ADDING DEVICES TO SPICE3

by

Thomas L. Quarles

Memorandum No. UCB/ERL M89/45

24 April 1989

COVER PAGE

5-7

ADDING DEVICES TO SPICE3

by

Thomas L. Quarles

Memorandum No. UCB/ERL M89/45

24 April 1989

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

ADDING DEVICES TO SPICE3

by

Thomas L. Quarles

Memorandum No. UCB/ERL M89/45

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 M89/44	The SPICE3 Implementation Guide
UCB/ERL M89/46	SPICE3 Version 3C1 Users Guide
UCB/ERL M89/47	Benchmark Circuits: Results for SPICE3

This memo was originally Appendices D and E of the dissertation and contains a description of the procedure which must be followed to add a device model to SPICE3. The release of SPICE3 described is SPICE3, version 3C1

Table of Contents

CHAPTER 1

Adding a Device

When adding a new device, there are three types of changes that must be made: new routines which must be written specifically to support the device, modifications to existing routines to give them some detailed knowledge of the device for parsing, and changes necessary to integrate the device into the main loops of the simulation algorithms.

1.1. Writing device specific routines

The new devices for which routines must be written typically come from two different sources, those written completely from scratch, and those being moved from SPICE2 to SPICE3. This section contains an overview of the routines which must be written followed by a few guidelines for moving an existing SPICE2 implementation into SPICE3. For more detailed information on how the routines should be written, see the detailed descriptions of them in the next chapter.

Each device is described by a data structure which contains pointers to functions which provide the device specific operations and tables which describe the parameters of the device. This structure also contains pointers to a variety of tables and size data that are needed by code at the user-interface level and by higher-level SPICE routines. This data structure is the **only** externally visible portion of the device specific code; everything else is referenced through the function pointers contained in the structure. The other substructures to be defined are presented first, followed by a description of how they are all placed into this master data structure.

1.2. Device Specific Data Structures

For any device, it is necessary to define the internal data structures. There are two such structures required for each device, one for the device model, and one for the instance. The model structure will contain all the parameters which a number of devices are likely to have in common, such as ohms per square or capacitance per square meter. Typically, these parameters are all specific to the process used to produce the devices or to a class of similar devices, such as short channel transistors, rather than to a specific device instance. There is a standard header which describes the first four entries in the model structure which must be present in exactly the given order, but everything after that is up to the implementor. These four entries serve to connect the model and its instances together into the overall data structure, and by forcing their exact placement in the structure, generic routines can be used to traverse the structure.[1] The prototype model header is:

```
typedef struct sGENmodel {
    int GENmodtype;
    struct sGENmodel *GENnextModel;
    GENinstance * GENinstances;
    IFuid GENmodName;
} GENmodel;
```

Where *GEN* is replaced throughout by the package prefix used for the device. The GENmodType field is used to store the index number of the device type, allowing location of the routines needed to manipulate any given model. The GENnextModel field points to the next model of the same type of device, thus forming a linked list which can be traversed to process all models of a given type. GEN-instances points to the first instance of the model, and is the starting point for a linked list of the instances of the model, again allowing a simple traversal to process all such devices. GENmodName is the unique identifier assigned by the front end to represent the name of the model. Following GEN-modName, the implementor may add any additional fields desired.

For an instance, the standard header contains three entries which must be present, and additional entries which, if present, must be in specific positions. The *GENmodPtr, GENnextInstance*, and *GEN-name* fields are required and must be first. GENmodPtr is a back-pointer to the model structure this structure represents an instance of, and allows easy access to model parameters as well as the device type field contained in the model. The GENnextInstance field contains the pointer to the next instance in the linked list started by the GENinstances pointer in the model structure above. GENname is the

[1] There would be a requirement that the first item in the structure be an instance of this GENmodel structure, but the C language also guarantees that if two structures share a common prefix then references within that prefix can use either structure

unique identifier assigned to the name of this instance by the front end, and thus identifies the instances. If the device has any external nodes, they must appear next and in the same order as they will appear on the device card. At this time, devices with up to 5 external nodes are supported, although this may be increased by modifying GENdefs.h and CKTbindNode.c.

```
typedef struct sGENinstance {
    GENmodel *GENmodPtr;
    struct sGENinstance *GENnextInstance;
    IFuid GENname;
    int GENnode1;                       /* only use as */
    int GENnode2;                       /* many of these as */
    int GENnode3;                       /* you need and */
    int GENnode4;                       /* change their names to */
    int GENnode5;                       /* be more appropriate */
} GENinstance ;
```

Following this required portion of the structure may be any number of fields to contain instance data.

1.3. Parameter Descriptors

The next pair of structures that must be created are the parameter descriptor arrays. Each device has an array of parameter descriptors for the device and an array for the model. These arrays list all the parameters which are legal for the device and model, along with information needed by routines which will need to handle these parameters, such as their type and a simple integer that can be used to refer to them to avoid the overhead of multiple string compare operations. The integer is usually given as a symbolic constant rather than an actual constant to make the code more readable. The type is one of the symbolic constants *IF_INTEGER, IF_REAL, IF_FLAG, IF_NODE, IF_COMPLEX, IF_STRING, IF_INSTANCE,* or *IF_PARSETREE* combined with one or more modifiers. These symbolic constants indicate that the parameter requires an integer value, a real value, no value at all, a circuit node, a complex number, a character string, the unique identifier of another device, or a parse tree representing an expression respectively. These types are the basic datatypes supported by the parser. The modifiers pass additional information about the types used and are bitwise OR'd into the field as necessary. IF_VECTOR indicates that a vector of values of the given type is expected. IF_REQUIRED

definition, and the extra field in every reference within the device code would be unnecessary and distracting[Kern78a].

indicates that the value must be supplied if the model is to function correctly. IF_SET and IF_ASK indicate, respectively, that the parameter can be set by the front end and returned by the simulator in response to a front end query. If neither IF_SET nor IF_ASK is set, it indicates a parameter which the programmer recognizes but does not implement at this time, and should generate a warning from the front end but not an 'unrecognized parameter' error. The actual values of all these different types are passed in a single structure, and knowledge of the type field allows the program to access the correct field or fields of the structure. Specifying these incorrectly may result in errors when an attempt is made to extract data from this structure since it will not be in the expected form. Finally, IF_SELECT and IF_VSELECT indicate that a parameter is used to set or examine the value of a single element of a vector or array, and the additional *select* parameter to the corresponding *set* or *ask* function is an integer or vector of integers which will be used to select the correct elements from the appropriate vector or array entity.

1.4. The overall device structure

Once the basic structures have been defined, the SPICEdev structure must be filled in. This structure contains pointers to routines described in the device interface chapter of this dissertation, along with a number of tables and constants. The following gives the declaration of the structure and a brief comment on the purpose or source of each entry.

```
typedef struct SPICEdev {
   struct {
      char *name;              /* name of this type of device */
      char *description;       /* description of this type of device */

      int terms;               /* number of terminals on this device */
      int numNames;            /* number of names in the termNames array*/
      char **termNames;        /* pointer to array of pointers to names */
                               /* array contains 'terms' pointers */

      int numInstanceParms;    /* number of instance parameter descriptors */
      IFparm *instanceParms;   /* array  of instance parameter descriptors */

      int numModelParms;       /* number of model parameter descriptors */
      IFparm *modelParms;      /* array  of model parameter descriptors */

   } DEVpublic;

   int (*DEVparam)();          /* routine to input a parameter to a device instance */
   int (*DEVmodParam)();       /* routine to input a parameter to a model */
   int (*DEVload)();           /* routine to load the device into the matrix */
   int (*DEVsetup)();          /* setup routine to preprocess devices once before
                               /* solution begins */
   int (*DEVpzSetup)();        /* setup routine specifically for pole-zero anal. */
   int (*DEVtemperature)();    /* subroutine to do temperature dependent
                               /* setup processing */
   int (*DEVtrunc)();          /* subroutine to perform truncation error calculate. */
   int (*DEVfindBranch)();     /* subroutine to search for device branch eq.s */
   int (*DEVacLoad)();         /* ac analysis loading function */
   int (*DEVaccept)();         /* subroutine to call on acceptance of a timepoint */
   void (*DEVdestroy)();       /* subroutine to destroy all models and instances */
   int (*DEVmodDelete)();      /* subroutine to delete a model and all instances */
   int (*DEVdelete)();         /* subroutine to delete an instance */
   int (*DEVsetic)();          /* routine to pick up device init. conditions from RHS */
   int (*DEVask)();            /* routine to ask about device details*/
   int (*DEVmodAsk)();         /* routine to ask about model details*/
   int (*DEVpzLoad)();         /* routine to load for pole-zero analysis */
   int (*DEVconvTest)();       /* convergence test function */
   int (*DEVsenSetup)();       /* routine to setup the device sensitivity info */
   int (*DEVsenLoad)();        /* routine to load the device sensitivity info */
   int (*DEVsenUpdate)();      /* routine to update the device sensitivity info */
   int (*DEVsenAcLoad)();      /* routine to load the device ac sensitivity info */
   void (*DEVsenPrint)();      /* subroutine to print out sensitivity info */
   int (*DEVsenTrunc)();       /* subroutine to print out sensitivity info */

   int DEVinstSize;            /* size of an instance */
   int DEVmodSize;             /* size of a model */

} SPICEdev;                    /* instance of structure for each possible type of
                               /* device */
```

The structure is divided up into two major sections. The first part, *DEVpublic* is designed to comply with the requirements for the front end interface, and provides the information it needs to parse the input. The remainder of the structure is for SPICE3 itself to identify the specific subroutines to call for each operation that needs to be performed on the specific device. The breakdown of the fields follows. For a complete description of the functions in the main part of the structure, see the device interface chapter, for the details of the public part of the structure, see the front end interface documentation.

1.4.1. name

```
char *name;
```

This is a character string constant which is the name of the class of devices, such as "Resistor". This field will be used for identification of the devices internally, via CKTtypelook, and for error and debugging messages as necessary.

1.4.2. description

```
char *description;
```

This is a longer character string which should provide a more detailed description of the device implemented by this package.

1.4.3. terms

```
int terms;
```

This is the number of actual terminals on the device, and is positive for an exact number. The front end interface supports a negative number here indicating a variable number of terminals, but SPICE3 does not support this option.

1.4.4. numNames

```
int numNames;
```

This is the actual number of names in the termNames array.

1.4.5. termNames;

```
char **termNames;
```

This is an array of character strings which provides names for the terminals of the device.

1.4.6. numInstanceParms

```
int numInstanceParms;
```

This is the count of the number of IFparm records in the instanceParms array.

1.4.7. instanceParms

```
IFparm *instanceParms;
```

This is a table giving the parameters and possible queries for the device. Each entry of the table is an IFparm structure which looks like:

```
typedef struct {
   char *keyword;
   int id;
   int dataType;
   char *description
}IFparm;
```

Where keyword is the name the user is expected to supply in the input, id is a simple integer code used to refer to the option internally, and is the unique identification of the parameter. The keyword should always be given entirely in lower case, and user input keywords are then converted to lower case to provide a case independent match. DataType indicates the type of argument the keyword takes, and indicates to the parser what type of value to read and pass to the DEVparam routine, such as integer, real, a vector of reals, or a device name. Description is a longer, but hopefully less than one line, description of the parameter.

1.4.8. numModelParms

```
int numModelParms;
```

The number of parameters described in the modelParms structure.

1.4.9. modelParms

```
IFparm *modelParms;
```

A table describing the model parameters and questions similar to the instanceParms table.

1.4.10. DEVparam

```
int (*DEVparam)();
```

A function which, given a specific device instance, a parameter, and a value, will assign that value to that field of the device. This function must be supplied by the implementor if there are any instance parameters, otherwise a value of NULL will inform SPICE3 that there are no legal device parameters.

1.4.11. DEVmodParam

```
int (*DEVmodParam)();
```

This function corresponds to the DEVparam function, but applies to model parameters.

1.4.12. DEVload

```
int (*DEVload)();
```

This function is used in the inner loop of the dc and transient analyses to load the sparse matrix and right hand side for solution. If this function pointer is left NULL, then no action will be taken for this device during dc and transient analysis.

1.4.13. DEVsetup

int (*DEVsetup)();

This function will be called once during parameter preprocessing, and all the one-time operations should be done here, such as allocating sparse matrix entries and getting pointers to them, as well as allocating space in the state table for data which must be retained from timepoint to timepoint.

1.4.14. DEVpzSetup

int (*DEVpzSetup)();

This function will generally be exactly the same as the DEVsetup function, and is called during the pole-zero setup operation. For almost all devices, this routine can be identical to, or simply another pointer to the same routine as the DEVsetup routine, since the only devices that need special processing during pole-zero setup are input voltage sources which must be removed from the circuit.

1.4.15. DEVtemperature

int (*DEVtemperature)();

This is the function where most of the parameter preprocessing takes place, and will be called before simulation takes place and whenever parameters or the simulation temperature have been changed.

1.4.16. DEVtrunc

int (*DEVtrunc)();

This function must perform the truncation error calculation on any energy storage elements or components of elements. If the device has no energy storage elements, this subroutine pointer should be NULL.

1.4.17. DEVfindBranch

```
int (*DEVfindBranch)();
```

This is a routine supplied by all devices which introduce a branch current equation and used by current controlled elements to find the equation corresponding to a named source.

1.4.18. DEVacLoad

```
int (*DEVacLoad)();
```

This routine corresponds to DEVload, but is used for ac analysis where complex quantities may need to be loaded. For a device which has exactly the same loading needs during ac analysis, this may be a pointer to the same routine as DEVload.

1.4.19. DEVaccept

```
int (*DEVaccept)();
```

This is a function which is called once at each timepoint after SPICE3 has decided that the current solution is acceptable. This can be used for one time calculations and breakpoint scheduling.

1.4.20. DEVdestroy

```
void (*DEVdestroy)();
```

This function is used for dismantling the data structures and should free all space used by all models and instances of the device.

1.4.21. DEVmodDelete

```
int (*DEVmodDelete)();
```

This function is used for partially dismantling the data structures by freeing a model and all instances to it, leaving the rest of the devices of the same type alone.

1.4.22. DEVdelete

```
int (*DEVdelete)();
```

This function deletes a single instance from the data structures, leaving everything else alone.

1.4.23. DEVsetic

```
int (*DEVsetic)();
```

This function is used to get device initial conditions from node initial conditions. Before calling this, the right hand side vector will be loaded with all the node initial conditions, so each device should look at the nodes it is attached to and set its initial condition fields to the proper value based on those voltages if the user has not specified them directly on the device.

1.4.24. DEVask

```
int (*DEVask)();
```

This is the function that allows user access to internal values of the device. Items from the instanceParms array with the IF_ASK bit set can be passed to it, and the corresponding value should be computed and returned if such a value is valid to compute at the time the routine is called.

1.4.25. DEVmodAsk

```
int (*DEVmodAsk)();
```

This function is exactly like DEVask, but applies to models, and is intended primarily to access the parameter set.

1.4.26. DEVpzLoad

```
int (*DEVpzLoad)();
```

This is the function used for evaluating and loading the matrix during pole-zero analysis.

1.4.27. DEVconvTest

```
int (*DEVconvTest)();
```

This function performs the per device convergence test. The function may return as soon as it has determined that a device has not converged and it has incremented ckt->CKTnoncon.

1.4.28. DEVinstSize

```
int DEVinstSize;
```

This is usually given as sizeof(DEVinstance), and is the amount of memory SPICE3 will allocate and link into the data structures for each instance that is created.

1.4.29. DEVmodSize

```
int DEVmodSize;
```

This is the size of a model structure which SPICE3 will allocate and link into the list structure for each model created.

1.4.30. DEVsenSetup

```
int (*DEVsenSetup)();
```

This is the special setup routine that is needed for sensitivity analysis, and performs additional computations and allocations needed by the sensitivity code.

1.4.31. DEVsenLoad

```
int (*DEVsenLoad)();
```

This routine loads the matrix and right hand side with the information needed for sensitivity analysis.

1.4.32. DEVsenUpdate

```
int (*DEVsenUpdate)();
```

This routine updates the matrix for sensitivity analysis.

1.4.33. DEVsenAcLoad

```
int (*DEVsenAcLoad)();
```

This is the modified ac load subroutine that is needed when sensitivity analysis is in progress.

1.4.34. DEVsenTrunc

```
int (*DEVsenTrunc)();
```

This subroutine performs truncation error computation during sensitivity analysis.

1.5. Converting a SPICE2 model

This section is a supplement to the above for those users who have existing SPICE2 models and wish to translate them to SPICE3. The places where corresponding sections of code are found in a SPICE2 implementation and the special precations that must be taken when translating a SPICE2 model will be shown.

There are no counterparts for the param or modParam functions in SPICE2, but these routines are quite straightforward to write, mostly requiring replication of prototype code and replacing a few key words.

The load function is almost identical to the SPICE2 subroutine LOAD or those subroutines it calls such as MOSFET, BJT, or JFET, requiring the same basic calculations, but specifically excluding those calculations related solely to convergence testing.

The setup, temperature, and setic functions are combinations of of the SPICE2 routines SETUP, ERRCHK, MATPTR, MATLOC, MODCHK, and TMPUPD. The operations of MATPTR and MATLOC are combined by using the SMP function SMPmakeElt which not only allocates the memory, but also returns a pointer to the data field at the requested matrix location. The one time only operations from these functions, such as the MATPTR and MATLOC code, should be placed in the setup function, the conversion of node initial conditions to device initial conditions from ERRCHK goes into setic, while the rest should be combined into the temperature function, including the temperature correction in all calculations.

The trunc function can simply call the CKTterr function to compute the truncation error on each of the energy storage elements within the device.

If the device adds a branch equation to the matrix which could be used to control a source, the findBranch function must be able to get the equation number for it from the name of the device, otherwise the findBranch function is not relevant.

The acLoad function is comparable to the code in ACLOAD in SPICE2.

The accept function is a new function which is used primarily by the incremental breakpoint table code to ensure that additional breakpoints are not added every iteration, but instead only once per timepoint.

The destroy, modDelete, and delete functions are new with SPICE3, and are quite simple to copy from existing implementations. No extra code is needed unless the device allocates additional memory beyond the basic data structure. The standard subroutine provided as a prototype breaks down all of the standard structure elements.

The ask and modAsk functions are also new, and are used to not only provide new functionality, but also to provide better access to many values used throughout the program.

1.6. Adapting the parser to the new device

The SPICE2 input language currently handled by SPICE3 is sufficiently irregular that the parser requires customization for each device. These customizations are relatively simple, and are spelled out in detail here.

1.6.1. INPdomodel.c

In the INP directory, the file INPdomodel.c provides a mapping from the model types which can appear on the ".MODEL" card, such as NPN, to the internal device names, such as BJT, and then a conversion using CKTtypelook to the device type number. If the device does not have a ".MODEL" card associated with it, this routine can be ignored entirely. The routine currently consists of a sequence of strcmp's in an if-then-elseif structure into which an additional elseif clause can easily be inserted for the new model type between any two existing clauses.

```
} else if(strcmp(typename,"new-device-model-name") == 0) {
   type = CKTtypelook("new-internal-type-name");
   INPmakeMod(modname, type, image);
```

1.6.2. INPfindLev.c

The file INPfindLev.c in the INP directory is used for MOSFETs to distinguish between the various levels of models (It is used by INPdomodel after determining that the device is a MOSFET of some kind). If another level of MOS model is being added, another case similar to the existing ones can be added for any additional levels which will be supported. Note that SPICE3 already has an additional level 4 model for BSIM.

1.6.3. inp_numnodes.c

This subroutine is used to translate from the first letter of the device name in SPICE2 format input to a number of nodes. This routine may go away at a later date, and is currently used only by the front end in subcircuit expansion. This subroutine is used to compensate for the difference between the SPICE3 device structure and the SPICE2 input format which allows a variable number of terminals on a bipolar junction transistor. Another case like the existing ones must be inserted for the first letter of the instance name of the new device. Note that unlike all of the other routines described in this section, this routine is in the FTE subdirectory, and is at the end of the file subckt.c instead of in a file by itself.

1.6.4. INPpas2.c

This subroutine does the bulk of the parsing of SPICE2 format input lines. For each possible first letter of a device instance, there is a case which processes the device's input line. INPpas2 calls a series of subroutines, one for each type of device to do the local parsing in order to make things simpler for compilers and to reduce the amount of code that must be recompiled when working on the parsing of a single device. INPpas2 should call a routine INP2*x*.c to parse the line corresponding to a first character of *x*. This subroutine should then parse the first part of the line which is usually quite irregular, and then use INPdevParse to handle the rest of the line. The subroutine INPdevParse can handle all of the parameters to the device that are of the form

```
keyword=value
keyword2 value
keyword3=value, value .... value
```

by reading the (keyword, value type) table provided in the previous section. All of the parameters that don't fall into this category, such as node names, model names, or parameters that don't have a keyword preceding them must be parsed directly in INP2*x*. The normal sequence is to use the utility routines supplied in the INP package to break off and properly handle the tokens on the first part of the line, up to but not including an optional keyword-less numeric parameter followed by keyword parameters, create the device with CKTcrtElt, bind the nodes identified with CKTbindNode, and hand the rest of the line to INPdevParse. The following subset of the routines in the INP package may be useful in writing the parser - see the section on the INP package for more details of these routines.

INPdomodel - determine device type from .model card
INPerrCat - concatenate error messages
INPevaluate - numerically evaluate the next token on the line
INPfindLev - determine MOS model level from .model card
INPgetMod - find a model given its name
INPgetTok - get the next token from the input line
INPgetValue - given an argument type, get an argument of that type
INPlookMod - look to see if a model with a given name has been defined
INPtermInsert - insert into terminal/node name symbol table
INPinsert - insert into device/model name symbol table

The simplest way to generate the code for this is to copy the code for an existing device and modify it as necessary.

After modifying all the necessary routines, those which were changed must be recompiled and the library rebuilt, then a new executable built.

1.7. Linking the new routines into the program

Linking the routines written above into the simulator is quite simple, requiring only a few simple changes to configuration files. In the CKT directory, there is a file called SPIinit.c which must be modified. The general form of this file is:

```
header and includes
external declarations
initialization of DEVices
additional constant declarations
```

The first and last sections need not be changed. The external declarations section contains a block of lines of the form:

```
extern SPICEdev XXXinfo;
```

where the XXX's are the device prefixes. Another similar line describing the new device must be added to this section.

The *initialization of DEVices* section includes the initialization of an array of pointers to the device descriptors which the program will use to access all of the device dependent data and subroutines. This section consists of a sequence of lines of the form:

```
&XXXinfo,
```

where XXX is the device prefix. The new device is simply added to this list, noting that the last device in the list is not followed by a comma.

The only step that remains is to recompile and relink. The exact details of this step depend on the operating system.

CHAPTER 2

The device to simulator interface

Integrating devices into the simulator is relatively easy as outlined in the previous chapter; Somewhat more complicated is writing the actual implementation code for the devices to be added. This chapter provides a detailed description of the routines needed by the simulator.

2.1. Preliminaries

The first step in implementing a new device is usually to look through the descriptions of the old devices and determine which one looks the most like the new device. Having done that, all the code for it should be copied. It is much easier to implement a new device by walking through the framework of an existing device and changing things as needed than it is to build the framework from scratch, and this procedure further enhances the consistency of the resulting system. After making the copy, the prefix used throughout should be changed to agree with the prefix chosen for the new device. Following this, each section of this chapter should be examined in detail along with the corresponding file being modified.

2.2. Data structures

The data structures for the devices are relatively unrestricted. There are very few strict rules, but many conventions which make life easier for anyone working on the code. There are four sources of data that will be available to each device at different times during its operation, with different characteristics for each one.

Data placed in the per model data structure will only be examined by the code implementing the device and will be relatively static - any data to be moved must be moved by the device code. Since the size of this structure is fixed when the model is defined, it must contain only data that is completely independent of any instance of the model. Space in this structure is reserved by adding elements to the model data structure definition in the device specific header file.

Data placed in the per instance data structure will also be examined only by the code implementing the device and will be similarly static. Per model data should generally not be copied to the instance structure, but precomputed values which will be used frequently and which combine several input device and model parameters or which are expensive to compute may be computed and placed here. Space in this structure is reserved by adding elements to the per instance data structure definition in the device specific header file.

During the iterative stages of the algorithm, the solution for the terminal voltages and branch currents of the previous iteration or timepoint will be available in the CKTrhsOld vector, indexed exactly as the corresponding current and voltage equations are in the new CKTrhs vector being built. These values **must not** be changed under any circumstances. Space in these vectors is allocated automatically for any circuit node either described by the front end or created internally through a call to CKTmkVolt, and to any circuit branch equation created by a call to CKTmkCur.

Finally, during transient solutions, there are a set of vectors known internally to SPICE3 as the *state* vectors. These vectors are all of the same size and are shifted around by the higher level portions of the simulator. At any given timepoint, the zero'th state vector contains data for the current timepoint, the first state vector contains data for the previous timepoint, the second one for the second previous timepoint, etc. These vectors are **automatically** switched around as needed, thus any data left in the zero'th vector at the end of a timepoint will be in the first vector at the first step of the next timepoint. Space in these vectors is allocated by the higher level portions of SPICE3 at the request of the device implementation code. In the DEVsetup routine, the device code is given the opportunity to mark off an arbitrary sized block of the vectors for its use. This space is **not** immediately available, but will be allocated after all the setup routines have run and the simulator has computed the total size of the vectors needed. Note that because of the way these vectors are automatically shifted by the simulator, the device code can only keep an offset into the vector of its data, not an address, and that because of the way they are defined, all possible state vectors may not have valid values in them during early timepoints. When running with predictor-corrector algorithms, the zeroth vector is filled with a predicted value before the first iteration at a new timepoint.

2.2.1. Model data structure

```
typedef struct sRESmodel {
    int RESmodType;                    /* device index for this device type */
    struct sRESmodel *RESnextModel;    /* pointer to next possible model in
                                       /* linked list */
    RESinstance * RESinstances;        /* pointer to list of instances that have this
                                       /* model */
    char *RESmodName;                  /* pointer to character string naming this model */

    double REStnom;                    /* temperature at which resistance measured */
    double REStempCoeff1;              /* first temperature coefficient of resistors */
    double REStempCoeff2;              /* second temperature coefficient of resistors */
    double RESsheetRes;                /* sheet resistance of devices in ohms/square */

                                       /* flags to indicate whether above parameters */
                                       /* were specified by the user (1) or obtained */
                                       /* from default mechanism (0). 1 bit values */
                                       /* used to save space. */
    unsigned REStnomGiven:1;           /* nominal temperature given? */
    unsigned REStc1Given:1;            /* tc1 given? */
    unsigned REStc2Given:1;            /* tc2 given? */
    unsigned RESsheetResGiven:1;       /* sheet resistance given? */
} RESmodel;
```

Figure 2.1
Example of a model data structure

The structure used to describe a device model is usually fairly straightforward since there should be no time-dependent values or values which vary from iteration to iteration, just a set which can be given by the user or computed by the setup or temperature routine. The fields of this structure generally are sorted into four major blocks: The required prefix, a block of double precision values, a block of integer values, and a block of single bit flags that are used to keep track of the source of the values. The required prefix is maintained by higher level code in SPICE3. By convention, the flags have the same name as the corresponding value with the word *Given* added on the end. Since these names are all internal, and thus subject only to a 31 character uniqueness constraint, this should not cause a conflict between the two names. Similarly, convention indicates that these flags should have a zero value, as produced when the structure is created, when the corresponding data location contains a program generated value and a non-zero value when the corresponding location contains user specified data.

2.2.2. Instance data structure

```
typedef struct sCAPinstance {
    struct sCAPmodel *CAPmodPtr;    /*pointer to our model */
    struct sCAPinstance
        *CAPnextInstance;           /* pointer to next instance of
                                    /* current model*/
    char *CAPname;                  /* pointer to character string naming this instance */

    int CAPposNode;                 /* number of positive node of capacitor */
    int CAPnegNode;                 /* number of negative node of capacitor */

    double CAPcapac;                /* capacitance */
    double CAPinitCond;             /* initial capacitor voltage if specified */

    int CAPstate;                   /* pointer to start of capacitor state vector */
#define CAPqcap CAPstate            /* charge on the capacitor */
#define CAPccap CAPstate+1          /* current through the capacitor */

    double *CAPposPosptr;           /* pointer to sparse matrix diagonal at
                                    /* (positive, positive) */
    double *CAPnegNegptr;           /* pointer to sparse matrix diagonal at
                                    /* (negative, negative) */
    double *CAPposNegptr;           /* pointer to sparse matrix off-diagonal at
                                    /* (positive, negative) */
    double *CAPnegPosptr;           /* pointer to sparse matrix off-diagonal at
                                    /* (negative, positive) */

    unsigned CAPcapGiven:1;         /* capacitance specified? */
    unsigned CAPicGiven:1;          /* init. cond. specified? */

} CAPinstance;
```

Figure 2.2
Example of an instance data structure

The per instance data structure is very similar to the per model data structure, but it includes a few additional features. The integer values will, if necessary, end in the offset of the first entry the instance has reserved in the state vectors. This will usually be followed by a set of defines for the names of the variables stored in these locations. Access to the data stored in these locations would then be through an expression such as *(ckt->CKTstate0+here->CAPqcap). The integers will then be followed by a section containing pointers. These pointers will primarily be to positions within the sparse matrix which are located in advance to eliminate the need to hunt through the entire matrix structure every iteration.

2.3. Input routines

These routines are used by the front end to communicate input parameters to the device. This means of communication eliminates the need for the front end to have any knowledge of the data structures used by the device internally.

2.3.1. DEVparam

```
int
RESparam(param, value, here, select)
    int param;
    IFvalue *value;
    RESinstance *here;
    IFvalue *select;
{
    switch(param) {
        case RES_RESIST:
            here->RESresist = value->rValue;
            here->RESresGiven = TRUE;
            break;
        default:
            return(E_BADPARM);
    }
    return(OK);
}
```

Figure 2.3
Example of a DEVparam function

The DEVparam function takes parameter values from the input parser and sets the appropriate field in the per instance data structure of the device. Four arguments will be provided: *param* is the integer identifier of the parameter to be set. The values for these identifiers will be chosen at the end of this appendix, but will be a set of unique integers, thus allowing a switch statement, and there will be symbolic constants for each of them which should be used for case labels. The symbolic constants are conventionally given names of the form XXX_PARMNAME where XXX is the unique prefix being used for the device and PARMNAME is a simple abbreviated name for the parameter. There is no need to assign values to these symbolic constants at this point, simply to maintain a list of them. *Value* is a union which can contain any type of value that the interface supports: flags, integers, reals, instances, character strings, nodes, and parse trees representing expressions, along with vectors of these types. Assume that the most convenient data type is provided at this point and use it, keeping track of that data type in the list of symbolic constants. *Here* is a pointer directly to the instance the parameter value is to be applied to. *Select* is another union which may contain either an integer or vector of integers as desired to specify a parameter more exactly. The intention is that select be used to isolate a single element in a vector or array to simplify the modification of a single element in a large structure without having to respecify the entire structure.

2.3.2. DEVmodParam

```
int
RESmParam(param, value, model, select)
   int param;
   IFvalue *value;
   register RESmodel *model;
   IFvalue *select;
{
   switch(param) {
      case RES_MOD_TC1:
         model->REStempCoeff1 = value->rValue;
         model->REStc1Given = TRUE;
         break;
      case RES_MOD_TC2:
         model->REStempCoeff2 = value->rValue;
         model->REStc2Given = TRUE;
         break;
      default:
         return(E_BADPARM);
   }
   return(OK);
}
```

Figure 2.4
Example of a DEVmodParam function

The DEVmodParam function is exactly analogous to the DEVparam function, but provides values for model parameters instead of instance parameters. The symbolic constants chosen should be distinct from those chosen for the DEVparam function and by convention take the form XXX_MOD_PARMNAME. Note that since in the parsing of SPICE2 format inputs, the model type is a parameter (NPN, PNP, D, etc), this parameter should be accepted, even if it carries no useful information, as the 'D' in the diode model.

2.4. Output routines

These routines are used by SPICE3 to obtain data from the device. Rather than giving SPICE3 knowledge of the data structures of the devices, these routines are provided to allow the higher level routines to query the devices for values of their internal variables. In some sense, these routines are exactly the opposite of the DEVparam and DEVmodParam routines.

2.4.1. DEVask

```
int
RESask(ckt, here, which, value, select)
   CKTcircuit *ckt;
   RESinstance *here;
   int which;
   IFvalue *value;
   IFvalue *select;
{
   switch(which) {
      case RES_CONDUCT:
         value->rValue = here->RESconduct;
         return(OK);
      case RES_RESIST:
         value->rValue = here->RESresist;
         return(OK);
      case RES_POWER:
         value->rValue = (*(ckt->CKTrhsOld + here->RESposNode) -
               *(ckt->CKTrhsOld + here->RESnegNode)) *
               here->RESconduct *
               (*(ckt->CKTrhsOld + here->RESposNode) -
               *(ckt->CKTrhsOld + here->RESnegNode));
         return(OK);
      default:
         return(E_BADPARM);
   }
   /* NOTREACHED */
};
```

Figure 2.5
Example of a DEVask function

This function is the reverse of the DEVsetParm function and has the same parameters with one addition. *Ckt* is the pointer to the circuit the device is in, and is used to get access to additional data such as that in the right hand side and state vectors. Generally, there will be far more legal queries to a device than there are parameters to the device, since this provides a way for both users and the program maintainer to get access to internal circuit variables as well as the parameters already input. Note that although the tables describing the queries available will be static, it may not always be valid to ask any given question, so the routine should watch for such a situation and provide an E_BADPARM error return.

2.4.2. DEVmodAsk

```
int
RESmodAsk(ckt, model, which, value, select)
   CKTcircuit *ckt;
   RESmodel*model;
   int which;
   IFvalue *value;
   IFvalue *select;
{
   switch(which) {
      case RES_MOD_TC1:
         value->rValue = model->REStempCoeff1;
         return(OK);
      case RES_MOD_TC2:
         value->rValue = model->REStempCoeff2;
         return(OK);
      default:
         return(E_BADPARM);
   }
};
```

Figure 2.6
Example of a DEVmodAsk function

The DEVmodAsk function provides access to model parameters in exactly the same manner as the DEVask function provides access to instance parameters. The DEVmodAsk function will generally only provide access to model parameters and precomputed functions of those parameters, since there are usually no other interesting values associated with the models. These parameters are not generally interesting to the user, but can be helpful for program maintenance and new device implementation testing.

2.4.3. DEVfindBranch

```
int
VSRCfindBr(ckt, inModel, name)
   register CKTcircuit *ckt;
   GENmodel *inModel;
   register IFuid name;
{
   register VSRCmodel *model = (VSRCmodel *)inModel;
   register VSRCinstance *here;
   int error;
   CKTnode *tmp;

   for( ; model != NULL; model = model->VSRCnextModel) {
      for (here = model->VSRCinstances; here != NULL;
            here = here->VSRCnextInstance) {
         if(here->VSRCname == name) {
            if(here->VSRCbranch == 0) {
               error = CKTmkCur(ckt, &tmp, heretmp, here->VSRCname, "branch");
               if(error) return(error);
               here->VSRCbranch = tmp->number;
            }
            return(here->VSRCbranch);
         }
      }
   }
   return(0);
}
```

Figure 2.7
Example of a DEVfindBranch function

The DEVfindBranch function is used to find the equation that represents a branch current in order to use that equation in another device. As an example, a current controlled voltage source needs to know the equation number of the current equation for the voltage source which will control it. If a device adds an additional equation to the circuit and that equation should be usable by a controlled source or similar construct, then a DEVfindBranch function must be written. This function should examine the source unique identifier given as its third argument and compare it with all possible device names and if a match is found, return the equation number. If the equation has not yet been generated, the DEVfindBranch function must create it and return its number. If no name match is found, zero should be returned. Note that rather than examining a single instance, this routine iterates through all instances of all models of the given device type given the pointer to the first model structure.

2.5. Structure decomposition routines

These routines are used to dismantle the data structures that have been built up. Failure to provide these should not cause the simulator to fail, but will cause a more rapid consumption of memory and possible problems with running out of memory when editing circuits and performing multiple runs.

2.5.1. DEVdestroy

```
void
RESdestroy(model)
   RESmodel **model;
{

   RESinstance *here;
   RESinstance *prev = NULL;
   RESmodel *mod = *model;
   RESmodel *oldmod = NULL;

   for( ; mod ; mod = mod->RESnextModel) {
      if(oldmod) FREE(oldmod);
      oldmod = mod;
      prev = (RESinstance *)NULL;
      for(here = mod->RESinstances ; here ; here = here->RESnextInstance) {
         if(prev) FREE(prev);
         prev = here;
      }
      if(prev) FREE(prev);
   }
   if(oldmod) FREE(oldmod);
   *model = NULL;
}
```

Figure 2.8
Example of a DEVdestroy function

The DEVdestroy functions all follow the same general pattern and loop through all instances and all models of the device type to free all memory used by them. For devices which have more complicated structures, such as those which allocate additional substructures or arrays, these structures and arrays should also be freed at this time, but only if they have already been created (do not free null pointers!). This function will only be used when deleting the entire circuit, so inter-device dependencies need not be considered in this function.

2.5.2. DEVmodDelete

```
int
RESmDelete(model, modname, modptr)
   RESmodel **model;
   char *modname;
   RESmodel *modptr;
{
   RESinstance *here;
   RESinstance *prev = NULL;
   RESmodel **oldmod;
   oldmod = model;
   for( ; *model ; model = &((*model)->RESnextModel)) {
      if( (*model)->RESmodName == modname |
            (modptr && *model == modptr) ) goto delgot;
      oldmod = model;
   }
   return(E_NOMOD);

delgot:
   *oldmod = (*model)->RESnextModel; /* cut deleted device out of list */
   for(here = (*model)->RESinstances ; here ; here = here->RESnextInstance) {
      if(prev) FREE(prev);
      prev = here;
   }
   if(prev) FREE(prev);
   FREE(*model);
   return(OK);

}
```

Figure 2.9
Example of a DEVmodDelete function

This function is designed to delete a model from the circuit. As a side effect, it should track down all instances of the model and delete them as well. At this point, node and matrix entry reference counting are not implemented, but in the future these reference counts will have to be decremented at this point as well. This function is provided for future extension capabilities since full reference counting will have to be performed to allow individual instances or models to be deleted, and as such is never called by the present front end.

2.5.3. DEVdelete

```
int
RESdelete(model, name, inst)
   RESmodel *model;
   char *name;
   RESinstance **inst;
{
   RESinstance **prev = NULL;
   RESinstance *here;

   for( ; model ; model = model->RESnextModel) {
      prev = &(model->RESinstances);
      for(here = *prev; here ; here = *prev) {
         if(here->RESname == name | (inst && here==*inst) ) {
            *prev= here->RESnextInstance;
            FREE(here);
            return(OK);
         }
         prev = &(here->RESnextInstance);
      }
   }
   return(E_NODEV);
}
```

Figure 2.10
Example of a DEVdelete function

This function must delete the single specified instance from the circuit. Reference counting will have to be taken into account in this function in a future version, but is not handled yet. This function is similarly never called by the present front end since full reference counting will have to be implemented and the specifications for this function extended before it can be used properly.

2.6. Processing routines

2.6.1. DEVsetup

```
    int
    CAPsetup(matrix, model, ckt, states)
        register SMPmatrix *matrix;
        register CAPmodel *model;
        CKTcircuit *ckt;
        int *states;
    {
        register CAPinstance *here;
        /* loop through all the capacitor models */
        for( ; model != NULL; model = model->CAPnextModel ) {

            /*Default Value Processing for Model Parameters */

            /* loop through all the instances of the model */
            for (here = model->CAPinstances; here != NULL ;
                    here=here->CAPnextInstance) {

/* macro to make elements with built in test for out of memory */
#define TSTALLOC(ptr, first, second) E
if((here->ptr = SMPmakeElt(matrix, here->first, here->second) E
        )==(double *)NULL){    return(E_NOMEM);}
                TSTALLOC(CAPposPosptr, CAPposNode, CAPposNode)
                TSTALLOC(CAPnegNegptr, CAPnegNode, CAPnegNode)
                TSTALLOC(CAPposNegptr, CAPposNode, CAPnegNode)
                TSTALLOC(CAPnegPosptr, CAPnegNode, CAPposNode)

                /* grab our share of space in the state vector */
                here->CAPqcap = *states;
                *states += 2;
            }
        }
        return(OK);
    }
```

Figure 2.11
Example of a DEVsetup function

This function performs the first step of preparing the device for simulation. By the time this function is called, the device will be attached to all necessary nodes and have most of its parameters set, although some parameters may be changed later. At this point, space in the simulator state vectors is reserved by saving the current value of **states* and then incrementing it by the number of double precision values needed. All parameter defaulting to constant values may also be done at this point, but not defaulting to computed values which depend on other parameters or model parameters. In addition, any additional equations needed should be allocated if they haven't been allocated already. Note that a DEVfindBranch function call earlier may have forced a device to allocate its equation early and it must not be allocated again. Finally, the entries in the sparse matrix where contributions from this devise will be added should be reserved and their addresses saved to speed performance during the simulation.

2.6.2. DEVpzSetup

This function is almost exactly the same as the DEVsetup function and, except for input voltage sources, should do exactly the same thing as the setup routine. For all other devices, except for those which replace themselves with instances of other devices during setup, the DEVsetup routine may be used as the DEVpzSetup routine.

2.6.3. DEVtemperature

This function completes the parameter preprocessing and prepares the device for simulation to begin. All model and instance parameters should have their final default values assigned here, either constants or based on computed values, even dependent on which other values have or have not been specified by the user at this point. It is recommended that all unconditional constant valued defaults (i.e. those based only on whether the specific parameter was or was not given and with a constant, not computed value) be performed in the DEVsetup routine.

This routine may be called several times, so it must make sure that it can properly handle being called again and again. Every time a model or instance parameter is changed, this routine will be called, additionally, every time the circuit temperature changes this routine will be called. The way temperature dependence is handled currently is to perform the following defaults in this routine.

1 If the model nominal temperature (temperature at which parameters were measured) is not specified, it should be set to the circuit nominal temperature (ckt->CKTnomTemp).

```
int
REStemp(model, ckt)

    register RESmodel *model;
    register CKTcircuit *ckt;
        /* perform the temperature update to the resistors
         * calculate the conductance as a function of the
         * given nominal and current temperatures.
         */
{
    register RESinstance *here;
    double factor;
    double difference;

    /*  loop through all the resistor models */
    for( ; model != NULL; model = model->RESnextModel ) {

        /* Default Value Processing for Resistor Models */
        if(!model->REStnomGiven) model->REStnom = ckt->CKTnomTemp;
        if(!model->REStc1Given) model->REStempCoeff1 = 0;
        if(!model->REStc2Given) model->REStempCoeff2 = 0;

        /* loop through all the instances of the model */
        for (here = model->RESinstances; here != NULL ;
                here=here->RESnextInstance) {

            /* Default Value Processing for Resistor Instance */
            if(!here->REStempGiven) here->REStemp = ckt->CKTtemp;
            if(!here->RESresGiven)  {
                (void)sprintf(emsg, "Resistor %s resistance=0, set to 10000
                        , here->RESname);
                (*(SPfrontEnd->IFerror))(ERR_WARNING,
                        "%s: resistance=0, set to 1000", &(here->RESname));
                here->RESresist=1000;
            }

            difference = here->REStemp - model->REStnom;
            factor = 1.0 + (model->REStempCoeff1)*difference +
                    (model->REStempCoeff2)*difference*difference;

            here->RESconduct = 1.0/(here->RESresist * factor);
        }
    }
    return(OK);
}
```

Figure 2.12
Example of a DEVtemperature function

2 If the instance temperature (temperature at which the specific instance will operate) is not specified, it should be set to the circuit operating temperature (ckt->CKTtemp).

Finally, all parameters should be adjusted for temperature and any pre-computation of derived values may be performed.

2.6.4. DEVsetic

```
int
CAPgetic(model, ckt)
   CAPmodel *model;
   CKTcircuit *ckt;
{
   CAPinstance *here;
   /*
    * grab initial conditions out of rhs array.   User specified, so use
    * external nodes to get values
    */

   for( ; model ; model = model->CAPnextModel) {
      for(here = model->CAPinstances; here ; here = here->CAPnextInstance) {
         if(!here->CAPicGiven) {
            here->CAPinitCond =
                  *(ckt->CKTrhs + here->CAPposNode) -
                  *(ckt->CKTrhs + here->CAPnegNode);
         }
      }
   }
   return(OK);
}
```

Figure 2.13
Example of a DEVsetic function

This function is used to convert node initial conditions to device initial conditions. Before calling the DEVsetic function, the simulator will put all of the node initial condition values in the ckt->CKTrhs array, so for any device initial condition that was not individually specified, the condition should be computed from the node voltages present in ckt->CKTrhs.

2.6.5. DEVload

```
int
CAPload(inModel,ckt)
    GENmodel *inModel;  /* starting model pointer */
    register CKTcircuit *ckt;   /* the circuit to work on */
        /* actually load the current capacitance value into the
         * sparse matrix previously provided
         */
{
    register CAPmodel *model = (CAPmodel*)inModel;  /* current model */
    register CAPinstance *here; /* current instance */
    register int cond1; /* the condition for using initial condition */
    double vcap;    /* voltage across the capacitor */
    double geq; /* equivalent conductance */
    double ceq; /* equivalent current */
    int error;  /* Var. to hold error return codes */

    /* check if capacitors are in the circuit or are open circuited */
    if(ckt->CKTmode & (MODETRAN|MODEAC|MODETRANOP) ) {
        /* evaluate device independent analysis conditions */
        cond1=
            ( ( (ckt->CKTmode & MODEDC) &&
                (ckt->CKTmode & MODEINITJCT) )
            |( ( ckt->CKTmode & MODEUIC) &&
                ( ckt->CKTmode & MODEINITTRAN) ) ) ;
        /*  loop through all the capacitor models */
        for( ; model != NULL; model = model->CAPnextModel ) {

            /* loop through all the instances of the model */
            for (here = model->CAPinstances; here != NULL ;
                    here=here->CAPnextInstance) {

                if(cond1) {
                    vcap = here->CAPinitCond;
                } else {
                    vcap = *(ckt->CKTrhsOld+here->CAPposNode) -
                        *(ckt->CKTrhsOld+here->CAPnegNode) ;
                }
                if(ckt->CKTmode & (MODETRAN | MODEAC)) {
#ifndef PREDICTOR
                    if(ckt->CKTmode & MODEINITPRED) {
                        *(ckt->CKTstate0+here->CAPqcap) =
                            *(ckt->CKTstate1+here->CAPqcap);
                    } else { /* only const caps - no polynomials */
#endif /* PREDICTOR */
                        *(ckt->CKTstate0+here->CAPqcap) = here->CAPcapac * vcap;
                        if((ckt->CKTmode & MODEINITTRAN)) {
                            *(ckt->CKTstate1+here->CAPqcap) =
                                *(ckt->CKTstate0+here->CAPqcap);
                        }
#ifndef PREDICTOR
                    }
#endif /* PREDICTOR */
                        error = NIintegrate(ckt,&geq,&ceq,here->CAPcapac,
                                here->CAPqcap);
                        if(error) return(error);
                        if(ckt->CKTmode & MODEINITTRAN) {
                            *(ckt->CKTstate1+here->CAPccap) =
                                *(ckt->CKTstate0+here->CAPccap);
                        }
                        *(here->CAPposPosptr) += geq;
                        *(here->CAPnegNegptr) += geq;
                        *(here->CAPposNegptr) -= geq;
                        *(here->CAPnegPosptr) -= geq;
                        *(ckt->CKTrhs+here->CAPposNode) -= ceq;
                        *(ckt->CKTrhs+here->CAPnegNode) += ceq;
                    }
                }
            }
        }
    return(OK);
}
```

Figure 2.14
Example of a DEVload function

The DEVload function is the most important function in the device implementation. The DEVload function is responsible for evaluating all instances at each iteration in dc and transient analyses and for loading the sparse matrix and right hand side vector with the appropriate values. Because of the variety of startup conditions and analysis type special conditions, this function can be quite complicated, but as the most frequently called function must be as efficient as possible.

There are six major forms of operation of the DEVload function, controlled by the value of the variable ckt->CKTmode. To begin, the mode determines where the DEVload function obtains the terminal voltages of the device for the evaluation. The six cases are:

MODEINITFLOAT

This is the most common case and is used when iterating to convergence. The terminal voltages should be obtained by looking in the previous solution vector ckt->CKTrhsOld.

MODEINITPRED

This is the case used for the first iteration at any given timepoint. In this case, historically, SPICE has had to predict the terminal voltages. For non-linear devices where junction voltages have been being saved, a linear prediction of the junction voltages is performed, otherwise, the values from the previous solution vector are used. The most recent version of SPICE3 treats this case in two different ways. If the preprocessor symbol "PREDICTOR" is defined, this case can be treated exactly as MODEINITFLOAT since the simulator will perform the needed prediction, while if it is not defined, the prediction should be done as in SPICE2.

MODEINITTRAN

This is a special case of MODEINITPRED and is used for the first iteration at the first timepoint after the dc solution. This is needed because the prediction code historically used two previous solutions to perform the prediction, but for the first timepoint only one previous solution is available. This special case is no longer important since SPICE3 now either performs the prediction as in the MODEINITPRED case described above, generates data to fill in the data for the non-existent second previous timepoint, thus MODEINITTRAN can always be considered equivalent to MODEINITPRED.

MODEINITFIX

This is the case which considers the effect of the "off" specifications on device lines. Other than forcing devices to be off if necessary, this case behaves exactly as MODEINITFLOAT.

MODEINITJCT

This is used for the first iteration of the circuit and is used to initialize the junction voltages to something reasonable. Where there is no advance knowledge, such as from initial conditions, the best results have been obtained in the past from initializing the junctions to V_{t_0} or its equivalent to allow the device to readily move either direction in a single iteration.

MODEINITSMSIG

This case is used to store special values needed for small signal analyses. Traditionally, SPICE2 did not save very many of the values computed as intermediate values, but some of them were needed during the small signal analyses, where such things as capacitor charges were not needed. This mode triggered the DEVload function to save these extra values *in place of* the normally saved information in the state vector. This makes the use of things in the state vector confusing since it is necessary to know the exact context of the call to the function to know what values are stored in the vector. In SPICE3, there is a separate location in the per instance data structure used to store each of these additional values, and most are simply computed in place during the course of the evaluation. If a call is made with this flag set, any values which may occasionally not be computed that are needed for small signal analysis should be explicitly computed and stored. Terminal voltages are again picked up from the previous right hand side vector, but the matrix and new right hand side vector need not be recomputed.

Once the operating condition of the device is determined, the necessary derivatives need to be computed. The matrix ckt->CKTmatrix must be filled with the jacobian of of the circuit. In the absence of the preprocessor define ''NEWCONV'' the convergence tests described as making up the DEVconvTest function should be performed here, otherwise they are performed in that function.

2.6.6. DEVtrunc

```
int
MOS1trunc(inModel, ckt, timeStep)
   GENmodel *inModel;
   register CKTcircuit *ckt;
   double *timeStep;
{
   register MOS1model *model = (MOS1model *)inModel;
   register MOS1instance *here;

   for( ; model != NULL; model = model->MOS1nextModel) {
      for(here = model->MOS1instances ; here != NULL ;
            here = here->MOS1nextInstance) {
         CKTterr(here->MOS1qgs, ckt, timeStep);
         CKTterr(here->MOS1qgd, ckt, timeStep);
         CKTterr(here->MOS1qgb, ckt, timeStep);
      }
   }
   return(OK);
}
```

Figure 2.15
Example of a DEVtrunc function

The DEVtrunc function is used to compute the truncation error for each energy storage device in the circuit. The CKTterr function is generally used. For a given set of charge/current vectors CKTterr computes the acceptable truncation error, and from that it computes the maximum timestep that could have been taken to reach this point consistent with the acceptable error. The DEVtrunc function reduces its timeStep argument to be the minimum of its previous value and the smallest timestep found for any of the instances it processes.

2.6.7. DEVconvTest

```
int
DIOconvTest(inModel, ckt)
   GENmodel *inModel;
   CKTcircuit *ckt;
      /* Check the diodes for convergence */
{
   register DIOmodel *model = (DIOmodel*)inModel;
   register DIOinstance *here;
   double delvd, vd, cdhat, cd;
   double tol;
   /*  loop through all the diode models */
   for( ; model != NULL; model = model->DIOnextModel ) {
      /* loop through all the instances of the model */
      for (here = model->DIOinstances; here != NULL ;
            here=here->DIOnextInstance) {
         /*
          *  initialization
          */
         vd = *(ckt->CKTrhsOld+here->DIOposPrimeNode)-
               *(ckt->CKTrhsOld + here->DIOnegNode);

         delvd=vd- *(ckt->CKTstate0 + here->DIOvoltage);
         cdhat= *(ckt->CKTstate0 + here->DIOcurrent) +
               *(ckt->CKTstate0 + here->DIOconduct) * delvd;
         cd= *(ckt->CKTstate0 + here->DIOcurrent);
         /*
          *  check convergence
          */
         tol=ckt->CKTreltol*
               MAX(FABS(cdhat), FABS(cd))+ckt->CKTabstol;
         if (FABS(cdhat-cd) > tol) {
            ckt->CKTnoncon++;
            return(OK); /* don't need to check any more diodes */
         }
      }
   }
   return(OK);
}
```

Figure 2.16
Example of a DEVconvTest function

The DEVconvTest function performs the necessary convergence testing to determine whether the currents in each device have converged. If the device has no energy storage elements, this routine is not necessary, but if there are energy storage elements, this function should test the change in the currents since the last iteration and increment ckt->CKTnoncon if the change exceeds the permitted tolerance. Code to perform this testing is integrated into the DEVload function if the preprocessor symbol NEWCONV is not defined, but if it defined it is in this routine so that unnecessary evaluations of the device are not performed.

2.6.8. DEVaccept

The DEVaccept function is called by the simulator at each timepoint once it has been determined that the current solution is adequate and will be accepted. This allows the device code to do any clean up or one time only operations necessary. Typically, this is used by devices to check for breakpoints and set future breakpoints.

2.6.9. DEVacLoad

```
int
CAPacLoad(inModel, ckt)
   GENmodel *inModel;
   register CKTcircuit *ckt;
{
   register CAPmodel *model = (CAPmodel*)inModel;
   double val;
   register CAPinstance *here;

   for( ; model != NULL; model = model->CAPnextModel) {
      for( here = model->CAPinstances ; here != NULL;
            here = here->CAPnextInstance) {

         val = ckt->CKTomega * here->CAPcapac;
         *(here->CAPposPosptr +1) += val;
         *(here->CAPnegNegptr +1) += val;
         *(here->CAPposNegptr +1) -= val;
         *(here->CAPnegPosptr +1) -= val;
      }
   }
   return(OK);
}
```

Figure 2.18
Example of a DEVacLoad function

The DEVacLoad function is a variation of the DEVload function that is used when performing ac analysis. The DEVacLoad function will not evaluate the device based on current information in the

```
int
TRAaccept(ckt, inModel)
   register CKTcircuit *ckt;
   GENmodel *inModel;
{
   register TRAmodel *model = (TRAmodel *)inModel;
   register TRAinstance *here;
   register int i=0, j;
   double v1, v2, v3, v4;
   double v5, v6, d1, d2, d3, d4;
   double *from, *to;
   int error;
   /*  loop through all the transmission line models */
   for( ; model != NULL; model = model->TRAnextModel ) {
      /* loop through all the instances of the model */
      for (here = model->TRAinstances; here != NULL ;
            here=here->TRAnextInstance) {
         if( (ckt->CKTtime - here->TRAtd) > *(here->TRAdelays+6)) {
            /* shift! */
            for(i=2;i<here->TRAsizeDelay &&
               (ckt->CKTtime - here->TRAtd > *(here->TRAdelays+3*i));i++)
                  { /* loop does it all */ ; }
            i -= 2;
            for(j=i; j<=here->TRAsizeDelay; j++) {
               from = here->TRAdelays + 3*j;
               to = here->TRAdelays + 3*(j-i);
               *(to) = *(from);
               *(to+1) = *(from+1);
               *(to+2) = *(from+2);
            }
            here->TRAsizeDelay -= i;
         }
         if(ckt->CKTtime - *(here->TRAdelays+3*here->TRAsizeDelay) >
               ckt->CKTminBreak) {
            if(here->TRAallocDelay <= here->TRAsizeDelay) {
               /* need to grab some more space */
               here->TRAallocDelay += 5;
               here->TRAdelays = (double *)REALLOC((char *)here->TRAdelays,
                  (here->TRAallocDelay+1)*3*sizeof(double));
            }
            here->TRAsizeDelay ++;
            to = (here->TRAdelays  +3*here->TRAsizeDelay);
            *to = ckt->CKTtime;
            to = (here->TRAdelays+1+3*here->TRAsizeDelay);
            *to = ( *(ckt->CKTrhsOld + here->TRAposNode2)
                -*(ckt->CKTrhsOld + here->TRAnegNode2))
               + *(ckt->CKTrhsOld + here->TRAbrEq2)*
                  here->TRAimped;
            *(here->TRAdelays+2+3*here->TRAsizeDelay) =
               ( *(ckt->CKTrhsOld + here->TRAposNode1)
                    -*(ckt->CKTrhsOld + here->TRAnegNode1))
                   + *(ckt->CKTrhsOld + here->TRAbrEq1)*
                      here->TRAimped;
                v1 = *(here->TRAdelays+1+3*here->TRAsizeDelay);
                v2 = *(here->TRAdelays+1+3*(here->TRAsizeDelay-1));
                v3 = *(here->TRAdelays+1+3*(here->TRAsizeDelay-2));
                v4 = *(here->TRAdelays+2+3*here->TRAsizeDelay);
                v5 = *(here->TRAdelays+2+3*(here->TRAsizeDelay-1));
                v6 = *(here->TRAdelays+2+3*(here->TRAsizeDelay-2));
                d1 = (v1-v2)/ckt->CKTdeltaOld[0];
                d2 = (v2-v3)/ckt->CKTdeltaOld[1];
                d3 = (v4-v5)/ckt->CKTdeltaOld[0];
                d4 = (v5-v6)/ckt->CKTdeltaOld[1];
                if( (FABS(d1-d2) >= here->TRAreltol*MAX(FABS(d1), FABS(d2))+
                      here->TRAabstol) |
                      (FABS(d3-d4) >= here->TRAreltol*MAX(FABS(d3), FABS(d4))+
                      here->TRAabstol) ) {
                   /* derivative changing - need to schedule after delay */
                   error = CKTsetBreak(ckt,
                         *(here->TRAdelays+3*here->TRAsizeDelay -3) +
                         here->TRAtd);
                   if(error) return(error);
                }
            }
        }
    }
    return(OK);
}
```

Figure 2.17
Example of a DEVaccept function

right hand side vector, but will load the matrix and right hand side based on the data at the last dc analysis, which will have been followed by a call to the DEVload function with the MODEINITSMSIG bit set. This should have saved enough data in the instance's own data structures to allow the matrix to be quickly loaded during an ac analysis with the proper conductance evaluated at the frequency f=i×ckt→CKTomega.

2.6.10. DEVpzLoad

```
int
CAPpzLoad(inModel,ckt,s)
   GENmodel *inModel;
   CKTcircuit *ckt;
   register SPcomplex *s;
{
   register CAPmodel *model = (CAPmodel*)inModel;
   double val;
   register CAPinstance *here;

   for( ; model != NULL; model = model->CAPnextModel) {
      for( here = model->CAPinstances;here != NULL;
            here = here->CAPnextInstance) {

         val = here->CAPcapac;
         *(here->CAPposPosptr ) += val * s->real;
         *(here->CAPposPosptr +1) += val * s->imag;
         *(here->CAPnegNegptr ) += val * s->real;
         *(here->CAPnegNegptr +1) += val * s->imag;
         *(here->CAPposNegptr ) -= val * s->real;
         *(here->CAPposNegptr +1) -= val * s->imag;
         *(here->CAPnegPosptr ) -= val * s->real;
         *(here->CAPnegPosptr +1) -= val * s->imag;
      }
   }
   return(OK);
}
```

Figure 2.19
Example of a DEVpzLoad function

The DEVpzLoad function is very similar to the DEVacLoad function, but evaluates the conductance at the complex frequency s which is not the purely imaginary frequency f used in the ac analysis.

2.7. Sensitivity routines There are five subroutines which are entirely dedicated to performing sections of the sensitivity analysis. The modifications to SPICE3 to support sensitivity analysis are covered in an additional report Chou88a which provides the details of that implementation. The implementation of sensitivity analysis for a device requires that five routines, DEVsenSetup, DEVsenLoad, DEVsenUpdate, DEVsenAcLoad, and DEVsenTrunc be written as described in that report. If no sensitivity capability is required, these routines can be left out.

2.8. More data structures and constants

In addition to the basic per model and per instance data structures, there are a few other structures that must be defined to provide the complete description of the device type to the higher levels of the simulator.

2.8.1. DEVinstSize

The DEVinstSize field should be initialized to the size of the per instance data structure needed by the device, and is usually initialized using the preprocessor construct *sizeof* to automatically track changes.

2.8.2. DEVmodSize

The DEVmodSize field must be initialized to the size of the per model data structure needed by the device, and is usually initialized using *sizeof* as the DEVinstSize field is.

2.8.3. instanceParms

```
static IFparm RESpTable[] = {
    IOP("resistance",   RES_RESIST,     IF_REAL,   "Resistance"),
    IOP("w",            RES_WIDTH,      IF_REAL,   "Width"),
    IOP("l",            RES_LENGTH,     IF_REAL,   "Length"),
    IOP("c",            RES_CURRENT,    IF_REAL,   "Current"),
    IOP("p",            RES_POWER,      IF_REAL,   "Power")
};
```

Figure 2.20
Example of a instanceParms definition

The instanceParms element in the structure is eventually exported all the way to the front end and is an array which describes the acceptable parameters and queries for the device instances. The array should be a static, initialized array containing instances of the macros IP, OP, and IOP. These macros are defined in the DEVdefs.h header file and are used to specify parameters which are Input Parameters, Output Parameters, or Input and Output Parameters. These macros perform two functions:

They shorten the text needed to describe a parameter, so that it can frequently be put on a single line, by automatically putting the IF_SET and IF_ASK bit in the proper subfield.

They allow the program to fit on a machine such as the IBM PC/AT which has a limited data area for initialized variables where the descriptive text must be left out.

The contents of the structure comes from the lists of parameters and questions prepared while writing the DEVparam and DEVask functions. For each legal *param* argument to the DEVparam function, a line of the form:

```
IP(name, const, type, description)
```

should be present in the instanceParms array. The *name* field is the short name or abbreviation that the user will use to refer to the parameter, *const* is the corresponding symbolic constant used in the DEVparam function as a case label, *type* is a coded variable type field that corresponds to the type of data required in the DEVparam function, and *description* is a longer description of the parameter designed to help a user unfamiliar with the abbreviations used. Further details of the coding of the type field can be found in the description of the IFparm data structure in the front end to simulator interface appendix.

2.8.4. modelParms

```
static IFparm RESmPTable[] = {
    IOP("tc1",    RES_MOD_TC1,    IF_REAL,    "First order temp. coefficient"),
    IOP("tc2",    RES_MOD_TC2,    IF_REAL,    "Second order temp. coefficient"),
    IP ("r",      RES_MOD_R,      IF_FLAG,    "Device is a resistor model")
};
```

Figure 2.21
Example of a modelParms definition

The modelParms structure is exactly comparable to the instanceParms structure, but contains descriptions of the parameters to the device models instead of the instances of those models.

2.8.5. Symbolic constants

```
/* instance parameters */
#define RES_RESIST 1
#define RES_WIDTH 2
#define RES_LENGTH 3
#define RES_CONDUCT 4
#define RES_RESIST_SENS 5
#define RES_CURRENT 6
#define RES_POWER 7
#define RES_SENS_REAL 8
#define RES_SENS_IMAG 9
#define RES_SENS_MAG 10
#define RES_SENS_PH 11
#define RES_SENS_CPLX 12
#define RES_SENS_DC 13

/* model parameters */
#define RES_MOD_TC1 101
#define RES_MOD_TC2 102
#define RES_MOD_RSH 103
#define RES_MOD_DEFWIDTH 104
#define RES_MOD_NARROW 105
#define RES_MOD_R 106
```

Figure 2.22
Example of a set of
symbolic constant definitions

Once all of the functions to implement the device are complete, the symbolic constants used to refer to parameters must be defined. The constants should be grouped into those which apply to instances and those which apply to models. The exact values used for these symbolic constants is not important, but it is recommended that they be distinct to help catch errors and that the two sets of symbolic constants each use a contiguous set of integers to make it possible for compilers to generate better code.

2.8.6. Function declarations

```
extern int RESask();
extern int RESdelete();
extern void RESdestroy();
extern int RESload();
extern int RESmAsk();
extern int RESmDelete();
extern int RESmParam();
extern int RESparam();
extern int RESpzLoad();
extern int RESsenAcLoad();
extern int RESsenLoad();
extern int RESsenSetup();
extern void RESsenPrint();
extern int RESsetup();
extern int REStemp();
```

Figure 2.23
Example of a set of
function declarations

The device specific header file should contain a set of declarations for all of the functions used in the device implementation.

2.9. The SPICEdev structure

```
SPICEdev RESinfo = {
   {
      "Resistor",
      "Simple linear resistor",

      sizeof(RESnames)/sizeof(char *),
      sizeof(RESnames)/sizeof(char *),
      RESnames,

      sizeof(RESpTable)/sizeof(IFparm),
      RESpTable,

      sizeof(RESmPTable)/sizeof(IFparm),
      RESmPTable,
   },

   RESparam,%/* the DEVparam function */
   RESmParam,%/* the DEVmodParam function */
   RESload,%/* the DEVload function */
   RESsetup,%/* the DEVsetup function */
   REStemp,%/* the DEVtemp function */
   NULL,%/* the DEVtrunc function */
   NULL,%/* the DEVfindBranch function */
   RESload,%/* the DEVacLoad and DEVload functions are identical */
   NULL,%/* the DEVaccept function */
   RESdestroy,%/* the DEVdestroy function */
   RESmDelete,%/* the DEVmDelete function */
   RESdelete,%/* the DEVdelete function */
   NULL,%/* the DEVsetic function */
   RESask,%/* the DEVask function */
   NULL,%/* the DEVmodAsk function */
   RESpzLoad,%/* the DEVpzLoad function */
   NULL,%/* the DEVconvTest function */
   RESsSetup,%/* the DEVsenSetup function */
   RESsLoad,%/* the DEVsenLoad function */
   NULL,%/* the DEVsenUpdate function */
   RESsAcLoad,%/* the DEVsenAcLoad function */
   RESsPrint,%/* the DEVsenPrint function */
   NULL,%/* the DEVsenTrunc function */

   sizeof(RESinstance),
   sizeof(RESmodel),

};
```

Figure 2.24
Example of a SPICEdev
structure initialization

The SPICEdev structure is an initialized constant structure which contains pointers to all of the other descriptive data structures and functions defined in this appendix. By exporting this one structure to the higher level routines, the details of which functions are required, the exact names of them, and even the set of functions that will be used at run time is not fixed at this time. The next level of software can use any or all of the functions provided by the device interface and will only have a single reference to the entire device rather than about 30 of them distributed through the code.

References

Chou88a.

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

Kern78a.

Kernighan, Brian W. and Ritchie, Dennis M., *The C Programming Language,* Prentice-Hall, Inc., Englewood Cliffs, New Jersey (1978).