THE FRONT END TO SIMULATOR INTERFACE

by

Thomas L. Quarles

Memorandum No. UCB/ERL M89/43

24 April 1989

COVER PAGE

50

THE FRONT END TO SIMULATOR INTERFACE

by

Thomas L. Quarles

Memorandum No. UCB/ERL M89/43

24 April 1989

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

THE FRONT END TO SIMULATOR INTERFACE

by

Thomas L. Quarles

Memorandum No. UCB/ERL M89/43

24 April 1989

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

TITLE PAGE

Preface

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

Memo number	Title
UCB/ERL M89/44	The SPICE3 Implementation Guide
UCB/ERL M89/45	Adding Devices to SPICE3
UCB/ERL M89/46	SPICE3 Version 3C1 Users Guide
UCB/ERL M89/47	Benchmark Circuits: Results for SPICE3

This memo was originally Appendix A of the dissertation and contains a description of the interface between the Numeg front end and the SPICE3 circuit simulator. This version describes the interface as of the release of SPICE3, version 3C1.

Table of Contents

CHAPTER 1

The Front End to Simulator Interface

The remainder of this chapter is the detailed description of the interface from a front end to a circuit simulator. This interface was originally designed for SPICE3 as one of the major module separation points, but as the utility of the idea became apparent, the idea was expanded to encompass other simulators. Almost all of the development work for this interface was done in various versions of SPICE3, but the extensions of the interface to support many other simulators has been a joint effort with Ken Kundert, Tom Laidig, and Don Webber.

The changes described in the final section of this chapter refer to differences since a preliminary version of this interface (V1.0) was described in a manual accompanying the SPICE3B1 release, and are provided in an attempt to provide continuity in documentation from release to release. This chapter currently describes version 2.0.

1.1. Introduction

This paper is an attempt to bring all the circuit simulators being written together under one common interface so that work being done on the development of an input language, graphics output, data management, and user interface need not be duplicated. The assumption is made that the front end will be linked with the simulator and will then interact with it through a series of subroutine calls. In order to make it possible to simply relink the front end with a different simulator, we present here a complete set of definitions of routines that we believe are sufficient for all current simulators.

1.1.1. Terminology

For this to work, we make a few assumptions about the simulator and about terminology that makes everything much easier, and that all current simulators can readily agree on.

References to the front end in this document can either refer to the front end itself or to a separate back end which does data collection. Since the purpose of this document is to explain how these front and back end systems interact with the simulator, not how they interact with each other, we have chosen to refer to them collectively as the front end. When we refer to a **circuit** we mean a set of instances and their models which are all interconnected to form a single connected circuit that will be treated as one in a simulation. While some simulators may only support a single circuit, most simulators allow several circuits to be defined at a time and switch between them based on a parameter passed on all of the function calls. When we refer to a **device** we mean a category of circuit elements, such as 'resistor', 'capacitor', or 'bipolar transistor'. When we refer to a **model** we mean a named set of parameters selected by the programmer for a device which are likely to be common to several instances. When we refer to an **instance** we mean a specific instantiation of a model which has terminals connected to other instances making up a circuit. A **node** is a connection point between elements of the circuit, sometimes known in other systems as a net. A **signal** is a circuit variable which is naturally computed by the simulator and is output to the front end. All node voltages are signals, and additional signals such as currents may be created by the simulator. An **analysis** is the smallest component of a simulation which the simulator will perform. Generally, an analysis will result in a single output, and will be independent of other analyses, but an analysis may also modify and change the results of another analysis, such as a sensitivity analysis which causes additional computation and output to result during ac, dc, and transient analyses, or the 'options' analysis generally used to modify the simulation program's overall parameters. A **task** is a collection of analyses which should be performed together by the simulator in a single call from the front end. Finally, when we refer to a **plot** we mean a collection of data associated with single analysis which would be used to generate one or more actual plots.

1.2. System overview

The interface between the simulator and the front end is a fairly simple interface, using a relatively small set of routines for manipulating each type of object. The simulator provides a library of subroutines to the front end which supplies the main program to interact with the user and call the simulator as needed. The objects can generally be split into two categories, those the simulator maintains at the request of the front end, and those the front end maintains at the request of the simulator.

1.2.1. Simulator objects

The simulator and front end agree on the existence of six different types of simulator objects, a *circuit,* a *node,* a *model,* an *instance,* a *task,* and an *analysis.* A circuit is the top-level construct which a simulator manipulates, and thus contains all the other objects. Functions are provided to create and delete circuits. A circuit is made up of a set of interconnected nodes and instances. Nodes are second level objects, and are attached directly to the circuit. Functions are provided to create and delete nodes, set parameters on them and query their parameters. Additionally, since instances and nodes must be connected together, functions are provided to connect a terminal of an instance to a node and to locate a node within the circuit, either by name or by reference to the terminal of an instance it is connected to. Models are second level objects, and functions are provided to create and delete them, set and query their parameters, and to find them by name within the circuit. Instances are always instances of a specific model, and thus third level objects. Functions are provided to create and delete instances, to set and query their parameters, and to find them by name within the circuit. Finally, analyses and tasks have a relationship very similar to that between instances and models. Functions are provided to create and delete tasks, as well as find them by name within the circuit they were declared in. Analyses can be created, found within the circuit or task, and have parameters set and queried.

Although there are a large number of total functions, their operations are all quite similar, making them simple to use.

The only moderately complicated aspect of using these functions is the technique used to find the arguments to use in them. Some objects are relatively simple, circuits and tasks come in only one variety, and have no operations which apply directly to them. Nodes come in only one variety, but have attributes which can be modified by the program, thus their set and query functions. For each object which has set and query functions, there will be a description of the capabilities of that object in an array of IFparm structures. These capabilities will be in the form of keywords to identify the capabilities to the user, integer id's to be used when communicating to the simulator, longer descriptive messages, and a type field which signals the type of data needed or supplied and other properties of the transaction. Generally, the front end will search through such a table until the proper keyword match is found, then use the type information to properly format the parameter for the simulator and the id to identify the transaction to the simulator in a call to the proper function.

Analyses are more complicated yet, in that there are several types of analyses, thus an array of IFanalysis structures is provided to allow the front end to search through and identify the type of analysis it wants. Actual identification of the analysis is by the index, starting from zero, of the desired analysis in this array. Each analysis then provides an array of IFparm structures to describe the operations that can be performed on that specific type of analysis.

Devices are the most complicated object of all and provide an array of IFdevice device descriptors. The front end uses the zero based index of the proper device descriptor in this array to identify the type of device. Each of these IFdevice descriptors provides a set of IFparm arrays of parameters for both the models and instances of the device it describes.

While the set of functions provided is quite extensive, many of the functions are expected to be used frequently, and thus should be fast, while others are provided for completeness and to help a front end which wishes to trade off efficiency for simplicity or memory use. Specifically, the find functions will generally be slow since they must search through large data structures, while the create, delete, bind, set parameter, and query parameter functions should be simple to implement and can be expected to be quite fast.

1.2.2. Front end objects

The front end manipulates sets of data known as *plots* and parse trees for complicated expressions, as well as maintaining the symbol tables for the overall system.

1.2.2.1. Name space management

The front end is responsible for maintaining all names used in the entire system. Rather than manipulate names directly, the front end and simulator pass around tokens known as IFuid's, interface unique identifiers. The front end is able to map an IFuid into its corresponding name or other identification on demand, but as far as the simulator is concerned, this simple object which can be compared for equality using the standard == operator in C is all that it needs to know about. Whenever the simulator wishes to create an object with a new name, it must go through the front end to have an IFuid generated correctly. IFnewUid generates an IFuid for the simulator. It is assumed that the simulator will rarely need to generate names from scratch, but that they will usually be related to an IFuid provided by the front end from user data. For example, an internal node in a device should be identified by the device name plus some further distinguishing characters. Since the front end will keep any two devices from having the same name, this will prevent name conflicts in simulator generated names as long as, for example, a character which is illegal in user supplied names is added in the process. Thus, IFuid requires an existing IFuid as one argument along with an additional character string to be used to make the name unique. Note that IFuid's are especially useful to graphical interfaces where nodes may not have names which are meaningful to the user, but the system can easily translate some identifier to a piece of geometry to highlight.

1.2.2.2. Parse trees

The front end provides parse tree support to the simulator to allow for complicated expressions that are evaluated at run time. The front end will parse such an expression into a parse tree data structure, and on demand, through a function provided in that structure, evaluate the expression and its partial derivatives. This function, known generically as IFeval, does not necessarily exist by this name, but is simply the name of the function pointer in the parse tree structure. Making this a per-parse tree pointer allows for a system which compiles and loads code for parse tree evaluation.

1.2.2.3. Plots

Each analysis will generally produce a single plot as output. A plot may be as simple as a single vector of data at an operating point of the circuit, or a multi dimensional plot of one or more dependent variables as one or more independent variables changes through a range. Functions are provided to indicate the beginning and end of a plot, to provide data within the plot, to define an additional dimension on the plot, and to provide further information about the data being output, such as the units it is measured in or the type of scale it is appropriate to plot it against.

1.3. Error handling

All the functions used to interface between the front end and the simulator return an error code. Error codes are simple integers with specific meanings as defined in the header file *IFerrmsgs.h*. A return value of zero indicates no error, error codes from 1 to E_PRIVATE are generic codes that apply to many simulators, and codes above E_PRIVATE are simulator specific. Currently, there is expansion space reserved between the highest defined shared error code and E_PRIVATE, but simulator specific error codes should explicitly add the constant E_PRIVATE to each of their codes to allow for further expansion of the shared message list. In addition to the simple integer error codes used for errors, there are two additional sources of information about errors which can be used whenever the error code is non-zero. The global variable errRtn can be set to point to a constant character string and will be interpreted by the front end as the point in the analysis where the error was detected. The global variable errMsg is used to contain a descriptive message about the error. Since most messages will contain variable length descriptive information which will be assembled into a buffer using sprintf, the calling routine will free it after printing the message, thus the message pointed to by errMsg should be dynamically allocated with malloc. Both errRtn and errMsg should be defined in the front end and referenced as external variables by the simulator. This will ensure against multiple definitions or no definition. The name pointed to by errRtn should not be dynamically allocated since this will not be freed by the front end, thus allowing it to be used even in the case of a malloc error. After output, both errMsg and errRtn will be cleared to NULL by the front end to prevent incorrect future messages from a simulator that chooses not to set both in every case. In the interests of maximum flexibility, the simulator is allowed to set either, both, or neither of errMsg and errRtn, and the front end should do something sensible in all cases. It is strongly recommended that the simulator set both unless it is using a standard error code, in which case it should still set errRtn. In the case of returns from front end or back end routines called by the simulator, the return codes and conventions are the same, and the simulator may wish to simply pass the codes on up the calling hierarchy to the front end.

The following is a summary of the error codes currently defined in *IFerrmsgs.h*. It is expected that this list will grow, so any messages you add to it locally will probably need to be renumbered and added in again with each successive version of this header file from Berkeley. If you find the need for additional public error codes which should be produced by several simulation programs, please let us know so we can put them in this document and in the master copy of the header file.

1.3.1. OK

No error.

1.3.2. E_PAUSE

E_PAUSE is a special return code provided for the simulator to indicate to various levels of its own code as well as the front end that it is pausing as a result of a request from the front end, and not terminating normally or in error.

1.3.3. E_PANIC

This is a very vague message that is provided for errors that don't fit into any other category and are of the type that can never occur, but paranoid programming practice dictates that a test for them be made anyway.

1.3.4. E_EXISTS

An attempt has been made to create a model, instance, node, or analysis that already exists. This can be either an error or a warning, depending on the amount of state maintained by the front end. As a warning, it allows a front end to maintain a minimal amount of state about models, instances, and analyses, and still correctly handle cases where a user makes a second reference to one. If the front end is careful about state and never repeats a creation, then this indicates an error in that data. The creation routine will not create a new model, instance, or analysis, but will instead find and return the existing one.

Not all simulators provide this level of checking, and it is undefined what will happen in such a simulator if a duplicate of a device is created.

1.3.5. E_NODEV

An attempt has been made to reference a device which doesn't exist.

1.3.6. E_NOMOD

An attempt has been made to reference a model which doesn't exist.

1.3.7. E_NOANAL

An attempt has been made to reference an analysis which doesn't exist.

1.3.8. E_NOTERM

An attempt has been made to bind a circuit node to a terminal that isn't defined for the specified device.

1.3.9. E_BADPARM

A parameter specification is in error, because the specified parameter number is not valid in this circumstance.

1.3.10. E_NOMEM

The simulator has run out of memory. This is usually a fatal error, and the simulation in question can not be resumed. The exact state of the circuit involved is undefined, although it is guaranteed that the circuit can be freed with the *deleteCircuit* function, and if the circuit has reached the simulation stage, the simulation can be restarted without difficulty, although continuation is not possible.

1.3.11. E_NODECON

A node binding for a terminal of an instance has been specified for a terminal that is already bound. The simulator must bind the terminal to the new node, but this error is returned as a warning that a rebinding is occurring.

1.3.12. E_UNSUPP

The front end called a function which the simulator has not implemented. For full functionality, all the described functions should be implemented, but in some implementations some of the functions will be left out either because the simulator can't support them or because the interface is only partially implemented.

1.3.13. E_PARMVAL

The parameter specified is not in the legal range for this parameter. The specification has been rejected by the simulator.

1.3.14. E_NOTEMPTY

The device, model, analysis, or node specified to a deletion function is still referenced by something and cannot be deleted.

1.3.15. E_NOCHANGE

The simulator has reached a point where it can no longer accept additions to or changes in the structure of the circuit, and the operation requested by the user would have made such an addition or change. The requested operation is rejected.

1.3.16. E_NOTFOUND

The simulator has been asked to find something and has not been successful in locating it.

1.3.17. E_BAD_DOMAIN

The output package has detected some incompatibility between the pairing of OUTbeginDomain and OUTendDomain calls or the nesting level of the calls at the time OUTpData or OUTwData was called.

1.4. Data Structures

A variety of data structures are used to pass data back and forth between the simulator and the front end. These data structures are all defined in *IFsim.h*, which should be included by all routines using this interface.

1.4.1. IFparm structure

The IFparm data structure is a low level structure, arrays of which are used within other structures to describe the bulk of the data concerning the simulator's capabilities to the front end by describing the parameters available. The IFparm data structure consists of four fields used to describe the data involved.

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

The *keyword* entry is a one word character string which would be used on an input line to indicate the beginning of an entry of this type. To avoid problems with special characters which may be used by some parsers, the keyword should use a very restricted character set. We recommend restricting the keyword to letters, digits, and underscores, with the first character a letter. The *id* field is a simple integer which uniquely identifies this parameter within the set of parameters in this array of IFparm structures. This is a simple integer since integer comparison is much faster than string comparison, and it allows simple aliasing of keywords. The *dataType* field is an integer that holds one of the values defined in the IFsim.h header file. The legal values for *dataType* currently include the primitive types IF_FLAG, IF_INTEGER, IF_REAL, IF_COMPLEX, IF_NODE, IF_STRING, IF_INSTANCE, IF_PARSETREE, and vectors of these primitive types (indicated by or'ing IF_VECTOR with the proper type). This identifies the type of argument the keyword should be associated with in an input or the type of value to be returned in response to a query. The parser should use this to determine how to interpret the argument and to place it in the correct field of the union used to pass data to the simulator or extract the result from the returned union. The dataType field also may have a bit set indicating the parameter is required (IF_REQUIRED), which allows the front end to do some additional error checking by not asking the simulator to run an analysis if all parameters are not specified, but the simulator should not depend on this level of checking, it is simply provided as a level of optimization for front ends which wish to save some time processing bad inputs. There are also bits for indicating whether the description is of an input data item valid in a parameter call (IF_SET), is an output data item valid in a question call (IF_ASK), or is both. As a special case, if it is not indicated as an input or output data item, it is an input parameter which the simulator recognizes as being a normal parameter under the circumstance, but one which this simulator chooses not to support. Finally, the simulator can indicate that additional information is required to specify the parameter more precisely by setting the IF_SELECT or IF_VSELECT bit. If either of these bits are set, then the 'selector' parameter may contain an integer or vector of integers respectively to further specify the parameter. It is intended that these be used to provide access to individual elements in arrays of parameters without having to set or output the entire array. See setNodeParm and askNodeQuest for a more detailed description of this mechanism.

The *description* is a longer, more descriptive character string that describes the parameter. The description should be short enough to easily fit on a single line of a standard terminal, but sufficiently detailed to identify the function of the parameter to a user of the program. Note that the order in which parameters are passed to the simulator is not defined, and may be the order in which the user specified them, the order the simulator defined them in or completely random at the option of the front end.

1.4.2. IFuid datatype

The datatype IFuid is defined in the header file to facilitate the exchange of the unique identifiers used in place of names. This datatype is intended to be a pointer, thus the front end can have it point directly to the name in question, can have it point to some data structure, or can, in a machine dependent implementation, have it contain some other data such as an integer.

1.4.3. IFparseTree structure

This structure is designed to allow the front end to read and partially parse a complex expression which the simulator can later call to have evaluated. This structure contains all the information the simulator needs to determine the variables involved and how to have the tree evaluated.

```
typedef struct sIFparseTree {
    int numVars;                /* number of variables used */
    int *varTypes;              /* array of IFvalue types describing values */
    IFvalue *vars;              /* array of IFvalues describing values */
    int ((*IFeval)());          /* function to call to get evaluated */
} IFparseTree;
```

numvars is the number of variables used in the parse tree. *vars* is an array of the actual parameters used in the expression. Note that all of the variables must be names of signals, either those generated normally by the circuit or those in the list of special signals provided in the IFsimulator structure. *IFeval* is a function provided by the front end which will evaluate the equation represented by the parse tree structure and its partial derivatives. IFeval is described in greater detail in the chapter on front end functions.

1.4.4. IFvalue structure

The structure used to pass data values to and from the simulator is defined as:

```
typedef union uIFvalue{
    int iValue;             /* integer valued data */
    double rValue;          /* real valued data */
    complex cValue;         /* complex valued data */
    char *sValue;           /* string valued data */
    IFuid uValue;           /* instance valued data */
    IFnode nValue;          /* node valued data */
    IFparseTree *tValue;    /* parse tree */
    struct {
        int numValue;       /* length of vector */
        union {
            int *iVec;      /* pointer to integer vector */
            double *rVec;   /* pointer to real vector */
            complex *cVec   /* pointer to complex vector */
            char **sVec;    /* pointer to string vector */
            IFuid *uVec;    /* pointer to instance vector */
            IFnode *nVec;   /*pointer to node vector */
        }vec;
    }v;
} IFvalue;
```

All of the possible parameter types can be placed in this structure, integer values in the iValue field, reals in the rValue field, node pointers in the nValue field, and strings in the sValue field. Vectors are given as a length in v.numvalue field with an array of values in one of the v.vec.Xvec fields. Flags are passed in the iValue field, with the convention that any non-zero value corresponds to setting the flag, and zero corresponds to clearing it.

1.4.5. IFdevice structure

This structure is used to describe a device to be parsed, along with its parameters and the corresponding information, if any, about the model used for the device.

```
typedef struct sIFdevice {
    char *name;                 /* name of this type of device */
    char *description;          /* description of what the device is */

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

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

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

} IFdevice;
```

Name is the character string used to describe all devices of this type, such as 'resistor'. Again, to prevent problems with special characters in parsers, the name should be limited to letters, digits, and underscores, with the first character a letter. *Description* is a one-line description of the device. *terms* is the number of terminals allowed on the device. To allow for devices with a variable number of terminals, if *terms* is negative, then at least *-terms* terminals must be present, but any number larger than that is legal. *TermNames* is a pointer to an array of length *numNames* of character pointers, each of which points to a character string describing a node of the device. If terms is nonnegative, then it is equal to numNames, while if it is negative, its absolute value must be less than or equal to numNames. There are two arrays of IFparm structures for each device. The instanceParms array is of length numInstanceParms, and contains parameters and questions that apply to individual instances. The modelParms array is of length numModelParms, and contains parameters and questions that apply to a device model. The front end will use these structures to determine what operations are legal on each device and pass the appropriate IFparm id back to the simulator as necessary.

1.4.6. IFanalysis structure

```
typedef struct sIFanalysis {
    char *name;                  /* name of this analysis type */
    char *description;           /* description of this analysis type */

    int numParms;                /* number of analysis parameter descriptors */
    IFparm *analysisParms;       /* array  of analysis parameter descriptors */

} IFanalysis;
```

This structure is used to describe an analysis that can be performed by the simulator. *Name* and *description* are respectively one word and one line descriptions of the analysis being offered. Once again, name should be composed only of letters, digits, and underscores with a leading letter. The *analysisParms* array describes the parameters and queries that are applicable to this particular analysis.

1.4.7. IFsimulator structure This structure consists of a number of pointers to functions that the parser may use, as well as tables of data. The parser will obtain this simulator data by calling the simulator routine *XXXinit()*. This routine returns a pointer to a copy of the following IFsimulator structure.

```
typedef struct sIFsimulator {
    char *simulator;             /* the simulator's name */
    char *description;           /* description of the simulator */
    char *version;               /* version or revision of simulator */

    int ((*newCircuit)());       /* create new circuit */
    int ((*deleteCircuit)());    /* destroy old circuit's data structures */
    int ((*newNode)());              /* create new node */
    int ((*groundNode)());           /* create ground node */
    int ((*bindNode)());             /* bind a node to a terminal */
    int ((*findNode)());             /* find a node by name */
    int ((*instToNode)());           /* find node attached to a terminal */
    int ((*setNodeParm)());          /* set a parameter on a node */
    int ((*askNodeQuest)());         /* ask a question about a node  */
    int ((*deleteNode)());           /* delete a node from the circuit */

    int ((*newInstance)());          /* create new instance */
    int ((*setInstanceParm)());      /* set a parameter on an instance */
    int ((*askInstanceQuest)());     /* ask a question about an instance */
    int ((*findInstance)());         /* find a specific instance */
    int ((*deleteInstance)());       /* delete an instance from the circuit*/

    int ((*newModel)());             /* create new model */
    int ((*setModelParm)());         /* set a parameter on a model */
    int ((*askModelQuest)());        /* ask a questions about a model */
    int ((*findModel)());            /* find a specific model */
    int ((*deleteModel)());          /* delete a model from the circuit*/

    int ((*newTask)());              /* create a new task */
    int ((*newAnalysis)());          /* create new analysis */
    int ((*setAnalysisParm)());      /* set a parameter on an analysis  */
    int ((*askAnalysisQuest)());     /* ask a question about an analysis */
    int ((*findAnalysis)());         /* find a specific analysis */
    int ((*findTask)());             /* find a specific task */
    int ((*deleteTask)());           /* delete a task */

    int ((*doAnalyses)());           /* run a specified task */

    int numDevices;                  /* number of device types supported */
    IFdevice **devices;              /* array of device type descriptors */

    int numAnalyses;                 /* number of analysis types supported */
    IFanalysis **analyses;           /* array of analysis type descriptors */

    int numNodeParms;                /* number of node parameters supported */
    IFparm *nodeParms;               /* array of node parameter descriptors */

    int numSpecSig;                  /* number of special signals legal in parse trees */
    char **specSigs;                 /* names of special signals legal in parse trees */
} IFsimulator;
```

The functions pointed to by this structure are described in the chapter on interface subroutines. The *simulator* character string is the name of the simulator itself. *NumDevices* contains the number of devices which are described in the devices array. *Devices* is an array of pointers to IFdevice structures for the *numDevices* devices supported by the simulator. *Analyses* is an array of pointers to

IFanalysis structures describing the *numAnalyses* different analyses supported by the simulator. *numNodeParms* is the number of parameters supported on nodes in the circuit, and *nodeParms* is the array of descriptors of those parameters. *numSpecSig* and *specSigs* provide a list of special signal names which identify signals which may be used in parse trees even through they may not normally occur in the circuit as dependent variables. Typically, these will be the independent variables, such as time, of the various types of analyses which the simulator may wish to allow to occur in parse trees. Use of these variables in parse trees requires special action on the part of the simulator, particularly in the case of multiple analyses where a particular variable may not be active in certain situations. Some reasonable value should be used, and the action documented in the description of the function.

1.4.8. IFfrontEnd structure

```
typedef struct sIFfrontEnd {
    int ((*IFnewUid)());                   /* create a new UID in the circuit */
    int ((*IFpauseTest)());                /* should we stop now? */
    double ((*IFseconds)());               /* CPU time so far */
    int ((*IFerror)());                    /* output an error or warning message */
    int ((*OUTpBeginPlot)());              /* start pointwise output plot */
    int ((*OUTpData)());                   /* data for pointwise plot */
    int ((*OUTwBeginPlot)());              /* start windowed output plot */
    int ((*OUTwReference)());              /* independent vector for windowed plot */
    int ((*OUTwData)());                   /* data for windowed plot */
    int ((*OUTwEnd)());                    /* signal end of windows */
    int ((*OUTendPlot)());                 /* end of plot */
    int ((*OUTbeginDomain)());             /* start nested domain */
    int ((*OUTendDomain)());               /* end nested domain */
                                       int ((*OUTattributes)()); specify attributes of node */
} IFfrontEnd;
```

This structure is used by the front end to describe itself and the back end to the simulator. The functions whose pointers are required in the structure are all described in the front end subroutines section.

1.5. Simulator subroutines

Since we want to allow for several simulators to be supported by the front end in as modular a fashion as possible, we define the following subroutines for communication between the front end and the simulator. These are the subroutines that must be implemented by the simulator for the front end to call. Note that only the 'generically named' subroutine *XXXinit* is defined as having a known name while all others may have any name at all, but are listed by the names given to them in the *IFsimulator* and *IFfrontEnd* data structures described in the previous section for ease of identification. It is STRONGLY recommended that for simplicity, each simulator or front end or major section thereof have a unique two or three letter prefix that is prepended to all external names, including the names of these functions, to prevent name clashes.

In the description of the functions below, many parameters are described as *GENERIC* * or *GENERIC* **. This should be the ANSI standard *void* * or *void* ** notation, but many C compilers do not support this yet, so you may wish to use *char* * and *char* ** instead. To make this substitution easier, everything has been declared with *GENERIC* which is *typedef*'d to *void* in the IFsim.h header file if the preprocessor symbol *__STDC__* is defined, or to *char* otherwise.

1.5.1. XXXinit

```
int XXXinit(frontEnd, description)
   IFfrontEnd *frontEnd;
   IFsimulator **description;
```

The function XXXinit perfoms whatever simulator specific initialization is needed, and allows the front end and simulator to exchange information about each other. The actual name of the function should be derived from the simulator, thus SPIinit initializes spice, while RLXinit would initialize relax. By naming the simulator initialization function this way, several simulators can be used from within one front end at one time. The front end supplies a pointer to an IFfrontEnd structure which describes itself and the (possibly separate, possibly integrated) back end to the simulator. In return, the simulator provides a pointer to it's *IFsimulator* structure which describes to the front end all the remaining interface facilities. This function is the only one in the simulator or front end which has a fixed name, all other functions are accessed through the *IFsimulator* and *IFfrontEnd* structures, thus allowing for the possibility of multiple simulators accessible from one front end or multiple front ends for one simulator by changes in this one routine at some point in the future.

1.5.2. newCircuit

```
int newCircuit(circuit)
    GENERIC **circuit;
```

NewCircuit is a function that allocates the structure necessary to describe a circuit, and sets *circuit to be a pointer to the newly allocated structure. Each succeeding call from the parser will contain this pointer as one of its arguments to identify which circuit the call applies to.

1.5.3. deleteCircuit

```
int deleteCircuit(circuit);
    GENERIC *circuit;
```

DeleteCircuit is a cleanup function that recursively traverses all the data structures that may be associated with *circuit* and frees them. After return, *circuit* is no longer a valid circuit pointer and its use in any subsequent subroutine calls will cause unpredicatable results.

1.5.4. newNode

```
int newNode(circuit, node, nodeUid);
    GENERIC *circuit;
    GENERIC **node;
    IFuid nodeUid;
```

This function will be called by the front end whenever a new circuit node is to be created. The pointer pointed to by *node* will be set to a value to be used by the front end for all future references to this node. It is expected that the simulator will keep the node IFuid it receives in order to describe its output and to produce error messages. The simulator may also wish to assign its own node numbering based on these calls. *node* is the unique identifier assigned by the front end for this node.

Note that it is an error to call this function with a IFuid that matches that passed to a previous call, although it is not necessary for the simulator to check for this error.

1.5.5. groundNode

```
int groundNode(circuit, node, nodeUid);
    GENERIC *circuit;
    GENERIC **node;
    IFuid nodeUid;
```

This function creates a node exactly as *newNode* does, but it indicates that this is the ground node, and thus must be called exactly once for each circuit. This subroutine will return E_EXISTS if a ground node has been previously defined for this circuit and the value of the ground node is NOT changed. This subroutine must be called before any other reference is made to the ground node by any other call to the simulator.

1.5.6. bindNode

```
int bindNode(circuit, instPtr, terminal, node)
    GENERIC *circuit;
    GENERIC *instPtr;
    int terminal;
    GENERIC *node;
```

This function connects a terminal of an instance to a node in the circuit. The *terminal* parameter is used to select a terminal by number. Terminals of the device are numbered from 1 to the maximum terminal number allowed, while nodes are passed by the pointers returned by *newNode* and *groundNode*. Note that the terminal numbering is assumed to correspond to the ordering of the termNames field in the device structure.

1.5.7. findNode

```
int findNode(circuit, node, nodeUid)
    GENERIC *circuit;
    GENERIC **node;
    IFuid *nodeUid;
```

This function allows the front end to find a node pointer from the node's unique identifier. The arguments are all exactly as in newNode, but the node must already exist or the error E_NOTFOUND will be returned.

1.5.8. instToNode

```
int instToNode(circuit, instPtr, terminal, node, nodeUid)
    GENERIC *circuit;
    GENERIC *instPtr;
    int terminal;
    GENERIC **node;
    IFuid *nodeUid;
```

This function allows the front end to find the node pointer and node unique identifier of the circuit node a particular terminal of an instance is bound to. If the node has not been bound yet, *instToNode* will return the node this terminal will be connected to by default if no call to *bindNode* is made and the simulator provides default bindings, otherwise it will set *node* and *nodeUid* to NULL.

1.5.9. deleteNode

```
int deleteNode(circuit, node);
    GENERIC *circuit;
    GENERIC * node;
```

This function is used to indicate to the simulator that the last attachment to the specified node has been deleted and the node will no longer be used. Since the simulator may keep reference counts on nodes, it may detect a call to deleteNode that MUST be ignored because the node is still in use and return the E_NOTEMPTY error code, but the simulator's behavior in such a situation is undefined since it need not do such reference counting. Note that it is always illegal to delete the ground node.

1.5.10. setNodeParm

```
int setNodeParm(circuit, node, parm, value, selector)
    GENERIC *circuit;
    GENERIC *node;
    int parm;
    IFvalue *value;
    IFvalue *selector;
```

The node *node* in circuit *circuit* has its parameter *parm* set to the appropriate value taken from the *value* structure. The *selector* structure provides additional information about the way the parameter is to be modified. Specifically, it is either an integer or vector of integers which selects a specific entry from the set of possible values to set. For example, if a possible parameter is a multi-dimensional matrix, and a user wishes to change only a single entry in the matrix, rather than passing the entire matrix it is possible to set the selector structure to have the array of matrix subscripts to specify the element to be changed and place just its value in the value field. Another parameter with a similar but not identical name may be used to manipulate the entire array or matrix. Note that for any parameter not declared with the IF_SELECT or IF_VSELECT bit in its IFparm structure, this argument is ignored.

1.5.11. askNodeQuest

```
int askNodeParm(circuit, node, quest, value, selector)
    GENERIC *circuit;
    GENERIC *node;
    int quest;
    IFvalue *value;
    IFvalue *selector;
```

The node *node* in circuit *circuit* is queried for the value of its parameter *quest* which is returned in the appropriate field in the supplied *value* structure. As with the setNodeParm function, the selector allows the specification of a single value within a uni- or multi- dimensional array, but is ignored unless IF_SELECT or IF_VSELECT is specified in the IFparm structure for the parameter.

1.5.12. newInstance

```
int newInstance(circuit, modelPtr, instPtr, nameUid)
    GENERIC *circuit;
    GENERIC *modelPtr;
    GENERIC **instPtr;
    IFuid nameUid;
```

Circuit is the circuit structure describing the circuit this instance is to be a part of. *ModelPtr* will be a pointer to the actual model that is to be instantiated. *InstPtr* points to the pointer the newInstance routine should initialize to point to the new instance. *Nameid* is the unique identifier of the instance to be created.

1.5.13. setInstanceParm

```
int setInstanceParm(circuit, instPtr, parm, value, selector)
    GENERIC *circuit;
    GENERIC *instPtr;
    int parm;
    IFvalue *value;
    IFvalue *selector;
```

This function is used to set a single parameter on a instance to a specified value. *instPtr* is the instance pointer provided by the *newInstance* function.

1.5.14. askInstanceQuest

```
int askInstanceQuest(circuit, instPtr, parm, value, selector)
    GENERIC *circuit;
    GENERIC *instPtr;
    int parm;
    IFvalue *value;
    IFvalue *selector;
```

This function can be used at any time to determine the current value of any of the queryable parameters described in the instanceParms array. The first two parameters are used simply to identify the instance in question, *parm* is one of the id's from the instanceParms array, and identifies the exact parameter to be returned, while *value* points to an empty IFvalue structure which will be filled in as appropriate by the simulator.

1.5.15. findInstance

```
int findInstance(circuit, devIndex, instPtr, instUid, modelPtr, modelUid)
    GENERIC *circuit;
    int *devIndex;
    GENERIC **instPtr;
    IFuid instUid;
    GENERIC *modelPtr;
    IFuid modelUid;
```

This function is used to allow the front end to find an instance pointer from whatever information it has available about the instance. The pointer pointed to by *instPtr* will be set to the necessary instance pointer if the information is sufficient and the instance is found, otherwise *findInstance* will return E_NOTFOUND. The information required to identify the instance is simply the minimum of the following needed to uniquely identify it, provided in any sensible combination. The more information provided, the more efficiently the instance can be found, with the most important information being described first. *instUid* is the unique identifier of the instance to be found and MUST be supplied. *devIndex* is a pointer to the (0 based) index into the device array in the IFsimulator structure of the description of this device type. If devIndex points to -1, it will be set to the correct value on return. *modelPtr* is the model pointer for the model this instance is thought to be an instance of. A value of NULL indicates the model pointer is not known. *modelUid* is the unique identifier of the model this instance is thought to be an instance of. A value of NULL indicates the model identifier is unknown.

1.5.16. deleteInstance

```
int deleteInstance(circuit, instPtr)
    GENERIC *circuit;
    GENERIC *instPtr;
```

This function removes the specified instance from the circuit. After return, instPtr is no longer valid and should not be used in any further subroutine calls.

1.5.17. newModel

```
int newModel(circuit, devIndex, modelPtr, modUid)
    GENERIC *circuit;
    int devIndex;
    GENERIC **modelPtr;
    IFuid modUid;
```

This function is similar to the newInstance function, except it creates a model instead of an instance, thus treating *modUid* and *modelPtr* as *nameUid* and *instPtr* were treated in newInstance. *devIndex* is a specification to the simulator of the type of model which is being created. The model type is defined as the (0 based) index to the description of the device in the *devices* field of the

IFsimulator structure.

1.5.18. setModelParm

```
int setModelParm(circuit, modPtr, parm, value, selector)
    GENERIC *circuit;
    GENERIC *modPtr;
    int parm;
    IFvalue *value;
    IFvalue *selector;
```

This function is identical to setInstanceParm, except it applies to parameters of a model instead of an instance.

1.5.19. askModelQuest

```
int askModelQuest(circuit, modelPtr, parm, value, selector)
    GENERIC *circuit;
    GENERIC * modelPtr;
    int parm;
    IFvalue *value;
    IFvalue *selector;
```

This is similar to the askInstanceQuest function, but applies to parameters of a device model instead of an instance.

1.5.20. findModel

```
int findModel(circuit, devIndex, modelPtr, modelUid)
    GENERIC *circuit;
    int *devIndex;
    GENERIC **modelPtr;
    IFuid *modelUid;
```

This function allows the front end to attempt to get a model pointer from whatever information it has about the model. *devIndex* is the device index of the device type the model is an instance of, and may be specified as -1 if it is unknown. The actual index value will be returned to facilitate obtaining information from the IFdevice structure for further operations on the model. *modelUid* specifies the unique identifier of the model being looked for, and must be supplied. *modelPtr* points to a pointer which will be set to the desired model pointer if the specified model is found. If the specified model can't be found either because it doesn't exist or because not enough data has been supplied, *findModel* returns E_NOTFOUND.

1.5.21. deleteModel

```
int deleteModel(circuit, modelPtr);
    GENERIC *circuit;
    GENERIC *modelPtr;
```

This function removes the specified model from the circuit. After return, modelPtr is no longer valid and should not be used in any further subroutine. If any instances of the specified model exist, *deleteModel* should return E_NOTEMPTY without deleting anything.

1.5.22. newTask

```
int newTask(circuit, taskPtr, taskUid);
    GENERIC *circuit;
    GENERIC **taskPtr;
    IFuid taskUid;
```

A task is a grouping of analyses that are to be performed as a group by a single call to *doAnalyses*. The simulator is free to re-arrange analyses within a task if doing so will improve its efficiency, but may not move analyses between tasks. This function creates a new task associated with the given *circuit* with the unique identifier specified and returns a pointer through *taskPtr*.

1.5.23. newAnalysis

```
int newAnalysis(circuit, analIndex, analysisUid, analysisPtr, taskPtr);
    GENERIC *circuit;
    int analIndex;
    IFuid analysisName;
    GENERIC **analysisPtr;
    GENERIC *taskPtr;
```

This function creates a new analysis within the specified task, and sets the pointer pointed to by *analysisPtr* to be the appropriate analysis pointer.

1.5.24. setAnalysisParm

```
int setAnalysisParm(circuit, analysisPtr, parm, value, selector)
    GENERIC *circuit;
    GENERIC *analysisPtr;
    int parm;
    IFvalue *value;
    IFvalue *selector;
```

This function sets a parameter for the specified analysis.

1.5.25. askAnalysisQuest

```
int askAnalysisQuest(circuit, analysisPtr, parm, value, selector)
    GENERIC *circuit;
    GENERIC *analysisPtr;
    int parm;
    IFvalue *value;
    IFvalue *selector;
```

This function provides information about the specified analysis request.

1.5.26. findAnalysis

```
int findAnalysis(circuit, analIndex, analysisPtr, analysisUid, taskPtr, taskUid)
    GENERIC *circuit;
    int *analIndex;
    GENERIC **analysisPtr;
    IFuid analysisUid;
    GENERIC *taskPtr;
    IFuid taskUid;
```

This function is provided to allow the front end to locate a specific analysis from limited information. *analysisUid* is the unique identifier of the analysis to be found and must be specified. *analIndex* is a pointer to the index used in the newAnalysis call, and is set on return. *taskPtr* is a pointer to the task this analysis is a member of, and may be passed as NULL if it is not known. *taskUid* is the unique identifier of the task this analysis is a member of, and may be passed as NULL if it is not known. If both *taskPtr* and *taskUid* are passed as null, then for the return value to be defined, the analysisUid must be unique over all analyses in all tasks.

1.5.27. findTask

```
int findTask(circuit, taskPtr, taskUid)
    GENERIC *circuit;
    GENERIC **taskPtr;
    IFuid taskUid;
```

This function allows the front end to get a task pointer from the task unique identifier.

1.5.28. deleteTask

```
int deleteTask(circuit, taskPtr)
    GENERIC *circuit;
    GENERIC *taskPtr;
```

This function deletes the specified task and all analyses in it.

1.5.29. doAnalyses

```
int doAnalyses(circuit, restart, taskPtr)
    GENERIC *circuit;
    int restart;
    GENERIC *taskPtr;
```

This function instructs the simulator to perform the analyses within the specified task that have been requested by previous function calls.

The restart parameter has several different values which determine how the simulator will proceed. If restart is RESUME, the simulator should attempt to resume from where it stopped if possible even if that is in the middle of a paused or aborted analysis, and failing that, restart the aborted simulation without rerunning the entire task. If restart is RESTART, the simulator should throw away any analysis results obtained so far for this task and start all over again. If restart is SKIPTONEXT, the simulator should give up on the analysis in progress, but attempt to continue with any other scheduled analyses in this task. When starting an analysis from the beginning, the value RESTART should be given.

1.6. Front end subroutines

All output from the simulator must pass through a consistent interface since the front end should have complete control of the display at all times to eliminate confusion due to terminals with graphics and alphanumeric modes, intermixed output, and other such difficulties. The functions below fall into three categories, those which are generic, and thus used by all simulators, those which are tuned for simulators which generate their output as a set of values for a single value of their independent variables (pointwise output), and those which generate their output as a set of solutions for a single signal at a set of values for their independent variables (windowed output). Note that calls to the pointwise and windowed functions may NOT be intermixed.

These calls all return an error condition as described earlier in this interface manual. If the error return is not OK, then the operation requested may not have been completed, and future calls may or may not succeed depending on the type of error. The simulator must NOT terminate, but should try to interpret the error code. If this is not possible, the simulator may wish to return this code to the front end. The output package will have set errRtn and errMsg, thus if you do not pass the error on, you must clear errRtn and free errMsg.

Note that each output call has an analysisPtr with it. This pointer serves to uniquely identify the analysis the plot is associated with and should be checked on each call. Some simulators and/or simulations may produce output for two or more of their analyses intermixed, and the output package should be prepared for it.

Finally, note that as an optimization for the simulator, since the output system will, in general, not want to save the entire output and the simulator will need to generate additional identically sized vectors, the output system is to copy the contents of the vector which it wants but does NOT become responsible for the vector or for freeing it as is usual for a passed object of type IF_VECTOR. This applies to the vectors passed to OUTpData, OUTwReference, and OUTwData.

1.6.1. IFnewUid

```
int IFnewUid(circuit, newuid, olduid, suffix, type, pointer)
    GENERIC *circuit;
    IFuid *newuid;
    IFuid olduid;
    char *suffix;
    int type;
    GENERIC *pointer;
```

This function allows a simulator to generate a new IFuid from an existing IFuid. The new IFuid will be named as if the name corresponding to *olduid* had the character string *suffix* appended to it with a front end specified character added to ensure uniqueness. The suffix must be made entirely from alphanumeric characters, upper and lower case letters and digits. The *type* field is used to specify to the front end the use that will be made of the IFuid. In the event that names must be generated from scratch for such things as time, the olduid should be specified as the reserved IFuid value 0. Current legal types are UID_ANALYSIS, UID_TASK, UID_INSTANCE, UID_MODEL, UID_SIGNAL, and UID_OTHER for an id to be used to name an analysis, task, instance, model, any node or other naturally computed output of the circuit, and any other type of IFuid that is needed respectively. *pointer* is an optional pointer supplied to the simulator to point to the structure to be associated with the IFuid being created. This pointer may be left out completely, or may be ignored by the front end, but is supplied for the front end's convenience in maintaining its data structures in which it may wish to keep a pointer to the structures associated with an IFuid.

1.6.2. IFeval

```
int IFeval(tree, result, vals, derivs)
    IFparseTree *tree;
    double *result;
    double *vals;
    double *derivs;
```

This function is a generic function passed by the front end in the IFparseTree structure which is used to evaluate the parse tree given. The given parse tree is to be evaluated with its *tree->numVars* variables given the values from the vector *vals*. The scalar result of evaluating the tree is placed in *result*, while a set of partial derivatives with respect to the input variables is placed in the vector *derivs*.

1.6.3. IFpauseTest

```
int IFpauseTest()
```

This function is a simple function passed from the front end to the simulator as part of the IFfrontEnd structure and is used to signal the simulator that the front end wants control. Since the front end can not take control away from the simulator and expect to return without damage to the simulator's data structures, the simulator must relinquish control when necessary. This is accomplished by having the simulator call the IFpauseTest function frequently (at least as frequently as it can conveniently stop). This function can allow the front end to perform short operations which require no access to the simulator (window redraws by the back end, for example), and to indicate by a non-zero return value that it wants total control at the simulator's earliest convenience.

1.6.4. IFseconds

```
double IFseconds()
```

This function is provided to allow simulators and front ends which agree to do so to provide the facilities for statistics gathering. This function should return the run time for the job in seconds as a double precision floating point number. The run time does not have to be absolute, but the difference between successive calls should provide accurate elapsed CPU time. If the front end does not wish to provide such capabilities or is unable to do so because of the operating system, it may return the constant 0.0, and should expect **and document** that all time related statistics from the simulator will be meaningless.

1.6.5. IFerror

```
int IFerror(flags, format, names);
    int flags;
    char *format;
    IFuid *names;
```

This function provides a way for the simulator to output messages to the user through the front end without interfering with whatever the front end may be doing with the terminal display. *flags* is one of ERR_INFO, ERR_WARNING, ERR_FATAL, or ERR_PANIC, indicating that the message is a simple informational status message that the front end should provide a way of enabling and disabling based on user preference, a warning message, an error message that will probably cause the simulator to fail, and an extremely serious error from which the simulator won't even try to recover. Note that after an ERR_PANIC, the simulator may be in such a bad state that it is impossible to perform further operations on the current circuit. *Format* is a textual message that should be displayed to the user. The format string will be processed by a mechanism similar to printf, and all printf conventions should be observed, including modifiers on %s and using %% to place a % in the output. Only %s format items will be interpreted properly, and the data for them comes from the array of IFuid's pointed to by *names*. This function is provided in this form to allow for the greatest flexibility in error message output, since it permits the front end to get the IFuid's directly. One benefit of this is that a graphic front end may wish to highlight the objects corresponding to the IFuid's in different colors and fill in some descriptive information in the message rather than using a possibly cryptic name which the user may not understand in the message itself.

1.6.6. Pointwise output

Two functions are intended specifically to support pointwise output, OUTpBeginPlot and OUTpData. OUTpBeginPlot describes the plot to be produced to the output routines, and is called once per plot. OUTpData is called repeatedly and supplies data for one set of values of the independent variables per call. OUTendPlot indicates that the plot is completed and no more calls to OUTpData will be made.

1.6.6.1. OUTpBeginPlot

```
int OUTpBeginPlot(ckt, analysisPtr, analUid, refName, refType,
        numNames, dataNames, dataType, plotPtr);
    GENERIC *ckt;
    GENERIC *analysisPtr;
```

```
        IFuid analUid;
        IFuid refName;
        int refType;
        int numNames;
        IFuid *dataNames;
        int dataType;
        GENERIC **plotPtr;
```

This function describes the plot to be produced to the output package. *ckt* is the circuit pointer used by the front end to identify the circuit to the simulator. analysisPtr is an analysis pointer as returned to the front end in a previous call to the simulator's newAnalysis function. IFuid is the unique identifier for the analysis that analysisPtr refers to. These two should serve to uniquely identify the analysis to the front end and thus allow access to any additional information it may have about the plot. refName is the IFuid of the slowest varying reference or independent variable of the plot, or is NULL in the case of a single point output, such as an operating point. refType is the type (as used in the IFparm structure) of the reference variable. numNames is the number of variables named in the dataNames array. dataNames is an array of the unique identifiers which are associated with the data to be output in the vectors handed to OUTpData. A NULL IFuid (i.e. (IFuid) 0) indicates that the corresponding data point is not interesting and should not be saved. The IFuid refName MAY appear in the dataNames array, but does not need to. If refName does appear in dataNames, then the two values provided for it in the OUTpData call MUST be the same or the results will be unpredictable. dataType is the type (as used in the IFparm structure) of all of the values to be returned by OUTpData (without the IF_VECTOR attribute). Finally, *plotPtr* is an identifier created by the back end to identify the plot, and will be supplied by the simulator in all future OUT* calls regarding this plot.

1.6.6.2. OUTpData

```
    int OUTpData(plotPtr, refValue, valuePtr)
        GENERIC *plotPtr;
        IFValue *refValue;
        IFvalue *valuePtr;
```

This function actually delivers the data described in OUTpBeginPlot for a single value of the reference variable. plotPtr is the pointer returned by OUTpBeginPlot above and is used to ensure that plot data doesn't get mixed up between plots since multiple plots may be active at once. refValue is the value (of the type indicated by refType in the OUTpBeginPlot call) of the reference vector. Note that for multi-dimensional plots (more than one independent variable), this will be the value for the independent variable which is changing MOST rapidly, values for the other independent variables are obtained from previous calls to OUTbeginDomain. If refUid was given as the reserved IFuid 0, then the refValue is uninteresting and should be ignored. valuePtr is a pointer to an IFvalue structure of type IF_VECTOR of whatever type was given for the dataType in the call to OUTpBeginPlot. The length of the vector valuePtr MUST be the same as the value of the numNames parameter passed to OUTpBeginPlot.

1.6.7. Windowed output

When outputting windowed data, provision is made for the possibility that different nodes in the circuit may have a different time scale (reference vector) associated with them, and that there may be several different groupings of these during the course of the simulation. Four functions are intended specifically to support windowed output, OUTwBeginPlot, OUTwReference, OUTwData, and OUTwEnd. These functions declare an output, present values for the independent variable and the dependent variables, and signal the end of output for a specific window. The output routines may assume that windows are non-overlapping and that all reference and data values from one window are given before any from the next window. When a new window begins (as indicated by a call to OUTwEnd) the data from the previous window may be considered complete.

1.6.7.1. OUTwBeginPlot

```
int OUTwBeginPlot(ckt, analysisPtr, analUid, refName, refType,
     numNames, dataNames, dataType, plotPtr);
   GENERIC *ckt;
   GENERIC *analysisPtr;
   IFuid analName;
   IFuid refName;
```

```
    int refType;
    int numNames;
    IFuid *dataNames;
    int dataType;
    GENERIC **plotPtr;
```

This function begins a windowed output plot. The arguments are the same as for OUTpBeginPlot, but the data for the plot will be delivered using the windowing output functions instead of the pointwise functions. The vector dataNames is provided for the convenience of the output package, so that all the dependent variables are known in advance. Also, the dependent variables are referred to using their index in this vector by calls to the function OUTwData.

1.6.7.2. OUTwReference

```
int OUTwReference(plotPtr, valuePtr, refPtr)
    GENERIC *plotPtr;
    IFvalue *valuePtr;
    GENERIC **refPtr;
```

This routine outputs a window's worth of the reference vector. Several calls will usually be made to this routine with overlapping values when different nodes have different time scales. The plotPtr specifies the analysis this applies to, valuePtr provides a vector of the type given in OUTwBeginPlot of data covering one window of values for the reference variable. refPtr is used to return to the simulator a pointer it will use to refer to this particular reference vector during the remainder of this window.

1.6.7.3. OUTwData

```
int OUTwData(plotPtr, index, valuePtr, refPtr)
    GENERIC *plotPtr;
    int index;
    IFvalue *valuePtr;
    GENERIC *refPtr;
```

OUTwData supplies output data for a single signal for an entire window. The plotPtr gives the analysis this data is for, the index gives the index of the output vector being provided within the dataNames array provided in the call to OUTwBeginPlot, valuePtr is a vector of whatever type was specified in OUTwBeginPlot providing the values. The number of elements in the vector in valuePtr and the number of elements in the vector provided to OUTwReference when the given refPtr was returned must match.

1.6.7.4. OUTwEnd

```
int OUTwEnd(plotPtr)
   GENERIC *plotPtr;
```

This call indicates that all data for the current window is complete and the output system should prepare to receive data for the next window. This function may optionally be called at the end of the plot or a domain, but is not needed at those points since OUTendDomain and OUTendPlot both imply the end of a window.

1.6.8. General routines

There are a number of routines which are used by both windowed and pointwise output.

1.6.8.1. OUTendPlot

```
int OUTendPlot(plotPtr)
   GENERIC *plotPtr;
```

This call indicates to the output system that all the data for the specified plot has been delivered already and any clean-up operations may be performed.

1.6.8.2. OUTbeginDomain

```
int OUTbeginDomain(plotPtr, refUid, refType, outerRefValue)
   GENERIC *plotPtr;
   IFuid refUid;
   int refType;
   IFvalue *outerRefValue;
```

When a plot has two or more independent variables, it is assumed that they will be swept in the fashion of nested loops, with one swept through its entire range before the next one is changed. This function specifies to the output package that this sort of sweeping is taking place. Each call to OUTbeginDomain indicates a variable that will be swept and must match up with a corresponding call to OUTendDomain. The most recent OUTbeginDomain not yet closed by the corresponding OUTend-Domain specifies the only reference variable which may change, with the values of all less rapidly changing reference variables specified by the intervening OUTbeginDomain calls. An arbitrary nesting level is permitted. There is an implied OUTbeginDomain/OUTendDomain pair in the OUTwBeginPlot or OUTpBeginPlot and OUTendPlot function calls, so an explicit call is not required for the outermost reference variable or the only reference variable in simple plots. Note that this structure requires that all OUTwBeginPlot or OUTpBeginPlot calls within one plot occur at exactly the same nesting level and with exactly the same set of reference variables active at each call.

1.6.8.3. OUTendDomain

```
int OUTendDomain(plotPtr)
    GENERIC *plotPtr;
```

Closes the innermost pending OUTbeginDomain call, thus allowing the reference variable at the next higher nesting level to change.

1.6.8.4. Domain use Example

```
IFvalue innerRef, outerRef;
IFvalue dataValues;
GENERIC *pltptr;
IFuid iid, jid;

IFnewUid(ckt, &iid, (IFuid)0, "i", UID_OTHER)
IFnewUid(ckt, &jid, (IFuid)0, "j", UID_OTHER)
/* initialize the plot */
OUTpBeginPlot(ckt, analPtr, analUid , iid, IF_REAL, numdata, dataNames, IF_REAL, &pltptr)
for ( i = istart ; i <= istop ; i++ ) {
    outerRef.rValue = i;
    /* tell the output package that we are
     * going into a nested loop, and give the
     * value of the outer loop since it won't
     * change within this inner loop
     */
    OUTbeginDomain(pltptr, jid, IF_REAL, &outerRef);
    for ( j = jstart ; j <= jstop ; j+= jstep ) {
        /* crunch away, leave results in dataValues */
        dataValues.vec.dvec = compute(i, j);
        innerRef.rValue = j;
```

```
            /* and output the results */
            OUTpData(pltptr, &innerRef, &dataValues);
        }
        OUTendDomain(pltptr);
    }
    outEndPlot(pltptr);
```

Note that the call to OUTbeginDomain gives the value of the reference variable in the enclosing (implied) domain (i), and the name and type of the reference variable in the inner domain (j), while the OUTpData call only provides a value for the innermost domain's reference variable j.

1.6.8.5. OUTattributes

```
int OUTattributes(plotPtr, varName, param, value)
    GENERIC *plotPtr;
    IFuid *varUid;
    int param;
    IFvalue *value;
```

This function allows the simulator to provide additional information to the output system about the plot in general or about specific variables in the plot. If varUid is NULL, then the data is a default for this plot which overrides the defaults described here only for this plot. If varUid is not NULL, then it is the unique identifier of one of the variables in the plot specified by plotPtr. and provides overriding values for just that variable. The possible values for the parameters and their meanings are:

parameter	value type	value meaning	default
OUT_SUBTITLE	char *	An optional one line descriptive title. Only applicable to the entire plot.	none
OUT_INTERP_POLY	int	The number of points used to interpolate values using polynomial interpolation. A value of zero implies interpolation is not appropriate. For values of 1, 2, 3, 4 specify respectively piecewise constant, linear, quadratic, or cubic interpolation.	2
OUT_INTERP_SINE	int	The number of points used to interpolate values using sinusoidal interpolation. The sinusoid used should be contiguous harmonics of a fundamental, whose frequency equals the reciprocal of the difference between the maximum and minimum values of the independent variable.	
OUT_INTERP_EXP	int	The number of points used to interpolate values using exponential interpolation.	
OUT_UNITS	char *	The name of the units the data is in.	none
OUT_SCALE_LIN	none	The output should be plotted on a linear scale	
OUT_SCALE_LOG	none	The output should be plotted on a log scale	
OUT_GRID_RECT	none	The output should be plotted on a rectangular grid.	*
OUT_GRID_SMITH	none	The output should be plotted on a smith chart.	
OUT_GRID_POLAR	none	The output should be plotted on a polar grid.	
OUT_STYLE_POINTS	none	The plot should show only points without a connecting curve	
OUT_STYLE_CURVES	none	The plot should have a smooth curve fitted through the points based on the interpolation parameter above.	*
OUT_STYLE_COMB	none	The plot should be a comb or bar-graph type plot.	
OUT_RELTOL	double	The relative precision of a variable. Values of a variable smaller than this threshold times the maximum value of the variable are considered noise.	0
OUT_ABSTOL	double	The absolute precision of a variable. Values of a variable smaller than this threshold are considered noise.	0

The interpolation attributes are usually specified for the independent variables only. When given for a particular independent variable, they are applied when interpolating any dependent variable over intervals of that independent variable.

Calls to this function must follow the call to OUTwBeginPlot or OUTpBeginPlot immediately, and precede all calls to outBeginDomain or any data output functions. Note that this function may thus provide attributes for a reference variable which has not yet been encountered in an OUTbegin-Domain call.

For consistency, it is recommended that the value of OUT_UNITS be in a standard form. For simple units, the standard symbol should be used. Below is a partial list of standard symbols, with some alterations to avoid unusual characters such as Greek letters.

Standard Abbreviation	Meaning
V	Volt
A	Ampere
W	Watt
J	Joule
ohm	ohm
S	Siemens
F	Farad
H	Henry
s	second
Hz	Hertz
K	degree Kelvin
C	degree Celsius
Coul	Coulomb
Wb	Weber
T	Tesla
m	meter
in	inch

For more complicated units, the standard symbols should be combined according to a fixed set of rules whenever possible: a product of units is formed by placing a hyphen between them; a ratio of units is formed by placing a slash between them, and only one slash is allowed in the entire units string; a power of a unit is represented by a caret and a number following the unit; as special cases of powers, the square root of a unit is written as *sqrt(unit)* and the cube root of a unit is written as *cbrt(unit)*; parentheses can be used for grouping; no spaces should appear in the units string. Exponentiatian has the highest precedence, followed by multiplication, the division. Some examples of units strings formed according to these rules are:

```
V/sqrt(Hz)
ohm^2-V/(m-K)^3
```

A clever plotting package can take advantage of this standardization to do some formatting, such as raising the exponents or displaying a ratio with the numerator over the denominator.

1.7. Changes since version 1.0

This section summarizes the changes made in this document since the first released version, V1.0. Subsections will provide summaries of changes from each version to the next, with a detailed breakdown of each of the chapters.

1.7.1. V2.0

The most significant change is the introduction of the output features. Also affecting existing code, the 'type' field passed to many of the simulator routines has been deleted. It is now assumed that the simulator can determine the type of an instance or model it is passed. Additionally, the routines pauseTest, IFnodeName, and IFdevName have been moved into the new front end data structure, and their descriptions moved to the chapter on front end routines. In the process, IFnodeName and IFdevName have been replaced by a new more general routine, IFnewUid. Finally, IFuid's, as described in the overview, have been introduced in place of explicit pointers to names throughout the system. There is nothing to prevent the IFuid's from being the pointers to the names as used in previous versions, but this permits the front end more flexibility. Some changes in the naming of the parameters in the documentation has also been made to make their purpose clearer, along with some typographical corrections. These last changes are not described in more detail since they should not affect existing code.

1.7.1.1. Introduction

The terminology section has been expanded to cover more of the terms used throughout the document.

1.7.1.2. Overview

The symbol which indicates the presence of the ANSI standard C constructs such as void* or void** has been changed from 'ANSI' to __STDC__ to comply with the latest draft ANSI standard.

The notion of a restricted character set for node and device names was introduced, along with the concept of IFuid's, InterFace Unique IDentifiers, instead of pointers to device names. This change allows systems that don't have a simple textual format as their base to use the simulators more easily.

1.7.1.3. Error Handling

Additional description of errRtn and errMsg was added making it clear that the front end must declare them in such a way as to actually allocate storage, and that various combinations of these two may be set or not set by the simulator.

The error code E_BAD_DOMAIN was added to the list of standard error codes.

1.7.1.4. Data structures

The description of the IFparm structure has been expanded, and new bits in the dataType field have been defined.

The IFdevice structure has had a field added to it. This field indicates the length of the termNames array independent of the number/minimum number of terminals as indicated by the terms field.

The names newCkt and newCircuit were used interchangeably in the previous draft. They have been consistently renamed to newCircuit.

The IFfrontEnd data structure was introduced.

The names numDevices and numAnalyses were used throughout the description, but the actual code and header files used numdevices and numanalyses. The code and header files have been corrected. The subroutine instToNode was mistakenly described under the name devToNode. The description has been corrected to refer to the instToNode subroutine correctly now.

1.7.1.5. Simulator subroutines

All of the subroutines in the front end that were previously described here have been moved to the front end interface section, while the general description at the beginning of this chapter has been moved to the overview.

The SIMinit function has been renamed to have a simulator specific name. The old functionality can be obtained by writing a function for each simulator called SIMinit which reads:

```
int SIMinit(frontEnd, simulator)
   IFfrontEnd *frontEnd;
   IFsimulator **simulator;
   {
      return ( SPIinit(frontEnd, simulator) );
   }
```

where SPIinit is a simulator dependent name. Similarly, an expanded front end may call more than one XXXinit function.

The subname argument to newNode has been deleted.

The findInstance and findModel functions have been augmented to allow them to return the type of the device they have found if the type is unknown, thus that parameter has been changed from an int to an int*

The taskType argument to the newTask function has been deleted since it was put in accidentally and only in the documentation.

The second argument to deleteTask is only a GENERIC *, not a GENERIC**.

1.7.1.6. Front end subroutines

The pauseTest function has been renamed IFpauseTest and described completely.

The IFfrontEnd structure has been introduced, and the routines IFnodeName, IFdevName, and IFpauseTest have been moved that that structure. SIMinit has been modified to pass this entire structure instead of just the pauseTest function.