the next frequency point.

2.1.3.2. dc operating point analysis

The dc operating point analysis provides a simple dc analysis with all capacitances open-circuited and all inductances shorted. The result is both output as an analysis and left in the CKTrhsOld vector for future use by other analyses. This analysis does not require any parameters, and thus can be called by other analyses as a preliminary step to their work.

2.1.3.2.1. DCOaskQuest

```
int DCOaskQuest(ckt, anal, which, value)
    CKTcircuit *ckt;    /* circuit to operate on */
    GENERIC *anal;      /* Analysis to query parameter of */
    int which;          /* parameter to query */
    IFvalue *value;     /* value of parameter to return */
```

DCOaskQuest always returns E_BADPARM, since the dc operating point analysis does not support any options. The routine itself is provided to satisfy the analysis interface requirement.

2.1.3.2.2. DCOsetParm

```
int DCOsetParm(ckt, restart)
    CKTcircuit *ckt;    /* circuit to operate on */
    GENERIC *anal;      /* Analysis to modify parameter of */
    int which;          /* parameter to modify */
    IFvalue *value;     /* new value of parameter */
```

There are no parameters to the dc operating point analysis, so this function returns E_BADPARM on all calls.

2.1.3.2.3. DCop

```
int DCop(ckt, restart)
    CKTcircuit *ckt;    /* circuit to operate on */
    int restart;        /* start over or resume? */
```

Although the dc operating point should be a relatively simple analysis, DCop is fairly complicated. In addition to a simple Newton-Raphson iteration to solve the circuit, code is also included to handle the complicated cases where the simple Newton-Raphson iteration does not converge or converges too slowly by using G_{min} stepping and source stepping as backup techniques.

2.1.3.3. dc transfer curve analysis

The dc transfer curve analysis allows the user to sweep a voltage or current source through a set of values and obtain dc analysis results at each of the points. Additionally, a set of nested sweeps

may be done, with each more deeply nested sweep varying throughout its entire range for each point in the outer sweep, producing families of curves.

2.1.3.3.1. DCTaskQuest

```
int DCTaskQuest(ckt, anal, which, value)
    CKTcircuit *ckt;    /* circuit to operate on */
    GENERIC *anal;      /* Analysis to query parameter of */
    int which;          /* parameter to query */
    IFvalue *value;     /* value of parameter to return */
```

DCTaskQuest can return the values accepted by DCTsetParm, and an additional value "maxnest" which indicates the maximum nesting level used, with zero indicating no sweep specified, one indicating no nesting, just a single sweep, and two indicating one sweep nested within another. Note that this is not exact, but based on the assumption that the parameters corresponding to source one will always be set before source two. If no values are set for source one, but values are given for source two, "maxnest" will still indicate a nesting level of two.

2.1.3.3.2. DCTsetParm

```
int DCTsetParm(ckt, anal, which, value)
    CKTcircuit *ckt;    /* circuit to operate on */
    GENERIC *anal;      /* Analysis to modify parameter of */
    int which;          /* parameter to modify */
    IFvalue *value;     /* new value of parameter */
```

This routine assumes only two levels of nesting in the loop, although the actual analysis routine can handle more. It is also assumed that the front end will specify all of the parameters necessary for a given level without leaving out values.

2.1.3.3.3. DCtrCurv

```
int DCtrCurv(ckt, restart)
    GENERIC *ckt;    /* circuit to operate on */
    int restart;     /* start over or resume? */
```

DCtrCurv is fairly straightforward, except for the stepping rules which have to handle positive and negative steps, rounding error, and missing nesting levels. Although previous versions of SPICE have only supported two levels of nesting, this routine supports an arbitrary level of nesting should such a capability be desired in the future and the input system modified to support it. The only other unusual feature of this code is the attempt made to use MODEINITPRED to predict a value in the transfer curve and the placement of the voltage differences in the array of time deltas to allow such predictions. These predictions frequently produce much faster convergence, although a standard dc solution is performed if any difficulties are encountered.

2.1.3.4. Transfer function analysis

This analysis allows for a small signal dc transfer function to be computed.

2.1.3.4.1. TFanal

```
int TFanal(ckt, restart)
    GENERIC *ckt;    /* circuit to operate on */
    int restart;     /* start over or resume? */
```

TFanal performs an actual transfer function analysis. The code includes a partial copy of the DCop code without the output calls to find the operating point. The code to find the actual transfer function is fairly simple, with most of the actual code going into output name generation.

2.1.3.4.2. TFaskQuest

```
int TFaskQuest(ckt, anal, which, value)
    CKTcircuit *ckt;    /* circuit to operate on */
    GENERIC *anal;      /* Analysis to query parameter of */
    int which;          /* parameter to query */
    IFvalue *value;     /* value of parameter to return */
```

TFaskQuest is used to query the parameters of a transfer function analysis. This function is only a shell, with the actual query code still to be written as the need for it arises.

2.1.3.4.3. TFsetParm

```
int TFsetParm(ckt, anal, which, value)
    CKTcircuit *ckt;    /* circuit to operate on */
    GENERIC *anal;      /* Analysis to modify parameter of */
    int which;          /* parameter to modify */
    IFvalue *value;     /* new value of parameter */
```

TFsetParm simply stores the specified parameters into the transfer function analysis structure. No checking is done to prevent specification of both an output source and an output node pair, but whichever is specified last will override.

2.1.3.5. Transient analysis

The transient analysis is the largest and most complicated simulation currently supported by SPICE3. This analysis permits the time dependent behavior of the circuit to be analyzed.

2.1.3.5.1. TRANaskQuest

```
int TRANaskQuest(ckt, anal, which, value)
    CKTcircuit *ckt;    /* circuit to operate on */
    GENERIC *anal;      /* Analysis to query parameter of */
    int which;          /* parameter to query */
    IFvalue *value;     /* value of parameter to return */
```

TRANaskQuest allows the front end to query any of the parameters to a transient analysis.

2.1.3.5.2. TRANsetParm

```
int TRANsetParm(ckt, anal, which, value)
    CKTcircuit *ckt;    /* circuit to operate on */
    GENERIC *anal;      /* Analysis to modify parameter of */
    int which;          /* parameter to modify */
    IFvalue *value;     /* new value of parameter */
```

TRANsetParm allows the front end to set the parameters needed to perform a transient analysis.

2.1.3.5.3. DCtran

```
int DCtran(ckt, restart)
    GENERIC *ckt;     /* circuit to operate on */
    int restart;      /* start over or resume? */
```

This routine actually performs the transient analysis. The algorithms are adequately explained elsewhere, and this routine contains relatively straightforward implementations of them. The overall code is moderately complicated since so many of the algorithms come together and interact here, particularly the stop and restart code which takes up much of the first third of the function.

2.1.3.6. Pole-zero analysis

This analysis computes the poles and zeroes of the ac small signal transfer function.

2.1.3.6.1. PZan

```
int PZan(ckt, restart)
    GENERIC *ckt;     /* circuit to operate on */
    int restart;      /* start over or resume? */
```

PZan performs the actual pole-zero analysis itself using Muller's method.

2.1.3.6.2. PZaskQuest

```
int PZaskQuest(ckt, anal, which, value)
    CKTcircuit *ckt;     /* circuit to operate on */
    GENERIC *anal;       /* Analysis to query parameter of */
    int which;           /* parameter to query */
    IFvalue *value;      /* value of parameter to return */
```

PZaskQuest provides access from the front end to all of the internal parameters of the pole-zero analysis.

2.1.3.6.3. PZsetParm

```
int PZsetParm(ckt, anal, which, value)
    CKTcircuit *ckt;    /* circuit to operate on */
    GENERIC *anal;      /* Analysis to modify parameter of */
    int which;          /* parameter to modify */
    IFvalue *value;     /* new value of parameter */
```

PZsetParm sets all of the parameters used to control the pole-zero analysis.

2.1.3.7. Sensitivity analysis

The sensitivity analysis is not truly a separate analysis, but is a set of modifications to other analyses which extracts sensitivity information from them during the course of their operation. Since it is logically a separate operation and should have no effect on normal operation when no sensitivity options have been selected, the code to perform the sensitivity analysis is controlled by a separate set of structures which are independent of the other analysis structures.

2.1.3.7.1. SENaskQuest

```
int SENaskQuest(ckt, anal, which, value)
    CKTcircuit *ckt;    /* circuit to operate on */
    GENERIC *anal;      /* Analysis to query parameter of */
    int which;          /* parameter to query */
    IFvalue *value;     /* value of parameter to return */
```

This is the routine which should provide data about the sensitivity analyses requested. At this time, the routine is just a framework that does not return any values.

2.1.3.7.2. SENsetParm

```
int SENsetParm(ckt, restart)
    CKTcircuit *ckt;    /* circuit to operate on */
    GENERIC *anal;      /* Analysis to modify parameter of */
    int which;          /* parameter to modify */
    IFvalue *value;     /* new value of parameter */
```

This routine actually sets up the parameters in the sensitivity structures that the simulator will later use to perform the sensitivity analysis in convert with the other analyses being performed.

2.1.3.8. Analysis control

Although not strictly an analysis, the overall control of various parameters affecting the analysis operation, such as numerical tolerances, is treated as another analysis with the same interface as the standard analysis routines, but with variables having values global to the task the analysis is in.

2.1.3.8.1. CKTsetOpt

```
int CKTsetOpt(ckt, anal, opt, val)
    CKTcircuit *ckt;    /* circuit to operate on */
    GENERIC *anal;      /* Analysis/task to modify parameter of */
    int which;          /* parameter to modify */
    IFvalue *value;     /* new value of parameter */
```

CKTsetOpt is used to set the values of parameters to the "options" analysis in a task. The "options" analysis is always performed first in any task, and replaces default system parameters with those specified in the "options" analysis. The parameters which can be set are the numerical control parameters which SPICE2 placed on a ".options" card, and those additional parameters of SPICE3 which similarly apply to the entire circuit. The OPTinfo data structure definition and initialization is also kept in the same source file for convenience.

2.1.3.8.2. CKTacct

```
int CKTacct(ckt, anal, which, val)
    GENERIC *ckt;       /* circuit to operate on */
    GENERIC *anal;      /* pointer to the options analysis */
    int which;          /* Which option/accounting value do we want*/
    IFvalue *val;       /* value to return */
```

This is the routine used for querying overall information about the circuit. Initially used for accounting information, thus the strange name, it is the opposite of CKTsetOpt for setting option values in the circuit. Note that since the option values are not transferred into the circuit itself until the task is actually started, this function will not return useful data until the task has been started even though options may have been set in the options analysis. This behavior permits different tasks associated with the same circuit to have different options since the options themselves associate with the

task, but this function returns data about the actual current state of the circuit.

2.1.4. Utility routines

These routines are used primarily as simple utilities to manipulate the data structures in SPICE3. These remove the details of the small substructures from the rest of the code as well as making them easier to use.

2.1.4.1. CKTclrBreak

```
int CKTclrBreak(ckt)
    CKTcircuit *ckt;    /* circuit to operate on */
```

This subroutine is used to clear a breakpoint. When called, it will delete the first breakpoint in the table for the specified circuit, leaving the breakpoint table otherwise unchanged.

2.1.4.2. CKTdump

```
int CKTdump(ckt, ref, plot)
    CKTcircuit *ckt;    /* circuit to operate on */
    double ref;         /* value of the reference variables */
    GENERIC *plot;      /* plot pointer returned by OUTpBeginPlot */
```

CKTdump is used by all of the routines in SPICE3 which produce a real valued solution vector in the CKTrhsOld vector. *Ref* is the value of the reference vector which is varying the most frequently, *plot* is the plot identifier, and *ckt* identifies the circuit from which the output vectors are to be taken. The appropriate real-valued solution is constructed and passed to the output interface routine.

2.1.4.3. CKTlinkEq

```
int CKTlinkEq(ckt, node)
    CKTcircuit *ckt;    /* circuit to operate on */
    CKTnode *node;      /* Node structure to be linked */
```

CKTlinkEq links an equation into the circuit's node data structures. The node data structures require an IFuid in them as the node "name" , but since the front end wants to keep a pointer in the

the structure associated with an object when it stores the name away, it is necessary to allocate that structure before calling IFuid. For the internal procedures for generating various kinds of equations in the circuit, the procedure has thus been broken into two parts, creation of the node structure, and linking of the structure into the overall node list. CKTlinkEq provides the latter half of that capability.

2.1.4.4. CKTmkCur

```
int CKTmkCur(ckt, node, basename, suffix)
    CKTcircuit *ckt;      /* circuit to operate on */
    CKTnode **node;       /* node to create and return */
    IFuid basename;       /* base UID of new name to create */
    char *suffix;         /* suffix to add to the name */
```

CKTmkCur is used to create a new circuit equation corresponding to a current in the circuit. All necessary calls are made to ensure that the IFuid needed to identify the node is properly registered with the front end and the node is initialized as a current type equation.

2.1.4.5. CKTmkNode

```
int CKTmkNode(ckt, node)
    CKTcircuit *ckt;      /* circuit to operate on */
    CKTnode **node;       /* pointer for new node */
```

CKTmkNode allocates an empty node structure. The node structure must be filled in with the IFuid and type of node, and then be linked into the node linked list by an appropriate call to CKTlinkEq. This is usually done automatically by CKTmkVolt or CKTmkCur instead of using CKTmkNode directly.

2.1.4.6. CKTmkVolt

```
int CKTmkVolt(ckt, node, basename, suffix)
    CKTcircuit *ckt;      /* circuit to operate on */
    CKTnode **node;       /* node to create and return */
    IFuid basename;       /* base UID of new name to create */
    char *suffix;         /* suffix to add to the name */
```

CKTmkVolt works exactly as CKTmkCur does, but allocates a node voltage type equation.

2.1.4.7. CKTnames

```
int CKTnames(ckt, numNames, nameList)
    CKTcircuit *ckt;       /* circuit to operate on */
    int *numNames;         /* return: number of names output */
    IFuid **nameList;      /* return:  vector of names */
```

CKTnames is used to generate an IFuid list in the form needed by the interface package's OUTpBeginPlot function. The *nameList* which is returned has been malloc'ed, and thus should be free'd after use.

2.1.4.8. CKTnum2nod

```
CKTnode *CKTnum2nod(ckt, node)
    CKTcircuit *ckt;    /* circuit to operate on */
    int node;           /* number of equation */
```

CKTnum2nod is used to convert a node number to a node structure on which further operations may be performed. This is not a very efficient routine as it must linearly search a linked list, but is available where such a conversion must be performed.

2.1.4.9. CKTpModName

```
int CKTpModName(parm, val, ckt, type, name, model)
    char *parm;             /* the name of the parameter to set */
    IFvalue *val;           /* parm union containing value */
    CKTcircuit *ckt;        /* circuit model is in */
    int type;               /* type of model */
    IFuid name;             /* name of the model */
    GENmodel **model;       /* pointer to the model */
```

This is a routine used by devices which are implemented by creating instances of other devices to replace themselves. This routine allows such a device to readily find parameters in the other device's model description and set them. This function eventually uses the same model parameter setting routines exported to the front end, but contains a front end to find a parameter in the model's

description by name rather than by its id number.

2.1.4.10. CKTpName

```
int CKTpName(parm, val, ckt, dev, name, inst)
    char *parm;          /* name of the parameter */
    IFvalue *val;        /* parm union containing value */
    CKTcircuit *ckt;     /* circuit instance is in */
    int type;            /* device type */
    char *name;          /* name of the instance */
    GENinstance **inst;  /* pointer to the instance */
```

This is another routine used for modifying instances of other devices used to implement part of a new device, much as CKTpModName, but this routine modifies instance parameters instead of model parameters.

2.1.4.11. CKTsenAC

```
int CKTsenAC(ckt)
    CKTcircuit *ckt;    /* circuit to operate on */
```

CKTsetAC is the main driver routine for the ac sensitivity calculations.

2.1.4.12. CKTsenComp

```
int CKTsenComp(ckt)
    CKTcircuit *ckt;    /* circuit to operate on */
```

CKTsenComp performs the actual sensitivity computations.

2.1.4.13. CKTsenDCtran

```
int CKTsenDCtran(ckt)
    CKTcircuit *ckt;    /* circuit to operate on */
```

CKTsenDCtran is the main driver routine for the sensitivity calculations for dc and transient analysis.

2.1.4.14. CKTsetBreak

```
int CKTsetBreak(ckt, time)
    CKTcircuit *ckt;    /* circuit to operate on */
    double time;        /* time at which to set breakpoint */
```

CKTsetBreak is used to set a new breakpoint in the system breakpoint table. A *time* in the past will generate an internal error and a diagnostic message. A breakpoint that is within CKTminBreak of an existing breakpoint will either be dropped or replace it, with the one at the earlier time taking precedence. Otherwise, the breakpoint table will be extended and the new breakpoint added in the appropriate place to keep the breakpoint times in order.

The breakpoint table maintenance is currently performed entirely within CKTsetBreak and CKTclrBreak, and uses a very simple algorithm for the table, allocating a new table of the correct size and copying the old table every time. This algorithm can certainly be improved, but has proven adequate so far, and can be changed with no modifications elsewhere in the code.

2.1.4.15. CKTterr

```
int CKTterr(qcap, ckt, timeStep)
    int qcap;           /* offset of charge in state vector */
    CKTcircuit *ckt;    /* circuit to operate on */
    double *timeStep;   /* current time step */
```

This routine is used to calculate the truncation error in a single capacitor. *Qcap* is the offset within the set of CKTstate vectors where the charge on the capacitor can be found. The current through the capacitor must be the next item in the state vector. *TimeStep* is the limiting time found so far, and will be reduced if the capacitor in question requires a smaller timestep. This subroutine should probably be called NIterr and placed in that package, but is here for historical reasons.

2.1.4.16. CKTtypelook

```
int CKTtypelook(type)
    char  *type;    /* name of type */
```

This is a simple device type lookup function that iterates through the table of all devices supported and returns the type index number of the device type whose name is given by *type*. If the type is not in the current executable, a code of negative one is returned.

2.1.4.17. PZDCop

```
int PZDCop(ckt)
    CKTcircuit *ckt;    /* circuit to operate on */
```

This is a stripped down version of DCop designed for use within the pole-zero analysis. No calls are made to the I/O routines and no special steps are taken to improve convergence, thus making G_{min} and source stepping unavailable.

2.1.4.18. SENdestroy

```
int SENdestroy(senPtr)
    SENstruct *senPtr;    /*sensitivity structure to destroy */
```

SENdestroy frees all suballocations of the sensitivity structure passed as an argument. The structure itself can't be freed since it is also the analysis structure and part of a task.

2.1.4.19. SENstartup

```
int SENstartup(ckt, restart)
    GENERIC *ckt;    /* circuit to operate on */
    int restart;     /* start over or resume? */
```

SENstartup is the first of the sensitivity analysis startup routines. SENstartup performs the initial structure setup, and locates the devices needed within the circuit, since they must all have been created by the time setup starts.

2.1.5. Obsolete routines

These routines are present for historical reasons. In some cases, they show alternate ways to do things, in some they show how they were done in previous versions, and in some cases, they provide

debugging capabilities that a program maintainer may wish to use.

2.1.5.1. CKTbreakDump

```
int CKTbreakDump(ckt)
    CKTcircuit *ckt;    /* circuit to operate on */
```

This is a debugging routine used for diagnosing problems with breakpoints. It will print a table giving every entry currently in the breakpoint table on the standard output device, typically the terminal. This table is not pretty, nor is it efficient or useful for normal operations, but it can be indispensable for tracking down mysterious breakpoint troubles.

2.1.5.2. CKTmapNode

```
int CKTmapNode(ckt, node, name)
    GENERIC *ckt;       /* circuit to operate on */
    GENERIC **node;     /* node to be returned */
    IFuid name;         /* name of node */
```

CKTmapNode is a version of CKTmkVolt that first looks through the entire list of existing nodes to find out of the node already exists before creating it. This function is much less efficient than CKTmkVolt due to the need to search through the entire list of circuit nodes, and is no longer used since the front end is now required to perform the calls to the interface package's newNode function exactly once for each node, thus making the search unnecessary.

2.1.5.3. CKTnewEq

```
int CKTnewEq(ckt, node, name)
    GENERIC *ckt;       /* circuit to operate on */
    GENERIC **node;     /* new node name */
    IFuid name;         /* name node should have */
```

This is an obsolete routine used to generate a circuit equation. This was formerly used in the V1.0 front end to simulator interface, but has been replaced in that context by the new CKTnewNode function which uses the primitives CKTmkNode and CKTlinkEq instead of attempting to do all the

work itself. Instead of using CKTnewEq, all parts of SPICE3 now use the functions CKTmkVolt or CKTmkCur.

2.1.5.4. CKTnodName

```
IFuid CKTnodName(ckt, nodeNum)
    CKTcircuit *ckt;    /* circuit to operate on */
    int nodeNum;        /* Name of node to get info about */
```

CKTnodName is a routine used to return the "name" of an equation given its equation number. This function is intended only for debugging output, as it is quite inefficient, having to search down a linked list, and returning a token which can still only be interpreted by the front end package.

2.1.5.5. SPerror

```
int SPerror(ckt, restart)
    GENERIC *ckt;    /* circuit to operate on */
    int restart;     /* start over or resume? */
```

SPerror is the predecessor of INPerror. SPerror is now obsolete since the front end specifies the errors associated with the various codes, and the simulator no longer needs to provide translations to the messages.

2.1.5.6. CKTdltAnal

```
int CKTdltAnal(ckt, anal, task)
    GENERIC *ckt;    /* circuit to operate on */
    GENERIC *anal;   /* Analysis to delete */
    GENERIC *task;   /* Task analysis is in */
```

CKTdltAnal is an unimplemented, and now obsolete function used by the V1.0 front end interface to delete an analysis from a task. The routine simply returns the error code E_UNSUPP to indicate that the delete analysis function is not implemented yet. The newer version of the front end interface does not have to capability of deleting a single analysis, so the full code to delete the analysis has never been implemented.

2.2. DEV

Descriptions of the device package are not provided here. A detailed description of the interface between the simulator and each device is provided in the device to simulator interface chapter. A detailed description of each of the models used for the various devices is beyond the scope of this dissertation.

2.3. NI

The NI package contains the numerical algorithms by SPICE. Originally containing only Numerical Integration routines, the package was expanded to encompass all of the major numerical routines, including the Newton-Raphson iteration loops.

2.3.1. NIacIter

```
int NIacIter(ckt)
    CKTcircuit *ckt     /* circuit to operate on */
```

NIacIter loads the complex circuit matrix with the circuit at the frequency currently set in the CKTcircuit structure passed, then performs the L-U decomposition and solve, yielding the AC solution. Despite what the name implies, there is no iteration in this routine, it is simply named for its functional correspondence to NIiter.

2.3.2. NIcomCof

```
void NIcomCof(ckt)
    CKTcircuit *ckt     /* circuit to operate on */
```

NIcomCof computes the timestep, order, and integration method dependent terms used in the numerical integration formulas. In addition to the corrector coefficients computed by the corresponding routine in SPICE2, this routine also computes the predictor coefficients for use in the predictor-corrector technique.

2.3.3. NIconvTest

```
int NIconvTest(ckt)
    CKTcircuit *ckt     /* circuit to operate on */
```

This function performs the node and current equation convergence tests, and in the new arrangement of device convergence testing, calls CKTconvTest to perform the necessary per device convergence tests.

2.3.4. NIdestroy

```
void NIdestroy(ckt)
    CKTcircuit *ckt     /* circuit to operate on */
```

NIdestroy is used to free all of the data structures allocated by the numeric package and the sparse matrix sub-package.

2.3.5. NIinit

```
int NIinit(ckt)
    CKTcircuit *ckt     /* circuit to operate on */
```

This is the initialization routine for the numeric package. Since the numeric data structure has been absorbed into the CKTcircuit structure, it does not have to be allocated, but it does need to be initialized, and sub-packages such as the sparse matrix package must be called to perform their initialization.

2.3.6. NIintegrate

```
int NIintegrate(ckt, geq, ceq, cap, qcap)
    CKTcircuit *ckt     /* circuit to operate on */
    double *geq;        /* equivalent conductance */
    double *ceq;        /* equivalent current source */
    double cap;         /* capacitance */
    int qcap;           /* offset in state vector of charge */
```

This routine performs the numerical integration required for an individual energy storage device. The integration coefficients computed by NIcomCof and stored in the CKTcircuit structure and the capacitance of the capacitor in question, along with the current charge and previous charge and current values are used to compute the new current and the equivalent circuit current source and conductance values.

2.3.7. NIiter

```
int NIiter(ckt, maxIter)
    CKTcircuit *ckt     /* circuit to operate on */
    int maxIter;        /* maximum number of iterations to attempt */
```

NIiter is the main subroutine for performing the dc and transient Newton-Raphson iterations. This routine maintains a state machine to manage the various modes the load routines support, returning only when the state machine attains the proper state for exit or if the circuit fails to converge.

2.3.8. NIpzMuller

```
int NIpzMuller(ckt, ptrlistptr, type)
    CKTcircuit *ckt     /* circuit to operate on */
    root **ptrlistptr;  /* list of the roots already found */
    int type;           /* the type of analysis step being performed */
```

NIpzMuller provides an implementation of Muller's method of finding the roots of a complex polynomial.

2.3.9. NIpzSolve

```
int NIpzSolve(ckt, s, listptr, f, type, power)
    CKTcircuit *ckt     /* circuit to operate on */
    SPcomplex *s;       /* Frequency at which to solve */
    root *listptr;      /* list of roots already found */
    SPcomplex *f;       /* mantissa of answer */
    int type;           /* type of pz analysis in progress */
    int *power;         /* power of 10 to multiply mantissa by */
```

NIpzSolve performs the main step of the pole-zero analysis. The determinant of the parse matrix at frequency *s* is computed and divided by the product of the terms s–root*i* where the roots are in the list pointed to by *listptr*.

2.3.10. NIreinit

```
int NIreinit(ckt)
   CKTcircuit *ckt     /* circuit to operate on */
```

This function is used to perform any additional initialization required by the numeric package which can not be performed before the setup phase of the analysis is complete. The vectors which are dependent on the size of the matrix, such as the right hand side vector, are allocated here.

2.3.11. NIsenReinit

```
int NIsenReinit(ckt)
   CKTcircuit *ckt     /* circuit to operate on */
```

NIsenReinit allocates all of the additional memory required by the sensitivity package that is dependent on the size of the circuit, and thus can not be allocated at the time the sensitivity analysis is initialized.

2.4. Sparse Matrix Package

The SMP or Sparse Matrix Package is used by SPICE3 to manipulate the sparse matrices generated by the application of the modified nodal analysis algorithm to the circuit. This package is NOT a general sparse matrix package, but is intended specifically to handle the problems of solving MNA problems. All of the function descriptions below assume that the header file "SMPdefs.h" has already been included in the current source file. This header file not only declares all of the data types involved, but also provides complete declarations of all of the functions involved and will include any subsidiary header files which are needed.

2.4.1. SMPnewMatrix

```
int SMPnewMatrix(matrix)
    SMPmatrix **matrix;
```

SMPnewMatrix allocates a new set of data structures for a sparse matrix to be managed by the *SMP* package. *Matrix* is set to point to the newly allocated structure. No use is made of the current value of matrix. SMPnewMatrix returns 0 for success, E_NOMEM for failure to allocate the required space.

The allocated matrix will consist of only the basic SMPmatrix structure initialized as necessary. No elements are created, but a token allocation of each of the other dynamically allocated substructures is made since most *malloc* functions do not perform correctly when given a null pointer or a request for an allocation of size zero.

2.4.2. SMPdestroy

```
void SMPdestroy(matrix)
    SMPmatrix *matrix;
```

SMPdestroy dismantles and frees the data structures allocated to the sparse matrix *matrix*.

2.4.3. SMPaddElt

```
int SMPaddElt(matrix, row, col, value)
    SMPmatrix *matrix;
    int row;
    int col;
    double value;
```

SMPaddElt adds the specified value *value* to the sparse matrix *matrix* at the position (*row*, *col*). If necessary, the new element is allocated automatically. Returns 0 for success, E_NOMEM for insufficient memory to allocate a new matrix element.

2.4.4. SMPclear

```
void SMPclear(matrix)
    SMPmatrix *matrix;
```

SMPclear clears to zero the real part of all entries in the specified sparse matrix *matrix*.

2.4.5. SMPfindElt

```
SMPelement *SMPfindElt(matrix, row, col, flag)
    SMPmatrix *matrix;
    int row;
    int col;
    int flag;
```

SMPfindElt finds the specified location in the sparse matrix *matrix* at position (*row, col*) and returns a pointer to it. If the location is not present in the matrix, it will be created if *flag* is non-zero. Returns NULL if the location is not present and *flag* is zero or if no memory is available for allocation, otherwise returns a pointer to the internal matrix structure.

Note: *Row* and *col* are interpreted as **internal** row and column numbers, not external. This function is **not** intended for general use, but is an internal function within the matrix package.

2.4.6. SMPmakeElt

```
double *SMPmakeElt(matrix, row, col)
    SMPmatrix *matrix;
    int row;
    int col;
```

SMPmakeElt returns a pointer to the double precision value field at the location (*row, col*) in the matrix *matrix*. If the location does not exist, it is added. If the matrix does not contain the specified row or column it is expanded to encompass it. If the allocation to add the element or to expand the matrix fails, NULL is returned, and the matrix may be corrupted.

2.4.7. SMPnewNode

```
int SMPnewNode(node, matrix)
    int node;
    SMPmatrix *matrix;
```

SMPnewNode makes the necessary changes in the data structures of the sparse matrix *matrix* to increase the size of the matrix by one. *Node* is taken as the external row and column number of the new row and column to be added. If the necessary memory allocation fails, the matrix will be corrupted and *SMPnewNode* will return NULL.

2.4.8. SMPgetError

```
int SMPgetError(matrix, i, j)
    SMPmatrix *matrix;
    int *i;
    int *j;
```

SMPgetError is used to query the matrix package for the location in the matrix *matrix* where the last reordering failed. *I* and *j* will be set to the external row and column numbers of the point in the matrix where the reordering algorithm gave up and declared the matrix singular.

2.4.9. SMPmatSize

```
int SMPmatSize(matrix);
    SMPmatrix *matrix;
```

SMPmatSize returns the number of rows and columns in the sparse matrix *matrix*.

2.4.10. SMPpreOrder

```
int SMPpreOrder(matrix)
    SMPmatrix *matrix;
```

SMPpreOrder performs preliminary reordering of the sparse matrix *matrix* based on knowledge of the characteristics of MNA sparse matrices. Voltage sources cause zero valued entries on the diagonal which can cause severe problems during the decomposition, but the source must also have a corresponding pair of symmetric off-diagonal entries which a simple row swap can put on the

diagonal. This routine locates these structures purely from information in the matrix and performs the necessary row swaps.

2.4.11. SMPreorder

```
int SMPreorder(matrix, pivtol, pivrel, gmin)
    SMPmatrix *matrix;
    double pivtol;
    double pivrel;
    double gmin;
```

SMPreorder performs a reordering and L-U factorization of the sparse matrix *matrix*. *Pivtol* is the minimum absolute value which is acceptable for a diagonal element in the decomposition and *pivrel* is the factor by which a pivot element may be non-optimal numerically and yet still be selected to maintain sparsity. *Gmin* will be added to each diagonal element before computation to allow the Gmin stepping convergence aid.

2.4.12. SMPluFac

```
int SMPluFac(matrix, pivtol, gmin)
    SMPmatrix *matrix;
    double pivtol;
    double gmin;
```

SMPluFac performs an L-U decomposition of the sparse matrix *matrix*. *Pivtol* is the minimum absolute value which will be accepted as a pivot element during the factorization. If a smaller pivot is found, the decomposition will be aborted and the error code E_SINGULAR returned. As an aid to Gmin stepping, the value *gmin* is added to each diagonal element in the matrix before using it in the calculations.

2.4.13. SMPsolve

```
int SMPsolve(matrix, rhs, spare)
    SMPmatrix *matrix;
    double *rhs;
    double *spare;
```

SMPsolve solves the equation Ax=b where A is a sparse matrix. *matrix* is a sparse matrix which has already been L-U factored by SMPluFac or SMPreorder. *Rhs* is the column vector b on input and is replaced by the row vector x by SMPsolve. To provide working room for vector permutation and solution, a scratch vector the same size as *rhs*, called *spare* must be supplied. The contents of *spare* will not be used, but will be destroyed. Dynamic allocation of this vector is possible, but not acceptable since this routine will appear inside the inner loop of the Newton-Raphson iteration with the same temporary space needs each time, thus making the use of an externally supplied vector preferable.

2.4.14. SMProwSwap

```
int SMProwSwap(matrix, row1, row2)
   SMPmatrix *matrix;
   int row1;
   int row2;
```

SMProwSwap swaps the two rows *row1* and *row2* in the sparse matrix *matrix*. All necessary pointers are adjusted, and the internal permutation table associated with the matrix is adjusted to reflect the swap. The elements themselves are not moved, so pointers to them previously obtained are still valid. This routine is an internal routine to the sparse matrix package and is not intended to be called from outside.

2.4.15. SMPcolSwap

```
int SMPcolSwap(matrix, col1, col2)
   SMPmatrix *matrix;
   int col1;
   int col2;
```

SMPcolSwap is identical to SMProwSwap, but operates on columns instead of rows.

2.4.16. SMPprint

```
#include <stdio.h>
void SMPprint(matrix, file)
   SMPmatrix *matrix;
   FILE *file;
```

SMPprint is a diagnostic routine intended primarily for matrix package debugging, although it has also proven useful in debugging matrix loading routines. The contents of the matrix *matrix* will be dumped to the standard I/O system FILE pointer *file* in a reasonably human-readable format. The dump will include all row and column permutation data, pointers, and the real and imaginary parts of the matrix. Because of the quantity of data being printed and the size of "interesting" matrices, no attempt is made to print the matrix as an array, but simply as row, column coordinates and contents.

2.4.17. SMPfillup

```
int SMPfillup(matrix)
    SMPmatrix *matrix;
```

This very elementary debugging routine references every possible element in the sparse matrix *matrix*, thus making it a full matrix. This makes some L-U factorization and reordering code easier to debug by removing any possible fill-in effects.

2.4.18. SMPcClear

```
void SMPcClear(matrix)
    SMPmatrix *matrix;
```

SMPcClear clears to zero both the real and imaginary parts of all entries in the specified sparse matrix *matrix*.

2.4.19. SMPcLUfac

```
int SMPcLUfac(matrix, pivtol)
    SMPmatrix *matrix;
    double pivtol;
```

SMPcLUfac performs the same operations as SMPluFac does, but operates on the full complex matrix instead of only the real part of the matrix.

2.4.20. SMPcProdDiag

```
int SMPcProdDiag(matrix, det, power)
    SMPmatrix *matrix;
    complex *det;
    int *power;
```

SMPcProdDiag computes the product of the diagonal terms in the sparse matrix *matrix*. This product is reported as two numbers, an exponent *power* which is the power of ten which the mantissa must be multiplied by, and the mantissa, which is normalized so that the larger of its real and imaginary parts lies between .1 and 1. If this is applied to an L-U factored matrix, this product is equal to the determinant of the matrix.

2.4.21. SMPcReorder

```
int SMPcReorder(matrix, pivtol, pivrel, numswaps)
    SMPmatrix *matrix;
    double pivtol;
    double pivrel;
    int *numswaps;
```

SMPcReorder performs a reordering and L-U factorization of the complex sparse matrix *matrix*. As in SMPreorder, *pivtol* is the minimum magnitude which is acceptable for a diagonal element in the decomposition and *pivrel* is the factor by which a pivot element may be numerically non-optimal and still be selected to maintain sparsity. Additionally, *numswaps* is multiplied by $-1^{\#rowswaps+\#columnswaps}$.

2.4.22. SMPcSolve

```
int SMPcSolve(matrix, rhs, irhs, spare, ispare)
    SMPmatrix *matrix;
    double *rhs;
    double *irhs;
    double *spare;
    double *ispare;
```

SMPcSolve works exactly as SMPsolve, but operates on the complex sparse matrix *matrix* with the right hand side and solution vectors being the complex pair (*rhs* + *irhs* i), and two scratch vectors *spare* and *ispare* being required.

2.4.23. Complex math

These routines are part of the supporting complex number package used in the sparse matrix package. Since C does not supply basic operations on complex numbers, macros have been developed to provide them. In some cases these macros are too complex for the compilers involved, thus they have also been made into subroutines which are used directly if the preprocessor symbol SHORTMACRO is defined.

2.4.23.1. DCdiveq

```
void DCdiveq(a, b, c, d)
   double *a;
   double *b;
   double c;
   double d;
```

DCdiveq is equivalent to the C "/=" operator when applied to complex numbers, producing (a + bi) /= (c + di).

2.4.23.2. DCmult

```
void DCmult(a, b, c, d, x, y)
   double a;
   double b;
   double c;
   double d;
   double *x;
   double *y;
```

DCmult is equivalent to the C "*" operator when applied to complex numbers, producing (x + yi) = (a + bi) * (c + di).

2.4.23.3. DCminusEq

```
void DCminusEq(a, b, c, d)
   double *a;
   double *b;
   double c;
   double d;
```

DCminusEq is equivalent to the C "-=" operator when applied to complex numbers, producing $(a + bi)$ -= $(c + di)$.

2.5. INP

The INP or INPut package is properly a part of the front end, but was originally developed as a part of the SPICE3 simulator before being split off completely. This package provides the tools necessary to parse SPICE2 format input and produce the subroutine calls required by the front end to simulator interface standard.

The main component of the parser is in the file INPpas2.c which controls the main parsing step. During pass one of the parsing, the .model cards are collected since SPICE3, unlike SPICE2, needs to have the models described before they are instantiated. During pass 2, all remaining cards are parsed and, as needed, the collected model cards are actually parsed on demand as they are referenced.

2.5.1. INPpas1

```
void INPpas1(ckt, deck, tab)
    GENERIC *ckt;           /* circuit structure of circuit being parsed */
    card *deck;             /* The input deck to parse */
    INPtables *tab;         /* The symbol table for this circuit */
```

First pass of the SPICE2 compatible input parser. This routine currently only considers ".MODEL" cards, passing them to INPdoModel to save for pass 2.

2.5.2. INPpas2

```
void INPpas2(ckt, deck, tab, task)
    GENERIC *ckt;           /* The circuit structure to parse into */
    card *deck;             /* The input deck to parse */
    INPtables *tab;         /* The symbol table for this circuit */
    GENERIC *task;          /* The "task" to put analysis requests in */
```

The second pass of the SPICE2 compatible input parser, this routine examines all of the SPICE input lines except for ".MODEL" cards. Each type of input line, as determined by the first character of the line, is actually handled by a separate routine to keep this file from becoming too large. By

convention, the routines handling the individual device lines are named INP2*x* where *x* is the keyletter used by SPICE for the first letter of the device name. All of the control, or "dot" cards are handled by INP2dot.

2.5.2.1. INP2B

```
void INP2B(ckt, tab, current)
    GENERIC * ckt;              /* the current circuit */
    INPtables * tab;            /* the symbol table */
    card * current;             /* the current input line */
```

This routine parses a SPICE2 format input line describing an arbitrary source. Since the input language does not allow for the definition of a model for an arbitrary source, this routine generates and maintains in the INPtables structure a default model for the arbitrary source, creating it when needed and making all sources instances of that model.

2.5.2.2. INP2C

```
void INP2C(ckt, tab, current)
    GENERIC * ckt;              /* the current circuit */
    INPtables * tab;            /* the symbol table */
    card * current;             /* the current input line */
```

Parses a constant capacitor description, maintaining a default model in the INPtables structure since the SPICE2 input format does not permit a model name for a capacitor. SPICE3 also allows a model name to appear on the input line, thus allowing a capacitor to be specified by its size rather than by its capacitance, with the model specifying the relationship between the two.

2.5.2.3. INP2D

```
void INP2D(ckt, tab, current)
    GENERIC * ckt;              /* the current circuit */
    INPtables * tab;            /* the symbol table */
    card * current;             /* the current input line */
```

Parses the SPICE2 description of a diode. Since SPICE3 relaxes the input format to the extent of making the definition of a model unnecessary, INP2D maintains a default model definition in the INPt-

ables structure associated with the circuit.

2.5.2.4. INP2E

```
void INP2E(ckt, tab, current)
    GENERIC * ckt;              /* the current circuit */
    INPtables * tab;            /* the symbol table */
    card * current;             /* the current input line */
```

Parses a voltage controlled voltage source. Unlike SPICE2, SPICE3 does not support polynomial valued sources, but uses the arbitrary source instead. INP2E Also maintains its own model since the input format does not permit one and SPICE3 requires one.

2.5.2.5. INP2F

```
void INP2F(ckt, tab, current)
    GENERIC * ckt;              /* the current circuit */
    INPtables * tab;            /* the symbol table */
    card * current;             /* the current input line */
```

INP2F parses a current controlled current source. Support is only provided for the simple linear source, and a default model is maintained in the INPtables structure.

2.5.2.6. INP2G

```
void INP2G(ckt, tab, current)
    GENERIC * ckt;              /* the current circuit */
    INPtables * tab;            /* the symbol table */
    card * current;             /* the current input line */
```

Parses a linear voltage controlled current source, and maintains a default model in the INPtables structure.

2.5.2.7. INP2H

```
void INP2H(ckt, tab, current)
    GENERIC * ckt;              /* the current circuit */
    INPtables * tab;            /* the symbol table */
    card * current;             /* the current input line */
```

INP2H parses a linear current controlled voltage source, maintaining a model in the INPtables structure since the input syntax does not permit the specification of a model on the input line.

2.5.2.8. INP2I

```
void INP2I(ckt, tab, current)
    GENERIC * ckt;              /* the current circuit */
    INPtables * tab;            /* the symbol table */
    card * current;             /* the current input line */
```

INP2I parses an independent current source. Since SPICE2 and the input format don't support models for current sources, a default model is maintained in the INPtables structure.

2.5.2.9. INP2J

```
void INP2J(ckt, tab, current)
    GENERIC * ckt;              /* the current circuit */
    INPtables * tab;            /* the symbol table */
    card * current;             /* the current input line */
```

INP2J parses a JFET instance. Each instance must have a model specified. If no model with a corresponding name was found during pass 1, a model with the given name is created using entirely default parameters and a warning message is printed.

2.5.2.10. INP2K

```
void INP2K(ckt, tab, current)
    GENERIC * ckt;              /* the current circuit */
    INPtables * tab;            /* the symbol table */
    card * current;             /* the current input line */
```

INP2K parses mutual inductor cards. Since mutual inductors don't have models in the SPICE2 input format, a default model is maintained in the INPtables structure and used for all mutual inductors created by this routine.

2.5.2.11. INP2L

```
void INP2L(ckt, tab, current)
    GENERIC * ckt;              /* the current circuit */
    INPtables * tab;            /* the symbol table */
    card * current;             /* the current input line */
```

INP2L parses a single inductor. Only simple linear inductors are supported, and a default inductor model is maintained in the INPtables structure since a model is not permitted in the input.

2.5.2.12. INP2M

```
void INP2M(ckt, tab, current)
    GENERIC * ckt;              /* the current circuit */
    INPtables * tab;            /* the symbol table */
    card * current;             /* the current input line */
```

Parses a SPICE2 format MOSFET card. Due to the differences between SPICE2 and SPICE3 this routine is relatively ugly, first having to find the model the device is an instance of to determine the type of MOSFET it is instantiating, a level 1, level 2, level 3, or BSIM device. If no model can be found by the given name, a default level 1 model is created with that name and given no parameters.

2.5.2.13. INP2Q

```
void INP2Q(ckt, tab, current, gnode)
    GENERIC * ckt;              /* the current circuit */
    INPtables * tab;            /* the symbol table */
    card * current;             /* the current input line */
    GENERIC *gnode;             /* node structure for the ground node */
```

Parses a SPICE2 format bipolar junction transistor card. Due to the ambiguity present in the language after the SPICE2 restriction on node names is lifted, this routine must attempt to distinguish between the two cases

```
Q1 collector base emitter model other-parameters
Q2 collector base emitter bulk model other-parameters
```

To do this, the collected list of models is consulted and if a BJT model exists matching the fifth token on the line, it is assumed to be the model name, and that model is used. If the fifth token does not

match a BJT model found during pass 1, that token is assumed to be the bulk node and the sixth token to be the model name. Thus, it is very important to include a known model name in a BJT instantiation. The final argument to this function which distinguishes it from all the rest of the INP2*x* functions is the ground node of the circuit which is needed to connect the bulk node to if the bulk node is not specified.

2.5.2.14. INP2R

```
void INP2R(ckt, tab, current)
    GENERIC * ckt;              /* the current circuit */
    INPtables * tab;            /* the symbol table */
    card * current;             /* the current input line */
```

INP2R parses a resistor instantiation in the expanded SPICE3 format which allows for a model specification if desired but does not require it.

2.5.2.15. INP2S

```
void INP2S(ckt, tab, current)
    GENERIC * ckt;              /* the current circuit */
    INPtables * tab;            /* the symbol table */
    card * current;             /* the current input line */
```

INP2S parses an input line for a voltage controlled ideal switch. Switches require a model specification, but a default switch model is maintained in case the specified model does not exist and all switches using an unknown model are made instances of this model.

2.5.2.16. INP2T

```
void INP2T(ckt, tab, current)
    GENERIC * ckt;              /* the current circuit */
    INPtables * tab;            /* the symbol table */
    card * current;             /* the current input line */
```

INP2T parses a transmission line. Models are not permitted in the SPICE2 input format for this element, so a default model is maintained and all transmission lines are instances of this model.

2.5.2.17. INP2U

```
void INP2U(ckt, tab, current)
    GENERIC * ckt;              /* the current circuit */
    INPtables * tab;            /* the symbol table */
    card * current;             /* the current input line */
```

INP2U parses uniform distributed RC lines. All references to unknown models are changed to a default model maintained in the INPtables structure.

2.5.2.18. INP2V

```
void INP2V(ckt, tab, current)
    GENERIC * ckt;              /* the current circuit */
    INPtables * tab;            /* the symbol table */
    card * current;             /* the current input line */
```

INP2V handles specifications of independent voltage sources. No model is permitted in the input syntax, so a single default model is maintained and all voltage sources are instances of it.

2.5.2.19. INP2W

```
void INP2W(ckt, tab, current)
    GENERIC * ckt;              /* the current circuit */
    INPtables * tab;            /* the symbol table */
    card * current;             /* the current input line */
```

INP2W parses current controlled switch statements. A default model is maintained for all instances which refer to a non-existent model.

2.5.2.20. INP2Z

```
void INP2Z(ckt, tab, current)
    GENERIC * ckt;              /* the current circuit */
    INPtables * tab;            /* the symbol table */
    card * current;             /* the current input line */
```

INP2Z parses GaAs MESFET descriptions. The input format requires a model, but a default is maintained in the INPtables structure for all instances which refer to an unknown model.

2.5.2.21. INP2dot

```
void INP2dot(ckt, tab, current, task)
    GENERIC *ckt;                   /* The current circuit */
    INPtables *tab;                 /* The symbol table */
    card *current;                  /* The line to finish parsing */
    GENERIC *task;                  /* The task new analysis will be in */
```

This routine attempts to parse all of the control cards in the SPICE2 format which begin with a dot. Some of these cards are picked off by an earlier layer of nutmeg, but many reach this level, such as the analysis commands. All are trapped at this level to avoid ambiguity if nutmeg misses one or this parser is used outside of nutmeg.

2.5.3. Interface functions

These functions are provided to help implement the front end to simulator calling convention within nutmeg.

2.5.3.1. IFeval

```
int IFeval(tree, result, vals, derivs)
    IFparseTree *tree;              /* The parse tree to evaluate */
    double *result;                 /* The value of the equation */
    double *vals;                   /* The values needed for the evaluation */
    double *derivs;                 /* The derivatives computed during evaluation */
```

This function is part of the parse tree package which evaluates the parse trees. Using a list of names provided in the IFparseTree structure, the *vals* vector must be filled in with the actual values of the named variables before calling. Upon return from the call, result will be the value of the equation, and derivs will be a vector of partial derivatives of the equation with respect to each of the named values.

2.5.3.2. PTeval

```
static int PTeval(tree, results, vals)
    INPparseNode *tree;             /* Tree to evaluate */
    double *results;                /* Result of evaluating tree */
    double *vals;                   /* values needed to evaluate tree */
```

PTeval is an internal function used by the parse tree evaluation function IFeval to evaluate each of the individual subtrees of the IFparseTree structure to get the values for the overall equation and for each of the derivatives. This function **can not** be called by any function other than IFeval.

2.5.3.3. IFnewUid

```
int IFnewUid(ckt, newuid, olduid, suffix, type, nodedata);
    GENERIC *ckt;          /* The circuit the uid is to be contained in */
    IFuid *newuid;         /* The uid to be returned */
    IFuid olduid;          /* The old uid this id is to be based on */
    char *suffix;          /* The suffix to logically add to the olduid */
    int type;              /* The type of uid to create */
    GENERIC **nodedata;    /* an optional node structure if adding signal*/
```

This function provides a mapping from names to the unique identifiers used to distinguish between named entities. The unique identifiers are pointer sized objects which can be compared to determine the equality of the referenced items. This implementation simply generates the full name specified by the argument by concatenating the suffix with the existing id with the "#" character as a separator. This identifying character string is then inserted in one of two string tables and the pointer to the string in the string table used as the unique identifier. Two string tables are used to keep the name spaces for nodes and other signal names distinct from that for models, device names, and analysis names.

2.5.4. Convenience functions

These functions encapsulate many of the common operations the input package needs to perform frequently which are conceptually simply, but require a moderate amount of code which would otherwise be needlessly replicated.

2.5.4.1. INPaName

```
int INPaName(parm, val, ckt, dev, devname, inst, sim, dataType, selector)
    char *parm;        /* name of parameter to ask for */
    IFvalue *val;      /* value field to fill in */
    GENERIC *ckt;      /* current circuit */
    int *dev;          /* device type code for the device */
```

```
char *devname;      /* name of the device */
GENERIC **inst;     /* pointer to the device structure */
IFsimulator *sim;   /* the simulator data structure */
int *dataType;      /* the type of the returned value field */
IFvalue *selector;  /* sub-selector for questions */
```

INPaName is a convenience function used to ask for the value of an instance output variable by name rather than by the explicit id required by the DEVask function. By using the findInstance function, this function will find the correct instance given a minimal amount of data and then identify the correct parameter and call the required DEVask function with the necessary id. The *dataType* field is filled in with the type of the returned data, while the *val* field is filled in with the actual data. Other unknown data about the device such as the type code and the pointer to the instance structure will be filled in as appropriate.

2.5.4.2. INPapName

```
int INPapName(ckt, type, analPtr, parmName, value)
    GENERIC *ckt;                   /* the current circuit */
    int type;                       /* the type of analysis */
    GENERIC *analPtr;               /* The analysis to modify */
    char *parmName;                 /* The name of the analysis parameter */
    IFvalue *value;                 /* The value to give the parameter */
```

INPapName is a convenience function for setting a parameter on an analysis by name rather than having to computed the id of the parameter to be changed.

2.5.4.3. INPcaseFix

```
void INPcaseFix(string)
    char *string;           /* String to convert to lower case */
```

INPcaseFix operates on a character string, normally an input line record, and converts all characters to lower case. Since the SPICE2 input syntax is case insensitive, and SPICE3 has been written entirely in terms of lower case strings, the simplest step is to convert all SPICE2 format inputs all to lower case before proceeding.

2.5.4.4. INPdevParse

```
char *INPdevParse(line, ckt, dev, inst, leading, waslead, tab)
    char **line;        /* The line fragment to parse */
    GENERIC *ckt;       /* The current circuit */
    int dev;            /* The type index of the device to parse */
    GENERIC *inst;      /* pointer to the instance the line describes */
    double *leading;    /* The value of the unlabeled leading value */
    int *waslead;       /* Flag to indicate unlabeled leading val. present */
    INPtables *tab;     /* The symbol table for the current circuit */
```

Once a device input line has been reduced to a form where it consists of nothing but data of the form "keyword=value" or "flagkeyword" with a single optional unlabeled value, it can be parsed by a single routine. INPdevParse breaks down this very regular form, performing all necessary operations to attach the parameter values to the instance. The optional unlabeled value will be returned in *leading*, with *waslead* set to a non-zero value if that value was present.

2.5.4.5. INPdoOpts

```
void INPdoOpts(ckt, anal, optCard, tab)
    GENERIC *ckt;                   /* Circuit being parsed */
    GENERIC *anal;                  /* The options analysis to add to */
    card *optCard;                  /* The card image to get data from */
    INPtables *tab;                 /* The parse tables for this circuit */
```

INPdoOpts is used to process a .option card. The analysis *anal* is assumed to be an options analysis already created, and any legal options parameters found after the first token of the card image *optCard* are added to that options analysis. The first token on the card image is assumed to be the word '.options' or one of its legal synonyms and is ignored.

2.5.4.6. INPdomodel

```
char *INPdomodel(ckt, image, tab)
    GENERIC *ckt;               /* The current circuit */
    card *image;                /* The card structure to examine for data */
    INPtables *tab;             /* The parse tables for this circuit */
```

INPdomodel is used to perform the necessary pass one operations on a model. The model name is put into the symbol table, and the type of the model (NPN, PNP, NMOS, etc) is converted to a

SPICE3 device type name, which is then converted to the proper device type index and passed on to INPmakeMod to store the necessary data away in the model tables for future references during pass 2 of the parsing.

2.5.4.7. INPerrCat

```
char *INPerrCat(a, b)
    char *a;            /* First string to concatenate */
    char *b;            /* Second string to concatenate */
```

INPerrCat is used to concatenate error messages during the parsing operation. *a* and *b* are assumed to be either malloc'ed character strings or NULL pointers. If both strings are non-null, they will be concatenated together into a new string separated by a newline, and their original storage will be free'd. If either string is null, the other will be returned unchanged. If both are null, null will be returned.

2.5.4.8. INPerror

```
char *INPerror(type)
    int type;           /* type of error detected */
```

INPerror is used to generate an error message in a consistent format. INPerror knows about the list of error codes that can be returned by SPICE3 and converts the code into an appropriate message which is merged with any additional information, such as the routine detecting the problem, into a malloc'ed character string which is returned.

2.5.5. INPevaluate

```
double INPevaluate(line, error, eat)
    char **line;                /* line to get number from */
    int *error;                 /* returned error code */
    int eat;                    /* eat rest of line? */
```

INPevaluate takes the next token from the given input line fragment (advancing the pointer along the line) and evaluates it as a double precision floating point number which it returns. If the next token can not be evaluated as a legal floating point number, the pointer within the line is reset to

its original value and error is set to a non-zero value. If *eat* has a non-zero value, the remainder of the input token will be eaten up and thrown away. A value of zero for *eat* will cause INPevaluate to stop as soon as it has found the end of the number and leave the rest of the line alone.

2.5.6. INPfindLev

```
char *INPfindLev(line, level)
   char *line;          /* The .model line to scan for level */
   int *level;          /* the detected level number on return */
```

INPfindLev is used by INPdomodel to determine which level of MOSFET model is being used. Unlike other devices the model type keyword for MOSFETs, NMOS or PMOS, does not determine the type of the device, merely its polarity. To determine the type of the device, the entire ".MODEL" card must be scanned for an occurrence of the keyword level followed by a numerical parameter giving the actual level of the device. The level is returned in the *level* parameter, with the return value of the function being a textual error message if any problems were encountered.

2.5.7. INPgetMod

```
char *INPgetMod(ckt, name, model, tab)
   GENERIC *ckt;              /* The current circuit */
   char *name;                /* The name of the model */
   INPmodel **model;          /* The parser model structure */
   INPtables *tab;            /* the current symbol table */
```

INPgetMod searches the parser's database of model definitions built up during pass 1 for the model *name*. It is assumed that the model is to be used, so the necessary calls are made to create the model and parse all of its parameters. The front end model structure containing the SPICE model pointer is returned in *model*. The function return value is a character string containing all the error messages generated while parsing the model which are not directly attributable to the model. Errors directly attributable to the ".MODEL" card itself are added to the error field on that card directly.

2.5.8. INPgetTitle

```
int INPgetTitle(ckt, data)
    GENERIC **ckt;          /* The circuit to create & save title of */
    card **data;            /* The data structure representing the input */
```

INPgetTitle is an obsolete routine used by an earlier version of the parse to grab the title card off of the front of the input deck and call the initialization routine to create a new circuit with that name.

2.5.9. INPgetTok

```
int INPgetTok(line, token)
    char **line;            /* input line */
    char **token;           /* resulting token */
```

INPgetTok is the primary tokenizer for the parser. INPgetTok finds the next token on the input line given and returns in *token* a malloc'ed character string containing a copy of just that token. Tokens are assumed to be delimited by a reasonable set of separator characters, currently consisting of space, tab, comma, equals, and open and close parentheses.

2.5.10. INPgetValue

```
IFvalue *INPgetValue(ckt, line, type, tab)
    GENERIC *ckt;           /* The current circuit */
    char **line;            /* The line to get the argument from */
    int type;               /* The type of argument to get */
    INPtables *tab;         /* The symbol table for the current circuit */
```

INPgetValue is used to get a value of type *type* from the input line without having to know anything about type itself. INPgetValue performs all of the type specific operations, evaluating numbers, inserting strings into the string table, and placing the final result into an IFvalue structure. Since the IFvalue structure which is returned is a static, the structure must be used or its value copied out before any other calls are made to INPgetValue or any subroutine which calls it.

2.5.11. INPkillMods

```
void INPkillMods();
```

INPkillMods destroys the linked list of semi-parsed models maintained by INPmakeMod. INPkillMods should be called once before calling INPpas1 for a new circuit.

2.5.12. INPlist

```
void INPlist(file, deck, type)
    FILE *file;          /* The file to print the listing on */
    card *deck;          /* The input deck to list */
    int type;            /* The type of listing - logical or physical */
```

INPlist is an obsolete function used to produce a listing of an input deck, either printing all of the logical lines produced by stripping out the continuation cards and stringing the lines out to their full length, or a physical listing of the cards as they were input. This routine has been replaced by more general facilities inside of nutmeg.

2.5.13. INPlookMod

```
int INPlookMod(name)
    char *name;          /* model name to look up */
```

INPlookMod is used to search the list of ".model" cards found during pass 1 of the parser to determine whether *name* is the name of a known model or not. A return of zero indicates the name is not that of a model, while one indicates that it is a model name. The comparison is made by comparing the actual strings rather than pointers as is usually done, since this routine is used to solve the ambiguity in the BJT device card where the fifth token on the line may be either a model or node name, and thus hasn't been entered into a symbol table yet.

2.5.14. INPmakeMod

```
int INPmakeMod(token, type, line)
    char *token;         /* The model name */
    int type;            /* The type of model */
    card *line;          /* The line representing the model */
```

INPmakeMod is used to store the ".model" cards found during pass 1 of the parser into a list where they can be accessed during pass 2 without actually parsing the entire model.

2.5.15. INPmkTemp

```
char *INPmkTemp(string)
    char *string;              /* string to make copy of */
```

INPmkTemp is used to generate malloc'ed copies of constant strings. There are many places, such as error messages, where a string may need to be a literal, but the routines which manipulate the strings later on need to be able to treat all strings equally, and to free those that were obtained from malloc, so literals are copied into malloc'ed memory with INPmkTemp.

2.5.16. INPpName

```
int INPpName(par, val, ckt, dev, inst)
    char *parm;                /* The parameter name to set */
    IFvalue *val;              /* The value to set it to */
    GENERIC *ckt;              /* The current circuit */
    int dev;                   /* The device type of the device being parsed */
    GENERIC *inst;             /* The pointer to the device being parsed */
```

INPpName is a simple routine to set a parameter on an instance by specifying the name of the parameter instead of the parameter id as required by the lower level DEV routines. The *dev* parameter allows INPpName to search the parameter tables of the proper type of device for the name *parm* and then call the proper device specific setInstanceParm function to perform the actual work.

2.5.17. INPparseTree

```
void INPgetTree(line, pt, ckt, tab)
    char **line;               /* line to parse */
    INPparseTree **pt;         /* parse tree to return */
    GENERIC *ckt;              /* The current circuit */
    INPtables *tab;            /* The symbol tables */
```

INPgetTree fetches an expression from the given input line and breaks it into a parse tree. The trees necessary to evaluate the partial derivatives of the original tree with respect to each of the input variables will also be generated. sh 4 "PTdifferentiate"

```
static INPparseNode *PTdifferentiate(p, varnum)
   INPparseNode *p;                    /* Tree to differentiate */
   int varnum;                         /* number of variable to differentiate WRT */
```

INPparseNode differentiates an expression symbolically. The INPparseNode returned contains a parse tree which represents the derivative of the INPparseNode given as an argument with respect to the variable given as *varnum*. This function is local to the INPgetTree routine and can not be used elsewhere.

2.5.17.1. mkcon

```
static INPparseNode * mkcon(value)
   double value;                       /* The constant */
```

Mkcon generates a parse tree node corresponding to a constant with the given value. This function is local to the INPgetTree routine and can not be used elsewhere.

2.5.17.2. mkb

```
static INPparseNode *mkb(type, left, right)
   int type;                           /* type of node to generate */
   INPparseNode *left;                 /* left subtree */
   INPparseNode *right;                /* right subtree */
```

Mkb generates a parse tree for a binary operator connecting two other parse trees. An attempt is made to generate the simplest parse tree, so special cases for binary operators joining constants, multiplication by zero or one, division by one, zero divided by anything, addition or subtraction of zero, unary subtraction, and raising to the zeroth or first power are all handled correctly. This function is local to the INPgetTree routine and can not be used elsewhere.

2.5.17.3. mkf

```
static INPparseNode *mkf(type, arg)
   int type;                           /* Type of function */
   INPparseNode *arg;                  /* argument to function */
```

Mkf generates a parse tree for a function call operation. Simplifications are performed if possible, so functions of a constant will be replaced by their actual constant value. This function is local to the INPgetTree routine and can not be used elsewhere.

2.5.17.4. PTcheck

```
static int PTcheck(p)
   INPparseNode *p;     /* Parse tree to check */
```

PTcheck is used to check the validity of a parse tree. There are some ambiguities of the syntax used in our expressions which can not be resolved until the surrounding context has been examined. Such problem tokens are put in the tree unparsed and either handled later when appropriate, or generate an error when this routine checks to be sure they are gone. This function is local to the INPgetTree routine and can not be used elsewhere.

2.5.17.5. PTparse

```
static INPparseNode * PTparse(line)
   char **line;                    /* line to parse */
```

Actually parse an expression into a parse tree. This is the internal routine that does all the hard work for INPgetTree. This function is local to the INPgetTree routine and can not be used elsewhere.

2.5.17.6. makepnode

```
static INPparseNode *makepnode(elem)
   PTelement *elem;                /* Element to convert */
```

Convert the given element into a parse tree node. *Element* can be a string, a number, or a node. This function is local to the INPgetTree routine and can not be used elsewhere.

2.5.17.7. mkbnode

```
static INPparseNode *mkbnode(opnum, arg1, arg2)
   int opnum;                                  /* operator number */
   INPparseNode *arg1;                         /* left argument parse tree */
   INPparseNode *arg2;                         /* right argument parse tree */
```

Mkbnode generates a parse tree node for a binary operator which combines the given two subtrees as its left and right arguments. This function is local to the INPgetTree routine and can not be used elsewhere.

2.5.17.8. mkfnode

```
static INPparseNode *mkfnode(fname, arg)
   char *fname;                          /* The name of the function */
   INPparseNode *arg;                    /* The argument it is given */
```

mkfnode converts a function call into the necessary parse tree node. Since the old spice notations V(nodename) and I(sourcename) are supported, mkfnode must detect these situations and convert them to the proper reference to a node voltage or device current instead of a function call. This function is local to the INPgetTree routine and can not be used elsewhere.

2.5.17.9. mknnode

```
static INPparseNode *mknnode(number)
   double number;                        /* constant value for node */
```

mknnode generates a parse node for a constant with the given value. This function is local to the INPgetTree routine and can not be used elsewhere.

2.5.17.10. mksnode

```
static INPparseNode *mksnode(string)
   char *string;                         /* The unknown string */
```

Mksnode generates a parse tree node for a character string which may not be fully parsable yet. Cases for special signal names like 'time' are handled here as are well known constants like pi. Unknown names are left alone with a placeholder type so that they can be picked up later as arguments to the SPICE2 V(node) or I(source) notation. This function is local to the INPgetTree routine and can not be used elsewhere.

2.5.17.11. PTlexer

```
static PTelement *PTlexer(line)
   char **line;                    /* input line */
```

PTlexer is the lexical analyzer used for expression parsing. The tokens needed when parsing an expression are different from those used when parsing most of the rest of the SPICE input format. This function handles the different tokenizing needed for expressions. This function is local to the INPgetTree routine and can not be used elsewhere.

2.5.17.12. INPptPrint

```
void INPptPrint(str, ptree)
   char *str;              /* tree name */
   IFparseTree *ptree;     /* tree to print */
```

INPptPrint is a debugging routine used for printing an entire parse tree family, including all of the derivative trees. the string *str* is printed as a label on the top level tree. This function is local to the INPgetTree routine and can not be used elsewhere.

2.5.17.13. printTree

```
static void printTree(pt)
   INPparseNode *pt;       /* parse sub-tree to print */
```

PrintTree is a debugging function used to print a subtree of a parse tree. This function is local to the INPgetTree routine and can not be used elsewhere.

2.5.18. Symbol table handling

These routines are collected into the file INPsymTab.c, and implement the symbol tables used for node names, the terminal symbol table, and for all other names within SPICE3, the general symbol table.

2.5.18.1. INPtabInit

```
INPtables *INPtabInit(numlines)
   int numlines;                /* approximate number of input lines */
```

INPtabInit allocates and initializes the symbol tables. *numlines* is an approximation to the size of the input, and is used to pre-allocate most of the table.

2.5.18.2. INPtermInsert

```
int INPtermInsert(ckt, token, tab, node)
   GENERIC *ckt;                /* The current circuit */
   char **token;                /* The name to insert */
   INPtables *tab;              /* The current symbol table */
   GENERIC **node;              /* The node structure to return */
```

INPtermInsert is used to place the name of a node or a current equation into the symbol table. The name is first checked for uniqueness, and if it is not unique, is discarded and the existing node's information is returned. If the name is new, then a call is made to the simulator's newNode function with the resulting node pointer saved in the symbol table and returned to the caller.

2.5.18.3. INPmkTerm

```
int INPmkTerm(ckt, token, tab, node)
   GENERIC *ckt;                /* current circuit */
   char **token;                /* token representing node name */
   INPtables *tab;              /* current symbol table */
   GENERIC **node;              /* node structure to associate with name */
```

INPmkTerm is almost identical to INPtermInsert, but the node passed in is an existing node pointer which should be used instead of calling the newnode function to create a node pointer.

2.5.18.4. INPgndInsert

```
int INPgndInsert(ckt, token, tab, node)
   GENERIC *ckt;                /* Current circuit */
   char **token;                /* Token representing ground node */
   INPtables *tab;              /* Current symbol table */
   GENERIC **node;              /* Returned node structure for ground node */
```

INPgndInsert is a special case of INPtermInsert used only when creating the ground node of the circuit.

2.5.18.5. INPinsert

```
int INPinsert(token, tab)
    char **token;          /* Token to put in table */
    INPtables *tab;        /* Current table pointer */
```

INPinsert inserts a single token into the general symbol table.

2.5.18.6. INPtabEnd

```
void INPtabEnd(tab)
    INPtables *tab;     /* Table to free */
```

INPtabEnd frees the space allocated for the symbol tables and their subsidiary linked list structures. The actual names will not be freed since other references made to them may still exist.

2.5.18.7. hash

```
static int hash(name, tsize)
    char *name;            /* name to hash */
    int tsize;             /* size of hash table */
```

Hash computes a hash function in the range 0 to *tsize* from the given name to make the symbol table operation more efficient.

2.5.19. INPtypelook

```
int INPtypelook(type)
    char *type;          /* Name of type to look up */
```

This is a simple supporting function which provides a mapping from the names of device types to their type indexes.

2.5.20. PTfunctions

These functions are all used to implement the parse trees, and actually perform the operations represented by various nodes within the trees. Many of these are simply wrappers around math library routines, while others are actual implementations of the desired functions.

2.5.20.1. reduce

```
static double reduce(arg)
   double arg;              /* argument to reduce to 0-2*pi */
```

Reduce reduces the argument of trig functions to the range 0 to 2π since some library functions don't perform this reduction properly. This is a static function only used within the parse tree implementation functions.

2.5.20.2. PTplus

```
double PTplus(arg1, arg2)
   double arg1;             /* left argument */
   double arg2;             /* right argument */
```

PTplus implements simple addition and returns *arg* 1+*arg* 2.

2.5.20.3. PTminus

```
double PTminus(arg1, arg2)
   double arg1;             /* left argument */
   double arg2;             /* right argument */
```

PTminus implements simple subtraction and returns *arg* 1–*arg* 2.

2.5.20.4. PTtimes

```
double PTtimes(arg1, arg2)
   double arg1;             /* left argument */
   double arg2;             /* right argument */
```

PTtimes implements simple multiplication and returns *arg* 1×*arg* 2

2.5.20.5. PTdivide

```
double PTdivide(arg1, arg2)
   double arg1;              /* left argument */
   double arg2;              /* right argument */
```

PTdivide implements simple division and returns $\frac{arg1}{arg2}$

2.5.20.6. PTpower

```
double PTpower(arg1, arg2)
   double arg1;              /* left argument */
   double arg2;              /* right argument */
```

PTpower implements exponentiation, and returns $arg1^{arg2}$

2.5.20.7. PTacos

```
double PTacos(arg)
   double arg;          /* argument */
```

PTacos implements the arc cosine function by calling the corresponding function from the math library.

2.5.20.8. PTacosh

```
double PTacosh(arg)
   double arg;          /* argument */
```

PTacosh implements the hyperbolic arc cosine function by calling the corresponding function from the math library. If the "#define" name **HASATRIGH** is defined, this function will be available. If this name is not defined at compile time, this function and all internal references to it will be deleted automatically, since acosh is not a standard part of the C math library.

2.5.20.9. PTasin

```
double PTasin(arg1)
   double arg;          /* argument */
```

PTasin implements the arc sine function by calling the corresponding function from the math library.

2.5.20.10. PTasinh

```
double PTasinh(arg)
   double arg;          /* argument */
```

PTasinh implements the hyperbolic arc sine function using the corresponding function from the math library. If the "#define" name **HASATRIGH** is defined, this function will be available. If this name is not defined at compile time, this function and all internal references to it will be deleted automatically, since acosh is not a standard part of the C math library.

2.5.20.11. PTatan

```
double PTatan(arg)
   double arg;          /* argument */
```

PTatan implements the arc tangent function using the corresponding function from the math library.

2.5.20.12. PTatanh

```
double PTatanh(arg)
   double arg;          /* argument */
```

PTatanh implements the hyperbolic arc tangent function. If the "#define" name **HASATRIGH** is defined, this function will be available. If this name is not defined at compile time, this function and all internal references to it will be deleted automatically, since acosh is not a standard part of the C math library.

2.5.20.13. PTcos

```
double PTcos(arg)
   double arg;          /* argument */
```

PTcos implements the cosine function.

2.5.20.14. PTcosh

```
double PTcosh(arg)
   double arg;          /* argument */
```

sh implements the hyperbolic cosine function.

2.5.20.15. PTexp

```
double PTexp(arg)
   double arg;          /* argument */
```

PTexp implements e^{arg}.

2.5.20.16. PTln

```
double PTln(arg)
   double arg;        /* argument */
```

PTln implements the natural logarithm function.

2.5.20.17. PTlog

```
double PTlog(arg)
   double arg;        /* argument */
```

PTlog implements the base 10 logarithm function.

2.5.20.18. PTsin

```
double PTsin(arg)
   double arg;        /* argument */
```

PTsin implements the sine function.

2.5.20.19. PTsinh

```
double PTsinh(arg)
   double arg;          /* argument */
```

PTsinh implements the hyperbolic sine function.

2.5.20.20. PTsqrt

```
double PTsqrt(arg)
   double arg;          /* argument */
```

PTsqrt implements the square root function.

2.5.20.21. PTtan

```
double PTtan(arg)
   double arg;          /* argument */
```

PTtan implements the tangent function.

2.5.20.22. PTtanh

```
double PTtanh(arg)
   double arg;          /* argument */
```

PTtanh implements the hyperbolic tangent function.

2.5.20.23. PTuminus

```
double PTuminus(arg)
   double arg;          /* argument */
```

PTuminus implements the unary minus function, returning the negative of its argument.

2.5.21. Proc2Mod

This is the main function to an auxiliary program, Proc2Mod, which accepts the process description files described in the BSIM documentation and converts them into all possible BSIM .model cards which can then be used with SPICE3.

References

Chou88a.

Choudhury, Umakanta, "Sensitivity Analysis in SPICE3," Masters Report, University of California, Berkeley (December 1988).

Cohe76a.

Cohen, E., "Program Reference for SPICE2," *Electronics Res. Lab.*, University of California, Berkeley, (June 1976).

Nage75a.

Nagel, L., "SPICE2: A Computer Program to Simulate Semiconductor Circuits," *ERL Memo* **UCB/ERL M75/520**University of California, Berkeley, (May 1975).