SIMLAB^{TM} PROGRAMMER'S GUIDE

A. Lumsdaine, M. Silveira, J. White

Research Laboratory of Electronics
Department of Electrical Engineering and Computer Science
Massachusetts Institute of Technology
Cambridge, MA 02139

Abstract

SIMLAB is a circuit simulation environment consisting of a flexible, user-friendly front-end operating in conjunction with a sophisticated and versatile simulation engine. The program is specifically designed to be used as an educational tool and as a research platform. SIMLAB can be operated in either batch or interactive mode. This manual describes how to customize the SIMLAB program.

For information on using SIMLAB as a standalone program, see the *SIMLAB User's Guide*.

LICENSE AGREEMENT

This Agreement gives you, the LICENSEE, certain rights and obligations. By using the software, you indicate that you have read, understood, and will comply with the terms.

Permission to use, copy and modify for internal, noncommercial purposes is hereby granted. Any distribution of this program or any part thereof is strictly prohibited without prior written consent of M.I.T.

Title to copyright to this software and to any associated documentation shall at all times remain with M.I.T. and LICENSEE agrees to preserve same. LICENSEE agrees not to make any copies except for LICENSEE'S internal noncommercial use, or to use separately any portion of this software without prior written consent of M.I.T. LICENSEE agrees to place the appropriate copyright notice on any such copies.

Nothing in this Agreement shall be construed as conferring rights to use in advertising, publicity or otherwise any trademark or the name of "Massachusetts Institute of Technology" or "M.I.T."

M.I.T. MAKES NO REPRESENTATIONS OR WARRANTIES, EXPRESS OR IMPLIED. By way of example, but not limitation, M.I.T. MAKES NO REPRESENTATIONS OR WARRANTIES OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE OR THAT THE USE OF THE LICENSED SOFTWARE COMPONENTS OR DOCUMENTATION WILL NOT INFRINGE ANY PATENTS, COPYRIGHTS, TRADEMARKS OR OTHER RIGHTS. M.I.T. shall not be held liable for any liability nor for any direct, indirect or consequential damages with respect to any claim by LICENSEE or any third party on account of or arising from this Agreement or use of this software.

Contents

1 Introduction

SIMLAB™ is a circuit simulation environment consisting of a flexible, user-friendly front-end operating in conjunction with a sophisticated and versatile simulation engine. The program is specifically designed to be used as an educational tool and as a research platform. SIMLAB can be operated in either batch or interactive mode.

One of SIMLAB's primary uses is as a research platform for circuit simulation algorithms. As such, one of its major features is its modularity, which allows customization of the base program through a simple functional interface. There are three main areas of the program for which customization is easily facilitated:

- Adding a new numerical algorithm
- Adding a new device type
- Adding a new interactive command
- Adding a new environmental variable
- Adding a new circuit file option parameter

Note that SIMLAB already has a large collection of devices, algorithms, commands, variables and circuit file option parameters.

This guide describes how to go about customizing SIMLAB. It is assumed that the reader is at least somewhat familiar with circuit simulation (see [1, 2, 3]) and with using the SIMLAB program (see the *SIMLAB User's Guide*[4]). Familiarity with the C programming language and the UNIX operating system is highly recommended [5].

The SIMLAB program was developed at MIT under support from Analog Devices, the NSF PYI program, and DARPA contract N00014-87-K-825. Miguel Silveira was partially supported by the Portuguese INVOTAN committee and Andrew Lumsdaine was also supported by an AEA/Dynatech faculty development fellowship. Installing SIMLAB in your computer system is done with the aid of `makelinks`, a tool for avoiding code duplication developed at M.I.T. by G. Adams, D. Siegel and S. Narasimhan. The program uses a circuit parser written by P. Moore and a sparse matrix solver based on work by K. Kundert, both of whom were students in computer-aided design at U.C. Berkeley. The front-end interpreter for the program was inspired by the `xlisp` program written by David Michael Betz and uses the readline library from the Free Software Foundation's `bash` program. The plotting package included with the SIMLAB distribution is `xgraph`, a two-dimensional plotting package developed by D. Harrison at U.C. Berkeley.

2 Interfacing to Simlab

In this section, we will discuss, in general, how to interface to, and customize, SIMLAB. Detailed discussion of these procedures is contained in the rest of this guide.

2.1 Making Simlab

As delivered on the distribution medium, the SIMLAB directory hierarchy is as follows (let `$SIMLAB` be the SIMLAB root directory, e.g., `/usr/local/src/simlab`):

- `$SIMLAB/bin` contains SIMLAB executable
- `$SIMLAB/doc` contains SIMLAB documentation
- `$SIMLAB/custom` contains SIMLAB customizing hooks
- `$SIMLAB/examples` contains example circuit and configuration files for SIMLAB.
- `$SIMLAB/src` contains SIMLAB source code in several subdirectories
- `$SIMLAB/util` contains sources for some utility programs used by SIMLAB.

The configuration of the compiled version of SIMLAB is controlled with the file `$SIMLAB/src-/include/config.h`. There are several preprocessor macros in this file which control compilation options. It is in this file that you set up which algorithm packages (e.g., waveform newton, relaxation) you want to include in your compiled SIMLAB. At present, SIMLAB seems to be portable across several machine architectures and flavors of operating systems. However, it may be necessary to put control flags for these sorts of things into config.h as well.

As delivered with the distribution, `config.h` is set for the minimal configuration, i.e. no optional algorithm packages will be compiled in. This default version of SIMLAB includes forward-Euler, backward-Euler, trapezoidal, second-order Gear, and hybrid trapezoidal-Gear integration, Newton-Raphson nonlinear system solution, and sparse Gaussian elimination linear system solution.

To make SIMLAB, type "make all" at the command line in the `$SIMLAB` directory. This will do the following:

1. Create source directories `src-db`, `src-opt`, `src-prof` in which to do the actual compilation of the debugging, optimized, and profiling versions of SIMLAB, respectively.

These directories are created with the makelinks utility, which is included in the SIMLAB distribution. Makelinks creates copies of the `src` directory structure and makes symbolic links to the files contained in `src`. This allows us to have only one copy of the source code, but at the same time support multiple compilations.

2. Create library directories `lib-db`, `lib-opt`, `lib-prof` for the debugging, optimized, and profiling versions of the SIMLAB libraries, respectively.

3. Compile the programs `simlab`, `simlab.opt`, and `simlab.prof` , the debugging, optimized, and profiling versions of the SIMLAB program, respectively. The executables are put into `$SIMLAB/bin`.

4. Compile a version of a plotting program, called "`simgraph`" and put it into `$SIMLAB/bin`. SIMLAB uses `simgraph` for plotting output. The `simgraph` program is the `xgraph` program (a plotting package developed by D. Harrison at U.C. Berkeley), modified to handle SIMLAB output. The `simgraph` program is used by SIMLAB for plotting, but it can also be used as a standalone program for plotting SIMLAB output.

5. Create a `$SIMLAB/custom/Makefile` and `$SIMLAB/custom/src/Makefile`. These are the Makefiles that you will want to use in your work directory (see Section 2.2 below).

If for some reason, you do not want all three versions of SIMLAB, you can just specify "`make simlab`" (or whichever version you want). You should be sure then to specify "`make simgraph`" and "`make Makefile.custom`" (to create `$SIMLAB/custom/Makefile` and `$SIMLAB/custom/src/Makefile`) as well. The distribution `Makefile` allows you to specify "`make min`" in order to just make `simlab` (debugging version), `simgraph`, and `Makefile.custom`.

If there are any errors in the make, you should inspect `config.h` to make sure the options specified match the capabilities of your machine and operating system.

2.2 Making a Custom Simlab

This section describes the recommended method for making a customized version of SIMLAB.

When the SIMLAB libraries have been made, a `custom` subdirectory under `$SIMLAB` will also be created. The relevant contents of this subdirectory are:

```
$SIMLAB/custom/Makefile
$SIMLAB/custom/src/Makefile.multiple
$SIMLAB/custom/src/local.c
```

To make a customized version of SIMLAB, you must first set up your work directory (we will refer to it as `$WORK`). To do this, first make the directories `$WORK`, `$WORK/src`, and `$WORK/bin`, and then copy the necessary files from the SIMLAB distribution. You can do this with the following script:

```
mkdir $WORK $WORK/src $WORK/bin
cd $WORK
cp $SIMLAB/custom/Makefile ./Makefile
cp $SIMLAB/custom/src/Makefile.multiple src/Makefile
cp $SIMLAB/custom/src/local.c src/local.c
```

You should then have the following:

```
$WORK/Makefile
$WORK/src/Makefile
$WORK/src/local.c
```

Check the definition of the macro SIMLAB at the top of `$WORK/Makefile` it should be defined as the location of the SIMLAB root directory, i.e., `$SIMLAB`. This definition is automatically generated when SIMLAB is made and should be correct (check to make sure, though). Also check the definition of the macro TARGET at the top of `$WORK/Makefile` it should be defined as the name you would like for your custom version of SIMLAB (the default is "`simtest`").

The new modules you add to simlab must be put into the `$WORK/src`, and `$WORK/src/Makefile` must be modified to reflect the addition of the new modules. This is done simply by adding the source, object, and lint file names to `SRCS`, `OBJS`, and `LNS` macros at the top of `$WORK/src/Makefile`. For example, the top of `$WORK/src/Makefile` will initially look like:

```
# Add the names of sources for $(TARGET) here
SRCS= local.c

# Add the names of objects for $(TARGET) here
OBJS= local.o

# Add the names of .ln's for $(TARGET) here
LNS=  local.ln
```

If you want to add a new module, say, `test`, you would modify the top of `$WORK/src/Make-`

`file` to look like:

```
# Add the names of sources for $(TARGET) here
SRCS= local.c \
      test.c

# Add the names of objects for $(TARGET) here
OBJS= local.o \
      test.o

# Add the names of .ln's for $(TARGET) here
LNS=  local.ln \
      test.ln
```

The dependencies for the files comprising the `SRCS` macro are automatically generated by issuing the command:

```
make depend
```

in the directory `$WORK/src`. The automatically maintained dependency list is contained at the end of `$WORK/src/Makefile`. Whenever a new module is added to `$WORK/src/Makefile`, or whenever the dependencies of a particular module are modified, you should do a "`make depend`".

All that needs to be done to make your own custom version of SIMLAB is to do a make in `$WORK`. The advantage to this sort of setup is that you don't need to change the SIMLAB code itself. Furthermore, the only new compilations needed are for the additional files. There are three different targets that can be made in `$WORK`, a debugging version, an optimized version, and a profiling version, just as for SIMLAB itself. These are made, respectively, with the commands:

```
make simtest
make simtest.opt
make simtest.prof
```

(here, we assume that the name of the target is "`simtest`"). For each version of the target, a separate directory is created (if one does not already exist) in which the compilation is actually done. These directories are named `src-db`, `src-opt`, and `src-prof`. Symbolic links are then created (if they don't already exist) with the `makelinks` utility from the compilation directory to the `src` directory. Compilation is then performed in the appropriate directory. The executables are put into `$WORK/bin`. You can also do the makes in the appropriate compilation directories. This is particularly useful when going through the "code-compile-debug" cycle.

Editing of the source files can take place in any of the compilation directories, since those files are links to the "real" source files in `$WORK/src`. However, the Makefiles in the compilation directories are automatically generated from `$WORK/src/Makefile`, so `$WORK/src/Makefile`, is the only Makefile that should be edited.

Note that once the compilation directory is constructed, subsequent makes for the corresponding target will take place in that directory without reconstruction of all the links. For this reason, you must re-make the compilation directory whenever you add a new module to `$WORK/src`. The compilation directories can be re-made with the commands:

```
make src-db
make src-opt
make src-prof
```

If for some reason you recompile SIMLAB with a different configuration, you should also be sure to remake `Makefile.custom` in `$SIMLAB` (this will remake `$SIMLAB/custom/Makefile` and `$SIMLAB/custom/src/Makefile`). Do this either with a "`make all`" or "`make Makefile.custom`". It may then be necessary to copy `$SIMLAB/custom/Makefile` to your work directory again, since it may have changed in the recompilation.

Note that if you don't want to have separate directories for different compiled versions of your customized SIMLAB, you can just copy `$SIMLAB/src/Makefile.single` and `$SIMLAB/src/local.c` to `$WORK`. However, we feel that having multiple compilation directories is a powerful feature for program development and we encourage you to build your custom version of SIMLAB in this way.

2.3 Notation

Before jumping into the discussion about SIMLAB, some clarification of the notation used in this guide is necessary.

SIMLAB was written using an object-oriented paradigm, to some extent. In keeping with this paradigm, different data type definitions are generally local to a single module. In order to enable different objects to be passed around, a generic pointer type is used:

```
typedef char *GENERIC_PTR;
```

SIMLAB contains many, many identifiers. In order to identify each one as being globally accessible or as being local to a particular module, the following macros are used:

```
#define LOCAL       static
#define ENTRY       /* */
#define GLOBAL      /* */
```

Functions are either `ENTRY` or `LOCAL`. Variables are either `GLOBAL` or `LOCAL`.

SIMLAB is written to be ANSI-C compatible. Unfortunately, many existing compilers are not ANSI-C compatible. In order to accommodate both types of compilers, the following macro is defined:

```
#ifdef ANSI_C
#define ARGS(x) x
#else
#define ARGS(x) ()
#endif
```

This allows us to use the powerful function prototyping feature of ANSI-C (when using an ANSI-C compiler) while still maintaining backwards compatibility with older compilers.

For example, a function declaration will look like:

```
ENTRY int main ARGS((int argc, char *argv[]));
```

The corresponding definition might be:

```
ENTRY int
main(argc, argv)
  int argc;
  char *argv[];
{
  (void) printf("Hello, World\n");
  return 0;
}
```

All of the above definitions can be found in `cenvdefs.h`, which is included by `simlab.h`.

Naming of identifiers more or less follows this convention:

- Macros are in all-caps, i.e. `ENTRY`;
- Data types named with a `typedef` are in all-caps, e.g., `RESISTOR`;
- Elements of an enumerated data type have their first letter capitalized, e.g., `Unknown`;
- Global variables have their first letter capitalized, e.g., `Cmin`.
- The words in multi-word identifiers are generally separated by underscores, e.g., `load_main_circuit()`.
- Other identifiers are all lower case.

2.4 Hooks for Customization

The SIMLAB program can be compiled and used "as-is" and will still be a useful circuit simulation tool. However, much of SIMLAB's value comes from the fact that it can be easily customized. New numerical functions, new devices, and more can be added to SIMLAB by means of a simple functional interface. None of the existing SIMLAB code needs to be modified. Rather, SIMLAB assumes the existence of two functions: `init_local_devices()` and `local_init`. It is from within these two functions that new capabilities are added to SIMLAB. New devices are added from within `init_local_devices()` (see Section 5), everything else is added from within `local_init()` (see Sections 4, 6, and 7). The functions `init_local_devices()` and `local_init` are prototyped as:

```
ENTRY void
init_local_devices(void);

ENTRY void
local_init(void);
```

To create a custom version of SIMLAB, one needs to:

- Set up a working directory for the custom version of SIMLAB;
- Have a module — `local.c` — which contains `init_local_devices()` and `local_init()`;

- Have modules for the functions to be added to SIMLAB;
- Have the new modules properly declared in the Makefile within this working directory.

When "make" is invoked (see 2.2), the local modules will be compiled and linked with the SIMLAB libraries. Note that all you need are your new files and `local.c` — you don't even need `main()`.

The SIMLAB libraries contain empty versions of `init_local_devices()` and `local_init()` since the calls to these functions are part of the main program body. However, the explicit linking of local.o will keep the linker from extracting the empty definitions from the library. (This feature of the linker also allows developers to modify entire SIMLAB modules in their work directories without disturbing the original SIMLAB source. The local module will suppress extraction of the original module from the library. This practice is not necessary, nor is it recommended.)

3 Overview of Simlab

The SIMLAB program can be broken down into four separate sections:

Simulation Engine: The simulation engine is the collection of numerical algorithms (and their interconnections) which SIMLAB uses for circuit simulation. In this manual, we will describe the simulation engine in some detail, since, in some sense, it *is* SIMLAB. We will also describe how you can add new numerical functions to the simulation engine (see Section 4).

Device Library: The device library is just a collection of separate modules, one module per device type. Each module contains all the code necessary to describe the device type and to interface the device type to SIMLAB. In this manual, we will describe how to a new device type to SIMLAB (see Section 5).

User Front-End: The user front-end is accessed through a functional interface. Since the SIMLAB program is intended to be a circuit simulation research tool, the internal workings of the user front-end are a peripheral issue and will not be discussed in detail. However, the user interface does add quite alot of versatility and usability to SIMLAB, so we will describe how to interact with the user front-end through its functional interface (see Section 6 and Section 7).

Circuit Parser: The circuit parser comes to SIMLAB from another circuit simulation program, courtesy of Peter Moore. As with the user front-end, the internal workings are not very important. Also as with the user front-end, the circuit parser is accessed via a functional interface. In this case, however, we have put another level of code on top of that (in the module `ip.c`) and have established our own functional interface for interacting with the input circuit parser. These issues are important when adding a new device and are discussed in Section 5.

In the rest of this section, we will discuss how the pieces of SIMLAB all fit together.

3.1 The Simulation Process

A nodal analysis formulation of Kirchoff's current law (KCL) at each node of a circuit will produce a system of equations given by:

$$f(v) = 0, \tag{1}$$

for the DC case and

$$\frac{d}{dt}q(v(t)) + f(v(t)) = 0, \tag{2}$$

for the transient case. Here, $v(t) \in \mathbb{R}^N$ is a vector of node voltages, $q : \mathbb{R}^N \to \mathbb{R}^N$ is a nonlinear function of v representing the capacitive constitutive relations of the circuit, and $f : \mathbb{R}^N \to \mathbb{R}^N$ is a nonlinear function of v representing the resistive constitutive relations of the circuit.

We usually want to do one of two things when we simulate a circuit:

DC simulation: Solve $f(v) = 0$ for v.

Transient simulation: Solve $\frac{d}{dt}q(v(t)) + f(v(t)) = 0$ for $v(t)$ over some interval of interest.

3.1.1 DC Simulation

To solve (1), we apply Newton's method:

$$J_f(v^k)[v^{k+1} - v^k] = -f(v^k). \tag{3}$$

Here, J_f is the Jacobian of the system, expressed as:

$$J_f(v) = \frac{\partial}{\partial v} f(v). \tag{4}$$

3.1.2 Transient Simulation

To solve (2), we first discretize with, say, the trapezoidal rule to obtain:

$$q\left(v(t+h)\right) - q\left(v(t)\right) + \frac{h}{2}\left[f\left(v(t+h)\right) + f\left(v(t)\right)\right] = 0 \tag{5}$$

which can be re-written to group the terms which depend on t and $(t+h)$ as:

$$\left[\frac{2}{h}q\left(v(t+h)\right) - f\left(v(t+h)\right) +\right] + \left[\frac{2}{h}q\left(v(t)\right) + f\left(v(t)\right)\right] = F\left(v(t+h)\right) = 0 \tag{6}$$

To solve (6) for $v(t+h)$, we apply Newton's method:

$$J_F\left(v^k(t+h)\right)\left[v^{k+1}(t+h) - v^k(t+h)\right] = -F\left(v^k(t+h)\right). \tag{7}$$

Here, J_F is the Jacobian of the system, expressed as:

$$J_F(v) = \frac{2}{h}\frac{\partial}{\partial v}q(v) - \frac{\partial}{\partial v}f(v). \tag{8}$$

Note that the Jacobian is composed of two parts: the *capacitive part* $(\frac{\partial}{\partial v}q(v))$ and the *conductive part* $(\frac{\partial}{\partial v}f(v))$.

3.1.3 Terminology

For the purposes of discussion in the rest of this guide, we will use the following terms:

Charge: The term $q\left(v(t+h)\right)$.

Old Charge: The term $q\left(v(t)\right)$.

Current: The term $f\left(v(t+h)\right)$.

Old Current: The term $f\left(v(t)\right)$.

Right Hand Side (rhs): The right hand side term of the circuit simulation problem is the $-f(v)$ term in (3) for DC simulation and the $-F(v)$ term in (7) for transient simulation. Note that the exact form of this term in the transient case will depend on the particular integration method being used.

Old Right Hand Side (old rhs): The portion of the right hand side term in (6) which depends on $v(t)$ and not $v(t+h)$, i.e., $\left[\frac{2}{h}q\left(v(t)\right) + f\left(v(t)\right)\right]$ for the trapezoidal rule. Note that the exact form of this term will depend on the particular integration method being used.

When we speak of *evaluating the circuit*, we mean calculating the Jacobian matrix and right-hand side vector. We also speak of *evaluating the right-hand side*, in which case we mean calculating the right-hand side vector, but not the Jacobian. By *evaluating the old right-hand side*, we mean calculating the old right-hand side vector.

3.2 The SIMLAB Simulation Engine

The overall structure of SIMLAB is based on the following formulation of the circuit simulation problem:

DC simulation:

1. A numerical method to solve the nonlinear algebraic problem for the DC circuit solution is chosen, e.g. Newton-Raphson, Newton-Raphson with source-stepping.
2. A numerical method to solve the linear system of equations generated at each iteration of the nonlinear DC solution method is then chosen, e.g. sparse Gaussian elimination, relaxation methods.

Transient simulation:

1. A solution type for the circuit is initially chosen, e.g. point-wise time solution, multiple timepoint solution, waveform solution.
2. A numerical integration routine to support the solution type during transient solution is then chosen, e.g. trapezoidal, backward Euler.
3. A numerical method to solve the nonlinear algebraic problem generated at each time step is then chosen, e.g. Newton-Raphson, secant methods.
4. A numerical method to solve the linear system of equations generated at each iteration of the nonlinear solution method is then chosen, e.g. sparse Gaussian elimination, relaxation methods.

In addition, embedded within the above algorithms is the need for routines to evaluate the circuit being simulated. That is, we need routines to form $f(v)$ (in the DC case) or $\frac{d}{dt}q(v(t)) + f(v(t))$ (in the transient case).

Also, in order to start the nonlinear solution method one is usually required to extrapolate a guessed solution based on past data. Routines to that effect are included. Finally one needs to check for the convergence of the nonlinear method which requires both convergence of the solution and the residual, i.e., voltage and current.

3.2.1 Representing the Circuit

The initial step of simulating a circuit with SIMLAB is to read in a circuit specification file and construct an internal representation of the circuit. To represent the circuit internally, we need to represent the devices which comprise the circuit and represent their interconnections, i.e., represent the topology of the circuit.

The primary structures used for the internal circuit representation are the `CIRCUIT` structure, the`DEVICE` structure, and the `NODE` structure. Descriptions of these structures are given in the following sections.

The `CIRCUIT` Structure

The main structure for representing the circuit is the `CIRCUIT` structure.

Synopsis:

```
typedef struct circuit_struct CIRCUIT;
struct circuit_struct {
  char *name;
  int flags;
  int numfixed, numfree, numnets, numnodes;
  int numdevs[MAXDEVS];
  DEVICE *devices, *sources, *devlist;
  NODE *nodes, *fixed, *ground, *nodelist;
  SETIC *initial_conditions;
};
```

Description:

The `name` field is a pointer to a null terminated character string containing the name of the circuit file from which this circuit structure was created. The `flags` field is used to hold various bit-field flags about the circuit. The `numfixed` fields contain the number of fixed nodes (i.e., nodes with a voltage source connected). The `numfree` field contains the number of free nodes (i.e., nodes with no voltage source or initial condition). The `numnodes` field contains the number of internal circuit nodes (i.e., nodes with no voltage source but possibly with an initial condition). The `numnets` field contains the total number of nets (i.e., `numnodes` + numnodes). Note, in general, for DC simulation, the number of nodes we are simulating will be given by `numfree`, whereas in transient simulation, the number of nodes we are simulating will be given by `numnodes`. In the absence of any initial conditions, these will be equal. The `numdevs` field is an array of integers (indexed by device type), each entry of which contains the number of elements in the circuit for that particular device. The `devlist` field contains a pointer to an array of all `DEVICE` structures used by the circuit. These are separated into linked lists for voltage sources and non-sources, pointed to by the `sources` and `devices` fields, respectively. The `nodelist` field contains a pointer to an array of all `NODE` structures used by the circuit. These are separated into linked lists for internal circuit nodes, fixed circuit nodes (i.e., connected to a voltage source), and ground, pointed to by the `nodes`, `fixed`, and `ground` fields, respectively. The `initial_conditions` field contains a pointer to a list of `SETIC` structures, which contain information about any initial conditions specified in the circuit input file.

The `DEVICE` Structure

SIMLAB devices are handled uniformly through the use of the `DEVICE` structure. A SIMLAB `CIRCUIT` structure will point to a list of `DEVICE` structures, each of which represents a

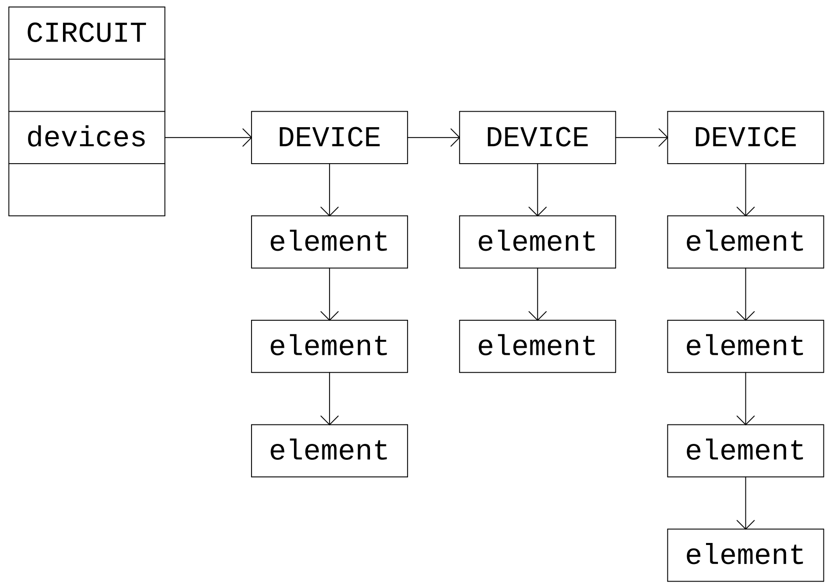

Figure 1: Block diagram showing a simplified example of the `CIRCUIT` and `DEVICE` structures.

different SIMLAB device type and each of which contains a pointer to a list of the circuit elements (of that type) which comprise the circuit. Figure 1 shows a simplified example.

Synopsis:

```
typedef struct device_struct DEVICE;
struct device_struct {
  int type, numdevs;
  GENERIC_PTR devlist;
  GENERIC_PTR modlist;
  int (*add)();
  void (*init)();
  void (*eval)();
  void (*rhseval)();
  void (*display)();
  void (*devfree)();
  void (*modfree)();
  GENERIC_PTR (*devdup)();
  DEVICE *next;
};
```

Description:

Each `DEVICE` structure in the circuit is used to represent a different device type. For each type, SIMLAB assigns a unique identifier, which is kept on the `type` field of the `DEVICE`. The `devlist` field contains a pointer to the head of a list of structures for the actual elements of this device type. (i.e. to a list of resistor structures or a list of capacitor structures). The `numdevs` field contains the number of elements in the list pointed to by `devlist`. The `add` field contains a pointer to the function that adds the devices in the list pointed to by `devlist` to the circuit structure. The `init` field contains a pointer to a function that initializes the devices in the list pointed to by `devlist` for simulation. The `eval` field contains a pointer to a function that evaluates the devices in the list pointed to by `devlist` and calculates their Jacobian and rhs entries. The `rhseval` field contains a pointer to a function that evaluates the devices in the list pointed to by `devlist` and calculates their rhs entries *only*. The `display` field contains a pointer to a function that displays all the devices in the list pointed to by `devlist`. The `devfree` field contains a pointer to a function that frees the structures in the list pointed to by `devlist`. The `devdup` field contains a pointer to a function that duplicates the structures in the list pointed to by `devlist` and returns a pointer to the duplicated list. The `next` field contains a pointer to the next `DEVICE` structure.

The `NODE` Structure

Synopsis:

```
typedef struct node_struct NODE;
struct node_struct {
  NODETYPE type;
  int flags;
  int num, renum, old;
  NODE *next;
};
```

Description:

The `type` field is an enumerated type (`NODETYPE`) indicating what type of node this structure represents. The possible types are `Unknown`, `Ground`, `Source`, `Internal`, and `NotFree`, with the obvious meanings. The `flags` field is used to hold various bit-field flags about the node. The `old` field contains the node number for this node assigned by the circuit parser. The `num` field contains the number for this node as originally assigned by SIMLAB. The `renum` field contains the present number for this node. In general, this will probably be the same as `num`; The `renum` field is included as a separate field in case the nodes are re-ordered after the SIMLAB's initial ordering. The next field is a pointer to another NODE.

3.2.2 Representing the Solution

Once the circuit description file is read in and the internal CIRCUIT structure is constructed, the circuit can be simulated. To do that, we must have an internal representation of the solution. The representation of the circuit is kept independent of the representation of the circuit since the circuit-related structures only have to do with the circuit devices and the circuit topology. The solution structures, on the other hand, will depend on what algorithms are chosen for simulating the circuit. Since we want to keep a high level of flexibility in the types of algorithms that can be used within SIMLAB, the solution is represented with a separate structure from the circuit.

The default data structure for representing the solution within SIMLAB is the SOLUTION structure. If the programmer wishes to replace the highest level simulation algorithms within SIMLAB, he may wish to use a different type of structure (see Section 4.2). However, if the programmer is interested in developing algorithms within the high-level simulation environment already within SIMLAB, he or she will need to use the SOLUTION structure.

Synopsis:

```
typedef struct solution_struct SOLUTION;
struct solution_struct {
  int solution_length;
  double *stepsize_history;
  double *v, **vhist, *vmax, *guess;
  double *current, **old_current, *abs_current, **abs_old_current;
  double *charge, **old_charge, *abs_charge, **abs_old_charge;
  double *rhs, *old_rhs, *abs_rhs, *abs_old_rhs;
  int num_xvects;
  double **xvect;
  GENERIC_PTR sm;
};
```

Description:

The SOLUTION structure holds pointers to the various vectors needed to perform both DC and transient simulations.

The solution_length field contains the length of the solution vectors. In general, this will be the same as the numnodes field in the CIRCUIT structure corresponding to this solution. The stepsize_history field contains a pointer to an array of numbers which are the values of the integration step sizes for the last few steps. The number of past steps will depend on the integration method. The values are indexed so that the most recent time step is in position 0, the next most recent in position 1, etc. This is the same indexing used for the vhist, old_current, and old_charge vector arrays.

The v field points to a double array which contains the values of the node voltages (for both internal and fixed nodes) for the circuit. The vmax field points to a double array which contains the maximum of the absolute value of all previous voltages for each node (for both internal and fixed). The guess field is used to hold the initial voltage guess for Newton's method. The vhist field points to an array of double pointers, each of which points to a vector containing past voltage information for each node of the circuit — vhist[0] contains the most recent past voltage vector, vhist[1] contains the next most recent, etc.

The current field points to a double array which contains the values of the sums of currents flowing into each internal circuit node, i.e., the value of $-f(v)$ at each internal circuit node (in both the DC and transient case). The old_current field points to an array of double pointers, each of which points to a vector containing past current information for each internal circuit node. The indexing is the same as for the vhist field. The abs_current field points to a double array which contains the sums of the absolute values of the currents flowing into each internal circuit node. The abs_old_current field point to an array of double pointers, each of which points to a vector containing past absolute current vectors.

The charge field points to a double array which contains the values of the sums of charges flowing into each internal circuit node, i.e., the value of $-f(v)$ at each internal circuit node (in both the DC and transient case). The old_charge field points to an array of double pointers, each of which points to a vector containing past charge information for each internal circuit node. The indexing is the same as for the vhist field. The abs_charge field points to a double array which contains the sums of the absolute values of the charges flowing into each internal circuit node. The abs_old_charge field point to an array of double pointers, each of which points to a vector containing past absolute charge vectors.

The rhs field points to a double array which contains the right-hand side values for the internal circuit node (see the definition of right-hand side in Section 3.1.3). The old_rhs field points to a double array which contains the old right-hand side values for the internal circuit node (see the definition of old right-hand side in Section 3.1.3). The abs_rhs and abs_old_rhs fields point to double arrays which contain the right-hand side and old right-hand side values formed with the abs_charge and abs_current entries instead of the charge and current entries. Note that the "abs" quantities are not the absolute value of the corresponding quantities, but are rather the sum of the absolute values of the components of the corresponding quantities.

The `xvect` field contains a pointer to an array of `double` pointers, each of which points to a scratch vector of length `solution_length`. The number of `xvects` is given by the `num_xvects` field.

The `sm` field points to the system Jacobian matrix.

In general, the vectors corresponding to voltage quantities have an entry for every node in the circuit, both internal and fixed (i.e., the length is specified by the `numnets` field of the `CIRCUIT` corresponding to this `SOLUTION`). The vectors corresponding to charge and current quantities have an entry only for the internal nodes in the circuit (i.e., the length is specified by the `numnodes` field of the `CIRCUIT` corresponding to this `SOLUTION`). The exception to this is the `guess` field, which contains voltage information — its length is `numnodes`.

3.2.3 Representing the Jacobian

SIMLAB is intended to be a testbed for new numerical algorithms; as such it makes few demands about the structures that need to be used for any of the stages of simulation. However, SIMLAB does include a good sparse matrix package, and we have found that the sparse matrix structures within that package are quite useful for other linear system solvers. See Section 9.5 for more information about SIMLAB's sparse matrix package.

SIMLAB's default sparse matrix structure is the `SMMATRIX`. structure. The matrix entries are represented by `SMELEMENT` structures.

The `SMMATRIX` Structure

Synopsis:

```
struct smmatrix_struct {
  int flags;
  int size, nextpivot;
  int *markprod, *numinrow, *numincol, numelements;
  SMELEMENT **firstinrow, **firstincol, **ninc, **diagonal;
  MEMLINK *alloclist;
  int allocindex, allocmax;
  double *allocptr;
  GENERIC_PTR graph;
};
```

Description:

The `flags` field is used to hold various bit-field flags about the matrix element. See sections 9.5.12 and 9.5.13. The `size` field contains the matrix size. This is usually equal to the number of free nodes in the circuit (`circuit->numfree` in DC simulation and `circuit->numnodes` in transient simulation). The `nextpivot` field contains the index of the next diagonal element to use as pivot during matrix factorization. The `markprod` field contains a pointer to an array of integers whose values are the Markowitz products for each diagonal pivot. The `numinrow` field contains a pointer to an array of integers whose values are the number of elements in the respective row. The `numincol` field contains a pointer to an array of integers whose values are the number of elements in the respective column. The `numelements` field contains the total number of matrix entries. The `firstinrow` field points to an array of matrix element pointers, each of which points to the first matrix element in the same row as the present element. The `ninc` field points to an array of matrix element pointers, each of which points to the next matrix element in the same column as the present element. The `diagonal` field points to an array of matrix element pointers, each of which points to diagonal matrix element in the respective row. The `alloclist` field contains a pointer to a list of allocated matrix elements. This is used in the memory management routines for the the sparse matrix package. The `allocindex`, `allocmax` and `allocptr` fields are used in the memory management routines for the sparse matrix package. The `graph` field contains a pointer to a graph interpretation of the sparse matrix which is used in some algorithms.

The `SMELEMENT` Structure

Synopsis:

```
struct smelement_struct {
  int row, col, rerow, recol, cost;
  double matval;
  SMELEMENT *ninr;
  SMELEMENT *ninc;
  double initcondpart;
  double initcappart;
  double condpart;
  double cappart;
  double savematval;
  GENERIC_PTR scratch;
  int fillin;
};
```

Description:

The `row` field contains the original row to which the element belongs. The `col` field contains the original column to which the element belongs. The `rerow` field contains the row to which the element belongs after some reordering is performed. The `recol` field contains the row to which the element belongs after some reordering is performed. The `cost` field contains an estimate of the cost The `ninr` field contains a pointer to the next matrix element in the present row. The `ninc` field contains a pointer to the next matrix element in the present column. The `initcondpart` field contains the initial conductance part of the matrix element as given by the linear conductances in the circuit. The `initcappart` field contains the initial capacitive part of the matrix element as given by the linear capacitors in the circuit. The `condpart` field contains the conductance part of the matrix element. The `cappart` field contains the initial capacitance part of the matrix element. The `matval` field is used for the merged values of the entry, i.e., the conductance part plus the scaled capacitance part. The `fillin` field contains a count of the number of fillins generated by the present matrix element

3.2.4 Admininistering the Algorithms

The SIMLAB simulation engine is essentially a framework for supporting the circuit simulation paradigm described earlier. The model behind the framework is that:

1. The program has a collection of algorithms for each step of the simulation process.
2. The user should be able to interactively specify which algorithm to use at a particular step of the simulation process.

3. The programmer should be able to easily add new algorithms to any simulation step and should in addition be able to add new simulation steps.

In the actual simulation code then, when a routine representing a particular simulation step is invoked, we don't want to hardwire in a particular function call. Rather, we want to use some data structure which holds information about which routine should be called (e.g., a function pointer). SIMLAB uses structures called (for lack of a better name) *simulation closures* for this task. Each separate simulation step has a corresponding simulation closure.

The structure used to represent a simulation closure is called (not surprisingly) a `CLOSURE` and is defined as:

```
typedef struct closure_struct {
  int (*fcn)();
  GENERIC_PTR data;
} CLOSURE, *CLOSURE_PTR;
```

The `fcn` field is a pointer to a function returning `int`. The `data` field is a `GENERIC_PTR` used to keep data upon which the `fcn` will operate. For SIMLAB, this is usually the current simulation mode. See Section 4 for more detailed information about how these `CLOSURE` structures are used.

In addition to the simulation closures, SIMLAB maintains some notion about how the various algorithms depend on each other. SIMLAB maintains internal tables for this and provides a functional interface which allows the programmer to add a new algorithm for a particular simulation step or to add a new closure to represent a new simulation step.

4 Adding a Numerical Function

SIMLAB contains a functional interface that allows the programmer to interpose new functions for performing the solution, integration, nonlinear system solution, and linear system solution. Furthermore, the framework is flexible enough so that one can even add new points of interposition (e.g., add a preconditioner for certain iterative linear system solvers).

For example, from within the nonlinear solution function, the following call might be seen:

```
SOLVE_LINEAR(circuit, solution)
```

The "function" `SOLVE_LINEAR` is not a function at all, but a macro, defined as:

```
#define SOLVE_LINEAR(c,s) U_SOLVE_LINEAR(Execute,(c),(s))
```

Here, the `SOLVE_LINEAR` macro just passes its arguments to `U_SOLVE_LINEAR`, another macro, with the `Execute` use prepended as a first argument.

Now, `U_SOLVE_LINEAR` is yet another macro, defined as:

```
#define U_SOLVE_LINEAR(u,c,s) \
  ((* ((int (*) \
    (USETYPE, MODETYPE, CIRCUIT *, SOLUTION *)) \
      _solve_linear->fcn)) \
        ((u), (MODETYPE) _solve_linear->data, (c), (s)))
```

This may look somewhat formidable, but all that is happening is that the function pointed to by `_solve_linear->fcn` is called with an argument list composed of the first argument to `U_SOLVE_LINEAR` (a `USETYPE`), `_solve_linear->data` (generally set to the current simulation mode), and the next two macro arguments (which should be a `CIRCUIT` pointer and a `SOLUTION` pointer).

The end effect of all this is that the function on the `_solve_linear` `CLOSURE` gets called with the `Execute` use, the `_solve_linear` argument, and then the two arguments given to the `SOLVE_LINEAR` macro, i.e., a circuit and a solution. Besides `SOLVE_LINEAR`, the following macros are also provided: `SOLVE_LINEAR_INIT` and `SOLVE_LINEAR_STATS`. These are defined as:

```
#define SOLVE_LINEAR_INIT()  U_SOLVE_LINEAR(InitUse, 0, 0)
#define SOLVE_LINEAR_STATS() U_SOLVE_LINEAR(Stats, 0, 0)
```

These macros essentially just call the `_solve_linear` function with different uses, `InitUse` and `Stats`. Similar macros exist for the other stages of the circuit simulation problem.

Besides `Execute`, `InitUse`, and `Stats`, SIMLAB may also call functions with `OpenUse` and `CloseUse` in response to `set` commands from the user. When a `set` command is issued, the function for that particular `CLOSURE` is called with `CloseUse` to notify the function that it will no longer be an "active" function. The new function (the one selected by the `set` command) is then called with `OpenUse` to notify it that it is now an "active" function. The calling with `OpenUse` and `CloseUse` is handled by SIMLAB's internal machinery, so macros are not provided for calling with these arguments.

It should be obvious from the preceding discussion, that for a given `CLOSURE`, the same function is called for execution, initialization, statistics, etc., but that a different first argument is given. The functions installed for use with the a simulation closure must perform the appropriate dispatching based on its first argument. In general, the dispatching should operate as follows:

`OpenUse`: The function is called with this argument when it is about to become an active function, i.e., when when the function pointer of its corresponding `CLOSURE` is set to this function. This allows function-specific tasks to be done. For instance, environment variables specific to a certain simulation method may be created.

`InitUse`: The function is called with this argument during the initialization of the simulation run. This allows simulation parameters (such as iteration counters) to be reset. Functions which during their own execution will access inferior closures for execution should also call those same inferior closures for initialization.

`Execute`: The function is called with this argument when the actual function task (such as linear system solution) is to be performed.

`CloseUse`: The function is called with this argument when it is about to become an inactive function, i.e., when when the function pointer of its corresponding `CLOSURE` is about to be set to another function. This allows function-specific tasks to be un-done. For instance, environment variables that were created in the `OpenUse` call may be destroyed.

`Stats`: The function is called with this argument to print out any statistics gathered during the simulation run. The statistics reported are those relevant to this particular function, e.g. the number of relaxation iterations in an iterative linear system solver. Functions which during their own execution will access inferior closures for execution should also call those same inferior closures for statistics.

A concrete example should make the dispatching operation clear. The following is a listing of the code for `pcg_smsolve()`, a function which solves a linear system with the preconditioned conjugate gradient method.

pcg_smsolve()

```
LOCAL int maxpcgiters = 32, maxdcpcgiters = -1;

ENTRY BOOL
pcg_smsolve(use, mode, circuit, solution)
  USETYPE use;
  MODETYPE mode;
  CIRCUIT *circuit;
  SOLUTION *solution;
{
  static double dtime, etime;
  BOOL retval = TRUE;

  switch (use) {
  case Execute:
    START_TIMER();
    retval = exec_pcg_smsolve(mode, circuit, solution);
    STOP_TIMER(dtime);
    etime += dtime;
    break;
  case InitUse:
    pcg_iter_count = 0;
    dtime = etime = 0.0;
    PRECONDITION_INIT();
    break;
```

pcg_smsolve() (continued)

```
  case OpenUse:
    if (mode == DCMode)
      install_variable("maxdcpcg", Integer, (GENERIC_PTR) &maxdcpcgiters);
    else if (mode == TransientMode)
      install_variable("maxpcg", Integer, (GENERIC_PTR) &maxpcgiters);
    break;
  case CloseUse:
    if (mode == DCMode)
      uninstall_variable("maxdcpcg");
    else if (mode == TransientMode)
      uninstall_variable("maxpcg");
    break;
  case Stats:
    sl_printf("      Linear Solution Method:  PCG\n");
    sl_printf("              PCG Iterations:  %d\n", pcg_iter_count);
    sl_printf("        Linear Solution Time:  %g\n", etime);
    PRECONDITION_STATS();
    break;
  default:
    bug("unknown usetype in pcg_smsolve");
  }

  return retval;
}
```

In the above function, the actual solution is accomplished by `exec_pcg_smsolve()`, declared as:

```
LOCAL BOOL
exec_pcg_smsolve(MODETYPE mode, CIRCUIT *circuit, SOLUTION *solution);
```

See `pcg.c` for a complete code listing. See Section 7 for information about `install_variable()` and `uninstall_variable()`. See Section 9 for information about `sl_printf()` and `bug()`.

Now we will examine how to add new functions for SIMLAB's existing simulation closures as well as what is expected of these functions (primarily during an `Execute` call).

4.1 environment

Adding your own environment function will be a long and arduous task. You will probably only want to do this when porting SIMLAB to an environment in which you would like to be able to use different computational hardware (e.g., parallel vs. serial) with the same program. The CMVSIM vision simulator is an example of such a program [6, 7]. We refer you to the *CMVSIM Programmer's Guide* [6] for further information.

The environment CLOSURE and macros are defined as:

```
EXTERN CLOSURE_PTR _environment;

#define U_ENVIRONMENT(u) \
  ((* ((int (*) \
    (USETYPE)) \
      _environment->fcn)) \
        ((u)))

#define ENVIRONMENT()       U_ENVIRONMENT(Execute)
#define ENVIRONMENT_INIT()  U_ENVIRONMENT(InitUse)
#define ENVIRONMENT_STATS() U_ENVIRONMENT(Stats)
```

The prototype (and default) environment function is serial_environment(), defined in pfunction.c.

The function is prototyped as:

```
LOCAL void
serial_environment(USETYPE use);
```

The use parameter is passed in by SIMLAB and determine the current use of the function (InitUse, Execute, etc.). If a new environment function is added, it will be called with the same arguments as serial_environment(), and should handle them appropriately.

A new environment function should be added to SIMLAB with the following function (called from within local_init()):

```
ENTRY void
add_environment_fcn(char *environment_name, void (*fcn_ptr)());
```

The `environment_name` argument specifies the name of this function. The `fcn_ptr` argument provides the address of the function.

4.2 simulate

Adding your own `simulate` function is probably almost as difficult as adding your own environment. The following discussion should clarify what is involved to some extent, but the best way to understand exactly what is involved is to carefully examine the code in `pt_solve.c` and `wn_solve.c`.

The `simulate` `CLOSURE` and macros are defined as:

```
EXTERN CLOSURE_PTR _simulate;

#define U_SIMULATE(u,c,t) \
  ((* ((int (*) \
    (USETYPE, MODETYPE, CIRCUIT *, RUNTYPE)) \
      _simulate->fcn)) \
        ((u), (MODETYPE) _simulate->data, (c), (t)))

#define SIMULATE(c,t)    U_SIMULATE(Execute,(c),(t))
#define SIMULATE_INIT()  U_SIMULATE(InitUse, 0, 0)
#define SIMULATE_STATS() U_SIMULATE(Stats, 0, 0)
```

The prototype (and default) `simulate` functions are `dc_pt_solve()` and `tran_pt_solve()`, both defined in `pt_solve.c`. These functions were installed separately for the `_simulate` `CLOSURE`, however one function can be installed for more than one mode on the same `CLOSURE`.

The functions are prototyped as:

```
ENTRY void
dc_pt_solve(USETYPE use, MODETYPE mode, CIRCUIT *circuit, RUNTYPE type);
ENTRY void
tran_pt_solve(USETYPE use, MODETYPE mode, CIRCUIT *circuit, RUNTYPE type);
```

The `use` and `mode` parameters are passed in by SIMLAB, and determine the current use of the function (`InitUse`, `Execute`, etc.) and/or the current mode of simulation (`DCMode` or `TransientMode`). The `circuit` parameter is a pointer to the circuit to be simulated. The `runtype` parameter indicates the type of simulation run to be performed in the transient case (either `Run` or `Continue`). If `usetype` is set to `InitUse`, the functions inferior to this level are also initialized. If `mode` is `DCMode`, we want to initialize the `solve_nonlinear` function, which is done with the call:

```
SOLVE_NONLINEAR_INIT();
```

in `dc_pt_solve()`. Similarly, if `mode` is `TransientMode`, we want to initialize the `integrate` function, which is done with the call:

```
INTEGRATE_INIT();
```

in `tran_pt_solve()`. If a new `simulate` function is added, it will be called with the same arguments as `dc_pt_solve()` or `dc_tran_solve()`, and should handle them appropriately.

A new `simulate` function should be added to SIMLAB with the following function (called from within `local_init()`):

```
ENTRY void
add_simulate_fcn(MODETYPE modetype, char *environment_name,
                 char *simulate_name, void (*fcn_ptr)());
```

The `modetype` argument indicates with which simulation mode this function is installed to work. The `environment_name` argument indicates with which `environment` this function is installed to work. The `simulate_name` argument specifies the name of this function. The `fcn_ptr` argument provides the address of the function.

Note that if a new `simulate` function is installed, other support functions which the new function will accessed via CLOSURE structures must also be installed.

The simulate function is responsible for the creation of the solution structure particular to the solution type selected. After creation of the solution structure, SIMULATE() runs the simulation, using that structure. Note that the default simulate functions create a SOLUTION structure. Other simulate functions might create a totally different type of solution representation which uses a different data structure than SOLUTION. There is no problem with this, however, in order to interface with the other parts of SIMLAB which are used in constructing the solution structure, there are certain CLOSURES which must be installed to work in conjunction with the simulate function: get_v_entry, get_rhs_entry, get_abs_rhs_entry, and get_jacobian_entry. Descriptions of these closures can be found in Sections 4.9–4.12.

Finally, one interesting statistic at this level of the simulation is the elapsed cpu time. Two macros are provided for this, START_TIMER() and STOP_TIMER(). Call the latter as STOP_TIMER(ElapsedTime), where ElapsedTime is a double which is returned with the number of seconds elapsed since the call to START_TIMER.

4.3 integrate

The integrate CLOSURE and macros are defined as:

```
EXTERN CLOSURE_PTR _integrate;

#define U_INTEGRATE(u,c,s,t) \
  ((* ((int (*) \
    (USETYPE, MODETYPE, CIRCUIT *, SOLUTION *, RUNTYPE)) \
      _integrate->fcn)) \
        ((u), (MODETYPE) _integrate->data, (c), (s), (t)))

#define INTEGRATE(c,s,t)  U_INTEGRATE(Execute,(c),(s),(t))
#define INTEGRATE_INIT()  U_INTEGRATE(InitUse, 0, 0, 0)
#define INTEGRATE_STATS() U_INTEGRATE(Stats, 0, 0, 0)
```

The prototype and default integrate function is trap(), defined in trap.c. The function is prototyped as:

```
ENTRY void
trap(USETYPE use, MODETYPE mode, CIRCUIT *circuit,
     SOLUTION *solution, RUNTYPE type);
```

The use and mode parameters are passed in by SIMLAB, and determine the current use of the function (InitUse, Execute, etc.) and/or the current mode of simulation (DCMode or TransientMode). The circuit parameter is a pointer to the circuit to be simulated. The solution parameter is a pointer to the solution structure used for the simulation. The runtype parameter indicates the type of simulation run to be performed (either Run or Continue). If usetype is set to InitUse, the functions inferior to this level are also initialized. In particular, we want to initialize the solve_nonlinear function, which is done with the call:

```
SOLVE_NONLINEAR_INIT();
```

If a new integrate function is added, it will be called with the same arguments as trap(), and should handle them appropriately.

A new integrate function should be added to SIMLAB with the following function (called from within local_init():

```
ENTRY void
add_integrate_fcn(MODETYPE modetype, char *environment_name,
                  char *simulate_name, char *integrate_name,
                  void (*fcn_ptr)());
```

The modetype argument indicates with which simulation mode this function is installed to work. The environment_name argument indicates with which environment this function is installed to work. The simulate_name argument specifies with which simulate this function is installed to work. The integrate_name argument is the name of this function. The fcn_ptr argument provides the address of the function.

Note that if a new integrate function is installed, other support functions which the new function will accessed via CLOSURE structures must also be installed.

4.4 solve_nonlinear

The solve_nonlinear CLOSURE and macros are defined as:

```
EXTERN CLOSURE_PTR _solve_nonlinear;

#define U_SOLVE_NONLINEAR(u,c,s,a,h) \
  ((* ((int (*) \
    (USETYPE, MODETYPE, CIRCUIT *, SOLUTION *, double, double)) \
      _solve_nonlinear->fcn)) \
        ((u), (MODETYPE) _solve_nonlinear->data, (c), (s), (a), (h)))

#define SOLVE_NONLINEAR(c,s,a,h) U_SOLVE_NONLINEAR(Execute,(c),(s),(a),(h))
#define SOLVE_NONLINEAR_INIT()   U_SOLVE_NONLINEAR(InitUse, 0, 0, 0, 0)
#define SOLVE_NONLINEAR_STATS()  U_SOLVE_NONLINEAR(Stats, 0, 0, 0, 0)
```

The prototype and default solve_nonlinear function is newton(), defined in newton.c. The function is prototyped as:

```
ENTRY BOOL
pt_newton(USETYPE use, MODETYPE mode, CIRCUIT *circuit,
          SOLUTION *solution, double alpha, double h);
ENTRY BOOL
dc_pt_newton(USETYPE use, MODETYPE mode, CIRCUIT *circuit,
             SOLUTION *solution, double alpha, double h);
```

The use and mode parameters are passed in by SIMLAB, and determine the current use of the function (InitUse, Execute, etc.) and/or the current mode of simulation (DCMode or TransientMode). The circuit parameter is a pointer to the circuit to be simulated. The solution parameter is a pointer to the solution structure used for the simulation. The parameter alpha is a quantity inversely proportional to the current timestep and is used to scale the charge and capacitance quantities when forming the rhs and old rhs. The exact value of alpha will depend on the integration method being used (the integration method will call SOLVE_NONLINEAR()). The parameter h is the value of the current timestep. In the DC simulation case, the parameters alpha and h will be irrelevant and should be ignored. If usetype is set to InitUse, the functions inferior to this level are also initialized. In particular, we want to initialize the solve_nonlinear function, which is done with the call:

```
SOLVE_LINEAR_INIT();
```

If a new solve_nonlinear function is added, it will be called with the same arguments as newton(), and should handle them appropriately.

A new solve_nonlinear function should be added to SIMLAB with the following function (called from within local_init():

```
ENTRY void
add_solve_nonlinear_fcn(MODETYPE modetype, char *environment_name,
                        char *simulate_name, char *solve_nonlinear_name,
                        BOOL (*fcn_ptr)());
```

The modetype argument indicates with which simulation mode this function is installed to work. The environment_name argument indicates with which environment this function is installed to work. The simulate_name argument specifies with which simulate this function is installed to work. The solve_nonlinear_name argument is the name of this function. The fcn_ptr argument provides the address of the function.

Note that if a new solve_nonlinear function is installed, other support functions which the new function will accessed via CLOSURE structures must also be installed.

4.5 solve_linear

The solve_linear CLOSURE and macros are defined as:

```
EXTERN CLOSURE_PTR _solve_linear;

#define U_SOLVE_LINEAR(u,c,s) \
  ((* ((int (*) \
    (USETYPE, MODETYPE, CIRCUIT *, SOLUTION *)) \
      _solve_linear->fcn)) \
        ((u), (MODETYPE) _solve_linear->data, (c), (s)))

#define SOLVE_LINEAR(c,s)    U_SOLVE_LINEAR(Execute,(c),(s))
#define SOLVE_LINEAR_INIT()  U_SOLVE_LINEAR(InitUse, 0, 0)
#define SOLVE_LINEAR_STATS() U_SOLVE_LINEAR(Stats, 0, 0)
```

The prototype and default `solve_linear` function is `direct_smsolve()`, defined in `direct.c`. The function is prototyped as:

```
ENTRY BOOL
direct_smsolve(USETYPE use, MODETYPE mode, CIRCUIT *circuit,
               SOLUTION *solution);
```

The `use` and `mode` parameters are passed in by SIMLAB, and determine the current use of the function (`InitUse`, `Execute`, etc.) and/or the current mode of simulation (`DCMode` or `TransientMode`). The `circuit` parameter is a pointer to the circuit to be simulated. The `solution` parameter is a pointer to the `solution` structure used for the simulation. If a new `solve_linear` function is added, it will be called with the same arguments as `direct_smsolve()`, and should handle them appropriately.

A new `solve_linear` function should be added to SIMLAB with the following function (called from within `local_init()`:

```
ENTRY void
add_solve_linear_fcn(MODETYPE modetype, char *environment_name,
                     char *simulate_name, char *solve_nonlinear_name,
                     char *solve_linear_name, BOOL (*fcn_ptr)());
```

The `modetype` argument indicates with which simulation mode this function is installed to work. The `environment_name` argument indicates with which `environment` this function is installed to work. The `simulate_name` argument specifies with which `simulate` this function is installed to work. The solve_nonlinear_name argument specifies with which solve_nonlinear this function is installed to work. The solve_linear_name argument is the name of this function. The fcn_ptr argument provides the address of the function.

Note that if a new solve_linear function is installed, other support functions which the new function will accessed via CLOSURE structures must also be installed.

4.6 evaluate

The evaluate CLOSURE and macros are defined as:

```
EXTERN CLOSURE_PTR _evaluate;

#define U_EVALUATE(c,s,a) \
  ((* ((int (*) \
    (MODETYPE, CIRCUIT *, SOLUTION *, double)) \
      _evaluate->fcn)) \
        ((MODETYPE) _evaluate->data, (c), (s), (a)))

#define EVALUATE(c,s,a) U_EVALUATE((c),(s),(a))
```

The prototype (and default) evaluate functions are direct_evaluate() and direct_-dc_evaluate(), both defined in direct.c. In general, one will need different evaluate functions for DC and for transient simulations. For DC simulation, the evaluate function is needed to support the solve_nonlinear function. For transient simulation, the evaluate function is needed to support the integrate function.

The functions are prototyped as:

```
ENTRY void
direct_evaluate(MODETYPE mode, CIRCUIT *circuit,
                SOLUTION *solution, double alpha);
ENTRY void
direct_dc_evaluate(MODETYPE mode, CIRCUIT *circuit,
                   SOLUTION *solution, double alpha));
```

The mode parameter is passed in by SIMLAB, and specifies the current mode of simulation (DCMode or TransientMode). The circuit parameter is a pointer to the circuit to be simulated. The solution parameter is a pointer to the solution structure used for the simulation. The alpha parameter is a quantity that specifies by how much the charge and capacitance portions of the rhs and Jacobian are to be scaled in the transient simulation case. This quantity has no meaning when modetype is set to DCMode. If a new evaluate function is added, it will be called with the same arguments as direct_evaluate() or direct_dc_evaluate(), and should handle them appropriately.

A new evaluate function should be added to SIMLAB with the following function (called from within local_init()):

```
ENTRY void
add_evaluate_fcn(MODETYPE modetype, char *environment_name,
                 char *simulate_name, char *solve_name, void (*fcn_ptr)());
```

The modetype argument indicates with which simulation mode this function is installed to work. The environment_name argument indicates with which environment this function is installed to work. The simulate_name argument specifies with which simulate this function is installed to work. The solve_name argument specifies with which integrate this function is installed to work in transient mode (i.e., when the modetype argument is TransientMode) or with which solve_nonlinear this function is installed to work in DC mode (i.e., when the modetype argument is DCMode). The fcn_ptr argument provides the address of the function. Note that since evaluate is not user-settable, it does not have a specified name.

4.7 evaluate_rhs

The evaluate_rhs CLOSURE and macros are defined as:

```
EXTERN CLOSURE_PTR _evaluate_rhs;

#define U_EVALUATE_RHS(c,s,a) \
  ((* ((int (*) \
    (MODETYPE, CIRCUIT *, SOLUTION *, double)) \
      _evaluate_rhs->fcn)) \
        ((MODETYPE) _evaluate_rhs->data, (c), (s), (a)))

#define EVALUATE_RHS(c,s,a) U_EVALUATE_RHS((c),(s),(a))
```

The prototype (and default) evaluate_rhs functions are direct_evaluate_rhs() and direct_dc_evaluate_rhs(), both defined in direct.c. In general, one will need different evaluate_rhs functions for DC and for transient simulations. For DC simulation, the evaluate_rhs function is needed to support the solve_nonlinear function. For transient simulation, the evaluate_rhs function is needed to support the integrate function.

The functions are prototyped as:

```
ENTRY void
direct_evaluate_rhs(MODETYPE mode, CIRCUIT *circuit,
                    SOLUTION *solution, double alpha);
ENTRY void
direct_dc_evaluate_rhs(MODETYPE mode, CIRCUIT *circuit,
                       SOLUTION *solution, double alpha));
```

The mode parameter is passed in by SIMLAB, and specifies the current mode of simulation (DCMode or TransientMode). The circuit parameter is a pointer to the circuit to be simulated. The solution parameter is a pointer to the solution structure used for the simulation. The alpha parameter is a quantity that specifies by how much the charge and capacitance portions of the rhs are to be scaled in the transient simulation case. This quantity has no meaning when modetype is set to DCMode. If a new evaluate_rhs function is added, it will be called with the same arguments as direct_evaluate_rhs() or direct_dc_evaluate_rhs(), and should handle them appropriately.

A new evaluate_rhs function should be added to SIMLAB with the following function (called from within local_init()):

```
ENTRY void
add_evaluate_rhs_fcn(MODETYPE modetype, char *environment_name,
                     char *simulate_name, char *solve_name,
                     void (*fcn_ptr)());
```

The modetype argument indicates with which simulation mode this function is installed to work. The environment_name argument indicates with which environment this function is installed to work. The simulate_name argument specifies with which simulate this function is installed to work. The solve_name argument specifies with which integrate this function is installed to work in transient mode (i.e., when the modetype argument is TransientMode) or with which solve_nonlinear this function is installed to work in DC mode (i.e., when the modetype argument is DCMode). The fcn_ptr argument provides the address of the function. Note that since evaluate_rhs is not user-settable, it does not have a specified name.

4.8 evaluate_old_rhs

The evaluate_old_rhs CLOSURE and macros are defined as:

```
EXTERN CLOSURE_PTR _evaluate_old_rhs;

#define U_EVALUATE_OLD_RHS(c,s,a) \
  ((* ((int (*) \
    (MODETYPE, CIRCUIT *, SOLUTION *, double)) \
      _evaluate_old_rhs->fcn)) \
        ((MODETYPE) _evaluate_old_rhs->data, (c), (s), (a)))

#define EVALUATE_OLD_RHS(c,s,a) U_EVALUATE_OLD_RHS((c),(s),(a))
```

The prototype (and default) evaluate_old_rhs function is trap_direct_evaluate_rhs() defined in trap.c. The exact form of the evaluate_old_rhs function will depend very heavily on the particular integration method being used. In general, one will need a separate evaluate_old_rhs to go along with each integrate.

The functions are prototyped as:

```
ENTRY void
trap_direct_evaluate_old_rhs(MODETYPE mode, CIRCUIT *circuit,
                             SOLUTION *solution, double alpha);
```

The mode parameter is passed in by SIMLAB, and specifies the current mode of simulation (DCMode or TransientMode). The circuit parameter is a pointer to the circuit to be simulated. The solution parameter is a pointer to the solution structure used for the simulation. The alpha parameter is a quantity that specifies by how much the charge and capacitance portions of the rhs are to be scaled in the transient simulation case. This quantity has no meaning when modetype is set to DCMode. If a new evaluate_old_rhs function is added, it will be called with the same arguments as trap_direct_evaluate_rhs() and should handle them appropriately.

A new evaluate_old_rhs function should be added to SIMLAB with the following function (called from within local_init()):

```
ENTRY void
add_evaluate_old_rhs_fcn(MODETYPE modetype, char *environment_name,
                         char *simulate_name, char *integrate_name,
                         void (*fcn_ptr)());
```

The modetype argument indicates with which simulation mode this function is installed to work. The environment_name argument indicates with which environment this function is installed to work. The simulate_name argument specifies with which simulate this function is installed to work. The integrate_name argument specifies with which integrate this function is installed to work. Note that since the old rhs only applies to transient mode, it is meaningless to call this function with modetype equal to DCMode. Note that since evaluate_old_rhs is not user-settable, it does not have a specified name.

4.9 get_v_entry

The get_v_entry CLOSURE and macros are defined as:

```
EXTERN CLOSURE_PTR _get_v_entry;
#define U_GET_V_ENTRY(i) \
  ((* ((double *(*) \
    (NODE *)) \
      _get_v_entry->fcn)) \
        ((i)))

#define GET_V_ENTRY(i) U_GET_V_ENTRY((i))
```

The prototype (and default) get_v_entry function is pt_get_v defined in pt_solve.c. The exact form of the get_v_entry function will depend very heavily on the particular simulation method and simulation data structure.

The functions are prototyped as:

```
ENTRY double *pt_get_v(NODE *i);
```

pt_get_v() returns a pointer to the voltage vector element with row indicated by i->renum. If a new get_v_entry function is added, it will be called with the same arguments as pt_get_v and should handle them appropriately.

A new get_v_entry function should be added to SIMLAB with the following function (called from within local_init()):

```
ENTRY void
add_get_v_entry_fcn(char *environment_name, char *simulate_name,
                    void (*fcn_ptr) ());
```

The environment_name argument indicates with which environment this function is installed to work. The simulate_name argument specifies with which simulate this function is installed to work. The fcn_ptr argument provides the address of the function. Note that since evaluate is not user-settable, it does not have a specified name. Also, since this function is not associated with any particular simulation mode, there is no modetype argument.

4.10 get_rhs_entry

The get_rhs_entry CLOSURE and macros are defined as:

```
EXTERN CLOSURE_PTR _get_rhs_entry;
#define U_GET_RHS_ENTRY(i,t) \
  ((* ((double *(*) \
    (NODE *, ENTRYTYPE)) \
      _get_rhs_entry->fcn)) \
        ((i), (t)))

#define GET_RHS_ENTRY(i,t) U_GET_RHS_ENTRY((i),(t))
```

The prototype (and default) get_rhs_entry function is pt_get_rhs defined in pt_solve.c. The exact form of the get_rhs_entry function will depend very heavily on the particular simulation method and simulation data structure.

The functions are prototyped as:

```
ENTRY double *pt_get_rhs(NODE *i, ENTRYTYPE type);
```

pt_get_rhs_entry() returns a pointer to the rhs vector element with row indicated by i->renum. The argument type can have enumerated value LinCond, LinCap, NonlinCond, or NonlinCap. LinCond and NonlinCond specify a location in the current vector; LinCap and NonlinCap specify a location in the charge vector. If a new get_rhs_entry function is added, it will be called with the same arguments as pt_get_rhs and should handle them appropriately.

A new get_rhs_entry function should be added to SIMLAB with the following function (called from within local_init()):

```
ENTRY void
add_get_rhs_entry_fcn(char *environment_name, char *simulate_name,
                      void (*fcn_ptr) ());
```

The environment_name argument indicates with which environment this function is installed to work. The simulate_name argument specifies with which simulate this function is installed to work. The fcn_ptr argument provides the address of the function. Note that since evaluate is not user-settable, it does not have a specified name. Also, since this function is not associated with any particular simulation mode, there is no modetype argument.

4.11 get_abs_rhs_entry

The get_abs_rhs_entry CLOSURE and macros are defined as:

```
EXTERN CLOSURE_PTR _get_abs_rhs_entry;
#define U_GET_ABS_RHS_ENTRY(i,t) \
  ((* ((double *(*) \
    (NODE *, ENTRYTYPE)) \
      _get_abs_rhs_entry->fcn)) \
        ((i), (t)))

#define GET_ABS_RHS_ENTRY(i,t) U_GET_ABS_RHS_ENTRY((i),(t))
```

The prototype (and default) get_abs_rhs_entry function is pt_get_abs_rhs defined in pt_solve.c. The exact form of the get_abs_rhs_entry function will depend very heavily on the particular simulation method and simulation data structure.

The functions are prototyped as:

```
ENTRY double *pt_get_abs_rhs(NODE *i, ENTRYTYPE type);
```

pt_get_abs_rhs_entry() returns a pointer to the abs rhs vector element with row indicated by i->renum. The argument type can have enumerated value LinCond, LinCap, NonlinCond, or NonlinCap. LinCond and NonlinCond specify a location in the abs current vector; LinCap and NonlinCap specify a location in the abs charge vector. If a new get_abs_rhs_entry function is added, it will be called with the same arguments as pt_get_abs_rhs and should handle them appropriately.

A new get_abs_rhs_entry function should be added to SIMLAB with the following function (called from within local_init()):

```
ENTRY void
add_get_abs_rhs_entry_fcn(char *environment_name, char *simulate_name,
    void (*fcn_ptr) ());
```

The environment_name argument indicates with which environment this function is installed to work. The simulate_name argument specifies with which simulate this function is installed to work. The fcn_ptr argument provides the address of the function. Note that since evaluate is not user-settable, it does not have a specified name. Also, since this function is not associated with any particular simulation mode, there is no modetype argument.

4.12 get_jacobian_entry

The get_jacobian_entry CLOSURE and macros are defined as:

```
EXTERN CLOSURE_PTR _get_jacobian_entry;
#define U_GET_JACOBIAN_ENTRY(i,j,t) \
  ((* ((double *(*) \
    (NODE *, NODE *, ENTRYTYPE)) \
      _get_jacobian_entry->fcn)) \
        ((i), (j), (t)))

#define GET_JACOBIAN_ENTRY(i,j,t) U_GET_JACOBIAN_ENTRY((i),(j),(t))
```

The prototype (and default) `get_jacobian_entry` function is `pt_get_jacobian` defined in `pt_solve.c`. The exact form of the `get_jacobian_entry` function will depend very heavily on the particular simulation method and simulation data structure.

The functions are prototyped as:

```
ENTRY double *pt_get_Jacobian(NODE *i, NODE *j, ENTRYTYPE type);
```

get_Jacobian() returns a pointer to the Jacobian element with row and column indicated by `i->renum` and `j->renum`, respectively. If `i->type` or `j->type` is not `Internal`, NULL is returned. The argument `type` can have enumerated value `LinCond`, `LinCap`, `NonlinCond`, or `NonlinCap`, specifying a linear conductive, linear capacitive, nonlinear conductive, or nonlinear capacitive entry, respectively. If a new `get_jacobian_entry` function is added, it will be called with the same arguments as `pt_get_jacobian` and should handle them appropriately.

A new `get_jacobian_entry` function should be added to SIMLAB with the following function (called from within `local_init()`):

```
ENTRY void
add_get_jacobian_entry_fcn(char *environment_name, char *simulate_name,
                           void (*fcn_ptr) ());
```

The `environment_name` argument indicates with which `environment` this function is installed to work. The `simulate_name` argument specifies with which `simulate` this function is installed to work. The `fcn_ptr` argument provides the address of the function. Note that since `evaluate` is not user-settable, it does not have a specified name. Also, since this function is not associated with any particular simulation mode, there is no `modetype` argument.

4.13 Adding a New CLOSURE

Since SIMLAB was written to allow arbitrary simulation methods to be added, a functional interface is provided so that the programmer can add new CLOSURE structures to the set of CLOSURE structures already included with SIMLAB. One might find it necessary to do this if a new algorithm that is added has an inferior function that the programmer would like to be settable from the user front-end. An example of this is in the iterative linear solution package that is included with the SIMLAB distribution. Many of the methods in this package are based on the conjugate gradient algorithm and may therefore be able to use a preconditioner. In general, it would be nice to be able to choose the particular preconditioner used, so the iterative linear solution package adds a preconditioning CLOSURE to SIMLAB.

These are the steps one needs to take to add a new CLOSURE.

1. Define the actual CLOSURE structure
2. Define the macros to access the structure
3. Define at least one function (the default) to use this closure
4. Define a function to add new functions to the closure's collection
5. Declare the add function
6. Add the desired functions to the database for the new closure
7. Initialize the closure to the default

4.13.1 Example: precondition

To demonstrate what is meant by each of these steps, we will take a walk-through of the steps used to create the precondition CLOSURE. The `precondition` closure is used in the iterative linear solution package that is include as part of the standard SIMLAB distribution (although it is not included in the standard SIMLAB compilation).

Define the CLOSURE

The actual CLOSURE_PTR to be used for the new simulation closure must be defined somewhere. If the use of a given CLOSURE is to be confined to a single module, the declaration can be confined to that module. On the other hand, if CLOSURE will be used in more than one module, it should be appropriately declared in a header file.

The `precondition` closure is defined as:

```
EXTERN CLOSURE_PTR _precondition;
```

Define the Macros

The macros for accessing the `precondition` closure are defined as:

```
#define U_PRECONDITION(_u,_sm,_b,_x) \
  ((* ((int (*) \
    (USETYPE, MODETYPE, SMMATRIX *, double *, double *)) \
      _precondition->fcn)) \
        ((_u), (MODETYPE) _precondition->data, (_sm), (_b), (_x)))
#define PRECONDITION(_sm, _b, _x) U_PRECONDITION(Execute, (_sm), (_b), (_x))
#define PRECONDITION_INIT()       U_PRECONDITION(InitUse, 0, 0, 0)
#define PRECONDITION_STATS()      U_PRECONDITION(Stats, 0, 0, 0)
```

Define the Default

The default `precondition` function is `ident_precondition()`, i.e., the identity preconditioner. This function is defined as:

```
ENTRY void
ident_precondition(use, mode, sm, b, x)
  USETYPE use;
  MODETYPE mode;
  SMMATRIX *sm;
  double *b;
  double *x;
{
  register int i, N;

  switch (use) {
  case Execute:
    N = sm->size;
    for (i = 1; i <= N; i++)
      x[i] = b[i];
    break;
  case InitUse:
    break;
  case OpenUse:
    break;
  case CloseUse:
    break;
  case Stats:
    sl_printf("             Preconditioner:  none\n");
    break;
  default:
    bug("unknown usetype in ident_precondition");
  }
}
```

Define the "Add" Function

It is often useful to have a separate function which will add new functions to the collection for a particular CLOSURE. Functions of this type are essentially "wrappers" for add_fcn() (see Section 9.6.6).

```
ENTRY void
add_precondition_fcn(modetype, environment_name,
                     simulate_name,
                     solve_nonlinear_name,
                     solve_linear_name,
                     precondition_name,
                     fcn_ptr)
  MODETYPE modetype;
  char *environment_name;
  char *simulate_name;
  char *solve_nonlinear_name;
  char *solve_linear_name;
  char *precondition_name;
  void (*fcn_ptr) ();
{
  add_fcn(modetype, TRUE, fcn_ptr, precondition_name,
          "environment", &_environment, environment_name,
          "simulate", &_simulate, simulate_name,
          "solve_nonlinear", &_solve_nonlinear, solve_nonlinear_name,
          "solve_linear", &_solve_linear, solve_linear_name,
          "precondition", &_precondition, NULL);
}
```

Here, `precondition` is made to be inferior to `environment`, `simulate`, `solve_nonlinear` and `solve_linear`.

Declare the "Add" Function

Next, `add_precondition_fcn()` must be declared so that it can be called from within `local_init()`, i.e.,

```
ENTRY void
add_precondition_fcn(MODETYPE modetype,
                     char *environment_name,
                     char *simulate_name,
                     char *solve_nonlinear_name,
                     char *solve_linear_name,
                     char *precondition_name,
                     void (*fcn_ptr)());
```

One can either just make the declaration from within `local_init()`, or make the declaration in a header file that is supporting the corresponding closure (this header file should be included from within `local.c`). For example, `add_precondition_fcn()` is prototyped in `relax.h`.

Add the Functions

Once the adding function is created, the functions should be added using this function, from within `local_init()`. This example shows `ident_precondition()` being added for DC mode.

```
ENTRY void
ident_precondition(USETYPE use, MODETYPE mode, SMMATRIX *sm,
                   double *b, double *x);

add_precondition_fcn(DCMode, "serial", "pt", "newton", "pcg",
                     "none", ident_precondition);
```

Note that the name used to specify `ident_precondition()` is `none`. Note also that the function to be installed must be declared at some point before the call for installation. Again, this can be done in `local.c` or in an appropriate header file.

Initialize the Closure

The last, and most tricky, step is to initialize the new closure. Whenever new closures are inserted into SIMLAB's database, they are in some sense being inserted into a "live" database. In order to initialize the closures, we have to mimic the behavior of the `set` commands from the user. To do this without causing errors, we have to initialize according to the closures' dependencies. This is tricky when only some functions belonging to a certain closure have another inferior closure.

To illustrate this point, let's examine the code for initializing SIMLAB's standard closures. The function for doing this is `set_default_fcn()`, which makes calls to the function `do_ew_set()` (see Section 9.6.7):

set_default_fcn()

```
LOCAL void
set_default_fcn()
{
  (void) do_ew_set("environment", NULL, "serial", TRUE);
  (void) do_ew_set("simulate", NULL, "pt", TRUE);
  (void) do_ew_set("integrate", NULL, "trap", TRUE);
  (void) do_ew_set("solve_nonlinear", NULL, "newton", TRUE);
  (void) do_ew_set("solve_linear", NULL, "direct", TRUE);
}
```

This function sets up the functions installed with names "`serial`", "`pt`", "`trap`", "`newton`", and "`direct`" as the `environment`, `simulate`, `integrate`, `solve_nonlinear`, and `solve_linear` functions, respectively.

Now, to set the default precondition function, we can't just make a call to `add_precondition_fcn()` in `local_init()`, because the function `set_default_fcn()` is called prior to `local_init()`, and the function installed with `direct` as the linear solver does not use a preconditioner. So, to initialize the `precondition` closure, we have to first set `solve_linear` to a function which uses a preconditioner, initialize the `precondition` closure, and then set `solve_linear` back to its desired default, if the previously set `solve_linear` is not the desired default.

For example, this is a code fragment showing how we would initialize `precondition` and retain `direct` as the `solve_linear` default:

```
  (void) do_ew_set("solve_linear", NULL, "pcg", TRUE);
  (void) do_ew_set("precondition", NULL, "none", TRUE);
  (void) do_ew_set("solve_linear", NULL, "direct", TRUE);
```

As an aside, note that the default functions for all the closures can be set from within `local.c` if the normal SIMLAB ones are not suitable for a particular application.

5 Adding a New Device Type

SIMLAB was written without any preconceived ideas about what device types would be supported. This was done in order to allow programmers to add an arbitrary number of arbitrary new device types to their own custom versions of SIMLAB. A functional interface (consisting of a single function call) is provided to allow programmers to add new devices to SIMLAB.

The basic process for adding a new device type is this:

1. Define a data structure to represent the element and a data structure to represent the element model (see Section 5.1);
2. Fill in appropriate static data structures for interacting with SIMLAB's circuit parser (see Section 5.2);
3. Create support functions for the element (see Section 5.3);
4. Call the function `define_device()` in `init_local_devices()` in the module `local.c` (see Section 5.4).

The convention we used in writing SIMLAB is to have one header file and one program file for each device type (i.e., a "`.h`" file and a "`.c`" file). The header file contains the definitions of the structures for the device and the device model. The header file also contains `extern` definitions for data or code in the program file which may need to be externally referenced (this is typically only one data structure — the `DEVICE_ENTRY` structure defined below in Section 5.2).

This section describes how to add new device types to SIMLAB. The procedure is essentially the same for all device types; the process can be simplified by using the code for an existing device type as a template — `res.h` and `res.c` or `diode.h` and `diode.c` are recommended.

In this section, we will be using a simplified diode as an example device. The complete code for this device can be found in Appendix A.

5.1 Data Structures for the New Device Type

SIMLAB elements need two data structures each — one to represent element models and one to represent the circuit elements that actually make up the circuit being simulated. An element model will generally contain information about relevant physical quantities for modeling a particular type of device. The actual element instance structure will generally contain information necessary for representing the incorporation of a particular element into a circuit.

For instance, a diode model structure would need to have information about saturation currents, maximum currents, etc. The diode element structure, on the other hand, would need to have information about the nodes it is connected to, the voltages at its terminals, etc.

5.1.1 The Element Model

The element model structure should have the following properties:

1. The structure should have fields to hold relevant quantities for numerically simulating the device being modeled.
2. The structure must have a pointer to the next model in the list (the last model's pointer must be `NULL`).

The following is an example for a simplified diode model:

```
typedef struct xdmod_struct XDMOD;

struct xdmod_struct {
  double area, is, cp, vmax;
  XDMOD *next;
};
```

5.1.2 The Element Instance

The element instance structure should have the following properties:

1. The structure should contain fields for those parameters which depend on the the particular element instance.
2. The structure should have a pointer to a device model structure which keeps the parameter information (the same model might be used for multiple element instances). Typically, one uses a model to hold information which will be the same for many of the devices in a circuit. The element instance structure would then have fields for information which is element-specific, such as area.
3. The structure should have `int` fields to record the node numbers for the element terminals assigned by the parser.
4. The structure should have pointers to the `NODE` structures corresponding to its terminal nodes.
5. The structure should have `double*` fields that point into the circuit voltage state vector.
6. The structure should have `double*` fields that point into the circuit Jacobian. Separate pointers should be maintained for capacitive and conductive contributions.
7. The structure should have `double*` fields that point into the circuit node charge and current vectors.
8. For each pointer to the circuit charge and current vectors, the structure should have a corresponding pointer into the appropriate absolute value vector.
9. The structure must have a pointer to the next device in the list (the last device's pointer must be `NULL`).

The following is an example for the simplified diode:

```
typedef struct xd_struct XD;

struct xd_struct {
  XDMOD *model;
  int anode, cnode;
  NODE *nodei, *nodej;
  double
    isat,                             /* to limit current       */
    vmax,                             /* to limit current       */
    *va, *vc,                         /* voltage vector pointers */
    *cmataa, *cmatcc, *cmatca, *cmatac, /* Cap mat pointers       */
    *rmataa, *rmatcc, *rmatca, *rmatac, /* Resistive mat ptrs     */
    *acharge, *ccharge,               /* Charge vector entries  */
    *acurrent, *ccurrent,             /* Current vector entries  */
    *abs_acharge, *abs_ccharge,       /* Charge vector entries  */
    *abs_acurrent, *abs_ccurrent      /* Current vector entries  */
  XD *next;
};
```

The `model` field contains a pointer to an example diode model. The `isat` field contains the value of the example diode's saturation current. The `vmax` field contains the value of the example diode's maximum voltage. The node numbers for the element terminals (as assigned by the parser) are kept in the fields `anode` and `cnode`. Pointers to the `NODE` structures for the terminals are kept in fields `nodei` and `nodej`. The voltage vector pointers are kept in fields `va` and `vc`. The Jacobian conductance pointers are kept in fields `rmataa`, `rmatac`, `rmatca`, and `cmatcc`. The Jacobian capacitance pointers are kept in fields `cmataa`, `cmatac`, `cmatca`, and `cmatcc`. The current vector pointers are kept in fields `acurrent`, and `cccurrent`; the absolute value current vector pointers are kept in fields `abs_acurrent`, and `abs_ccurrent`. The pointer to the next `XD` in the list is stored in the `next` field.

5.2 Data Structures to Interact with the Circuit Parser

5.2.1 The DEVICE_ENTRY structure

The primary means of interface for a new device with SIMLAB is through the `DEVICE_ENTRY` structure. The `DEVICE_ENTRY` structure fields contain information which allows the circuit parser to manipulate all device types in a uniform manner.

Synopsis:

```
typedef struct device_entry_struct DEVICE_ENTRY;
struct device_entry_struct {
  int type;
  int numdevs;
  char *name;
  BOOL is_source;
  GENERIC_PTR *parse_list;
  GENERIC_PTR *model_list;
  int (*add)();
  void (*init)();
  void (*eval)();
  void (*rhseval)();
  void (*display)();
  void (*devfree)();
  void (*modfree)();
  GENERIC_PTR (*devdup)();
  void (*devdefine)();
};
```

Description:

The `type` field contains a unique identifier (assigned by SIMLAB) for this device. The `numdevs` field contains the number of devices read in (also assigned by SIMLAB). The `name` field contains a character string of the device name. The `is_source` field indicates whether or not the device is a voltage source. The `parse_list` contains the address of a pointer to the head of the list of devices that are read in. `model_list` contains the address of a pointer to the head of the list of models that are read in. The `add` field contains a pointer a function that adds the devices to the circuit structure. The `init` field contains a pointer to a function that initializes the devices for simulation. The `eval` field contains a pointer to a function that evaluates the devices and calculates the Jacobian and rhs entries. The `rhseval` field contains a pointer to a function that evaluates the devices in the device list and calculates the rhs entries only. The `display` field contains a pointer to a function that displays all the devices in the device list. The `devfree` field contains a pointer to a function that frees the list of devices. The `modfree` field contains a pointer to a function that frees the list of device models. The `devdup` field contains a pointer to a function that duplicates the list of devices. The `devdefine` field contains a pointer to a function that defines the device to the circuit parser.

Example:

For the example diode, the DEVICE_ENTRY structure for the example diode might be filled in as follows:

```
GENERIC_PTR current_xd_list;
GENERIC_PTR current_xdmod_list;

GLOBAL DEVICE_ENTRY xd_entry =
{
  0, 0,
  "XD",                                /* char *name              */
  FALSE,                               /* BOOL is_source;         */
  (GENERIC_PTR *) &current_xd_list,    /* GENERIC_PTR *parse_list;*/
  (GENERIC_PTR *) &current_xdmod_list,/* GENERIC_PTR *model_list;*/
  add_xd,                              /* int (*add)();           */
  init_xd,                             /* void (*init)();         */
  eval_xd,                             /* void (*eval)();         */
  eval_xd_rhs,                         /* void (*rhseval)();      */
  display_xd,                          /* void (*display)();      */
  free_xd_list,                        /* void (*devfree)();      */
  free_xdmod_list,                     /* void (*devfree)();      */
  dup_xd,                              /* GENERIC_PTR (*devdup)();*/
  define_xd                            /* void (*devdefine)();    */
};
```

The functions in the various fields will be described ahead.

5.2.2 The PRIM structure

Synopsis:

```
typedef struct allmod {
  int type;
  GENERIC_PTR model;
  void (*readdev)();
  GENERIC_PTR (*readmod)();
} PRIM;
```

Description:

The only fields that are initialized are the `readdev` and `readmod` fields, which contain pointers to the functions for making the element and for making the element model, respectively. The other two fields are used by the input parser and should be initialized to zero.

Important: For any device that requires a `model` statement in the input circuit file, e.g., semiconductors, the `readdev` field of the local `PRIM` must be set to `NULL`.

Example:

For the example diode, the definition of `xd_prim` is:

```
LOCAL PRIM xd_prim = {
  0,
  NULL,
  NULL,
  make_xd_model
};
```

Note that the `readdev` field is `NULL` here, since the example diode requires that every element have a model.

5.2.3 The `IP_PARAM` structure

The `IP_PARAM` structure is used to hold information about model and device parameters. The circuit parser must have a list of valid parameters for each installed model and device. One usually provides such a list to the circuit parser by creating a static array of IP_PARAM structures, one entry per parameter. The model and device parameter lists are then passed to the circuit parser when the new device type is defined (with a call to `ip_define_primitive()`).

Synopsis:

```
typedef struct ip_param {
  int id;
  char *name;
  struct ip_pvalue *value;
} IP_PARAM;
```

Description:

The name field contains the literal name of the parameter. The id field contains an identifier for the parameter. The identifier is used later in the make_element() and make_element_model() functions and is used to uniquely identify the parameter with one piece of data, e.g., id field. The value field is used by the input parser and should be initialized to zero. Why use an id? When the input parser reads a model definition or device instantiation from the circuit file, it puts together an array of IP_PARAM structures corresponding to the parameters and values read in. It calls the make_element() or make_element_model() function with that array as an argument. One efficient method of decoding the information contained in the array is to sequentially index down it and decode each parameter individually. To decide which parameter has been read in, one could look at the name field, but the decoding process would then involve a convoluted series of string compares. However, since the id field contains information which is unique for each parameter in only a single data unit, we can use a switch statement. For instance, if a particular element has ten parameters, one might use the sequence of integers 1 through 10 as the identifiers. See the explanations for make_element() and make_element_model().

Example:

For the example device, the model and device parameter lists are defined as:

```
LOCAL ip_param xdplist[] =
{
  {1, "area", NULL},
  {2, "is", NULL},
  {3, "vmax", NULL},
  {4, "cp", NULL}
};

LOCAL ip_param xdilist[] =
{
  {1, "isat", NULL},
  {2, "vmax", NULL}
};
```

The circuit parser also uses the IP_PARAM structures to pass the parsed parameter values in to some of the device functions. In this case, one accesses the parameter value with the macro pareval(). See the examples in Sections 5.3.2 and 5.3.3)

5.3 The Support Functions for the New Device Type

When a new element type is added to SIMLAB, the following functions must be provided for it:

```
make_element();
make_element_model();
add_element();
init_element();
eval_element();
eval_element_rhs();
display_element();
free_element_list();
free_element_model_list();
define_element();
dup_element();
alloc_element();
free_element();
alloc_element_model();
free_element_model();
```

These functions are prototyped in the following sections.

The naming is not important here, since these are all local functions which are identified to SIMLAB by address only. You can substitute the name of the new device for "`element`" in the above function names (e.g. `make_resistor()` or `make_capacitor()`).

Now let us examine in turn what each of the element support functions must do.

5.3.1 The `define_element()` Function

The function `define_element()` is called to define the device to the input parser.

The function `define_element()` is called by the input parser just before the circuit file is read. For each device type, this function establishes the name of the device, the number of terminals for the device, the device's model parameters, and the device's instantiation parameters.

Synopsis:

```
LOCAL void
define_element(void);
```

Description:

There are only two things which must take place in this function:

1. The element type (contained in the static `DEVICE_ENTRY` data structure must be assigned to the element primitive structure.
2. The function must call `ip_define_primitive()` to define the device.

The function `ip_define_primitive()` is defined as follows:

```
ENTRY int
ip_define_primitive(char *name, char *prim,
                    int num_terminals,
                    int plist_size, IP_PARAM *plist,
                    int ilist_size, IP_PARAM *ilist);
```

`name` is the string used to identify the device in the circuit file (i.e. "r" for resistor). `num_terminals` contains the number of terminals for the device. `prim` is a pointer to a static `PRIM` structure. The `plist` and `ilist` arguments to `ip_define_primitive()` are pointers to (static) arrays of `IP_PARAM` structures. These structures contain the information for the parameters that can be given to the device or device model, respectively.

Example:

The following is a listing of the code for `define_xd()`, the function which defines the example diode:

define_xd()

```
LOCAL void
define_xd()
{
  xd_prim.type = xd_entry.type;
  (void) ip_define_primitive("r", (char *) &xd_prim, 2,
                             sizeof(xdplist) / sizeof(IP_PARAM),
                             xdplist,
                             sizeof(xdilist) / sizeof(IP_PARAM),
                             xdilist);
}
```

Examples of `xd_prim`, `xd_entry`, `xdplist`, and `xdilist` can be found in Section 5.1.

5.3.2 The make_element() Function

The function `make_element()` is called by the input parser when the parser reads in a device from the circuit file.

Synopsis:

```
LOCAL void
make_element(int netlist[], int numparams,
             IP_PARAM paramlist[], GENERIC_PTR modelptr);
```

Description:

netlist is a pointer to an array of ints which represent the node numbers for the device terminals. numparams is the number of parameters specified for the device in the input file. paramlist is a pointer to an array of IP_PARAMs which contain information about the parameters given to the device in the circuit file. modelptr is a pointer to the model specified for the device. modelptr should be set to NULL in the case of no model.

make_element() should allocate memory for each new device (one recommended technique for doing this is free-listing — see Section 9 for an example). The new device should then be attached to a current_element_list (see the discussion of define_element(). The exact form of the device structure will depend on the device (see Section 5.1).

There are five things which must happen in the make_element() function:

1. The function should allocate memory for the element structure – we recommend having specific functions (i.e., alloc_element()) for this allocation (see Section 5.3.12 for an example).
2. The function should assign the node numbers contained in netlist to the element.
3. The function should check if a model has been defined for this element (i.e. if modelptr != NULL). If so, the relevant parameters should be copied from the model to the element. For more complicated element types which share models, the pointer to the model should be copied to the element structure.
4. The parameter list should be evaluated and parameter values copied to the element structure. Parameter values not given by the model or the parameter list should be given sensible default values.
5. The new element should be inserted into the local element parse-list.

Example:

The following is a listing from the code for make_xd(), the function which makes the example diodes:

make_xd()

```
LOCAL void
make_xd(netlist, numparams, paramlist, gptr)
  int netlist[], numparams;
  IP_PARAM paramlist[];
  GENERIC_PTR gptr;
{
  XD *dptr;
  XDMOD *modelptr = (XDMOD *) gptr;
  int i;

  if (netlist[0] == netlist[1])
    return;

  if (gptr == NULL)
    error("Xd instance for which no model exists!");

  /* Allocate the xd.  */
  dptr = alloc_xd();

  /* Get default max current from the model.  */
  dptr->isat = modelptr->area * modelptr->is;
  dptr->vmax = modelptr->vmax;

  /* Get the xd node numbers.  */
  dptr->anode = netlist[0];
  dptr->cnode = netlist[1];

  /* Get the instance area parameter, if given.  */
  for (i = 0; i < numparams; i++) {
    switch (paramlist[i].id) {
    case 1:
      dptr->isat = pareval(paramlist[i]);
      break;
    case 2:
      dptr->vmax = pareval(paramlist[i]);
      break;
    }
  }
```

make_xd() (Continued)

```
  /* Copy a pointer to the xd model into the xd instance.  */
  dptr->model = modelptr;

  dptr->next = current_xd_list;
  current_xd_list = dptr;
}
```

5.3.3 The make_element_model() Function

The function make_element_model() is called by the input parser when the parser reads in a model definition from the circuit file.

Synopsis:

```
LOCAL GENERIC_PTR
make_element_model(int numparams, IP_PARAM *paramlist);
```

Description:

numparams contains the number of parameters read in for this model definition. paramlist is a pointer to an array of IP_PARAMs which contain information about the parameters read in for this model definition. The exact form of the model structure will depend on the device, but in general, all that is required is a field for each model parameter and a next pointer field.

There are five things which must happen in this function:

1. The function should allocate memory for the device model structure – we recommend having routines for this allocation (see Section 5.3.12 for an example).
2. The function should allocate memory for a PRIM structure (using the function alloc_prim()) and assign the device model pointer to the model field, the make_element function to the readdev field, and the make_element_model function to the readmod field.
3. The parameter list should be evaluated and parameter values copied to the model structure. Parameter values not given by the model or the parameter list should be given sensible default values.
4. The function should insert the model structure into the local model list.

5. The function must return the pointer to the allocated PRIM.

Example:

The following is a listing from the code for `make_xd_model()`, the function which makes models for the example diode:

`make_xd_model()`

```
LOCAL GENERIC_PTR
make_xd_model(numparams, paramlist)
  int numparams;
  ip_param *paramlist;
{
  PRIM *modptr;
  XDMOD *dptr;
  int i;

  /* Allocate space for the model.  */
  dptr = alloc_xdmod();
  dptr->next = current_xdmod_list;
  current_xdmod_list = dptr;

  modptr = alloc_prim();
  modptr->model = (GENERIC_PTR) dptr;
  modptr->readdev = make_xd;
  modptr->readmod = make_xd_model;

  /* Place in the defaults.  */
  dptr->area = 1.0;
  dptr->is = 0.0;
  dptr->cp = 0.0;
  dptr->vmax = 10;
```

make_xd_model() (Continued)

```
  /* Read user-specified parameters.  */
  for (i = 0; i < numparams; i++) {
    switch (paramlist[i].id) {
    case 1:
      dptr->area = pareval(paramlist[i]);
      break;
    case 2:
      dptr->is = pareval(paramlist[i]);
      break;
    case 3:
      dptr->vmax = pareval(paramlist[i]);
      break;
    case 4:
      dptr->cp = pareval(paramlist[i]);
      break;
    }
  } modptr->type = xd_entry.type;

  return ((GENERIC_PTR) modptr);
}
```

5.3.4 The add_element() Function

The function add_element() is called after all the devices are read in, while the circuit is being constructed. The total number of devices added to the circuit by this function is returned.

Synopsis:

```
LOCAL int
add_element(GENERIC_PTR list);
```

Description:

list is a pointer to a list of devices which are to be added to a circuit.

There are three things which need to happen in this function:

1. The function must get NODE structures corresponding to the net numbers assigned by the input parser. This is done with the calls to get_node().
2. The function must mark its unmarked nodes as being Internal (or as Source for voltage source devices).
3. The function must return the number of devices in list.

Example:

The following is a listing of the code for add_xd(), the function which adds the example diode:

add_xd()

```
LOCAL int
add_xd(vptr)
  GENERIC_PTR vptr;
{
  register XD *dptr, *fptr;
  int numdevs = 0;

  for (dptr = (XD *) vptr; dptr; dptr = dptr->next) {
    dptr->nodei = get_node(dptr->anode);
    dptr->nodej = get_node(dptr->cnode);
    if (dptr->nodei->type == Unknown)
      dptr->nodei->type = Internal;
    if (dptr->nodej->type == Unknown)
      dptr->nodej->type = Internal;
  }

  for (dptr = (XD *) vptr; dptr; dptr = dptr->next)
    numdevs++;

  return numdevs;
}
```

5.3.5 The init_element() Function

The function init_element() is called prior to simulation of the circuit to initialize the device (i.e. set up Jacobian and rhs pointers).

Synopsis:

```
LOCAL void
init_element(GENERIC_PTR list);
```

Description:

`list` is a pointer to a list of devices which are to be initialized.

There are five things which must happen in this function (six for a linear device):

1. The function must obtain pointers into the circuit voltage vector by calling the macro `get_v_entry()`. The argument to `get_v_entry()` specifies the row of the vector.
2. The function must obtain pointers into the circuit Jacobian by calling the macro `get_Jacobian_entry()`. Note that the Jacobian is actually sectioned into four "sections", the linear conductive, linear capacitive, nonlinear conductive, and nonlinear capacitive parts, respectively. The first two arguments to `get_Jacobian_entry()` specify the row and column of the Jacobian. The third argument (one of either `LinCond`, `LinCap`, `NonlinCond`, `NonlinCap`) specifies the "section."
3. The function must obtain pointers into the circuit rhs by calling the macro `get_rhs_entry()`. Note that the rhs is actually sectioned into two "sections", charge and current. The first argument to `get_rhs_entry()` specifies the row of the rhs vector. The third argument (one of either `LinCond`, `LinCap`, `NonlinCond`, `NonlinCap`) specifies the "section."
4. The function must obtain pointers into the circuit abs_rhs vector by calling the macro `get_abs_rhs_entry()`. Note that the abs_rhs is actually sectioned into two "sections", charge and current. The first argument to `get_rhs_entry()` specifies the row of the abs_rhs vector. The second argument (one of either `LinCond`, `LinCap`, `NonlinCond`, `NonlinCap`) specifies the "section."
5. The function must "stuff" the Jacobian contributions of linear devices (i.e. devices which obtain addresses into the linear conductive or capacitive sections of the Jacobian). Since the devices are linear, the conductance and/or capacitance only needs to be calculated and "stuffed" once.

Example:

The following is a listing of the code for `init_xd()`, the function which initializes the example diode:

init_xd()

```
LOCAL void
init_xd(vptr)
  GENERIC_PTR vptr;
{
  XD *dptr;
  NODE *a, *c;

  for (dptr = (XD *) vptr; dptr; dptr = dptr->next) {
    a = dptr->nodei;
    c = dptr->nodej;

    dptr->va = get_v_entry(a);
    dptr->vc = get_v_entry(c);

    dptr->acurrent = get_rhs_entry(a, Nonlincond);
    dptr->ccurrent = get_rhs_entry(c, Nonlincond);
    dptr->abs_acurrent = get_abs_rhs_entry(a, Nonlincond);
    dptr->abs_ccurrent = get_abs_rhs_entry(c, Nonlincond);

    dptr->acharge = get_rhs_entry(a, Nonlincap);
    dptr->ccharge = get_rhs_entry(c, Nonlincap);
    dptr->abs_acharge = get_abs_rhs_entry(a, Nonlincap);
    dptr->abs_ccharge = get_abs_rhs_entry(c, Nonlincap);

    dptr->rmataa = get_Jacobian_entry(a, a, Nonlincond);
    dptr->rmatcc = get_Jacobian_entry(c, c, Nonlincond);
    dptr->rmatac = get_Jacobian_entry(a, c, Nonlincond);
    dptr->rmatca = get_Jacobian_entry(c, a, Nonlincond);

    dptr->cmataa = get_Jacobian_entry(a, a, Nonlincap);
    dptr->cmatcc = get_Jacobian_entry(c, c, Nonlincap);
    dptr->cmatac = get_Jacobian_entry(a, c, Nonlincap);
    dptr->cmatca = get_Jacobian_entry(c, a, Nonlincap);
  }
}
```

5.3.6 The `eval_element()` Function

The function `eval_element()` is called to evaluate the circuit Jacobian and rhs.

Synopsis:

```
LOCAL void
eval_element(GENERIC_PTR list);
```

Description:

`list` is a pointer to a list of devices which are to be evaluated.

There are five things which must happen in this function (four for a linear device):

1. The function must obtain the relevant branch voltages from the circuit voltage vector. Since each element structure should contain pointers into the voltage vector, this can be done with a simple pointer de-reference.
2. The function must calculate the relevant branch currents (for a resistive element) and/or charges (for a capacitive element) and stuff them into the rhs vector. Since each element structure should contain pointers into the rhs vector, this can be done with a simple pointer de-reference. Note that rhs entries corresponding to source nodes will have `NULL` rhs pointers, so appropriate tests should be made so that `NULL` pointers are not de-referenced.
3. The function must stuff the absolute value of the branch current and charge into the abs_rhs vector.
4. The function must calculate the relevant incremental conductances and capacitances and stuff them into the circuit Jacobian. Since each element structure should contain pointers into the Jacobian, this can be done with a simple pointer de-reference. This only needs to be done for nonlinear devices, since the linear devices made their Jacobian contributions in the `init_element()` function. Note that rhs entries corresponding to source nodes will have `NULL` rhs pointers, so appropriate tests should be made so that `NULL` pointers are not de-referenced.

Example:

The following is a listing of the code for `eval_xd()`, the function which evaluates the example diode:

eval_xd()

```
LOCAL void
eval_xd(vptr)
  GENERIC_PTR vptr;
{
  register XD *dptr;
  register XDMOD *mptr;
  double vd, qd, id, gd, cd, temp1;
  register BOOL a, c;

  for (dptr = (XD *) vptr; dptr; dptr = dptr->next) {
    a = (dptr->acurrent != NULL);
    c = (dptr->ccurrent != NULL);

    mptr = dptr->model;
    vd = (*dptr->va) - (*dptr->vc);

    /* Compute dc current and derivatives */
    if (vd >= dptr->vmax)
      vd = dptr->vmax;

    temp1 = dptr->isat * exp(vd / VTHERM);
    id = temp1 - dptr->isat;
    gd = temp1 / VTHERM;

    /* Compute charge and capacitance.  */
    if (Dcflag == FALSE) {
      qd = mptr->cp * (vd + vd*vd*vd/3.0);
      cd = mptr->cp * (1 + vd*vd);
```

eval_xd() (Continued)

```
    /* Put xd currents, charges and conds in matrix and rhs */
    if (a) {
      *(dptr->rmataa) += gd;
      *(dptr->cmataa) += cd;
      *(dptr->acurrent) -= id;
      *(dptr->acharge) -= qd;
      *(dptr->abs_acurrent) += ABS(id);
      *(dptr->abs_acharge) += ABS(qd);
      if (c) {
        *(dptr->rmatcc) += gd;
        *(dptr->cmatcc) += cd;
        *(dptr->ccurrent) += id;
        *(dptr->ccharge) += qd;
        *(dptr->abs_ccurrent) += ABS(id);
        *(dptr->abs_ccharge) += ABS(qd);
        *(dptr->rmatac) -= gd;
        *(dptr->rmatca) -= gd;
        *(dptr->cmatac) -= cd;
        *(dptr->cmatca) -= cd;
      }
    } else if (c) {
      *(dptr->rmatcc) += gd;
      *(dptr->cmatcc) += cd;
      *(dptr->ccurrent) += id;
      *(dptr->ccharge) += qd;
      *(dptr->abs_ccurrent) += ABS(id);
      *(dptr->abs_ccharge) += ABS(qd);
    }
  } else {
```

```
eval_xd() (Continued)

      /* DC solution */
      if (a) {
        *(dptr->rmataa) += gd;
        *(dptr->acurrent) -= id;
        *(dptr->abs_acurrent) += ABS(id);
        if (c) {
          *(dptr->rmatcc) += gd;
          *(dptr->rmatac) -= gd;
          *(dptr->rmatca) -= gd;
          *(dptr->ccurrent) += id;
          *(dptr->abs_ccurrent) += ABS(id);
        }
      } else if (c) {
        *(dptr->rmatcc) += gd;
        *(dptr->ccurrent) += id;
        *(dptr->abs_ccurrent) += ABS(id);
      }
    }
  }
} }
```

5.3.7 The `eval_element_rhs()` Function

The function `eval_element_rhs()` is called to evaluate the circuit rhs only.

Synopsis:

```
LOCAL void
eval_element_rhs(GENERIC_PTR list);
```

Description:

`list` is a pointer to a list of devices which are to be evaluated.

There are three things which must happen in this function:

1. The function must obtain the relevant branch voltages from the circuit voltage vector. Since each element structure should contain pointers into the voltage vector, this can be done with a simple pointer de-reference.

2. The function must calculate the relevant branch currents (for a resistive element) and/or charges (for a capacitive element) and stuff them into the rhs vector. Since each element structure should contain pointers into the rhs vector, this can be done with a simple pointer de-reference. Note that rhs entries corresponding to source nodes will have `NULL` rhs pointers, so appropriate tests should be made so that `NULL` pointers are not de-referenced.
3. The function must stuff the absolute value of the branch currents and charge into the abs_rhs vector.

Example:

The following is a listing of the code for `eval_xd_rhs()`, the function which evaluates the example diode rhs:

eval_xd_rhs()

```
LOCAL void
eval_xd_rhs(vptr)
  GENERIC_PTR vptr;
{
  register XD *dptr;
  register XDMOD *mptr;
  double vd, id, qd, temp1, temp2;
  register BOOL a, c;

  for (dptr = (XD *) vptr; dptr; dptr = dptr->next) {
    a = (dptr->acurrent != NULL);
    c = (dptr->ccurrent != NULL);

    mptr = dptr->model;
    vd = (*dptr->va) - (*dptr->vc);

    /* Compute dc current and derivatives */
    if (vd >= dptr->vmax)
      vd = dptr->vmax;

    temp1 = dptr->isat * exp(vd / VTHERM);
    id = temp1 - dptr->isat;

    /* Compute charge.  */
    if (Dcflag == FALSE) {
      qd = mptr->cp * (vd + vd*vd*vd/3.0);
```

eval_xd_rhs() (Continued)

```
      /* Put the xd currents and charges in vector */
      if (a) {
        *(dptr->acurrent) -= id;
        *(dptr->acharge) -= qd;
        *(dptr->abs_acurrent) += ABS(id);
        *(dptr->abs_acharge) += ABS(qd);
      }
      if (c) {
        *(dptr->ccurrent) += id;
        *(dptr->ccharge) += qd;
        *(dptr->abs_ccurrent) += ABS(id);
        *(dptr->abs_ccharge) += ABS(qd);
      }
    } else {

      /* Load only current.  */
      if (a)
        *(dptr->acurrent) -= id;
      if (c)
        *(dptr->ccurrent) += id;
      if (a)
        *(dptr->abs_acurrent) += ABS(id);
      if (c)
        *(dptr->abs_ccurrent) += ABS(id);
    }
  }
}
```

5.3.8 The display_element() Function

The functions display_element() is called to display the contents of all the devices (this is used for debugging).

Synopsis:

```
LOCAL void
display_element(GENERIC_PTR list);
```

Description:

`list` is a pointer to a list of devices which are to be displayed.

The detailed operation of this function is more or less up to the individual programmer, since it is a debugging function, but it is suggested that all relevant parameters for all devices in `list` be displayed.

Example:

The following is a listing of the code for `display_xd()`, the function which displays the example diode:

display_xd()

```
LOCAL void
display_xd(vptr)
  GENERIC_PTR vptr;
{
  XD *dptr;
  int i;

  for (i = 0, dptr = (XD *) vptr; dptr; dptr = dptr->next, i++) {
    sl_printf("XD%d %d %d %e %e %e %e %e %e %ld %ld\n",
              i, dptr->anode, dptr->cnode, dptr->isat,
              dptr->vmax, dptr->model->area, dptr->model->is,
              dptr->model->vmax, dptr->model->cp,
              dptr->nodei, dptr->nodej);
  }
}
```

5.3.9 The `free_element_list()` Function

The function `free_element_list()` is called to destroy the list of devices and free their allocated memory.

Synopsis:

```
LOCAL void
free_element_list(GENERIC_PTR list);
```

Description:

list is a pointer to a list of devices which are to be freed.

Example:

The following is a listing of the code for free_xd_list(), the function which frees the example diode list:

free_xd_list()

```
LOCAL void
free_xd_list(vptr)
  GENERIC_PTR vptr;
{
  register XD *rptr, *fptr;

  for (rptr = (XD *) vptr; rptr; rptr = fptr) {
    fptr = rptr->next;
    free_xd(rptr);
  }
}
```

5.3.10 The free_element_model_list() Function

The function free_element_model_list() is called to destroy the list of element models and free their allocated memory.

Synopsis:

```
LOCAL void
free_element_model_list(GENERIC_PTR list);
```

Description:

list is a pointer to a list of device models which are to be freed.

Example:

The following is a listing of the code for `free_xdmod_list()`, the function which frees the example diode model list:

free_xdmod_list()

```
LOCAL void
free_xdmod_list(vptr)
  GENERIC_PTR vptr;
{
  register XDMOD *rptr, *fptr;

  for (rptr = (XDMOD *) vptr; rptr; rptr = fptr) {
    fptr = rptr->next;
    free_xdmod(rptr);
  }
}
```

5.3.11 The duplicate_element() Function

The function `duplicate_element()` is called to duplicate the device list. A pointer to the duplicated list is returned.

Synopsis:

```
LOCAL GENERIC_PTR
duplicate_element(GENERIC_PTR list, int numdevs);
```

Description:

`list` is a pointer to a list of devices which are to be duplicated.

For each element in the input list, this function should allocate a new element and copy the entire structure of the first. The duplicate list must be made to have the same ordering as the first.

Example:

The following is a listing of the code for `dup_xd()`, the function which duplicates the example diode list:

```
dup_xd()

LOCAL GENERIC_PTR
dup_xd(xd_list, numdevs)
  GENERIC_PTR xd_list;
  int numdevs;
{
  register XD *dptr, *aux;
  register int i;

  alloc_ntype(XD, dptr, numdevs);
  for (aux = (XD *) xd_list, i = 0; aux; aux = aux->next, i++) {
    if (i > numdevs - 1) {
      error("Number of xds specified doesn't match list size");
      break;
    }
    dptr[i].model = aux->model;
    dptr[i].anode = aux->anode;
    dptr[i].cnode = aux->cnode;
    dptr[i].nodei = aux->nodei;
    dptr[i].nodej = aux->nodej;
    dptr[i].isat = aux->isat;
    dptr[i].vmax = aux->vmax;
    if (i < numdevs - 1)
      dptr[i].next = (XD *) &dptr[i + 1];
    else
      dptr[i].next = (XD *) NULL;
  }

  return ((GENERIC_PTR) dptr);
}
```

5.3.12 The Memory Management Functions

In the above definitions of make_element(), make_element_model(), free_element_list(), and free_element_model_list(), some memory management routines were used. In the examples for the example diode, these were called, alloc_xd(), alloc_xdmod(), free_xd(), and free_xdmod(). These are all simple routines which use built-in utilities.

In this section, we will list the memory management routines for the example diode. They will be virtually identical for any other device — the only thing that will need to be changed are the obvious structure and variable names. The routines work by calling the generic free-listing routines provided by SIMLAB. The only thing you need to provide is the free list itself.

This is the memory management code for the example diode structure:

```
LOCAL XD *free_xds = (XD *) NULL;


LOCAL void
free_xd(rptr)
  XD *rptr;
{
  chunk_recycle((GENERIC_PTR) rptr, (GENERIC_PTR *) &free_xds);
}


LOCAL XD *
alloc_xd()
{
  return ((XD *) gimme_chunk(sizeof(XD),
  (GENERIC_PTR *) &free_xds));
}
```

This is the memory management code for the example diode model structure:

```
LOCAL XDMOD *free_xdmods = (XDMOD *) NULL;


LOCAL void
free_xdmod(rptr)
  XDMOD *rptr;
{
  chunk_recycle((GENERIC_PTR) rptr, (GENERIC_PTR *) &free_xdmods);
}


LOCAL XDMOD *
alloc_xdmod()
{
  return ((XDMOD *) gimme_chunk(sizeof(XDMOD),
  (GENERIC_PTR *) &free_xdmods));
}
```

5.4 Defining the Device

The final step in adding a new device type to SIMLAB is defining the device for the input parser. This is done with a call to the function `define_device()`, which installs information about the device in an internal table for later use by SIMLAB. The call to `define_device()` is made from within `init_local_devices()` in the file `local.c`.

Synopsis:

```
ENTRY int
define_device(DEVICE_ENTRY *sptr);
```

Description:

`sptr` points to a `DEVICE_ENTRY` structure which contains all the information that SIMLAB needs to handle an arbitrary device type. The `DEVICE_ENTRY` structure should be made known before `define_device()` is called with it via an `extern` declaration. For the existing SIMLAB devices, these declarations take place in each device's include file. For devices you add via `local.c`, it is sufficient to declare the device entry in `local.c`.

Example:

The example element would be installed with a call such as the following:

```
ENTRY void
init_local_devices()
{
  extern DEVICE_ENTRY xd_entry;

  define_device(&xd_entry);
}
```

6 Adding an Interactive Command

New commands can be added to the SIMLAB front-end with the function `install_command()`, which is prototyped as:

```
ENTRY void
install_command(char *name, void (*cmd_ptr)(), int min_args, int max_args);
```

`name` is a character string of the literal name of the command. `cmd_ptr` is a pointer to the function that is to be invoked when the user inputs the command contained in `name`. `min_args` and `max_args` contain the minimum and maximum number of arguments for this particular command. Commands should be installed from within `local_init()`.

Suppose we had a simple function called `hello()`:

```
LOCAL void
hello()
{
  sl_printf("Hello World\n");
}
```

We could install this function into the front-end like this:

```
install_command("hi", hello, 0, 0);
```

Now, when the user types `hi`, the function `hello()` will be called, and `"Hello World"` will be printed.

When the installed functions are called, they will also be passed any arguments given to them. The arguments will be passed as a linked list of strings. A `NULL` list will be passed in if no arguments were given. Let us modify `hello()` so that it takes an arbitrary number of arguments.

```
LOCAL void
new_hello(list)
  PN *list;
  { if (list == NULL)
    (void) sl_printf("Hello World\n");
  else {
    PN *pn;
    for (pn = list; pn; pn = getcdr(pn))
      sl_printf("Hello }
    }
```

The `PN` structure is defined in `interp.h`. It is not necessary to know any of the internals of the structure. One needs only a few macros to access relevant information:

`getcdr(pn)` returns the "cdr" of `pn`, that is, the next `PN` in the list.

`getcar(pn)` returns the "car" of `pn`, that is, `pn`'s data element.

`getpstr(pn)` returns the actual string associated with the data element.

The new definition of `hello()` will print a `Hello` with each argument given to `hi`. So if `hi` is installed with:

```
install_command("hi", new_hello, 0, 0);
```

invoking `hi President Bush` would produce the output:

```
Hello President
Hello Bush
```

Note that invoking `hi "President Bush"` would produce:

```
Hello President Bush
```

7 Adding an Environment Variable

Variables which are present in the SIMLAB environment can be set by SIMLAB input commands. For instance, when the user types `dodc = 1`, the `GLOBAL` variable `Dodc` is set to 1 by the front end. SIMLAB provide functions that allow you to associate environment variables with program variables.

Variables can be added to the SIMLAB environment with the function `install_variable()`, which is prototyped as:

```
ENTRY void
install_variable(char *name, PNTYPE type, GENERIC_PTR address);
```

`name` is a character string of the literal name of the variable. `type` is an enumerated data type representing the type of the variable, and can be one of `Real`, `Integer`, or `Boolean` (defined in `interp.h`). `address` is a pointer to a program variable to be associated with the environment variable. Unless a variable will be installed and uninstalled from within the same function call, `address` should not be the location of an automatic variable.

Variables can be removed from the SIMLAB environment with the function `uninstall_variable()`, which is prototyped as:

```
ENTRY void
uninstall_variable(char *name);
```

`name` is a character string of the literal name of the variable to be removed.

The use of `install_variable()` and `uninstall_variable()` allow the programmer to establish context-dependent environment variables. Consider the following code fragment:

```
LOCAL int maxpcgiters = 32, maxdcpcgiters = -1;

ENTRY BOOL
pcg_smsolve(use, mode, circuit, solution)
  USETYPE use;
  MODETYPE mode;
  CIRCUIT *circuit;
  SOLUTION *solution;
{
  .
  .
  .
  case OpenUse:
    if (mode == DCMode)
      install_variable("maxdcpcg", Integer, (GENERIC_PTR) &maxdcpcgiters);
    else if (mode == TransientMode)
      install_variable("maxpcg", Integer, (GENERIC_PTR) &maxpcgiters);
    break;
  case CloseUse:
    if (mode == DCMode)
      uninstall_variable("maxdcpcg");
    else if (mode == TransientMode)
      uninstall_variable("maxpcg");
    break;
  .
  .
  .
}
```

We assume in the above example that `pcg_smsolve` has been installed properly into SIMLAB as a `solve_linear` function for both the DC and transient case. When `pcg_smsolve()` is called with `OpenUse`, one of the context dependent variables for `pcg_smsolve()` will be installed into the SIMLAB environment. If the `OpenUse` call is with `DCMode`, the variable `"maxdcpcg"` will be installed. If the `OpenUse` call is with `TransientMode`, the variable `"maxpcg"` will be installed.

If the user inputs "`set dc solve_linear pcg`" then `pcg_smsolve()` will be called with `OpenUse` and `DCMode` and the user will see `maxdcpcg` in his environment. Similarly, If the user inputs "`set dc solve_linear pcg`" then `pcg_smsolve()` will be called with `OpenUse` and `TransientMode` and the user will see `maxpcg` in his environment. If the user inputs "set solve_linear pcg" then pcg_smsolve() will be called twice with OpenUse, once with DCMode and once with TransientMode. The user will see both maxdcpcg and maxpcg in his environment.

Conversely, the variables maxdcpcg and maxpcg will be removed from the SIMLAB environment when the pcg method is de-selected, i.e., after pcg_smsolve() is called with CloseUse and the corresponding mode argument. This action will happen when the user sets solve_linear to some other solution method.

Author's note: At some point, the functional interface for putting new functions into SIMLAB may be changed so that one can associate variables with a given function when the function is actually installed.

8 Adding a Circuit File Option Parameter

Circuit file option parameters which are present in SIMLAB can be set from the circuit file read by SIMLAB. For instance, the following line in the circuit file sets the length of the simulation interval and provides information on what type of simulation is required.

```
simlab options stop=100ns dodc=0 dtran=1
```

New circuit file option parameters can be added to SIMLAB with the function `install_option()`, which is prototyped as:

```
ENTRY void
install_option(char *name, int type, GENERIC_PTR address);
```

`name` is a character string of the literal name of the option parameter. `address` is a pointer to a program variable to be associated with the option parameter. `address` should never be the location of an automatic variable. `type` is a cast to type `int` of a `PNTYPE` enumerated data type representing the type of the variable corresponding to the option parameter, and can be one of `Real`, `Integer`, or `Boolean` (defined in `interp.h`).

9 Reference Guide

9.1 SIMLAB Enumerated Data Types

The following are some of the enumerated data types used in SIMLAB. Their definitions can be found in simlab.h.

9.1.1 ENTRYTYPE

Purpose:

Enumerated data type for differentiating Jacobian and rhs entries.

Definition:

```
typedef enum entrytype {
  Unknowntype, Lincond, Lincap, Nonlincond, Nonlincap, Just_a_number
} ENTRYTYPE;
```

9.1.2 MODETYPE

Purpose:

Enumerated data type for simulation modes.

Definition:

```
typedef enum modetype {
  UnknownMode, NoMode, DCMode, TransientMode, AllMode
} MODETYPE;
```

9.1.3 NODETYPE

Purpose:

Enumerated data type for nodes.

Definition:

```
typedef enum nodetype {
  Unknown, Ground, Source, Internal, NotFree
} NODETYPE;
```

9.1.4 RUNTYPE

Purpose:

Enumerated data type for differentiating transient simulation mode.

Definition:

```
typedef enum runtype {
  UnknownRun, Run, Continue
} RUNTYPE;
```

9.1.5 USETYPE

Purpose:

Enumerated data type for differentiating calls to functional control function pointers.

Definition:

```
typedef enum usetype {
  UnknownUse, Execute, OpenUse, CloseUse, InitUse, Stats
  } USETYPE;
```

9.2 SIMLAB Structures

The following are some of the structures used in SIMLAB. Their definitions can be found in simlab.h.

9.2.1 SOLUTION

Purpose:

Structure to hold the circuit state. This structure is used both in the pt and the waveform-newton solution methods.

Definition:

```
typedef struct solution_struct SOLUTION;
struct solution_struct {
  int solution_length;
  double *stepsize_history;
  double *v, **vhist, *vmax, *guess;
  double *current, **old_current, *abs_current, **abs_old_current;
  double *charge, **old_charge, *abs_charge, **abs_old_charge;
  double *rhs, *old_rhs, *abs_rhs, *abs_old_rhs;
  int num_xvects;
  double **xvect;
  GENERIC_PTR sm;
};
```

Description:

The SOLUTION structure holds pointers to the various vectors needed to perform both DC and transient simulations.

The solution_length field contains the length of the solution vectors. In general, this will be the same as the numnodes field in the CIRCUIT structure corresponding to this solution. The stepsize_history field contains a pointer to an array of numbers which are the values of the integration step sizes for the last few steps. The number of past steps will depend on the integration method. The values are indexed so that the most recent time step is in position 0, the next most recent in position 1, etc. This is the same indexing used for the vhist, old_current, and old_charge vector arrays.

The v field points to a double array which contains the values of the node voltages (for both internal and fixed nodes) for the circuit. The vmax field points to a double array which contains the maximum of the absolute value of all previous voltages for each node (for both internal and fixed). The `guess` field is used to hold the initial voltage guess for Newton's method. The `vhist` field points to an array of `double` pointers, each of which points to a vector containing past voltage information for each node of the circuit — `vhist[0]` contains the most recent past voltage vector, `vhist[1]` contains the next most recent, etc.

The `current` field points to a `double` array which contains the values of the sums of currents flowing into each internal circuit node, i.e., the value of $-f(v)$ at each internal circuit node (in both the DC and transient case). The `old_current` field points to an array of `double` pointers, each of which points to a vector containing past current information for each internal circuit node. The indexing is the same as for the `vhist` field. The `abs_current` field points to a `double` array which contains the sums of the absolute values of the currents flowing into each internal circuit node. The `abs_old_current` field point to an array of `double` pointers, each of which points to a vector containing past absolute current vectors.

The `charge` field points to a `double` array which contains the values of the sums of charges flowing into each internal circuit node, i.e., the value of $-f(v)$ at each internal circuit node (in both the DC and transient case). The `old_charge` field points to an array of `double` pointers, each of which points to a vector containing past charge information for each internal circuit node. The indexing is the same as for the `vhist` field. The `abs_charge` field points to a `double` array which contains the sums of the absolute values of the charges flowing into each internal circuit node. The `abs_old_charge` field point to an array of `double` pointers, each of which points to a vector containing past absolute charge vectors.

The `rhs` field points to a `double` array which contains the right-hand side values for the internal circuit node (see the definition of right-hand side in Section 3.1.3). The `old_rhs` field points to a `double` array which contains the old right-hand side values for the internal circuit node (see the definition of old right-hand side in Section 3.1.3). The `abs_rhs` and `abs_old_rhs` fields point to `double` arrays which contain the right-hand side and old right-hand side values formed with the `abs_charge` and `abs_current` entries instead of the `charge` and `current` entries. Note that the "abs" quantities are not the absolute value of the corresponding quantities, but are rather the sum of the absolute values of the components of the corresponding quantities.

The `xvect` field contains a pointer to an array of `double` pointers, each of which points to a scratch vector of length `solution_length`. The number of `xvects` is given by the `num_xvects` field.

The `sm` field points to the system Jacobian matrix.

In general, the vectors corresponding to voltage quantities have an entry for every node in the circuit, both internal and fixed (i.e., the length is specified by the `numnets` field of the `CIRCUIT` corresponding to this `SOLUTION`). The vectors corresponding to charge and current quantities have an entry only for the internal nodes in the circuit (i.e., the length is specified by the numnodes field of the CIRCUIT corresponding to this SOLUTION). The exception to this is the guess field, which contains voltage information — its length is numnodes.

9.2.2 CIRCUIT

Purpose:

Structure to hold the circuit.

Definition:

```
typedef struct circuit_struct CIRCUIT;
struct circuit_struct {
  char *name;
  int flags;
  int numfixed, numfree, numnets, numnodes;
  int numdevs[MAXDEVS];
  DEVICE *devices, *sources, *devlist;
  NODE *nodes, *fixed, *ground, *nodelist;
  SETIC *initial_conditions;
};
```

Description:

The name field is a pointer to a null terminated character string containing the name of the circuit file from which this circuit structure was created. The flags field is used to hold various bit-field flags about the circuit. The numfixed fields contain the number of fixed nodes (i.e., nodes with a voltage source connected). The numfree field contains the number of free nodes (i.e., nodes with no voltage source or initial condition). The numnodes field contains the number of internal circuit nodes (i.e., nodes with no voltage source but possibly with an initial condition). The numnets field contains the total number of nets (i.e., numnodes + numnodes). Note, in general, for DC simulation, the number of nodes we are simulating will be given by numfree, whereas in transient simulation, the number of nodes we are simulating will be given by numnodes. In the absence of any initial conditions, these will be equal. The numdevs field is an array of integers (indexed by device type), each entry of which contains the number of elements in the circuit for that particular device. The devlist field contains a pointer to an array of all DEVICE structures used by the circuit. These are separated into linked lists for voltage sources and non-sources, pointed to by the sources and devices fields, respectively. The nodelist field contains a pointer to an array of all NODE structures used by the circuit. These are separated into linked lists for internal circuit nodes, fixed circuit nodes (i.e., connected to a voltage source), and ground, pointed to by the `nodes`, `fixed`, and `ground` fields, respectively. The `initial_conditions` field contains a pointer to a list of `SETIC` structures, which contain information about any initial conditions specified in the circuit input file.

9.2.3 NODE

Purpose:

Structure to hold node information.

Definition:

```
typedef struct node_struct NODE;
struct node_struct {
  NODETYPE type;
  int flags;
  int num, renum, old;
  NODE *next;
};
```

Description:

The `type` field is an enumerated type (`NODETYPE`) indicating what type of node this structure represents. The possible types are `Unknown`, `Ground`, `Source`, `Internal`, and `NotFree`, with the obvious meanings. The `flags` field is used to hold various bit-field flags about the node. The `old` field contains the node number for this node assigned by the circuit parser. The `num` field contains the number for this node as originally assigned by SIMLAB. The `renum` field contains the present number for this node. In general, this will probably be the same as `num`; The `renum` field is included as a separate field in case the nodes are re-ordered after the SIMLAB's initial ordering. The `next` field is a pointer to another `NODE`.

9.2.4 DEVICE

Purpose:

Closure structure for devices.

Definition:

```
typedef struct device_struct DEVICE;
struct device_struct {
  int type, numdevs;
  GENERIC_PTR devlist;
  GENERIC_PTR modlist;
  int (*add)();
  void (*init)();
  void (*eval)();
  void (*rhseval)();
  void (*display)();
  void (*devfree)();
  void (*modfree)();
  GENERIC_PTR (*devdup)();
  DEVICE *next;
};
```

Description:

Each `DEVICE` structure in the circuit is used to represent a different device type. For each type, SIMLAB assigns a unique identifier, which is kept on the `type` field of the `DEVICE`. The `devlist` field contains a pointer to the head of a list of structures for the actual elements of this device type. (i.e. to a list of resistor structures or a list of capacitor structures). The `numdevs` field contains the number of elements in the list pointed to by `devlist`. The `add` field contains a pointer to the function that adds the devices in the list pointed to by `devlist` to the circuit structure. The `init` field contains a pointer to a function that initializes the devices in the list pointed to by `devlist` for simulation. The `eval` field contains a pointer to a function that evaluates the devices in the list pointed to by `devlist` and calculates their Jacobian and rhs entries. The `rhseval` field contains a pointer to a function that evaluates the devices in the list pointed to by `devlist` and calculates their rhs entries *only*. The `display` field contains a pointer to a function that displays all the devices in the list pointed to by `devlist`. The `devfree` field contains a pointer to a function that frees the structures in the list pointed to by `devlist`. The `devdup` field contains a pointer to a function that duplicates the structures in the list pointed to by `devlist` and returns a pointer to the duplicated list. The `next` field contains a pointer to the next `DEVICE` structure.

9.2.5 TV_PAIR

Purpose:

Time-voltage pair.

Definition:

```
typedef struct tv_pair_struct TV_PAIR;
struct tv_pair_struct {
  double time, voltage;
  struct tv_pair_struct *next;
  };
```

9.2.6 PLOTBUFF

Purpose:

Structure to hold voltage waveform for later plotting.

Definition:

```
typedef struct plotbuff_struct PLOTBUFF;
struct plotbuff_struct {
  int num, renum;
  double *v;
  char *name;
  TV_PAIR *tvpairs;
  TV_PAIR *last;
  PLOTBUFF *next;
};
```

9.2.7 SETIC

Purpose:

Structure to hold initial condition information.

Definition:

```
typedef struct setic_struct SETIC;
struct setic_struct {
  int old, num;
  double initval;
  SETIC *next;
};
```

9.2.8 DEVICE_ENTRY

Purpose:

Structure to hold device information.

Definition:

```
typedef struct device_entry_struct DEVICE_ENTRY;
struct device_entry_struct {
  int type;
  int numdevs;
  char *name;
  BOOL is_source;
  GENERIC_PTR *parse_list;
  GENERIC_PTR *model_list;
  int (*add)();
  void (*init)();
  void (*eval)();
  void (*rhseval)();
  void (*display)();
  void (*devfree)();
  void (*modfree)();
  GENERIC_PTR (*devdup)();
  void (*devdefine)();
};
```

Description:

The `type` field contains a unique identifier (assigned by SIMLAB) for this device. The `numdevs` field contains the number of devices read in (also assigned by SIMLAB). The `name` field contains a character string of the device name. The `is_source` field indicates whether or not the device is a voltage source. The `parse_list` contains the address of a pointer to the head of the list of devices that are read in. `model_list` contains the address of a pointer to the head of the list of models that are read in. The `add` field contains a pointer a function that adds the devices to the circuit structure. The `init` field contains a pointer to a function that initializes the devices for simulation. The `eval` field contains a pointer to a function that evaluates the devices and calculates the Jacobian and rhs entries. The `rhseval` field contains a pointer to a function that evaluates the devices in the device list and calculates the rhs entries only. The `display` field contains a pointer to a function that displays all the devices in the device list. The `devfree` field contains a pointer to a function that frees the list of devices. The `modfree` field contains a pointer to a function that frees the list of device models. The `devdup` field contains a pointer to a function that duplicates the list of devices. The `devdefine` field contains a pointer to a function that defines the device to the circuit parser.

9.2.9 PRIM

Purpose:

Closure structure for circuit parser.

Definition:

```
typedef struct allmod {
  int type;
  GENERIC_PTR model;
  void (*readdev)();
  GENERIC_PTR (*readmod)();
} PRIM;
```

9.3 SIMLAB'S Global Variables

9.3.1 `PresentTime`

Purpose:

Present simulation time.

Synopsis:

```
#include "simlab.h"

EXTERN double PresentTime;
```

Description:

`PresentTime` is a global variable which indicates the current time of a transient circuit simulation. For a DC simulation, `PresentTime` is zero. `PresentTime` corresponds to the front-end user variable "`time`".

9.3.2 `FinalTime`

Purpose:

Final simulation time.

Synopsis:

```
#include "simlab.h"

EXTERN double FinalTime;
```

Description:

`FinalTime` is a global variable which indicates the final time for a transient circuit simulation run. The final simulation timestep will equal `FinalTime`. `FinalTime` corresponds to the front-end user and circuit file option variable "`stop`".

Default:

```
FinalTime = 1.0;
```

9.3.3 Cmin, Gmin

Purpose:

Parasitic capacitance and conductance.

Synopsis:

```
#include "simlab.h"

EXTERN double Cmin;
EXTERN double Gmin;
```

Description:

Cmin and Gmin specify the parasitic conductance and capacitance values for aiding convergence of transient and DC simulation, respectively. Cmin and Gmin correspond to the front-end user and circuit file option variables "cmin" and "gmin", respectively.

Default:

```
Cmin = 1.e-18;
Gmin = 1.e-12;
```

9.3.4 NRVrel, NRVabs, NRIrel, NRIabs

Purpose:

Absolute and relative voltage and current convergence criteria.

Synopsis:

```
#include "simlab.h"

EXTERN double NRVrel;
EXTERN double NRVabs;
EXTERN double NRIrel;
EXTERN double NRIabs;
```

Description:

The global variables NRVrel, NRVabs NRIrel and NRIabs specify respectively the voltage and current relative and absolute convergence criteria. These variables can be set by the user from the front-end or from within the circuit specification file. `NRVrel`, `NRVabs`, `NRIrel`, `NRIabs` correspond to the front-end user and circuit file option variables "`nrvrel`" "`nrvabs`" "`nrcrel`" "`nrcabs`", respectively. respectively.

Default:

```
NRVrel = 1.e-3;
NRVabs = 1.e-3;
NRIrel = 1.e-3;
NRIabs = 1.e-9;
```

9.3.5 `LTErel`, `LTEabs`

Purpose:

Absolute and relative local truncation error criteria.

Synopsis:

```
#include "simlab.h"

EXTERN double LTErel;
EXTERN double LTEabs;
```

Description:

The global variables `LTErel` and `LTEabs` specify the absolute and relative local truncation error criteria. These variables should be used by any lte-timestep control routines. `LTErel` and `LTEabs` correspond to the front-end user and circuit file option variables "`lterel`" and "`lteabs`", respectively.

Default:

```
LTErel = 0.01;
LTEabs = 0.005;
```

9.3.6 `NRVlimit`

Purpose:

Newton-Raphson $\Delta\, V$ limit.

Synopsis:

```
#include "simlab.h"

EXTERN double NRVlimit;
```

Description:

The global variable `NRVlimit` specifies the maximum amount that any node voltage can be changed in one step of the Newton-Raphson iteration. The same variable is used for both DC and transient solution. `NRVlimit` corresponds to the front-end user and circuit file option variable "`nralpha`".

Default:

```
NRVlimit = 0.3;
```

9.3.7 `MaxtrNR`, `MaxdcNR`

Purpose:

Synopsis:

```
#include "simlab.h"

EXTERN int MaxtrNR;
EXTERN int MaxdcNR;
```

Description:

`MaxtrNR` and `MaxdcNR` correspond to the front-end user and circuit file option variables "`maxnr`" and "`maxdcnr`", respectively.

9.3.8 `NewJacob`

Purpose:

Number of Newton iterations per Jacobian evaluation.

Synopsis:

```
#include "simlab.h"

EXTERN int NewJacob;
```

Description:

NewJacob corresponds to the front-end user and circuit file option variable "newjacob".

9.3.9 Dodc, Dotran

Purpose:

Synopsis:

```
#include "simlab.h"

EXTERN BOOL Dodc;
EXTERN BOOL Dotran;
```

Description:

Dodc and Dotran correspond to the front-end user and circuit file option variables "dodc" and "dotran", respectively.

9.3.10 Verbose, Simdebug

Purpose:

Synopsis:

```
#include "simlab.h"

EXTERN BOOL Verbose;
EXTERN BOOL Simdebug;
```

Description:

Verbose and Simdebug correspond to the front-end user variables "verbose" and "simdebug", respectively. Verbose corresponds to the circuit file option variable "verbose".

9.3.11 `Minsteps`, `Maxsteps`

Purpose:

Synopsis:

```
#include "simlab.h"

EXTERN int Minsteps;
EXTERN int Maxsteps;
```

Description:

`Minsteps` and `Maxsteps` correspond to the front-end user variables "`minsteps`" and "`maxsteps`", respectively.

9.4 SIMLAB'S Standard Algorithms

9.4.1 dc_pt_solve(), tran_pt_solve()

Purpose:

To perform the DC or transient solution of the circuit.

Synopsis:

```
#include "simlab.h"

ENTRY void
dc_pt_solve(USETYPE use, MODETYPE mode, CIRCUIT *circuit,
           RUNTYPE type);
ENTRY void
tran_pt_solve(USETYPE use, MODETYPE mode, CIRCUIT *circuit,
             RUNTYPE type);
```

Description:

The `dc_pt_solve()` and `tran_pt_solve()` functions are dispatching functions operating in accordance with the use specified in `use`. If `use` equals `Execute`, the `dc_pt_solve()` function causes the invocation of the `SOLVE_NONLINEAR()` macro to solve the nonlinear set of equations that describe the DC behavior of the circuit pointed to by `circuit`. If `use` equals `Execute`, the `tran_pt_solve()` function causes the invocation of the `INTEGRATE()` to solve the nonlinear set of differential equations that describe the transient behavior of the circuit pointed to by `circuit`.

Note `dc_pt_solve()` is always called with `mode` set to `DCMode` since it is only installed to operate under that mode. `tran_pt_solve()` on the other hand is always called with `mode` set to `TransientMode` since it is only installed to operate under that mode.

9.4.2 beuler(), feuler, gear2(), mixed(), trap()

Purpose:

To perform the integration of the circuit in transient mode.

Synopsis:

```
#include "simlab.h"

ENTRY void
gear2(USETYPE use, MODETYPE mode, CIRCUIT *circuit, SOLUTION *solution,
      RUNTYPE type);
ENTRY void
mixed(USETYPE use, MODETYPE mode, CIRCUIT *circuit, SOLUTION *solution,
      RUNTYPE type);
ENTRY void
trap(USETYPE use, MODETYPE mode, CIRCUIT *circuit, SOLUTION *solution,
     RUNTYPE type);
ENTRY void
beuler(USETYPE use, MODETYPE mode, CIRCUIT *circuit, SOLUTION *solution,
       RUNTYPE type);
ENTRY void
feuler(USETYPE use, MODETYPE mode, CIRCUIT *circuit, SOLUTION *solution,
       RUNTYPE type);
```

Description:

The `gear2()`, `mixed`, `trap()`, `beuler` and `feuler` functions are dispatching functions operating in accordance with the use specified in `use`. If `use` equals `Execute`, all of these functions perform a time integration of the differential equations describing the transient behavior of the circuit pointed to by `circuit`. These functions implement time discretization algorithms and cause the invocation of the `SOLVE_NONLINEAR()` macro to solve the resulting nonlinear set of discretized equations. The `gear2` function uses the 2^{nd} order backwards-differentiation formula, also known as 2^{nd} order Gear, for accomplishing that discretization. The `trap` function uses the trapezoidal formula, also known as the 2^{nd} order Adams-Moulton algorithm. The `mixed` function uses an hybrid algorithm by performing one step of backwards differentiation followed by one step of the trapezoidal method. The `beuler` function uses the backward-Euler algorithm, also known as the 1^{st} order Adams-Moulton algorithm. Finally the `feuler` function uses the forward-Euler algorithm, i.e., the 1^{st} order Adams-Bashforth formula.

Diagnostics:

Note that all of these functions are called with `mode` set to `TransientMode` since they are only installed to operate under that mode. If these function are to be used under another simulation method, care must be taken to ensure that they are installed to operate only during transient mode.

9.4.3 dc_pt_newton(), pt_newton()

Purpose:

Performs the solution of a nonlinear set of equations using Newton-Raphson's multi-dimensional method.

Synopsis:

```
#include "simlab.h"

ENTRY BOOL
dc_pt_newton(USETYPE use, MODETYPE mode, CIRCUIT *circuit, SOLUTION *solution,
double alpha, double h);
ENTRY BOOL
pt_newton(USETYPE use, MODETYPE mode, CIRCUIT *circuit, SOLUTION *solution,
double alpha, double h);
```

Description:

The dc_pt_newton() and pt_newton() are dispatching functions operating in accordance with the use specified in use. If use equals Execute, all of these functions perform the solution of the nonlinear set of equations describing the pointwise behavior of the circuit pointed to by circuit. These functions implement the multi-dimensional Newton-Raphson algorithm and cause the invocation of the SOLVE_LINEAR() macro to solve the linear set of equations obtained at each Newton iteration.

Diagnostics:

dc_newton() is always called with mode set to DCMode since it was only installed to work under that mode. On the other side, the newton() function is always called with mode set to TransientMode for the same reasons.

9.4.4 direct_smsolve()

Purpose:

Performs linear system solution on a sparse matrix.

Synopsis:

```
#include "simlab.h"

ENTRY BOOL
direct_smsolve(USETYPE use, MODETYPE mode, CIRCUIT *circuit,
               SOLUTION *solution);
```

Description:

The `direct_smsolve()` function is a dispatching function operating in accordance with the use specified in `use`. If `use` equals `Execute`, this function will call the appropriate handler to perform a linear solution on the sparse matrix referred to in `solution`.

Diagnostics:

This function can be called either in DCMode or TransientMode.

9.4.5 `pt_smstart()`, `pt_smrhsstart()`, `pt_eval_gmin()`, `pt_eval_cmin()` `pt_eval_gmin_rhs()`, `pt_eval_cmin_rhs()`

Purpose:

Perform initialization on sparse matrix and right-hand side elements.

Synopsis:

```
#include "simlab.h"

ENTRY void pt_smstart(SOLUTION *solution, BOOL isdc);
ENTRY void pt_smrhsstart(SOLUTION *solution, BOOL isdc);
ENTRY void pt_eval_gmin(SOLUTION *solution, double gmin);
ENTRY void pt_eval_cmin(SOLUTION *solution, double cmin);
ENTRY void pt_eval_gmin_rhs(SOLUTION *solution, double gmin);
ENTRY void pt_eval_cmin_rhs(SOLUTION *solution, double cmin);
```

Description:

The functions `pt_smstart()`, `pt_smrhsstart()`, `pt_eval_gmin()`, `pt_eval_cmin()`, `pt_eval_gmin_rhs()` and `pt_eval_cmin_rhs()` perform some initialization steps needed to setup the matrix and right-hand side elements prior to matrix solution. Namely the `pt_smstart()` function is called prior to device evaluation and merges together the linear capacitive and conductive parts of the Jacobian matrix and the right-hand side. The function `pt_smrhsstart()` performs the same operation on the right-hand side only. These two functions are called from within the `direct_evaluate()`, `direct_evaluate_rhs()`, `direct_dc_evaluate()` and `direct_dc_evaluate_rhs()` functions. The function `pt_eval_gmin_rhs()` is called from `direct_dc_evaluate()` and `direct_dc_evaluate_rhs()` and causes the addition of a small minimum conductance to the right-hand side vector. This function should only be called when performing a DC simulation. `pt_eval_cmin_rhs()` has a similar effect on the right-hand side except that it adds a small capacitance term and is called from both `direct_dc_evaluate()` and `direct_dc_evaluate_rhs()`. This function should only be called when performing a DC simulation.

Diagnostics:

The functions pt_eval_gmin(), pt_eval_cmin() are provided for backwards compatibility. They perform similarly to their counterparts described above except that they add their values to the jacobian matrix diagonal entries in addition to the right-hand side. Currently this is being done explicitly in the initialization, from pt_solve and dc_pt_solve.

9.4.6 direct_evaluate(), direct_evaluate_rhs(), direct_dc_evaluate(), direct_dc_evaluate_rhs()

Purpose:

Performs device evaluation and recomputation of the Jacobian and right-hand side elements.

Synopsis:

```
#include "simlab.h"

ENTRY void
direct_evaluate(MODETYPE mode, CIRCUIT *circuit,
                SOLUTION *solution, double alpha);
ENTRY void
direct_evaluate_rhs(MODETYPE mode, CIRCUIT *circuit,
                    SOLUTION *solution, double alpha);
ENTRY void
direct_dc_evaluate(MODETYPE mode, CIRCUIT *circuit,
                   SOLUTION *solution, double alpha);
ENTRY void
direct_dc_evaluate_rhs(MODETYPE mode, CIRCUIT *circuit,
                       SOLUTION *solution, double alpha));
```

Description:

The direct_evaluate() and direct_dc_evaluate() functions call the appropriate device handling functions in order to perform an evaluation of all the circuit elements. After these functions are called, both the Jacobian matrix and right-hand side of the Newton iteration equation are recomputed by merging their capacitance and conductance parts. The functions direct_evaluate_rhs() and direct_dc_evaluate_rhs() have the same effect but only on the right-hand side of the Newton iteration equation. When evaluating the circuits devices in order to recompute the Jacobian or the right-hand side of the Newton iteration, appropriate functions are also called in order to add either cmin or gmin, respectively the minimum capacitance and conductance. During DC simulation, the device evaluation phase is preceded by a call to either pt_eval_gmin() or pt_eval_gmin_rhs() in order to add the minimum conductance to the appropriate positions. Similarly, during transient simulation, the device evaluation phase is preceded to a either pt_eval_cmin() or pt_eval_cmin_rhs() in order to add the minimum capacitance to the appropriate positions in the matrix and the right-hand side. See Subsection 9.4.5 for additional information.

Diagnostics:

The functions direct_evaluate() and direct_evaluate_rhs() are always called with mode set to DCMode since they were only installed to work under that mode. On the other the functions direct_dc_evaluate() direct_dc_evaluate_rhs() are always called with mode set to TransientMode for the same reasons.

9.4.7 init_plots(), close_plot(), update_plot(), DCplot(), clear_plots(), show_plot()

Purpose:

Routines for handling the plot of simulation results.

Synopsis:

```
#include "simlab.h"

ENTRY void init_plots(MODETYPE mode, CIRCUIT *circuit, RUNTYPE type);
ENTRY void close_plot(void);
ENTRY void update_plot(BOOL use_static);
ENTRY void add_plot_node(int num, char *name);
ENTRY void DCplot(void);
ENTRY void clear_plots(PLOTBUFF *fst_wptr);
ENTRY void show_plot(void);
ENTRY void plot_node_value(char *name, double time, double voltage);
```

Description:

The function init_plots() initializes the plotting structures with the information gathered from the parser. It sets up the necessary data for handling node plotting during simulation. The function clear_plots() clear this information returning the occupied memory to the system. The function close_plot() wraps up a plot in progression by causing all the buffered data to be output to the plot file. The function update_plot() causes a sample of the node voltages to be taken and saved in the plotting buffer. DCplot() is called at the end of DC simulation and outputs the DC voltages of the circuit nodes onto a file. Finally show_plot() causes the invocation of a waveform previewer to read in the simulation results stored in the plot file and plot them in the screen.

Diagnostics:

Presently the SIMGRAPH waveform previewer only works under the X window system. See the *SIMLAB User's Guide*[4] for more information about SIMLAB's plotting capabilities.

9.4.8 poly_extrapolate(), linear_extrapolate()

Purpose:

To obtain a guess solution to start the nonlinear solution method.

Synopsis:

```
#include "simlab.h"

ENTRY void
poly_extrapolate(CIRCUIT *circuit, SOLUTION *solution, double h);
ENTRY void
linear_extrapolate(CIRCUIT *circuit, SOLUTION *solution, double h);
```

Description:

The function poly_extrapolate() fits a second-order polynomial to the current data-point and the two previous data-points and then extrapolates the solution forward for the next time point. The function linear_extrapolate() fits a first-order polynomial to the current and the previous data-point and then extrapolates the solution forward for the next time point. In both cases, the next time-point is specified by the present time plus the new timestep h.

Diagnostics:

poly_extrapolate() should be used only for second-order integration formulas. Similarly linear_extrapolate() should be used only for first-order integration formulas. If one is to use formulas of order greater than two, one should also specify a correspondingly matching extrapolation routine.

9.4.9 tran_Iconverged(), tran_Vconverged(), dc_Iconverged(), dc_Vconverged()

Purpose:

Synopsis:

```
#include "simlab.h"

dc_pt_solve(USETYPE use, MODETYPE mode, CIRCUIT *circuit,
            RUNTYPE type);
LOCAL BOOL
tran_Iconverged(CIRCUIT *circuit, SOLUTION *solution, double alpha);
LOCAL BOOL
tran_Vconverged(CIRCUIT *circuit, SOLUTION *solution);
LOCAL BOOL
dc_Iconverged(CIRCUIT *circuit, SOLUTION *solution);
LOCAL BOOL
dc_Vconverged(CIRCUIT *circuit, SOLUTION *solution);
```

Description:

`tran_Iconverged()` `tran_Vconverged()` `dc_Iconverged()` `dc_Vconverged()` The functions `tran_Iconverged()`, `tran_Vconverged()`, `dc_Iconverged()` and `dc_Vconverged()` are used within the nonlinear solver to check for convergence. Checking for convergence requires requires checking for convergence of the solution and the residual, i.e., voltage and current. `tran_Iconverged()` and `dc_Iconverged()` check for convergence of the residual, i.e. the current, while `tran_Vconverged()` and `dc_Vconverged()` check for convergence of the solution, that is the voltage. In order to check for convergence both an absolute and a relative convergence criteria are required. The global variables `NRVrel`, `NRVabs` `NRIrel` and `NRIabs` specify respectively the voltage and current relative and absolute convergence criteria. These variables can be set by the user from the front-end or from within the circuit specification file. See Section 9.3 for additional information.

Diagnostics:

`dc_Iconverged()` and `dc_Vconverged()` should be used only in DC simulation, whereas `tran_Iconverged()` and `tran_Vconverged()` should be used only during transient simulation.

9.5 SIMLAB'S Sparse Matrix Package

9.5.1 The SMMATRIX Structure

Purpose:

Structure to represent a sparse matrix.

Definition:

```
struct smmatrix_struct {
  int flags;
  int size, nextpivot;
  int *markprod, *numinrow, *numincol, numelements;
  SMELEMENT **firstinrow, **firstincol, **ninc, **diagonal;
  MEMLINK *alloclist;
  int allocindex, allocmax;
  double *allocptr;
  GENERIC_PTR graph;
};
```

Description:

The `flags` field is used to hold various bit-field flags about the matrix element. See sections 9.5.12 and 9.5.13. The `size` field contains the matrix size. This is usually equal to the number of free nodes in the circuit (`circuit->numfree` in DC simulation and `circuit->numnodes` in transient simulation). The `nextpivot` field contains the index of the next diagonal element to use as pivot during matrix factorization. The `markprod` field contains a pointer to an array of integers whose values are the Markowitz products for each diagonal pivot. The `numinrow` field contains a pointer to an array of integers whose values are the number of elements in the respective row. The `numincol` field contains a pointer to an array of integers whose values are the number of elements in the respective column. The `numelements` field contains the total number of matrix entries. The `firstinrow` field points to an array of matrix element pointers, each of which points to the first matrix element in the same row as the present element. The `ninc` field points to an array of matrix element pointers, each of which points to the next matrix element in the same column as the present element. The `diagonal` field points to an array of matrix element pointers, each of which points to diagonal matrix element in the respective row. The `alloclist` field contains a pointer to a list of allocated matrix elements. This is used in the memory management routines for the the sparse matrix package. The `allocindex`, `allocmax` and `allocptr` fields are used in the memory management routines for the sparse matrix package. The `graph` field contains a pointer to a graph interpretation of the sparse matrix which is used in some algorithms.

9.5.2 The SMELEMENT Structure

Purpose:

Structure to represent a sparse matrix element.

Definition:

```
struct smelement_struct {
  int row, col, rerow, recol, cost;
  double matval;
  SMELEMENT *ninr;
  SMELEMENT *ninc;
  double initcondpart;
  double initcappart;
  double condpart;
  double cappart;
  double savematval;
  GENERIC_PTR scratch;
  int fillin;
};
```

Description:

The `row` field contains the original row to which the element belongs. The `col` field contains the original column to which the element belongs. The `rerow` field contains the row to which the element belongs after some reordering is performed. The `recol` field contains the row to which the element belongs after some reordering is performed. The `cost` field contains an estimate of the cost The `ninr` field contains a pointer to the next matrix element in the present row. The `ninc` field contains a pointer to the next matrix element in the present column. The `initcondpart` field contains the initial conductance part of the matrix element as given by the linear conductances in the circuit. The `initcappart` field contains the initial capacitive part of the matrix element as given by the linear capacitors in the circuit. The `condpart` field contains the conductance part of the matrix element. The `cappart` field contains the initial capacitance part of the matrix element. The `matval` field is used for the merged values of the entry, i.e., the conductance part plus the scaled capacitance part. The `fillin` field contains a count of the number of fillins generated by the present matrix element

9.5.3 sminit()

Purpose:

Create a sparse matrix.

Synopsis:

```
#include "sparse.h"

ENTRY SMMATRIX *sminit(int size);
```

Description:

`sminit()` creates a `SMMATRIX` structure for a `size` by `size` matrix. When first initialized, the matrix only has diagonal elements. The matrix is freed with `smfree()`.

9.5.4 smfree()

Purpose:

Free a sparse matrix.

Synopsis:

```
#include "sparse.h"

ENTRY void smfree(SMMATRIX *sm);
```

Description:

`smfree()` frees the memory associated with the `SMMATRIX` structure pointed to by `sm`, including all entries added to the matrix after it was created by `sminit()`.

9.5.5 smstuffmat(), smgetentry()

Purpose:

Add a value to sparse matrix.

Synopsis:

```
#include "sparse.h"

ENTRY double *
smstuffmat(SMMATRIX *sm, int row, int col, double val, ENTRYTYPE part);
ENTRY double *
smgetentry(SMMATRIX *, int row, int col, ENTRYTYPE part);
```

Description:

`smstuffmat()` adds the value in `val` to the entry specified by `row` and `col` in sparse matrix `sm`. The part of the `SMELEMENT` that the value is added to is specified by `part`. If the entry does not exist, it is created. The address of the `double` field in the specified `SMELEMENT` is returned (i.e., the `initcappart`, the `initcondpart`, the `cappart`, the `condpart`, or the `matval` field address).

`smgetentry()` is similar to `smstuffmat()`, except that no value is given to be added to the specified entry.

9.5.6 `smludecomp()`

Purpose:

Perform LU decomposition.

Synopsis:

```
#include "sparse.h"

ENTRY BOOL smludecomp(SMMATRIX *sm);
```

Description:

`smludecomp()` performs the LU decomposition of sparse matrix `sm`. The matrix is symbolically decomposed (with `sminitialdecomp()`) if it has not already been.

9.5.7 `smsolve()`

Purpose:

Solve linear system.

Synopsis:

```
#include "sparse.h"

ENTRY BOOL smsolve(SMMATRIX *sm, double *rhs);
```

Description:

smsolve solves the linear system $Ax = b$, where the matrix A is specified by the sparse matrix sm and the vector b is specified by rhs. The answer is returned in rhs.

9.5.8 smmerge()

Purpose:

Synopsis:

```
#include "sparse.h"

ENTRY void smmerge(SMMATRIX *sm, double alpha0);
```

Description:

9.5.9 smmatvect(), smtmatvect()

Purpose:

Matrix-Vector multiply.

Synopsis:

```
#include "sparse.h"

ENTRY void smmatvect(SMMATRIX *sm, double *src, double *dest);
ENTRY void smtmatvect(SMMATRIX *sm, double *src, double *dest);
```

Description:

smmatvect() forms the matrix-vector product of the sparse matrix sm and the vector src, putting the result in dest. smtmatvect() forms the matrix-vector product of the transpose of sparse matrix sm and the vector src, putting the result in dest.

9.5.10 `sminitialdecomp()`

Purpose:

Perform symbolic LU decomposition.

Synopsis:

```
#include "sparse.h"

ENTRY void sminitialdecomp(SMMATRIX *sm, BOOL dofill);
```

Description:

`sminitialdecomp()` performs the symbolic LU decomposition of the matrix `sm`, using Markowitz ordering for minimum fill-in. The `dofill` argument specifies whether fill-in should be created.

9.5.11 `sm_get_element()`, `sm_find_element()`

Purpose:

Access sparse matrix elements.

Synopsis:

```
#include "sparse.h"

ENTRY SMELEMENT *sm_get_element(SMMATRIX *sm, int row, int col);
ENTRY SMELEMENT *sm_find_element(SMMATRIX *sm, int row, int col);
```

Description:

`sm_get_element()` returns the address of the `SMELEMENT` in sparse matrix `sm` specified by `row` and `col`. If the element is not present in the sparse matrix structure, it is created. `sm_find_element()` is similar to `sm_get_element()`, except that it will return `NULL` for an entry that is not present.

9.5.12 `is_initial_decomposed()`, `mark_initial_decomposed()`, `clear_initial_decomposed()`

Purpose:

Checks and clear sparse matrix decomposition information.

Synopsis:

```
#include "sparse.h"

is_initial_decomposed();
mark_initial_decomposed();
clear_initial_decomposed();
```

Description:

`is_initial_decomposed()` returns true if the sparse matrix has been previously symbolically decomposed. `mark_initial_decomposed()` marks the sparse matrix has having been symbolically decomposed. `clear_initial_decomposed()`, clears the indication that the sparse matrix was previously symbolically decomposed. For further information see for example [1].

Diagnostics:

These are actually macros handling the `flags` field in the sparse matrix.

9.5.13 `has_new_values()` `mark_new_values()` `clear_new_values()`

Purpose:

Synopsis:

```
#include "sparse.h"

has_new_values();
mark_new_values();
clear_new_values();
```

Description:

`has_new_values()` returns true if the the sparse matrix was recomputed after the previous LU decomposition. `mark_new_values()` marks the sparse matrix as having been recomputed after the last LU decomposition. `clear_new_values()` clears the indication that the sparse matrix was previously recomputed after the last LU decomposition.

Diagnostics:

These are actually macros handling the `flags` field in the sparse matrix.

9.6 Utility Functions

9.6.1 alloc_nstruct(), alloc_ntype(), alloc_struct(), alloc_type()

Purpose:

Handy memory allocation macros.

Synopsis:

```
#include "simlab.h"

alloc_nstruct();
alloc_ntype();
alloc_struct();
alloc_type();
```

Description:

The macros are defined as:

```
#define alloc_nstruct(s,x,n) \
  do { \
    if ((x = (s *) calloc((unsigned)(n),sizeof(s))) == NULL) { \
    (void)fprintf(stderr,"Mem error %s line %d\n",__FILE__, __LINE__); \
    exit(1); \
  } \
} while(0)

#define alloc_struct(s,x)     alloc_nstruct(s,x,1)
#define alloc_type(type,x)    alloc_nstruct(type,x,1)
#define alloc_ntype(type,x,n) alloc_nstruct(type,x,n)
```

The call

```
alloc_ntype(double, d_array, N)
```

will allocate an array of N doubles, and point d_array to the beginning of it.

Diagnostics:

These may be somewhat slow, since they use calloc and not malloc.

9.6.2 `bug()`, `error()`, `fatal()`, `warning()`

Purpose:

SIMLAB error trapping functions.

Synopsis:

```
#include "simlab.h"

ENTRY void warning(...);
ENTRY void error(...);
ENTRY void fatal(...);
ENTRY void bug(...);
```

Description:

All of these functions are called with a format string and a variable length list of arguments. All of these functions print the message to the standard output (and to the diary file, if active), but have different behavior after that.

- `warning()` issues its message and returns to the calling function.
- `error()` issues its message and does a `longjmp` to the location of the last call to `init_error`. See the description of `init_error` in this section.
- `fatal()` issues its message and exits SIMLAB with an error code of -1.
- `bug()` issues its message and dumps core.

Diagnostics:

Make sure you understand how `init_error()` works.

9.6.3 `begin_context()`, `end_context()`, `init_error()`

Purpose:

SIMLAB error environment functions.

Synopsis:

```
#include "simlab.h"

ENTRY void begin_context(void);
ENTRY void end_context(void);
init_error();
```

Description:

init_error() sets up a location for error to longjmp to when called. init_error is actually a macro, defined as

```
extern jmp_buf *err_env;
#define init_error() setjmp(*err_env)
```

where err_env is a global variable maintained by SIMLAB. Calls to begin_context will push a new error environment onto an internal environment stack. Calls to end_context will pop a new error environment off the internal environment stack. When error is called, it will longjmp to the last place where init_error was called that has not been closed off by an end_context.

These functions should be used within a function in the following way:

```
begin_context();
if (init_error()) {

  /* Code to handle a call to error goes here */
  end_context();
  return;
}
/* Code for the normal operation of the function goes here */
end_context();
return;
```

Diagnostics:

You will probably only need to worry about using these functions if you write a new simulation type. Calling error from lower level routines is usually OK, since SIMLAB has an error environment set up in its default simulation function pt_solve. Furthermore, main has an init_error call.

Issuing a ctrl-c to the front-end will cause a longjmp to the location of the environment on the top of the stack.

Make sure to properly bracket code with begin_context and end_context. If a begin_context is called within a function, one (and only one) end_context must be called before the function returns. See the man page on longjmp and setjmp.

9.6.4 gimme_chunk(), chunk_recycle()

Purpose:

SIMLAB free-listing memory management functions.

Synopsis:

```
#include "simlab.h"

ENTRY GENERIC_PTR
gimme_chunk(unsigned size, GENERIC_PTR *freelist_ptr);
ENTRY void
chunk_recycle(GENERIC_PTR cp, GENERIC_PTR *freelist_ptr);
```

Description:

`gimme_chunk` returns a piece of memory of size `size`. The function works by first checking if there is memory on `freelist_ptr` that can be pulled off. If so, this piece is pulled of and returned. If not, multiple pieces of memory are obtained from the system and all but one are put on the `freelist_ptr`. The one not put on the `freelist_ptr` is returned. `freelist_ptr` must be a static `GENERIC_PTR` since its value is maintained and mutated by the calls to `gimme_chunk` and `chunk_recycle`.

`chunk_recycle` puts the memory pointed to by `cp` onto the `freelist_ptr`.

Example:

The following two functions (from res.c) use `gimme_chunk` and `chunk_recycle` to allocate and free `RESISTOR` structures:

```
LOCAL void
free_resistor(rptr)
  RESISTOR *rptr;
{
  chunk_recycle((GENERIC_PTR) rptr, (GENERIC_PTR *) &free_resistors);
}

LOCAL RESISTOR * alloc_resistor()
{
  return ((RESISTOR *) gimme_chunk(sizeof(RESISTOR),
                                   (GENERIC_PTR *) &free_resistors));
}
```

9.6.5 sl_message(), sl_printf()

Purpose:

SIMLAB printing functions.

Synopsis:

```
#include "simlab.h"

ENTRY void sl_message(...);
ENTRY void sl_printf(...);
```

Description:

`sl_message` prints a formatted string, with arguments, to the standard output. `sl_printf` prints a formatted string, with arguments, to the standard output and to a diary file, if a diary file is active.

9.6.6 add_fcn()

Purpose:

Add functions to SIMLAB's database for a given `CLOSURE`.

Synopsis:

```
#include "simlab.h"

ENTRY void add_fcn(MODETYPE mode, BOOL visible,
                   int (*fcn_ptr)(), char *fcn_name, ...);
```

Description:

`add_fcn()` adds new functions to simlab's database of functions for a given `CLOSURE`. The `mode` argument specifies for which simulation mode the function is to be added. The `visible` argument specifies whether the closure is to be visible to the user from the front end. The `fcn_ptr` argument is a pointer to the function to be installed. The `fcn_name` argument specifies the name to be used from the front-end to select the new function. If `visible` is `FALSE`, then `fcn_name` is ignored and should be set to `NULL`.

Following the first four arguments is an argument list which specifies the dependency path for the new function, i.e., the names and addresses of the `CLOSURE` pointers, as well as the names of the particular installed functions, to which the new function is inferior. The argument list comes in triples, each triple consists of the name of the superior `closure`, the address of the superior `CLOSURE_PTR`, and the name of the specific function associated with that `CLOSURE` with which the new function is to work. The final triple should represent the closure that the new function is to be added to — in that case, the name of the representative function must be set to NULL.

Example:

The function `add_solve_linear_fcn()` is defined as:

```
                        add_solve_linear_fcn()

ENTRY void
add_solve_linear_fcn(modetype, environment_name,
                     simulate_name,
                     solve_nonlinear_name,
                     solve_linear_name,
                     fcn_ptr)
  MODETYPE modetype;
  char *environment_name;
  char *simulate_name;
  char *solve_nonlinear_name;
  char *solve_linear_name;
  BOOL(*fcn_ptr) ();
{
  add_fcn(modetype, TRUE, fcn_ptr, solve_linear_name,
         "environment", &_environment, environment_name,
         "simulate", &_simulate, simulate_name,
         "solve_nonlinear", &_solve_nonlinear, solve_nonlinear_name,
         "solve_linear", &_solve_linear, NULL);
}
```

Here, the `solve_linear` closure is specified as being inferior to `environment`, `simulate`, and `solve_nonlinear`. The particular installed function will be inferior to the functions installed previously for these closures with the names specified by `environment_name`, `simulate_name`, and `solve_nonlinear_name`.

For instance, calling `add_solve_linear_fcn()` as:

```
add_solve_linear_fcn(DCMode, "serial", "pt", "newton",
                     "direct", direct_smsolve);
```

will install `direct_smsolve()` as a linear solver that will work with the `serial` environment, `pt` simulate, and `newton` nonlinear solver in DC mode. The function `direct_sm-`

solve() is specified with name direct.

Diagnostics:

Make sure that the path specified in the argument list is a valid one. The members in the path are created on the fly, so no complaints will be issued if the path is not valid. Also, make sure that all superior functions specified in the path are installed.

9.6.7 do_ew_set()

Purpose:

Set closure default values.

Synopsis:

```
#include "interp.h"

ENTRY PN *
do_ew_set(char *closure_name, char *mode_name,
          char *value_name, BOOL def);
```

Description:

do_ew_set() is used to set the value of simulation closures. The function mimics the behavior of the front-end set command. The arguments should be given just as to the set command. closure_name specifies the name of a previously installed simulation closure, whose value we would like to set. mode_name specifies the name of the mode for which we would like to set this closure — "dc" for DC mode, trans for transient mode. If mode_name is set to NULL, the closure will be set for all modes. value_name specifies the name of the particular function (previously installed) to which we would like to set the closure. def specifies whether the call to do_ew_set() is setting a default value.

Diagnostics:

Be careful that the call to do_ew_set is a valid set command.

Author's note: At some point, this part of the interface will hopefully be made simpler — especially the setting of default values for closures.

9.6.8 Debugging Routines

Purpose:

Synopsis:

```
#include "simlab.h"

ENTRY void display_circuit(CIRCUIT *circuit);
ENTRY void disjacobian(GENERIC_PTR sm);
ENTRY void display_save_jacobian(GENERIC_PTR sm);
ENTRY void disvect(double *vector, int length);
ENTRY void simdebug_jacobian_and_rhs(SOLUTION *solution);
ENTRY void simdebug_jacobian(SOLUTION *solution);
ENTRY void simdebug_save_jacobian(GENERIC_PTR vptr);
ENTRY void simdebug_rhs(SOLUTION *solution);
```

Description:

`display_circuit()` prints out information about the circuit, including information about every device in the circuit (by calling the display function for each device type). `disjacobian()` prints out information about the sparse matrix pointed to by `sm`, i.e., displays the `matval` field of the sparse matrix entries. `display_save_jacobian()` prints out information about the "saved" values of the sparse matrix pointed to by `sm`, i.e., displays the `savematval` field of the sparse matrix entries. `disvect()` prints out the first `length` entries of the vector pointed to by `vector`. `simdebug_jacobian_and_rhs()` checks the `Simdebug` flag, and if it is set, calls `disjacobian()` and `disvect()` on the sparse matrix and rhs associated with `solution`, respectively. `simdebug_jacobian()` `simdebug_jacobian_and_rhs()` checks the `Simdebug` flag, and if it is set, calls `disjacobian()` on the sparse matrix associated with `solution`. `simdebug_save_jacobian()` checks the `Simdebug` flag, and if it is set, calls `display_save_jacobian()` on the sparse matrix associated with `solution`. `simdebug_rhs()` checks the `Simdebug` flag, and if it is set, `disvect()` on the rhs associated with `solution`.

References

[1] L. O. Chua and P.-M. Lin, *Computer-Aided Analysis of Electronic Circuits*. Englewood Cliffs, New Jersey: Prentice Hall, 1975.

[2] J. Vlach and K. Singhal, *Computer Methods for Circuit Analysis and Design*. Berkshire, England: Van Nostrand Reinhold, 1983.

[3] J. K. White and A. Sangiovanni-Vincentelli, *Relaxation Techniques for the Simulation of VLSI Circuits*. Engineering and Computer Science Series, Norwell, Massachusetts: Kluwer Academic Publishers, 1986.

[4] M. Silveira, A. Lumsdaine, and J. White, "SIMLAB users' guide." Research Laboratory of Electronics, Massachusetts Institute of Technology, 1990.

[5] B. W. Kernighan and D. M. Ritchie, *The C Programming Language*. Englewood Cliffs, NJ: Prentice-Hall, second ed., 1986.

[6] A. Lumsdaine, M. Silveira, and J. White, "CMVSIM programmer's guide." Research Laboratory of Electronics, Massachusetts Institute of Technology. Unpublished, 1990.

[7] A. Lumsdaine, M. Silveira, and J. White, "CMVSIM users' guide." Research Laboratory of Electronics, Massachusetts Institute of Technology, 1990.

A Selected Code Listings

A.1 xd.h

xd.h

```
/*
File Name:  xd.h
            (c) 1991 MIT RLE
Author:     A. Lumsdaine
            M. Silveira
            J. White
Revision:   10 Jan 91

NAME

SYNOPSIS

DESCRIPTION

DIAGNOSTICS

*/


typedef struct xdmod_struct XDMOD;
typedef struct xd_struct XD;


struct xdmod_struct {
  double area, is, cp, vmax;
  XDMOD *next;
};


struct xd_struct {
  XDMOD *model;
  int anode, cnode;
  NODE *nodei, *nodej;
```

xd.h (Continued)

```
  double
    isat,                              /* to limit current        */
    vmax,                              /* to limit current        */
    *va, *vc,                          /* voltage vector pointers */
    *cmataa, *cmatcc, *cmatca, *cmatac, /* Cap mat pointers        */
    *rmataa, *rmatcc, *rmatca, *rmatac, /* Resistive mat ptrs      */
    *acharge, *ccharge,                /* Charge vector entries   */
    *acurrent, *ccurrent,              /* Current vector entries  */
    *abs_acharge, *abs_ccharge,        /* Charge vector entries   */
    *abs_acurrent, *abs_ccurrent;      /* Current vector entries  */
  XD *next;
};


extern DEVICE_ENTRY xd_entry;
```

A.2 xd.c

xd.c

```
/*
File Name:  xd.c
            (c) 1991 MIT RLE
Author:     A. Lumsdaine
            M. Silveira
            J. White
Revision:   10 Jan 91

NAME

SYNOPSIS

DESCRIPTION

DIAGNOSTICS

*/


#include "simlab.h"
#include "xd.h"


#define MIMAX (10.0)
#define ZREV -5.0
#define ZCOND 0.00673794699908546650
#define ZVT -0.129 /* ZREV (for xds, times vtherm).  */


LOCAL void make_xd
ARGS((int netlist[], int numparams,
      IP_PARAM paramlist[], GENERIC_PTR modelptr));
LOCAL GENERIC_PTR make_xd_model ARGS((int numparams, IP_PARAM *paramlist));
```

xd.c (Continued)

```
LOCAL int add_xd ARGS((GENERIC_PTR list));
LOCAL void init_xd ARGS((GENERIC_PTR list));
LOCAL void eval_xd ARGS((GENERIC_PTR list));
LOCAL void eval_xd_rhs ARGS((GENERIC_PTR list));
LOCAL void display_xd ARGS((GENERIC_PTR list));
LOCAL void free_xd_list ARGS((GENERIC_PTR list));
LOCAL void free_xdmod_list ARGS((GENERIC_PTR list));
LOCAL void define_xd ARGS(());
LOCAL GENERIC_PTR dup_xd ARGS((GENERIC_PTR list, int numdevs));
LOCAL XD *alloc_xd ARGS(());
LOCAL void free_xd ARGS((XD * rptr));
LOCAL XDMOD *alloc_xdmod ARGS(());
LOCAL void free_xdmod ARGS((XDMOD * rptr));


LOCAL ip_param xdplist[] =
{
  {1, "area", NULL},
  {2, "is", NULL},
  {3, "vmax", NULL},
  {4, "cp", NULL}
};


LOCAL ip_param xdilist[] =
{
  {1, "isat", NULL},
  {2, "vmax", NULL}
};


LOCAL XD *current_xd_list = NULL;
LOCAL XDMOD *current_xdmod_list = NULL;
```

```
GLOBAL DEVICE_ENTRY xd_entry =
{
  0, 0,
  "XD",                               /* char *name             */
  FALSE,                              /* BOOL is_source;        */
  (GENERIC_PTR *) &current_xd_list,   /* GENERIC_PTR *parse_list;*/
  (GENERIC_PTR *) &current_xdmod_list,/* GENERIC_PTR *model_list;*/
  add_xd,                             /* int (*add)();          */
  init_xd,                            /* void (*init)();        */
  eval_xd,                            /* void (*eval)();        */
  eval_xd_rhs,                        /* void (*rhseval)();     */
  display_xd,                         /* void (*display)();     */
  free_xd_list,                       /* void (*devfree)();     */
  free_xdmod_list,                    /* void (*devfree)();     */
  dup_xd,                             /* GENERIC_PTR (*devdup)();*/
  define_xd                           /* void (*devdefine)();   */
};


LOCAL PRIM xd_prim =
{
  0,
  NULL, /* GENERIC_PTR model; */
  NULL, /* GENERIC_PTR (*readdev)(); */
  make_xd_model, /* GENERIC_PTR (*readmod)(); */
};


LOCAL void
define_xd()
{
```

xd.c (Continued)

```
  xd_prim.type = xd_entry.type;
  (void) ip_define_primitive("d", (char *) &xd_prim, 2,
  sizeof(xdplist) / sizeof(IP_PARAM), xdplist,
  sizeof(xdilist) / sizeof(IP_PARAM), xdilist);
}


LOCAL void
make_xd(netlist, numparams, paramlist, gptr)
  int netlist[], numparams;
  IP_PARAM paramlist[];
  GENERIC_PTR gptr;
{
  XD *dptr;
  XDMOD *modelptr = (XDMOD *) gptr;
  int i;

  if (netlist[0] == netlist[1])
    return;

  if (gptr == NULL)
    error("Xd instance for which no model exists!");

  /* Allocate the xd.  */
  dptr = alloc_xd();

  /* Get default max current from the model.  */
  dptr->isat = modelptr->area * modelptr->is;
  dptr->vmax = modelptr->vmax;

  /* Get the xd node numbers.  */
  dptr->anode = netlist[0];
  dptr->cnode = netlist[1];
```

xd.c (Continued)

```
  /* Get the instance area parameter, if given.  */
  for (i = 0; i < numparams; i++) {
    switch (paramlist[i].id) {
    case 1:
      dptr->isat = pareval(paramlist[i]);
      break;
    case 2:
      dptr->vmax = pareval(paramlist[i]);
      break;
    }
  }

  /* Copy a pointer to the xd model into the xd instance.  */
  dptr->model = modelptr;

  dptr->next = current_xd_list;
  current_xd_list = dptr;
}


LOCAL GENERIC_PTR
make_xd_model(numparams, paramlist)
  int numparams;
  ip_param *paramlist;
{
  PRIM *modptr;
  XDMOD *dptr;
  int i;

  /* Allocate space for the model.  */
  dptr = alloc_xdmod();
  dptr->next = current_xdmod_list;
```

xd.c (Continued)

```
  current_xdmod_list = dptr;

  modptr = alloc_prim();
  modptr->model = (GENERIC_PTR) dptr;
  modptr->readdev = make_xd;
  modptr->readmod = make_xd_model;

  /* Place in the defaults.  */
  dptr->area = 1.0;
  dptr->is = 0.0;
  dptr->cp = 0.0;
  dptr->vmax = 10;
  /* Read user-specified parameters.  */
  for (i = 0; i < numparams; i++) {
    switch (paramlist[i].id) {
    case 1:
      dptr->area = pareval(paramlist[i]);
      break;
    case 2:
      dptr->is = pareval(paramlist[i]);
      break;
    case 3:
      dptr->vmax = pareval(paramlist[i]);
      break;
    case 4:
      dptr->cp = pareval(paramlist[i]);
      break;
    }
  }
  modptr->type = xd_entry.type;

  return ((GENERIC_PTR) modptr);
}
```

xd.c (Continued)

```
LOCAL int
add_xd(vptr)
  GENERIC_PTR vptr;
{
  register XD *dptr, *fptr;
  int numdevs = 0;

  for (dptr = (XD *) vptr; dptr; dptr = dptr->next) {
    dptr->nodei = get_node(dptr->anode);
    dptr->nodej = get_node(dptr->cnode);
    if (dptr->nodei->type == Unknown)
      dptr->nodei->type = Internal;
    if (dptr->nodej->type == Unknown)
      dptr->nodej->type = Internal;
  }

  for (dptr = (XD *) vptr; dptr; dptr = dptr->next)
    numdevs++;

  return numdevs;
}


LOCAL void
init_xd(vptr)
  GENERIC_PTR vptr;
{
  XD *dptr;
  NODE *a, *c;

  for (dptr = (XD *) vptr; dptr; dptr = dptr->next) {
```

xd.c (Continued)

```
    a = dptr->nodei;
    c = dptr->nodej;

    dptr->va = get_v_entry(a);
    dptr->vc = get_v_entry(c);

    dptr->acurrent = get_rhs_entry(a, Nonlincond);
    dptr->ccurrent = get_rhs_entry(c, Nonlincond);
    dptr->abs_acurrent = get_abs_rhs_entry(a, Nonlincond);
    dptr->abs_ccurrent = get_abs_rhs_entry(c, Nonlincond);

    dptr->acharge = get_rhs_entry(a, Nonlincap);
    dptr->ccharge = get_rhs_entry(c, Nonlincap);
    dptr->abs_acharge = get_abs_rhs_entry(a, Nonlincap);
    dptr->abs_ccharge = get_abs_rhs_entry(c, Nonlincap);

    dptr->rmataa = get_Jacobian_entry(a, a, Nonlincond);
    dptr->rmatcc = get_Jacobian_entry(c, c, Nonlincond);
    dptr->rmatac = get_Jacobian_entry(a, c, Nonlincond);
    dptr->rmatca = get_Jacobian_entry(c, a, Nonlincond);

    dptr->cmataa = get_Jacobian_entry(a, a, Nonlincap);
    dptr->cmatcc = get_Jacobian_entry(c, c, Nonlincap);
    dptr->cmatac = get_Jacobian_entry(a, c, Nonlincap);
    dptr->cmatca = get_Jacobian_entry(c, a, Nonlincap);
  }
}


LOCAL void
eval_xd(vptr)
  GENERIC_PTR vptr;
{
```

xd.c (Continued)

```
register XD *dptr;
register XDMOD *mptr;
double vd, qd, id, gd, cd, temp1;
register BOOL a, c;

for (dptr = (XD *) vptr; dptr; dptr = dptr->next) {
  a = (dptr->acurrent != NULL);
  c = (dptr->ccurrent != NULL);

  mptr = dptr->model;
  vd = (*dptr->va) - (*dptr->vc);

  /* Compute dc current and derivatives */
  if (vd >= dptr->vmax)
    vd = dptr->vmax;

  temp1 = dptr->isat * exp(vd / VTHERM);
  id = temp1 - dptr->isat;
  gd = temp1 / VTHERM;

  /* Compute charge and capacitance.  */
  if (Dcflag == FALSE) {
    qd = mptr->cp * (vd + vd*vd*vd/3.0);
    cd = mptr->cp * (1 + vd*vd);

    /* Put xd currents, charges and conds into matrix and rhs */
    if (a) {
      *(dptr->rmataa) += gd;
      *(dptr->cmataa) += cd;
      *(dptr->acurrent) -= id;
      *(dptr->acharge) -= qd;
      *(dptr->abs_acurrent) += ABS(id);
      *(dptr->abs_acharge) += ABS(qd);
```

xd.c (Continued)

```
      if (c) {
        *(dptr->rmatcc) += gd;
        *(dptr->cmatcc) += cd;
        *(dptr->ccurrent) += id;
        *(dptr->ccharge) += qd;
        *(dptr->abs_ccurrent) += ABS(id);
        *(dptr->abs_ccharge) += ABS(qd);
        *(dptr->rmatac) -= gd;
        *(dptr->rmatca) -= gd;
        *(dptr->cmatac) -= cd;
        *(dptr->cmatca) -= cd;
      }
    } else if (c) {
      *(dptr->rmatcc) += gd;
      *(dptr->cmatcc) += cd;
      *(dptr->ccurrent) += id;
      *(dptr->ccharge) += qd;
      *(dptr->abs_ccurrent) += ABS(id);
      *(dptr->abs_ccharge) += ABS(qd);
    }
  } else {

    /* DC solution */
    if (a) {
      *(dptr->rmataa) += gd;
      *(dptr->acurrent) -= id;
      *(dptr->abs_acurrent) += ABS(id);
      if (c) {
        *(dptr->rmatcc) += gd;
        *(dptr->rmatac) -= gd;
        *(dptr->rmatca) -= gd;
        *(dptr->ccurrent) += id;
        *(dptr->abs_ccurrent) += ABS(id);
```

```
        }
      } else if (c) {
        *(dptr->rmatcc) += gd;
        *(dptr->ccurrent) += id;
        *(dptr->abs_ccurrent) += ABS(id);
      }
    }
  }
}


LOCAL void
eval_xd_rhs(vptr)
  GENERIC_PTR vptr;
{
  register XD *dptr;
  register XDMOD *mptr;
  double vd, id, qd, temp1, temp2;
  register BOOL a, c;

  for (dptr = (XD *) vptr; dptr; dptr = dptr->next) {
    a = (dptr->acurrent != NULL);
    c = (dptr->ccurrent != NULL);

    mptr = dptr->model;
    vd = (*dptr->va) - (*dptr->vc);

    /* Compute dc current and derivatives */
    if (vd >= dptr->vmax)
      vd = dptr->vmax;

    temp1 = dptr->isat * exp(vd / VTHERM);
    id = temp1 - dptr->isat;
```

xd.c (Continued)

```
    /* Compute charge.  */
    if (Dcflag == FALSE) {
      qd = mptr->cp * (vd + vd*vd*vd/3.0);

      /* Put the xd currents and charges in vector */
      if (a) {
        *(dptr->acurrent) -= id;
        *(dptr->acharge) -= qd;
        *(dptr->abs_acurrent) += ABS(id);
        *(dptr->abs_acharge) += ABS(qd);
      }
      if (c) {
        *(dptr->ccurrent) += id;
        *(dptr->ccharge) += qd;
        *(dptr->abs_ccurrent) += ABS(id);
        *(dptr->abs_ccharge) += ABS(qd);
      }
    } else {

      /* Load only current.  */
      if (a)
        *(dptr->acurrent) -= id;
      if (c)
        *(dptr->ccurrent) += id;
      if (a)
        *(dptr->abs_acurrent) += ABS(id);
      if (c)
        *(dptr->abs_ccurrent) += ABS(id);
    }
  }
}
```

xd.c (Continued)

```
LOCAL void
display_xd(vptr)
  GENERIC_PTR vptr;
{
  XD *dptr;
  int i;

  for (i = 0, dptr = (XD *) vptr; dptr; dptr = dptr->next, i++) {
    sl_printf("XD%d %d %d %e %e %e %e %e %e %ld %ld\n",
              i, dptr->anode, dptr->cnode, dptr->isat,
              dptr->vmax, dptr->model->area, dptr->model->is,
              dptr->model->vmax, dptr->model->cp,
              dptr->nodei, dptr->nodej);
  }
}


LOCAL GENERIC_PTR
dup_xd(xd_list, numdevs)
  GENERIC_PTR xd_list;
  int numdevs;
{
  register XD *dptr, *aux;
  register int i;

  alloc_ntype(XD, dptr, numdevs);
  for (aux = (XD *) xd_list, i = 0; aux; aux = aux->next, i++) {
    if (i > numdevs - 1) {
      error("Number of xds specified doesn't match list size");
      break;
    }
    dptr[i].model = aux->model;
```

xd.c (Continued)

```
    dptr[i].anode = aux->anode;
    dptr[i].cnode = aux->cnode;
    dptr[i].nodei = aux->nodei;
    dptr[i].nodej = aux->nodej;
    dptr[i].isat = aux->isat;
    dptr[i].vmax = aux->vmax;
    if (i < numdevs - 1)
      dptr[i].next = (XD *) &dptr[i + 1];
    else
      dptr[i].next = (XD *) NULL;
  }

  return ((GENERIC_PTR) dptr);
}


LOCAL XD *free_xds = (XD *) NULL;


LOCAL void
free_xd(rptr)
  XD *rptr;
{
  chunk_recycle((GENERIC_PTR) rptr, (GENERIC_PTR *) &free_xds);
}


LOCAL XD *
alloc_xd()
{
  return ((XD *) gimme_chunk(sizeof(XD),
  (GENERIC_PTR *) &free_xds));
}
```

```
LOCAL void
free_xd_list(vptr)
  GENERIC_PTR vptr;
{
  register XD *rptr, *fptr;

  for (rptr = (XD *) vptr; rptr; rptr = fptr) {
    fptr = rptr->next;
    free_xd(rptr);
  }
}


LOCAL XDMOD *free_xdmods = (XDMOD *) NULL;


LOCAL void
free_xdmod(rptr)
  XDMOD *rptr;
{
  chunk_recycle((GENERIC_PTR) rptr, (GENERIC_PTR *) &free_xdmods);
}


LOCAL XDMOD *
alloc_xdmod()
{
  return ((XDMOD *) gimme_chunk(sizeof(XDMOD),
  (GENERIC_PTR *) &free_xdmods));
}
```

xd.c (Continued)

```
LOCAL void
free_xdmod_list(vptr)
  GENERIC_PTR vptr;
{
  register XDMOD *rptr, *fptr;

  for (rptr = (XDMOD *) vptr; rptr; rptr = fptr) {
    fptr = rptr->next;
    free_xdmod(rptr);
  }
}
```