# INTERFACE DESIGN DOCUMENT

for the

# XSPICE SIMULATOR

of the

# AUTOMATIC TEST EQUIPMENT SOFTWARE SUPPORT ENVIRONMENT (ATESSE)

Contract No. F09603-89-G-0077-0002
CDRL Sequence No. A00E

September 1992

Prepared for:

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

Prepared by:

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

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

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

# Contents


**1 Scope** **1**

1.1 Identification . . . . . . . . . . 1
1.2 System Overview . . . . . . . . . . 1
1.3 Document Overview . . . . . . . . . . 2
1.4 Acknowledgements . . . . . . . . . . 2

**2 Referenced Documents** **3**

**3 Interface Design** **5**

3.1 Interface Diagrams . . . . . . . . . . 5
3.2 Simulator Invocation . . . . . . . . . . 5
3.3 Interprocess Communication . . . . . . . . . . 7
3.3.1 Aegis Mailboxes . . . . . . . . . . 7
3.3.2 Unix Sockets . . . . . . . . . . 8
3.3.3 IPC Message Protocol . . . . . . . . . . 8
3.4 Simulator Inputs . . . . . . . . . . 9
3.4.1 IPC Data . . . . . . . . . . 9
3.4.2 Circuit Description Syntax . . . . . . . . . . 10
3.5 Simulator Outputs . . . . . . . . . . 13
3.5.1 IPC Data . . . . . . . . . . 13
3.5.2 Results Data . . . . . . . . . . 15
3.6 Code Model Interface . . . . . . . . . . 17
3.6.1 Interface Specification File . . . . . . . . . . 18
3.6.2 Model Definition File . . . . . . . . . . 30
3.6.3 Macro Translations . . . . . . . . . . 30
3.7 User-Defined Node Interface . . . . . . . . . . 31
3.7.1 Node Definition File . . . . . . . . . . 33
3.7.2 Macro Translations . . . . . . . . . . 35

**4 Notes** **37**

4.1 Glossary . . . . . . . . . . 37
4.2 Acronyms . . . . . . . . . . 39
4.3 Project Unique Identifiers . . . . . . . . . . 40

# List of Figures

3.1 XSPICE Interface Diagram . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 6

# List of Tables


3.1 IPC Input Control Words. . . . . . . . . . 10
3.2 Pre-defined Port Types. . . . . . . . . . 12
3.3 IPC Output Control Words. . . . . . . . . . 14
3.4 DC/Transient Data Record Format. . . . . . . . . . 16
3.5 AC Data Record Format. . . . . . . . . . 17
3.6 Current Ordering in Results Data. . . . . . . . . . 17
3.7 Origin of Data Held in MIFmPTable. . . . . . . . . . 28
3.8 Origin of Data Held in MIFpTable. . . . . . . . . . 29
3.9 Origin of Data Held in MIFconnTable. . . . . . . . . . 30
3.10 Origin of Data Held in MIFparamTable. . . . . . . . . . 31
3.11 Origin of Data Held in MIFinst_varTable. . . . . . . . . . 31
3.12 Code Model Macro Translations. . . . . . . . . . 32
3.13 User-Defined Node Macro Translations. . . . . . . . . . 36

# 1 Scope

## 1.1 Identification

This Interface Design Document describes the external interfaces of the XSPICE simulator. The XSPICE simulator is a software component developed as part of the Computer Software Configuration Item (CSCI) identified as the Simulator of the Automatic Test Equipment Software Support Environment (ATESSE).

## 1.2 System Overview

The ATESSE is an integrated set of software tools designed to support all stages of the life cycle of software used to control Automatic Test Equipment (ATE) in testing analog and hybrid (analog/digital) circuit cards.

The XSPICE simulator is the tool that performs mathematical simulation of a circuit specified by the user. It takes input in the form of commands and circuit descriptions and produces output data which predicts the circuit's behavior. The simulator is based on the industry standard SPICE program developed at the University of California at Berkeley and is enhanced and modified to provide board-level and system-level simulation capability.

The XSPICE simulator can be built to run either with the ATESSE system through interprocess communication (IPC), or in a "standalone" mode with the command-line "Nutmeg" interface. This document covers both versions but is written principally from the standpoint of the version that runs with the ATESSE system.

The version of the XSPICE simulator designed to run with the ATESSE system is configured to interface either to the ATESSE version 1 or the new ATESSE version 2 system. A

compile-time option in the simulator's IPC package selects which IPC interface will be used.

## 1.3 Document Overview

This document defines the external interfaces of the XSPICE simulator and is divided into four sections. This section (Section 1) covers background material and acknowledgements. Section 2 provides a list of related documents. Section 3 includes the detailed design of the interfaces to each of the components of the simulator. Section 4 includes a glossary of technical terminology used throughout this document, a list of acronyms, and a summary of all applicable Project Unique Identifiers (PUIs).

## 1.4 Acknowledgements

The XSPICE simulator is based on the SPICE3 program developed by the Electronics Research Laboratory, Department of Electrical Engineering and Computer Sciences, University of California at Berkeley.

# 2 Referenced Documents

1. Software Requirements Specification for the Simulator of the Automatic Test Equipment Software Support Environment (ATESSE), F. L. Cox, W. B. Kuhn, J. P. Murray, S. D. Tynor, M. J. Willis, Georgia Tech Research Institute, Atlanta, GA, November, 1991.

2. Software User's Manual for the XSPICE Simulator of the Automatic Test Equipment Software Support Environment (ATESSE), F. L. Cox, W. B. Kuhn, H. W. Li, J. P. Murray, S. D. Tynor, M. J. Willis, Georgia Tech Research Institute, Atlanta, GA, September, 1992.

3. SPICE3C.1 Nutmeg Programmer's Manual, Department of Electrical Engineering and Computer Sciences, University of California, Berkeley, CA, April, 1987.

4. SPICE3 Version 3C1 User's Guide, Thomas L. Quarles, Department of Electrical Engineering and Computer Sciences, University of California, Berkeley, CA, April, 1989.

5. SPICE3C1 Nutmeg Programmer's Guide, Department of Electrical Engineering and Computer Sciences, University of California, Berkeley, CA, April, 1989.

6. The Front End to Simulator Interface, Electronics Research Laboratory, College of Engineering, University of California, Berkeley, CA, April, 1989.

7. The SPICE3 Implementation Guide, Thomas L. Quarles, Department of Electrical Engineering and Computer Sciences, University of California, Berkeley, CA, April, 1989.

8. Adding Devices to SPICE3, Thomas L. Quarles, Department of Electrical Engineering and Computer Sciences, University of California, Berkeley, CA, April, 1989.

9. The C Programming Language, Second Edition, Brian Kernighan and Dennis Ritchie, Prentice-Hall, Englewood Cliffs, NJ, 1988.

10. Software Design Document for the XSPICE Simulator of the Automatic Test Equipment Software Support Environment (ATESSE), F. L. Cox, W. B. Kuhn, H. W. Li, J. P. Murray, S. D. Tynor, Georgia Tech Research Institute, Atlanta, GA, September, 1992.

11. Interface Design Document for the XSPICE Code Model Subsystem of the Automatic Test Equipment Software Support Environment (ATESSE), F. L. Cox, W. B. Kuhn, H. W. Li, J. P. Murray, S. D. Tynor, Georgia Tech Research Institute, Atlanta, GA, September, 1992.

12. Software Design Document for the XSPICE Code Model Subsystem of the Automatic Test Equipment Software Support Environment (ATESSE), F. L. Cox, W. B. Kuhn, H. W. Li, J. P. Murray, S. D. Tynor, Georgia Tech Research Institute, Atlanta, GA, September, 1992.

13. Program Design Specification (Volumes 1 and 2) for the Automatic Test Equipment Software Support Environment (ATESSE), F. L. Cox, R. M. Ingle, J. E. Doss, G. T. Fulton, A. M. Gilchrist, R. W. Kearney, W. B. Kuhn, D. A. Moreland, P. P. Warren, B. D. Williams, Georgia Tech Research Institute, Atlanta, GA, October 1988.

14. Data Base Design Document for the Automatic Test Equipment Software Support Environment (ATESSE), F. L. Cox, R. M. Ingle, J. E. Doss, G. T. Fulton, A. M. Gilchrist, R. W. Kearney, W. B. Kuhn, D. A. Moreland, P. P. Warren, B. D. Williams, Georgia Tech Research Institute, Atlanta, GA, October 1988.

15. Analysis of Performance and Convergence Issues for Circuit Simulation, Thomas L. Quarles, Department of Electrical Engineering and Computer Sciences, University of California, Berkeley, CA, April, 1989.

16. SPICE2: A Computer Program to Simulate Semiconductor Circuits, Lawrence W. Nagel, Electronics Research Laboratory, College of Engineering, University of California, Berkeley, CA, May, 1975.

# 3 Interface Design

## 3.1 Interface Diagrams

Figure 3.1 illustrates the interfaces between the Simulator and the rest of the ATESSE system. For an explanation of this diagram and the associated interfaces and "Project-Unique Identifiers", refer to the Software Requirements Specification for the Simulator of the Automatic Test Equipment Software Support Environment (ATESSE) identified in the references.

## 3.2 Simulator Invocation

The simulator is invoked to execute with the ATESSE system using the following command:

```
atesse_xspice -ipc [BATCH | INTERACTIVE] <IPC connection name>
```

When using the ATESSE version 1 interface, the <IPC connection name> is the name of the mailbox over which the SPICE deck and results data will be transmitted. When using the ATESSE version 2 interface, <IPC connection name> specifies a Unix socket. Either BATCH or INTERACTIVE mode must be chosen. When invoked with BATCH, the simulator behaves as a batch simulator and writes its results to a file in the Batch Control Process Database rather than transmitting them over the IPC connection as it would if invoked as INTERACTIVE.

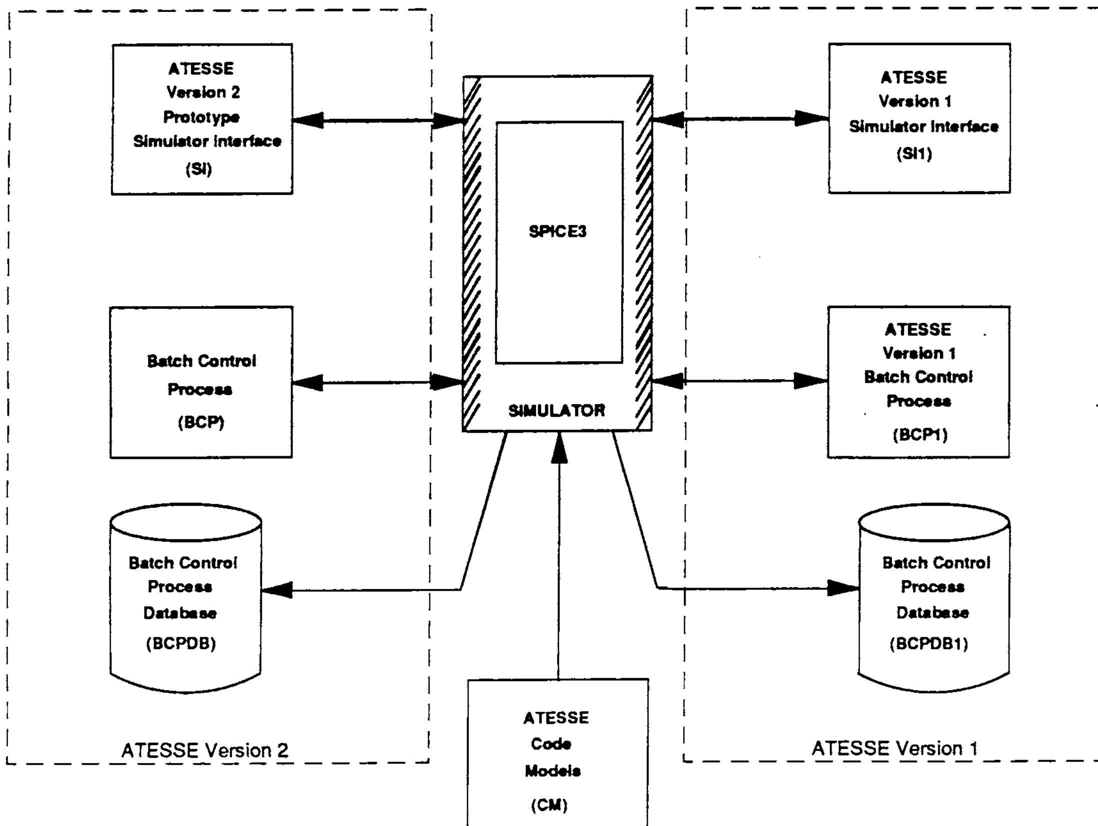


Figure 3.1 XSPICE Interface Diagram

For invoking the standalone version of the simulator, the following command is used from a shell:

```
xspice <filename>
```

Where <filename> is the name of the SPICE input deck to be run.

When a simulator executable is built, the user selects whether the simulator is to be created for use with the ATESSE system (atesse_xspice), or as a standalone version (xspice). The selection is made by the argument ("atesse_xspice" or "xspice", respectively) supplied to the "make" command in a simulator directory. The default, if no arguments are supplied to "make", is to create the standalone simulator.

## 3.3 Interprocess Communication

The "atesse_xspice" simulator executable includes interprocess communication (IPC) code for communicating with other processes in the ATESSE system. IPC is used for all inputs and outputs to and from the simulator in the ATESSE system. Hence, the "atesse_xspice" simulator is usually invoked as a background process and has no direct interaction with the user. All interaction is carried out through the ATESSE Simulator Interface process.

### 3.3.1 Aegis Mailboxes

The ATESSE version 1 IPC interface is implemented with Aegis mailboxes. A mailbox is a special type of file supported by the Apollo Domain/OS operating system and is used for two way communication between processes. The processes that use the mailbox are referred to as either servers (if they are responsible for creating the mailbox) or clients (if they are not). While a mailbox may have only one server, it may have more than one client. Each client talks to the server on a specific "channel" in the mailbox. The server determines how many channels are allowed.

There are two mailboxes used by the simulator:

- xspicesvr.mbx
- batchspice.mbx

The first mailbox resides in the `/user_temp` directory on the nodes running the ATESSE version 1 Simulator Interface (the Mentor Graphics MSPICE program). Hence, this mailbox file is present on every node in the ATESSE system network which is licensed to run the MSPICE software. The second mailbox file is located in `/user_temp` on any nodes on which a batch simulator is running.

### 3.3.2 Unix Sockets

The ATESSE version 2 IPC interface is implemented with Unix Sockets. Sockets are file-like abstractions which serve as a basis for flexible interprocess communication. There are three types of sockets: stream, datagram, and raw. A stream socket supports bidirectional, reliable, sequenced data flow with an interface very similar to that of Unix files and pipes. A datagram socket is used to broadcast messages. It is bidirectional, but is unsequenced and unreliable (users must provide their own error and redundancy checking). Finally, raw sockets are a low level interface to the underlying transport protocol. Their use is typically limited to implementing new communications protocols. The ATESSE version 2 IPC interface is designed to work with the ATESSE version 2 Simulator Interface program and is implemented with stream sockets since they are sequenced and reliable. Stream sockets, once created and connected, can be read from and written to with standard Unix I/O system calls (read() and write()).

### 3.3.3 IPC Message Protocol

The following protocol is used to sequence incoming SPICE cards and outgoing results and status codes.

```
The simulator creates the IPC connection (mailbox or socket)
     and waits for a client to connect to it.

Once connected to a client, the simulator reads the incoming
     text messages a line at a time.  The first message is
     a pathname to a results file.  Subsequent messages
     are individual SPICE cards or controls.  Except for the
     following cases, the simulator does not acknowledge the
     receipt of the messages and generates no return data.
     The following input cards are handled specially:

     >INQCON - Upon receipt of the >INQCON card, the
          simulator sends the >ABRTABL, >PAUSABL, and
          >KEEPABL messages.

     >PAUSE - Upon receipt of the >PAUSE card, the simulator
          suspends processing and waits for a >CONT card to
          signal that the simulation should be resumed.

     >STOP - Upon receipt of the >STOP card, the simulator
          aborts the simulation, sends the >ABORTED status
          message, and closes the IPC connection.

After receiving an >ENDNET card and processing the input
     deck, the simulator returns an #ERRCHK status code
     to indicate the results of circuit description (SPICE
     deck) error checking.
```

```
Simulation results data are then sent as they are computed.
     If the simulation is interactive, the data is
     transmitted over the IPC connection; if the simulation
     is batch, the data is written to the originating schematic's
     results database.

If the simulation is aborted, the simulator sends the
     >ABORTED result card and closes the IPC connection.

If the simulation is successfully completed, the simulator
     sends the >ENDANAL card and closes the IPC connection.
```

## 3.4 Simulator Inputs

Inputs to the simulator in the ATESSE environment consist of IPC control directives and the circuit description input deck as explained in the following sections.

### 3.4.1 IPC Data

The data passed via this interface is an augmented SPICE deck. The SPICE deck is a text file consisting of a number of lines, usually referred to as "cards" for historical reasons.

In addition to the standard cards (documented in the XSPICE User Manual) providing the circuit description, a number of IPC control-flow cards are recognized. These are listed in the Table 3.1 below.

Any other cards beginning with # or > are silently ignored. All other cards are assumed to be part of the circuit description (SPICE deck).

#### 3.4.1.1 ATESSE Version 1 Transport Protocol

The ATESSE version 1 interface is implemented with Aegis mailboxes. The Simulator is a server process and creates and monitors a well-known mailbox. Each incoming record is received as a separate Aegis mailbox record. The first card is expected to contain the name of the output data file (stripped of its file extension, .out) to which to write results and status information.

#### 3.4.1.2 ATESSE Version 2 Transport Protocol

The ATESSE version 2 interface is implemented with Unix sockets. The Simulator is a server process and creates and monitors a well-known socket. Once created, the socket is

| Control Word | Description |
|---|---|
| >INQCON | Request for configuration information from simulator. |
| >NETLIST | Marks beginning of SPICE netlist. |
| >ENDNET | Marks the end of SPICE netlist. |
| #RETURNI | Tells the simulator that currents should be monitored and returned for all components. |
| >PAUSE | Temporarily suspend an interactive simulation until a >CONT message word is received. |
| >CONT | Resume a paused interactive simulation. |
| >STOP | Abort a simulation in progress. |
| #VTRANS | Voltage source name translation information. Takes two string arguments which are the first five characters of the source name and the name to translate the source to. The sixth character in the source name is always a $ which identifies it as a source to be translated and the remaining two characters form a number which identifies the pin monitored by the source. |

Table 3.1 IPC Input Control Words.

monitored with ordinary Unix read() or fgets() function calls. Each incoming SPICE card is delimited with a newline (\n). The first card is expected to contain the name of the output data file (stripped of its file extension, .out) to which to write results and status information.

### 3.4.2 Circuit Description Syntax

The simulator's circuit description syntax is an extended version of Berkeley SPICE syntax. In particular, user-defined code models can be used in an XSPICE deck in a manner very similar to standard SPICE components. The subsections below specify the syntax used for instance and model cards in an input deck. The code model's allowed connections and parameters are specified in an Interface Specification (IFS) file. See the Interface Design Document for the Code Model Subsystem of the Automatic Test Equipment Software Support Environment (ATESSE) for details on the format of an IFS file.

#### 3.4.2.1 Parentheses

Parentheses may be used to clarify grouping of parameters or port connections. They are treated as whitespace by the XSPICE program and are completely ignored. They can be

particularly useful when specifying differential port connections; "(1 2) (3 4)" is easier to read than "1 2 3 4".

### 3.4.2.2 Vectors

Both instance ports and model parameter values may be scalars or vectors. Vectors are delimited by square brackets ("[" and "]"). Hence,

```
[1 2 3]
```

denotes a vector containing the values 1, 2, and 3, while

```
1 2 3
```

denotes three separate values 1, 2, and 3.

### 3.4.2.3 Instance Cards

Code model instance cards conform to the following format:

```
a<instance-name> <connections> <model-name>
```

that is, the instance name (which must begin with the letter "a"), a list of port connections (inputs and outputs), and the model name. The following sections describe the elements of an instance card in more detail.

#### 3.4.2.3.1 Instance Name

A code model instance's name is a (case-insensitive) alphanumeric string which is unique within the circuit. As specified above, a code model's instance name must start with the character "a". Hence, "a1", "Amp1", and "a22" are all valid model instance names.

#### 3.4.2.3.2 Connections

The connections specified on an instance card correspond to the respective ports in the IFS file. The ordering of connections on the instance card follows the order in which ports are defined in the IFS file.

The default type of a port specified in the IFS file can be overridden by preceding the port connection's node(s) with one of the type codes listed in Table 3.2. The "nodes" column

| Code | Description | Nodes | Direction |
|---|---|---|---|
| %v | single-ended voltage | 1 | in or out |
| %vd | differential voltage | 2 | in or out |
| %i | single-ended current | 1 | in or out |
| %id | differential current | 2 | in or out |
| %vnam | voltage source current | - | in |
| %g | single-ended voltage-controlled voltage source (VCCS) | 1 | inout |
| %gd | differential VCCS | 2 | inout |
| %h | single-ended current-controlled voltage source (CCVS) | 1 | inout |
| %hd | differential CCVS | 2 | inout |
| %d | digital | 1 | in, out, or inout |
| %<name> | user-defined type | 1 | in, out, or inout |

Table 3.2 Pre-defined Port Types.

specifies the number of nodes needed to specify the connection. As mentioned above, it is recommended that parentheses be used to delimit node pairs comprising a differential port connection.

A type code applies to the next port connection only. In the case of a vector port connection, the type code applies to each element within the vector.

#### 3.4.2.3.3 Model Name

The model name associates the model instance with a particular code model specified on a .MODEL card.

### 3.4.2.4 Model Cards

Code model cards conform to the following format:

```
.MODEL <model-name> <model-type> <parameters>
```

that is, a .MODEL keyword followed by the model name, the model type, and a list of parameters. The following sections describe the format and purpose of each of these fields.

#### 3.4.2.4.1 Model Name

The Model Name distinguishes the model from other models sharing the same functional definition, but having different parameters. For example, there may be several models which use the piecewise linear code model. The models may differ, however, in the values of their parameters.

#### 3.4.2.4.2 Model Type

The Model Type corresponds exactly to the SPICE_Model_Name: field in the code model's Interface Specification File. It specifies the model's functional definition.

#### 3.4.2.4.3 Parameters

The model's parameters are specified as name/value pairs. Each name and value are separated by an equal sign ("="). The name of each parameter must be one of the parameters specified in the IFS. The value of the parameter must be compatible with the definition in the IFS. That is, it must:

1. be of the proper type;
2. be a scalar or a vector, and if a vector must be of the proper size;
3. lie within the upper and lower limits (if specified).

If a parameter defined in the IFS has the property "Null_Allowed", then it is legal to omit that parameter on the .MODEL card. In this case, the parameter's value is set to the "Default_Value" specified in the IFS. If the parameter does not have a "Default_Value", then the value is undefined. If the parameter does not have the "Null_Allowed" property, it is illegal to omit it from the .MODEL card; an error will result.

## 3.5 Simulator Outputs

Outputs from the simulator consist of IPC control messages and the computed results data as explained in the following sections.

### 3.5.1 IPC Data

The data format used to return results and status information to the client process is identical to that described in the Data Base Design Document for the Automatic Test

Equipment Software Support Environment (ATESSE) for the ATESSE version 1. The result cards are summarized in Table 3.3.

| Control Word | Description |
|---|---|
| >ABRTABL | Configuration status information. Specifies that simulations can be aborted by the >STOP message word. |
| >PAUSABL | Configuration status information. Specifies that simulations can be paused by the >PAUSE message word. |
| >KEEPABL | Configuration status information. Specifies that voltage and current data can be selectively returned. |
| >DCOPB | Beginning of binary formatted DCOP data set. |
| >DATAB | Beginning of binary formatted results data set. Takes one argument which is the time or sweep point value for which the data was computed. |
| >ENDDCOP | End of DCOP data set. |
| >ENDDATA | End of results data set. |
| >ENDANAL | End of analysis (successful). Takes one argument which is the CPU time used. |
| >ABORTED | End of analysis (unsuccessful). Takes one argument which is the CPU time used. |
| #ERRCHK | Results of the topology error checking in the simulator. Takes one argument which may be one of the following: GO, NOGO. |

Table 3.3 IPC Output Control Words.

#### 3.5.1.1 ATESSE Version 1 Transport Protocol

The ATESSE version 1 interface is implemented with Aegis mailboxes. Output records are buffered and sent in chunks in Aegis mailbox messages. Each record is delimited within the buffered chunk with newlines (`\n`).

#### 3.5.1.2 ATESSE Version 2 Transport Protocol

The ATESSE version 2 interface is implemented with Unix sockets. Each record is delimited with a newline and is written to the socket with the standard Unix write() call.

### 3.5.2 Results Data

Tables 3.4 and 3.5 define the format for outputting results data from the simulator when operating in the ATESSE system. When the simulator is running under the direction of the Simulator Interface process for an interactive simulation, this results data is returned over the IPC channel. When operating under the direction of the ATESSE Batch Control Process for batch simulations, the data is written directly to the ".log" file whose pathname was sent to the simulator as the first record following opening of the IPC channel. Only IPC control words are returned to the Batch Control Process itself. Results data for the standalone version of the simulator is accessible through the SPICE3 "Nutmeg" interface and is documented in the Software User's Manual for the XSPICE Simulator of the Automatic Test Equipment Software Support Environment (ATESSE).

An example of results data returned by the simulator is given below (shown is ASCII for readability). Note that strings written to stderr or stdout by the simulator (including accounting information) or by code models may be interspersed between data blocks.

```
>ABRTABL
>PAUSABL
>KEEPABL
>DCOPB
1 5.3423E-3
2 6.9453E1
3 9.9982E1
10000 2.34598E1
V1 2.34523E-3
V2 3.42342E-3
D1 9.97543E-2
Q1 2.34983E-2
Q1 4.56921E-1
>ENDDCOP
>DATAB 0.000
1 5.3423E-3
2 6.9453E1
3 9.9982E1
10000 2.34598E1
V1 2.34523E-3
V2 3.42342E-3
D1 9.97543E-2
Q1 2.34983E-2
Q1 4.56921E-1
>ENDDATA
...
strings written to stdout and stderr
...
>DATAB 1.123E-9
1 5.3423E-3
2 6.9453E1
3 9.9982E1
```

```
10000 2.34598E1
V1 2.34523E-3
V2 3.42342E-3
D1 9.97543E-2
Q1 2.34983E-2
Q1 4.56921E-1
>ENDDATA
...
strings written to stdout and stderr
...
>ENDANAL 23.457
```

The data within a >DCOPB or >DATAB block contains the results of the initial DC operating point solution and is formatted in a mixed ASCII/binary format compatible with the format used by the ATESSE version 1 simulator. For Swept DC, AC, or Transient analysis types, similar mixed ASCII/binary messages from additional analysis points beyond the initial DCOP solution follow and are delimited by >DATA and >ENDDATA messages. In this case, the >DATA message contains a number (in ASCII) denoting an analysis time, DC voltage/current, or frequency.

For DC, Swept DC, or Transient analyses, each mixed ASCII/binary record between the delimiters consists of four logical fields as shown in Table 3.4: a length byte, a character string tag (a node or device name), a space, and a floating point number.

The length byte contains a byte count indicating the size of the record. It is equal to the decimal value of the ASCII letter "A", plus the length of the tag, plus 1 (for the space), plus the length of the binary data. The length byte and terminating newline of the message are not counted in the length. The length byte rather then the newline is used to locate the end of the message since the binary data could contain bytes with embedded ASCII newline values.

| Length | Node name | Space | Value |
|---|---|---|---|
| 1 byte | Length-5 bytes | 1 byte | 4 bytes |

Table 3.4 DC/Transient Data Record Format.

Similarly, for AC analyses, the data is packed as shown in Table 3.5. The binary data is a complex value and is stored with the real part first followed by the imaginary part.

The value fields in each of these tables contain floating point numbers in the Apollo Motorola 68000 4-byte floating point value format (a "float" in the C language).

For components which must report more than one current value (e.g., a transistor, which reports both collector current and base current), each value is reported using a separate value

| Length | Node name | Space | Real Value | Imag Value |
|---|---|---|---|---|
| 1 byte | Length-9 bytes | 1 byte | 4 bytes | 4 bytes |

Table 3.5 AC Data Record Format.

record as described above. The values are distinguished by the order in which they appear in the results. Table 3.6 describes the order of values in the results for each component type:

| Prefix | Component | Values |
|---|---|---|
| I | Current Source | current |
| C | Capacitor | current |
| L | Inductor | current |
| D | Diode | current |
| Q | BJT | collector current<br>base current |
| J | JFET | drain current<br>gate current |
| M | MOSFET | drain current<br>source current<br>bulk current |

Table 3.6 Current Ordering in Results Data.

## 3.6 Code Model Interface

The interface between the simulator and a code model is defined by an Interface Specification file and a Model Definition file for the code model. The formats for these files and the associated tools used to manipulate them are described in the Interface Design Document for the XSPICE Code Model Subsystem of the Automatic Test Equipment Software Support Environment (ATESSE), and the Software Design Document for the XSPICE Code Model Subsystem of the Automatic Test Equipment Software Support Environment (ATESSE).

### 3.6.1 Interface Specification File

The Interface Specification File is translated by the Code Model Preprocessor utility (cmpp) into a C language file (ifspec.c) which is compiled and linked into the XSPICE simulator executable. This section describes the format of that generated C file.

The ifspec.c file contains a number of local and global variables which, when linked into the XSPICE simulator executable, describe the interface to a particular code model. The types of these variables are defined in the file "MIF.h". The following sections will describe each of these types and show from where the values placed in the variables of these types are defined in the Interface Specification File.

#### 3.6.1.1 Types

A model is interfaced to the simulator core through a data structure of type "SPICEdev". The definition of this structure and its substructures is given below.

##### 3.6.1.1.1 SPICEdev

SPICEdev is a structure type defined in SPICE3C1 which defines the interface to a particular device. Its C definition is shown below. Note that not all of the SPICE3C1 function pointers are used in the XSPICE implementation of code models.

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

```
    int  (*DEVsenAcLoad)();   /* routine to load  the device ac sensitivity info*/
    void (*DEVsenPrint)();    /* routine to print out sensitivity info */
    int  (*DEVsenTrunc)();    /* routine to print out sensitivity info */
    int  DEVinstSize;         /* size of an instance struct */
    int  DEVmodSize;          /* size of a model struct */
 SPICEdev;
```

#### 3.6.1.1.2 IFdevice

IFdevice is a structure type defined in SPICE3C1 which, as a part of a SPICEdev variable, defines the interface to a particular device. This structure definition is augmented in the XSPICE simulator to allow it to contain the additional data required by XSPICE code models. Its C definition is:

```
typedef struct sIFdevice
    char *name;                          /* name of this type of device */
    char *description;                   /* description of this type of device */
    int terms;                           /* number of terminals on this device */
    int numNames;                        /* number of names in termNames */
    char **termNames;                    /* pointer to array of pointers to names
                                            array contains 'terms' pointers */
    int numInstanceParms;                /* number of instance parameter descriptors */
    IFparm *instanceParms;               /* array of instance parameter descriptors */
    int numModelParms;                   /* number of model param descriptors */
    IFparm *modelParms;                  /* array  of model param descriptors */
    void ((*cm_func)(Mif_Private_t *)); /* pointer to code model function */
    int num_conn;                        /* number of code model connections */
    Mif_Conn_Info_t  *conn;              /* array of connection info for mif parser */
    int num_param;                       /* number of parameters = numModelParms */
    Mif_Param_Info_t *param;             /* array of parameter info for mif parser */
    int num_inst_var;                    /* number of instance vars = numInstanceParms */
    Mif_Inst_Var_Info_t *inst_var;       /* array of instance var info for mif parser */
 IFdevice;
```

#### 3.6.1.1.3 Mif_Conn_Info_t

Mif_Conn_Info_t is a structure type which represents the port table information from the IFS file. Its C definition is:

```
typedef struct Mif_Conn_Info_s
    char             *name;               /* Name of this port */
    char             *description;        /* Description of this port */
    Mif_Dir_t        direction;           /* Is this port an input, output, or both? */
    Mif_Port_Type_t default_port_type;   /* The default port type */
    char             *default_type;       /* The default type in string form */
    int              num_allowed_types;   /* The size of the allowed type vector */
    Mif_Port_Type_t *allowed_type;       /* The allowed types */
```

```
    char            **allowed_type_str; /* The allowed types in string form */
    Mif_Boolean_t   is_array;           /* True if port is a vector */
    Mif_Boolean_t   has_lower_bound;    /* True if there is a vector size lower bound */
    int             lower_bound;        /* Vector size lower bound */
    Mif_Boolean_t   has_upper_bound;    /* True if there is an vector size upper bound */
    int             upper_bound;        /* Vector size upper bound */
    Mif_Boolean_t   null_allowed;       /* True if null is allowed for this port */
 Mif_Conn_Info_t;
```

#### 3.6.1.1.4 Mif_Param_Info_t

Mif_Param_Info_t is a structure type which represents the parameter table information from the IFS file. Its C definition is:

```
typedef struct Mif_Param_Info_s
    char               *name;            /* Name of parameter */
    char               *description;     /* Description of this parameter */
    Mif_Data_Type_t    type;             /* Is this a real, boolean, string, ... */
    Mif_Boolean_t      has_default;      /* True if there is a default value */
    Mif_Parse_Value_t  default_value;    /* The default value */
    Mif_Boolean_t      has_lower_limit;  /* True if there is a lower limit */
    Mif_Parse_Value_t  lower_limit;      /* The lower limit for this parameter */
    Mif_Boolean_t      has_upper_limit;  /* True if there is an upper limit */
    Mif_Parse_Value_t  upper_limit;      /* The upper limit for this parameter */
    Mif_Boolean_t      is_array;         /* True if parameter is a vector */
    Mif_Boolean_t      has_conn_ref;     /* True if parameter is associated with a port */
    int                conn_ref;         /* The subscript of the associated port */
    Mif_Boolean_t      has_lower_bound;  /* True if there is a vector size lower bound */
    int                lower_bound;      /* Vector size lower bound */
    Mif_Boolean_t      has_upper_bound;  /* True if there is a vector size upper bound */
    int                upper_bound;      /* Vector size upper bound */
    Mif_Boolean_t      null_allowed;     /* True if null is allowed for this parameter */
 Mif_Param_Info_t;
```

#### 3.6.1.1.5 Mif_Inst_Var_Info_t

Mif_Inst_Var_Info_t is a structure type which represents the static variable table information in the IFS file. Its C definition is:

```
typedef struct Mif_Inst_Var_Info_s
    char               *name;         /* Name of this static var */
    char               *description;  /* Description of this static var */
    Mif_Data_Type_t    type;          /* Is this a real, boolean, string, ...  */
    Mif_Boolean_t      is_array;      /* Is static var is a vector? */
 Mif_Inst_Var_Info_t;
```

#### 3.6.1.1.6 Mif_Port_Type_t

Mif_Port_Type_t is an enumerated type defining the type of data passed in a port. Its C definition is:

```
typedef enum
    MIF_VOLTAGE,            /* v - Single-ended voltage */
    MIF_DIFF_VOLTAGE,       /* vd - Differential voltage */
    MIF_CURRENT,            /* i - Single-ended current */
    MIF_DIFF_CURRENT,       /* id - Differential current */
    MIF_CONDUCTANCE,        /* g - Single-ended VCCS */
    MIF_DIFF_CONDUCTANCE,   /* gd - Differential VCCS */
    MIF_RESISTANCE,         /* h - Single-ended CCVS */
    MIF_DIFF_RESISTANCE,    /* hd - Differential CCVS */
    MIF_DIGITAL,            /* d - Digital */
    MIF_USER_DEFINED,       /* <identifier> - user defined type */
 Mif_Port_Type_t;
```

#### 3.6.1.1.7 IFparm

The structure IFparm is defined in SPICE3C1 and is used to describe both ports and parameters. Its C definition is:

```
typedef struct
     char *keyword;       /* name of this parameter */
     int id;              /* integer identifier for fast compares */
     int dataType;        /* type of data */
     char *description;   /* description of data */
 IFparam;
```

The keyword field is the name of the port or parameter. The id is an index which is used to identify the port or parameter. The dataType field indicates the type of data associated with the port or parameter. It is a bit mask constructed by ORing the following SPICE3C1 constants: `IF_REAL, IF_INTEGER, IF_COMPLEX, IF_STRING, IF_VECTOR`. Thus a scalar integer dataType field is specified by `IF_INTEGER` while an integer vector's is specified as (`IF_INTEGER | IF_VECTOR`). The description string is a short one-line description of the purpose of the parameter or port.

### 3.6.1.2 Data Structures and Variables

The following subsections describe the global variables and data structures used to interface code models with the simulator. The global variables DEVmaxnum and DEVices are instantiated in the compilation of file SPIinit.c when a new simulator is built. The remaining variables are defined in each code model's ifspec.c file which is generated from the model's Interface Specification file (ifspec.ifs) by the Code Model Preprocessor utility.

#### 3.6.1.2.1 DEVmaxnum

This is a global integer variable of type "int" defined in SPICE3 that defines the number of devices known to the simulator executable. A device corresponds to either one of SPICE3's internal device types (resistor, capacitor, diode, BJT, MOS1, etc.), or to an XSPICE code model.

This variable is allocated and initialized when CKT/SPIinit.c is compiled during the building of the xspice or atesse_xspice executable.

#### 3.6.1.2.2 DEVices

This is a global variable of type `SPICEdev*` defined in SPICE3. It contains an array of pointers to structures of type SPICEdev that allow the simulator core to access data and functions specific to a device (SPICE3 device or XSPICE code model).

This variable is allocated and initialized when CKT/SPIinit.c is compiled during the building of the xspice or atesse_xspice executable.

#### 3.6.1.2.3 XXX_info

This is the SPICEdev structure defined for the code model. Its name corresponds to the code model's C function name with an "_info" suffix.

This structure is generated automatically by the Code Model Preprocessor in the file "ifspec.c" based on the information supplied in the code model's Interface Specification file (ifspec.ifs). The XXX_info structure contains a number of substructures, each of which is discussed in the following subsections. An example of the code generated by the Code Model Preprocessor is shown below. The XXX_info structure appears at the end of the file, following the definition of the included substructures.

```
/*
 * Structures for model: new_mod
 *
 * Automatically generated by cmpp preprocessor
 *
 * !!! DO NOT EDIT !!!
 *
 */


#include "prefix.h"
#include <stdio.h>
#include "DEVdefs.h"
#include "IFsim.h"
#include "MIFdefs.h"
```

```
#include "MIFproto.h"
#include "MIFparse.h"
#include "suffix.h"


static IFparm MIFmPTable[] = {
    IOP("mode", 0, IF_STRING, "lpf, breakpoints, or errors"),
    IOP("gain", 1, IF_REAL, "Gain at DC"),
    IOP("cutoff_freq", 2, IF_REAL, "3db cutoff frequency in Hz"),
    IOP("temp_coeff", 3, IF_REAL, "temperature coefficient of cutoff freq (per deg C)"),
    IOP("debug", 4, IF_FLAG, "Print debug info switch"),
};


static IFparm MIFpTable[] = {
    OP("w0", 5, IF_REAL, "Radian frequency"),
    OP("breakpoints", 6, IF_INTEGER, "count of breakpoints hit"),
};


static Mif_Port_Type_t MIFportEnum0[] = {
MIF_VOLTAGE,
MIF_CURRENT,
MIF_DIFF_VOLTAGE,
MIF_DIFF_CURRENT,
MIF_VSOURCE_CURRENT,
};


static char *MIFportStr0[] = {
"v",
"i",
"vd",
"id",
"vnam",
};


static Mif_Port_Type_t MIFportEnum1[] = {
MIF_VOLTAGE,
MIF_CURRENT,
MIF_DIFF_VOLTAGE,
MIF_DIFF_CURRENT,
};


static char *MIFportStr1[] = {
"v",
"i",
"vd",
"id",
};
```

```
static Mif_Conn_Info_t MIFconnTable[] = {
  {
    "in",
    "input",
    MIF_IN,
    MIF_VOLTAGE,
    "v",
    5,
    MIFportEnum0,
    MIFportStr0,
    MIF_FALSE,
    MIF_FALSE,
    0,
    MIF_FALSE,
    0,
    MIF_FALSE,
  },
  {
    "out",
    "output",
    MIF_OUT,
    MIF_VOLTAGE,
    "v",
    4,
    MIFportEnum1,
    MIFportStr1,
    MIF_FALSE,
    MIF_FALSE,
    0,
    MIF_FALSE,
    0,
    MIF_FALSE,
  },
};


static Mif_Param_Info_t MIFparamTable[] = {
  {
    "mode",
    "lpf, breakpoints, or errors",
    MIF_STRING,
    MIF_FALSE,
    {MIF_FALSE, 0, 0.0, {0.0, 0.0}, NULL},
    MIF_FALSE,
    {MIF_FALSE, 0, 0.0, {0.0, 0.0}, NULL},
    MIF_FALSE,
    {MIF_FALSE, 0, 0.0, {0.0, 0.0}, NULL},
    MIF_FALSE,
    MIF_FALSE,
    0,
```

```
    MIF_FALSE,
    0,
    MIF_FALSE,
    0,
    MIF_FALSE,
  },
  {
    "gain",
    "Gain at DC",
    MIF_REAL,
    MIF_TRUE,
    {MIF_FALSE, 0, 1.000000e+00, {0.0, 0.0}, NULL},
    MIF_FALSE,
    {MIF_FALSE, 0, 0.0, {0.0, 0.0}, NULL},
    MIF_FALSE,
    {MIF_FALSE, 0, 0.0, {0.0, 0.0}, NULL},
    MIF_FALSE,
    MIF_FALSE,
    0,
    MIF_FALSE,
    0,
    MIF_FALSE,
    0,
    MIF_TRUE,
  },
  {
    "cutoff_freq",
    "3db cutoff frequency in Hz",
    MIF_REAL,
    MIF_FALSE,
    {MIF_FALSE, 0, 0.0, {0.0, 0.0}, NULL},
    MIF_TRUE,
    {MIF_FALSE, 0, 1.000000e-10, {0.0, 0.0}, NULL},
    MIF_FALSE,
    {MIF_FALSE, 0, 0.0, {0.0, 0.0}, NULL},
    MIF_FALSE,
    MIF_FALSE,
    0,
    MIF_FALSE,
    0,
    MIF_FALSE,
    0,
    MIF_FALSE,
  },
  {
    "temp_coeff",
    "temperature coefficient of cutoff freq (per deg C)",
    MIF_REAL,
    MIF_TRUE,
    {MIF_FALSE, 0, 0.000000e+00, {0.0, 0.0}, NULL},
    MIF_FALSE,
    {MIF_FALSE, 0, 0.0, {0.0, 0.0}, NULL},
```

```
    MIF_FALSE,
    {MIF_FALSE, 0, 0.0, {0.0, 0.0}, NULL},
    MIF_FALSE,
    MIF_FALSE,
    0,
    MIF_FALSE,
    0,
    MIF_FALSE,
    0,
    MIF_TRUE,
  },
  {
    "debug",
    "Print debug info switch",
    MIF_BOOLEAN,
    MIF_TRUE,
    {MIF_FALSE, 0, 0.0, {0.0, 0.0}, NULL},
    MIF_FALSE,
    {MIF_FALSE, 0, 0.0, {0.0, 0.0}, NULL},
    MIF_FALSE,
    {MIF_FALSE, 0, 0.0, {0.0, 0.0}, NULL},
    MIF_FALSE,
    MIF_FALSE,
    0,
    MIF_FALSE,
    0,
    MIF_FALSE,
    0,
    MIF_TRUE,
  },
};


static Mif_Inst_Var_Info_t MIFinst_varTable[] = {
  {
    "w0",
    "Radian frequency",
    MIF_REAL,
    MIF_FALSE,
  },
  {
    "breakpoints",
    "count of breakpoints hit",
    MIF_INTEGER,
    MIF_FALSE,
  },
};


extern void ucm_new_mod(Mif_Private_t *);

SPICEdev ucm_new_mod_info = {
```

```
    { "new_mod",
      "Low Pass Filter and Breakpoint Demonstrator",
      0,
      0,
      NULL,
      2,
      MIFpTable,
      5,
      MIFmPTable,
      ucm_new_mod,
      2,
      MIFconnTable,
      5,
      MIFparamTable,
      2,
      MIFinst_varTable,
    },
NULL,
MIFmParam,
MIFload,
MIFsetup,
NULL,
NULL,
MIFtrunc,
NULL,
MIFload,
NULL,
MIFdestroy,
MIFmDelete,
MIFdelete,
NULL,
MIFask,
MIFmAsk,
NULL,
MIFconvTest,
NULL,
NULL,
NULL,
NULL,
NULL,
NULL,
sizeof(MIFinstance),
sizeof(MIFmodel),

};
```

#### 3.6.1.2.4 MIFmPTable

The MIFmPTable is part of the original SPICE3C1 definition of structure type IFdevice within the SPICEdev structure. MIFmPTable is a static array of IFparam structures defining information about parameters that may be included on a .model card in a SPICE deck.

Each element of the MIFmPTable variable corresponds to one of the parameters in the IFS file. These are parameters supplied on a ".model" card in an XSPICE input deck. A macro defined in SPICE3C1 (IOP) is used to define these parameters as "input/output", meaning that they can be both set (by supplying them on a .model card), and queried (by using the SPICE3 ".save" command).

Table 3.7 indicates which fields in the IFS file are used by the Code Model Preprocessor in building this structure.

| C Structure Field | IFS File Field(s) |
|---|---|
| keyword | Parameter_Name: |
| id | <none> - generated by cmpp |
| dataType | Data_Type: + Array: |
| description | Description: |

Table 3.7 Origin of Data Held in MIFmPTable.

#### 3.6.1.2.5 MIFpTable

The MIFpTable is part of the original SPICE3C1 definition of structure type IFdevice within the SPICEdev structure. In SPICE3C1, the MIFpTable is a static array of IFparam structures defining information about parameters that are defined on an instance-by-instance basis. The XSPICE simulator uses this structure array for Static Variables defined in a code model's Interface Specification file. A macro defined in SPICE3C1 (OP) is used to define these parameters as "output", meaning that they can be queried by the SPICE3 ".save" command but they cannot be set in the XSPICE input deck (although they may, of course, be set by the model).

Table 3.8 indicates which fields in the IFS file are used by the Code Model Preprocessor in building this structure.

#### 3.6.1.2.6 <Code Model Function Name>

This is a new member of the IFdevice structure type introduced in XSPICE to hold a pointer to the function that defines the code model's behavior. The code model function is

| C Structure Field | IFS File Field(s) |
|---|---|
| keyword | Static_Var_Name: |
| id | <none> - generated by cmpp |
| dataType | Allowed_Types: + Array: |
| description | Description: |

Table 3.8 Origin of Data Held in MIFpTable.

defined in the Model Definition file (cfunc.mod) and its translated form (cfunc.c) associated with the code model.

The code model function name is read from the "C_Function_Name:" field of the Name Table in the Interface Specification file.

#### 3.6.1.2.7 MIFconnTable

This is a new member of the IFdevice structure type introduced in XSPICE to specify data used to parse connections to ports on a code model. It is a static array of Mif_Conn_Info_t structures.

Table 3.9 indicates which fields in the IFS file are used by the Code Model Preprocessor in building this structure.

#### 3.6.1.2.8 MIFparamTable

This is a new member of the IFdevice structure type introduced in XSPICE to specify data used to parse code model parameters supplied on a .model card. It is a static array of Mif_param_info_t structures.

Table 3.10 indicates which fields in the IFS file are used by the Code Model Preprocessor in building this structure.

#### 3.6.1.2.9 MIFinst_varTable

This is a new member of the IFdevice structure type introduced in XSPICE to hold the definition of Static Variables associated with a code model. It is a static array of Mif_inst_var_info_t structures.

Table 3.11 indicates which fields in the IFS file are used by the Code Model Preprocessor in building this structure.

| C Structure Field | IFS File Field(s) |
|---|---|
| name | Port_Name: |
| description | Description: |
| direction | Direction: |
| default_port_type | Default_Type: |
| default_type | Default_Type: |
| allowed_type | Allowed_Types: |
| num_allowed_types | Allowed_Types: |
| allowed_types_str | Allowed_Types: |
| is_array | Array: |
| lower_bound | Array_Bounds: |
| has_lower_bound | Array_Bounds: |
| upper_bound | Array_Bounds: |
| has_upper_bound | Array_Bounds: |
| null_allowed | Null_Allowed: |

Table 3.9 Origin of Data Held in MIFconnTable.

### 3.6.2 Model Definition File

The Model Definition file (cfunc.mod) is translated by the Code Model Preprocessor (cmpp) into a C language file (cfunc.c) which is compiled and linked into the XSPICE simulator executable. The translation involves mapping special XSPICE "accessor macros" into appropriate C language code referencing data in the code model's C function argument list.

The format of the Model Definition file is described in the Interface Design Document for the XSPICE Code Model Subsystem for the Automatic Test Equipment Software Support Environment (ATESSE). Each of the accessor macros is translated as described in the following sections.

### 3.6.3 Macro Translations

The Model Definition file uses "accessor macros" to insulate the user from the underlying C language structure definitions used to pass data to and from a model. These macros are translated by the Code Model Preprocessor (cmpp) utility as shown in Table 3.12 when the model is preprocessed. The names defined in a model's Interface Specification file and used as arguments to certain macros are translated by the preprocessor to an appropriate array index as determined by the order in which the names are defined in the Interface Specification.

| C Structure Field | IFS File Field(s) |
|---|---|
| name | Parameter_Name: |
| description | Description: |
| type | Data_Type: |
| has_default | Default: |
| default_value | Default_Value: |
| has_lower_limit | Limits: |
| lower_limit | Limits: |
| has_upper_limit | Limits: |
| upper_limit | Limits: |
| is_array | Array: |
| has_lower_bound | Array_Bounds: |
| lower_bound | Array_Bounds: |
| has_upper_bounds | Array_Bounds: |
| has_conn_ref | Array_Bounds: |
| conn_ref | Array_Bounds: |
| null_allowed | Null_Allowed: |

Table 3.10 Origin of Data Held in MIFparamTable.

## 3.7 User-Defined Node Interface

The interface between the XSPICE simulator and user-defined nodes is defined by a C source file, called the "Node Definition File". This C source file is edited by the user to describe the user-defined node data structure and functionality. Note that unlike the Model Definition File used in developing code models, the Node Definition File uses pure C macros and is not translated by the Code Model Preprocessor.

| C Structure Field | IFS File Field(s) |
|---|---|
| name | Static_Var_Name: |
| description | Description: |
| type | Data_Type: |
| is_array | <none> |

Table 3.11 Origin of Data Held in MIFinst_varTable.

| Macro | Translation |
|---|---|
| AC_GAIN(x,y) | private->conn[x]->port[xsub]->ac_gain[y].port[ysub] |
| ANALYSIS | private->circuit.anal_type |
| ARGS | Mif_Private_t *private |
| CALL_TYPE | private->circuit.call_type |
| INIT | private->circuit.init |
| INPUT(x) | <context dependent, depending on port type:><br>private->conn[x]->port[xsub]->input.rvalue<br>private->conn[x]->port[xsub]->input.pvalue |
| INPUT_STATE(x) | ((Digital_t*)(private->conn[x]->port[xsub]->input.pvalue))->state |
| INPUT_STRENGTH(x) | ((Digital_t*)(private->conn[x]->port[xsub]->input.pvalue))->strength |
| INPUT_TYPE(x) | private->conn[x]->port[xsub]->type_str |
| LOAD(x) | private->conn[x]->port[xsub]->load |
| MESSAGE(x) | private->conn[x]->port[xsub]->msg |
| NEW_TIMEPOINT | private->circuit.anal_init |
| OUTPUT(x) | <context dependent, depending on port type:><br>private->conn[x]->port[xsub]->output.rvalue<br>private->conn[x]->port[xsub]->output.pvalue |
| OUTPUT_CHANGED(x) | private->conn[x]->port[xsub]->changed |
| OUTPUT_DELAY(x) | private->conn[x]->port[xsub]->delay |
| OUTPUT_STATE(x) | ((Digital_t*)(private->conn[x]->port[xsub]->output.pvalue))->state |
| OUTPUT_STRENGTH(x) | ((Digital_t*)(private->conn[x]->port[xsub]->output.pvalue))->strength |
| OUTPUT_TYPE(x) | private->conn[x]->port[xsub]->type_str |
| PARAM(x) | <context dependent, depending on param type:><br>private->param[x]->element[xsub].ivalue<br>private->param[x]->element[xsub].dvalue<br>private->param[x]->element[xsub].cvalue<br>private->param[x]->element[xsub].bvalue<br>private->param[x]->element[xsub].svalue |
| PARAM_NULL(x) | private->param[x]->is_null |
| PARAM_SIZE(x) | private->param[x]->size |
| PARTIAL(x,y) | private->conn[x]->port[xsub]->partial[y].port[ysub] |
| PORT_NULL(x) | private->conn[x]->is_null |
| PORT_SIZE(x) | private->conn[x]->size |
| RAD_FREQ | private->circuit.frequency |
| STATIC_VAR(x) | <context dependent, depending on static var type:> |
| | private->inst_var[x]->element[xsub].ivalue<br>private->inst_var[x]->element[xsub].dvalue<br>private->inst_var[x]->element[xsub].cvalue<br>private->inst_var[x]->element[xsub].bvalue<br>private->inst_var[x]->element[xsub].svalue<br>private->inst_var[x]->element[xsub].pvalue |
| STATIC_VAR_SIZE(x) | private->inst_var[x]->size |
| T[x] | private->circuit.t[x] |
| TEMPERATURE | private->circuit.temperature |
| TIME | private->circuit.time |
| TOTAL_LOAD(x) | private->conn[x]->port[xsub]->total_load |

Table 3.12 Code Model Macro Translations.

### 3.7.1 Node Definition File

The Node Definition File is created as "udnfunc.c" by the User-Defined Node Directory Generator utility and is edited by the user. This file defines "User-Defined Nodes" that allow a user to perform event-driven simulation with arbitrary data structures. These nodes are integrated into the simulator similar to the manner in which models are linked into SPICE 3C1, so that new node types can be relatively easily added. A special data structure ("Evt_Udn_Info_t") holds pointers to functions that define operations on user-defined node data structures.

#### 3.7.1.1 Types

A user-defined node type is interfaced to the simulator core through a data structure of type "Evt_Udn_Info_t". The definition of this structure is given below.

##### 3.7.1.1.1 Evt_Udn_Info_t

```
typedef struct
    char        *name;
    char        *description;
    void        ((*create)(CREATE_ARGS));
    void        ((*dismantle)(DISMANTLE_ARGS));
    void        ((*initialize)(INITIALIZE_ARGS));
    void        ((*invert)(INVERT_ARGS));
    void        ((*copy)(COPY_ARGS));
    void        ((*resolve)(RESOLVE_ARGS));
    void        ((*compare)(COMPARE_ARGS));
    void        ((*plot_val)(PLOT_VAL_ARGS));
    void        ((*print_val)(PRINT_VAL_ARGS));
    void        ((*ipc_val)(IPC_VAL_ARGS));
 Evt_Udn_Info_t;
```

This data structure holds name and description strings for the user-defined node, plus pointers to the functions (required and optional) that define operations on it. The functions called through this data structure are listed below. Optional functions can be left undefined, in which cases the pointer placed into the Evt_Udn_Info_t structure would be specified as NULL. The function arguments are defined in terms of macros, which are explained in a subsequent section.

**Required functions:**

| | |
|---|---|
| create | Allocate data structure used as inputs and outputs to code models. |
| initialize | Set structure to appropriate initial value for first use as model input. |
| copy | Make a copy of the contents into created but possibly uninitialized structure. |
| compare | Determine if two structures are equal in value. |

**Optional functions:**

| | |
|---|---|
| dismantle | Free allocations inside structure (but not structure itself). |
| invert | Invert logical value of structure. |
| resolve | Determine the resultant when multiple outputs are connected to a node. |
| plot_val | Output a real value for specified structure component for plotting purposes. |
| print_val | Output a string value for specified structure component for printing. |
| ipc_val | Output a binary representation of the data and an int giving its size. |

#### 3.7.1.2 Variables

The following subsections describe the global variables and data structures used to interface user-defined nodes with the simulator. The global variables g_evt_num_udn_types and g_evt_udn_info are instantiated in the compilation of file SPIinit.c when a new simulator is built. The remaining variables are defined in each user-defined node's udnfunc.c file which is edited by the user to define the node type.

##### 3.7.1.2.1 g_evt_num_udn_types

This variable is a global of type "int", and defines the number of user-defined node types included in the XSPICE simulator executable when it is built.

This variable is allocated and initialized when CKT/SPIinit.c is compiled during the building of the XSPICE executable.

##### 3.7.1.2.2 g_evt_udn_info

This variable is a global array of type "Evt_Udn_Info_t" and contains pointers to the "udn_XXX_info" structures associated with the user-defined node types included in the simulator executable.

This variable is allocated and initialized when CKT/SPIinit.c is compiled during the building of the XSPICE executable.

#### 3.7.1.2.3 udn_XXX_info

Each user-defined node type included in the simulator executable is described by a data structure of type "Evt_Udn_Info_t" as described above. The individual instantiations of this structure type are named udn_XXX_info, where "XXX" is the name of the user-defined node type given by the user when a user-defined node directory is created with "mkudndir".

This variable is allocated and initialized when CKT/SPIinit.c is compiled during the building of the XSPICE executable.

### 3.7.2 Macro Translations

User-Defined Node Definition Files use macros similar to those used in defining models. However, unlike the "accessor macros" used by models, the macros used with user-defined nodes are true C language macros and are translated by the standard C preprocessor.

The macro translations shown in Table 3.13 are defined in the "EVTudn.h" header file for use with user-defined nodes. The group in the first half of the table defines the translation of macros used in function argument lists. The following group defines the translation of macros used within functions to reference data in the argument list.

| Macro | Translation |
|---|---|
| CREATE_ARGS | void **evt_struct_ptr |
| INITIALIZE_ARGS | void *evt_struct_ptr |
| COMPARE_ARGS | void *evt_struct_ptr_1,<br>void *evt_struct_ptr_2,<br>Mif_Boolean_t *evt_equal |
| COPY_ARGS | void *evt_input_struct_ptr,<br>void *evt_output_struct_ptr |
| DISMANTLE_ARGS | void *evt_struct_ptr |
| INVERT_ARGS | void *evt_struct_ptr |
| RESOLVE_ARGS | int evt_input_struct_ptr_array_size,<br>void **evt_input_struct_ptr_array,<br>void *evt_output_struct_ptr |
| PLOT_VAL_ARGS | void *evt_struct_ptr,<br>char *evt_struct_member_id,<br>double *evt_plot_val |
| PRINT_VAL_ARGS | void *evt_struct_ptr,<br>char *evt_struct_member_id,<br>char **evt_print_val |
| IPC_VAL_ARGS | void *evt_struct_ptr<br>void **evt_ipc_val<br>int *evt_ipc_val_size |
| MALLOCED_PTR | (*evt_struct_ptr) |
| STRUCT_PTR | evt_struct_ptr |
| STRUCT_PTR_1 | evt_struct_ptr_1 |
| STRUCT_PTR_2 | evt_struct_ptr_2 |
| EQUAL | (*evt_equal) |
| INPUT_STRUCT_PTR | evt_input_struct_ptr |
| OUTPUT_STRUCT_PTR | evt_output_struct_ptr |
| INPUT_STRUCT_PTR_ARRAY | evt_input_struct_ptr_array |
| INPUT_STRUCT_PTR_ARRAY_SIZE | evt_input_struct_ptr_array_size |
| STRUCT_MEMBER_ID | evt_struct_member_id |
| PLOT_VAL | (*evt_plot_val) |
| PRINT_VAL | (*evt_print_val) |
| IPC_VAL | (*evt_ipc_val) |
| IPC_VAL_SIZE | (*evt_ipc_val_size) |

Table 3.13 User-Defined Node Macro Translations.

# 4 Notes

## 4.1 Glossary

**ATESSE** — Automatic Test Equipment Software Support Environment. An integrated set of software tools designed to aid in development of programs for testing mixed-mode (analog/digital) printed circuit cards.

**C** — A programming language developed in the early 70's at Bell Labs. It is the standard programming language used for system development on UNIX machines.

**Code Model Library** — The set of code models supplied with the ATESSE simulator.

**Code Model Preprocessor** — A code model development tool that assists in adding new code models to the ATESSE simulator. It is automatically run by "make" to convert user-written files into C language source files that are compiled and linked with the simulator.

**Code Model Toolset** — A set of tools that assist in adding new code models to the ATESSE simulator.

**Interprocess Communication** — Communication between two separate programs running concurrently on a one or more computers.

**lex** — A UNIX operating system utility used to develop lexical analyzers (scanners).

**Make** — A UNIX software development tool that automates the compilation and linking of a program.

**Makefile** — A file that instructs the UNIX make utility how to compile and link a program.

**malloc** — A standard C library function for dynamically allocating memory in a program.

**Model Directory Generator** — A code model development tool that assists in adding new code models to the ATESSE simulator. It creates a directory in which a code model will be created.

**Simulator Directory Generator** — A code model development tool that assists in adding new code models to the ATESSE simulator. It creates a directory in which a copy of the simulator will be built.

**SPICE** — An analog simulation program developed at the University of California at Berkeley.

**SPICE3** — Version 3 of the SPICE simulator.

**XSPICE** — Extended SPICE program developed at the Georgia Tech Research Institute. XSPICE is based on SPICE3 from the University of California at Berkeley.

**yacc** — Yet Another Compiler Compiler. A UNIX operating system utility used in developing language parsers.

## 4.2 Acronyms

| | |
|---|---|
| **ANSI** | American National Standards Institute. |
| **ATE** | Automatic Test Equipment. |
| **ATESSE** | Automatic Test Equipment Software Support Environment. |
| **BJT** | Bipolar Junction Transistor. |
| **CCVS** | Current-Controlled Voltage Source. |
| **CDRL** | Contract Deliverable Requirement List. |
| **CPU** | Central Processing Unit. |
| **CSCI** | Computer Software Configuration Item. |
| **IPC** | Interprocess Communication. |
| **JFET** | Junction Field-Effect Transistor. |
| **MNA** | Modified Nodal Analysis. |
| **MOSFET** | Metal-Oxide-Semiconductor Field-Effect Transistor. |
| **PUI** | Project Unique Identifier. |
| **PWL** | Piecewise-Linear. |
| **SPICE** | Simulation Program with Integrated Circuit Emphasis. |
| **SPICE3C1** | Version 3C1 of the SPICE program. |
| **VCCS** | Voltage-Controlled Current Source. |
| **XSPICE** | Extended SPICE. |

## 4.3 Project Unique Identifiers

**SI-SIM-IN** Output of the Simulator (SIM) to the version 2 Simulator Interface (SI).

**SI-SIM-OUT** Input to the Simulator (SIM) from the version 2 Simulator Interface (SI).

**SI1-SIM-IN** Output of the Simulator (SIM) to the version 1 Simulator Interface (SI).

**SI1-SIM-OUT** Input to the Simulator (SIM) from the version 1 Simulator Interface (SI).

**SIM-BCP-IN** Input to the Simulator (SIM) from the version 2 Batch Control Process (BCP).

**SIM-BCP-OUT** Output from the Simulator (SIM) to the version 2 Batch Control Process (BCP).

**SIM-BCP1-IN** Input to the Simulator (SIM) from the version 1 Batch Control Process (BCP).

**SIM-BCP1-OUT** Output from the Simulator (SIM) to the version 1 Batch Control Process (BCP).

**SIM-BCPDB-IN** Input to the Simulator (SIM) from the version 2 Batch Control Process Database (BCPDB).

**SIM-BCPDB-OUT** Output from the Simulator (SIM) to the version 2 Batch Control Process Database (BCPDB).

**SIM-BCPDB1-IN** Input to the Simulator (SIM) from the version 1 Batch Control Process Database (BCPDB).

**SIM-BCPDB1-OUT** Output from the Simulator (SIM) to the version 1 Batch Control Process Database (BCPDB).

**SIM-CM-IN** Input to the Simulator (SIM) from the Code Model Subsystem (CM).