9100 Series

Programmer's
Manual

FLUKE ®

CUSTOMER NOTICE

THROUGHOUT THIS MANUAL, ALL INSTANCES OF 9100A
AND 9105A ALSO APPLY TO THE 9100FT AND 9105FT.

Contents

Figures

Figure	Title	Page

Where Am I?

Getting Started		A description of the parts of the 9100A/9105A, what they do, how to connect them, and how to power up.
Automated Operations Manual		How to run pre-programmed test or troubleshooting procedures.
Technical User's Manual		How to use the 9100A/9105A keypad to test and troubleshoot your Unit Under Test (UUT).
Applications Manual		How to design test or troubleshooting procedures for your Unit Under Test (UUT).
Programmer's Manual	*You Are Here*	How to use the programming station with the 9100A to create automated test or troubleshooting procedures.
TL/1 Reference Manual		A description of all TL/1 commands arranged in alphabetical order for quick reference.
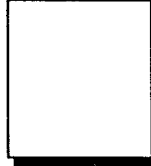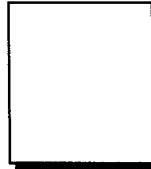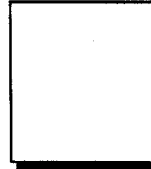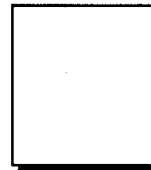
Section 1
Overview

By writing TL/1 programs, you can integrate a wide range of operations that use and expand on the built-in functions of the 9100A/9105A. The amount of time you invest in creating programs pays off in increased efficiency for test and trouble-shooting operations. This manual describes 9100A/9105A programs and how to create them using the programmer's interface on a 9100A. For a summary of each TL/1 command and its syntax, refer to the *TL/1 Reference Manual.*

If you have written programs before, you will find programming on the 9100A to be similar to other programming you have done. If you have no previous programming experience, you will find the 9100A system to be easy to learn, although you may want to refer to a programming text for help in understanding some of the fundamental programming principles used.

Before you begin to program, you will want to familiarize yourself with the operation of the 9100A/9105A and with the proper functioning of the circuit boards you wish to test so that you can define the tasks you want your programs to perform.

The remaining sections of this manual are organized in the following order:

2. Editor - The physical programming environment. You learn how you use the programmer's interface to create, modify, and store programs and other information.

3. Overview of TL/1 - A guide to the features of the TL/1 programming language. TL/1 is a structured language specifically designed for convenient use in developing test and troubleshooting routines.

4. Debugger - The 9100A facility for fine tuning a program. Much of programming effort is devoted to verifying that a program does what it is supposed to do. The debugger is an aid to this process.

5. Guided Fault Isolation (GFI) - How to program the system to perform Guided Fault Isolation. The GFI troubleshooting feature can be customized for your UUT designs.

6. Terminal Emulator - How to use the programmer's interface as a remote terminal. This feature is useful for transferring information between the 9100A and other computer systems.

7. CAD Translator - How to use the 9100A to download a CAD system output file and to convert it into the proper format for use with the 9100A/9105A.

8. Glossary - Definitions of commonly used terms.

An index is provided at the end of the manual for reference.

Section 2
Editor

INTRODUCTION 2.1.

With the editor, you create, store, or change the data and programs required for testing and troubleshooting with the 9100A/9105A. The editor follows the userdisk organization shown in Figure 2-1. A "userdisk" is the formatted storage space on a physical disk (the hard disk or a floppy disk) allocated for user-accessible information. Each physical disk incorporates a userdisk, which can contain data and programs for one or more UUTs. To provide additional userdisks, you add more floppy disks.

USERDISK ORGANIZATION 2.2.

A userdisk consists of the following:

- **Userdisk Text Files:** Files that can contain any text. These files are used for information which is not specific to just one UUT (Unit Under Test).

- **UUT Directories:** Directories that include all test programs and Guided Fault Isolation (GFI) information for a single UUT.

- **Program Library:** Stores programs that can be used by all UUTs.

- **Pod Library:** Contains pod descriptions. Each description includes a database for the pod and sometimes special TL/1 programs to be used with the pod.

- **Part Library:** Stores descriptions of different types of components.

- **Help Library:** Contains the text of the operator's keypad help information.

Figure 2-1 summarizes the organization of a userdisk. Each of the items listed above is described in more detail starting below.

Userdisk Text Files - Operator's instructions or program documentation that is associated with more than one UUT can be stored in userdisk text documents. Userdisk text files may also be written to or read from by a TL/1 program.
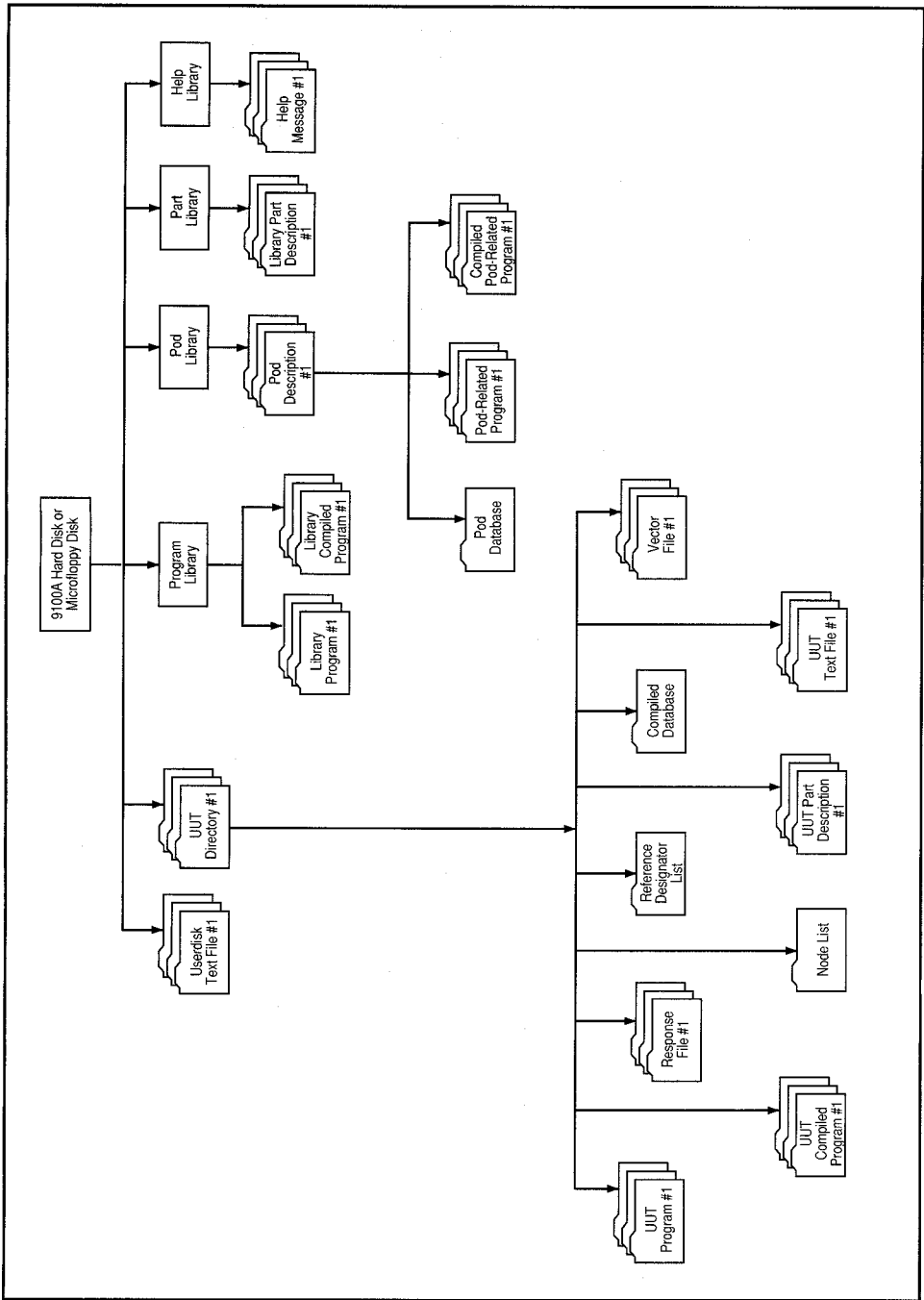
Figure 2-1: Userdisk Organization

UUT Directories - Each UUT directory includes the following items:

- Programs for testing or troubleshooting.
- Stimulus program response files.
- A node list (NODELIST).
- A reference designator list (REFLIST).
- Part descriptions.
- A compiled database (GFIDATA).
- UUT text files.
- Test vector files.

A program is used to test the functionality of an area of the UUT (or of the whole UUT.) Programs are also used in GFI to stimulate an individual node.

A stimulus program response file contains the correct data measurements that result from the application of a stimulus program to a particular node. In a complete UUT directory, each stimulus program is paired with a response file. GFI uses one or more stimulus programs and response files to determine whether a node is good or bad.

The node list describes all the interconnections of the UUT. Each UUT directory contains only one node list. The node list is used by GFI.

The reference designator list contains names which represent devices on the UUT. With this list, you assign a unique name and a part description to every device on the UUT. Each UUT directory contains only one reference designator list.

A compiled database contains stimulus program responses, reference designators, part descriptions, and the node list converted to a form that the GFI program can use for isolating faults. You cannot edit a compiled database. Stimulus programs themselves are not compiled into the database. You must copy them separately whenever you copy a compiled database to another disk.

A part description contains a description of a component, such as a 7400 quad NAND gate or a resistor. The part descriptions are used by GFI.

UUT text files may be used either to describe the UUT or the tests, or to contain source notes about programming. UUT text files also may be written to or read from by a TL/1 program.

In addition, UUT text files are manipulated by the READ BLOCK and WRITE BLOCK commands entered at the operator's keypad. And, UUT text files can be deleted using the MAIN MENU key on the operator's keypad.

Test vector files are used to describe the test vectors to be driven out by a vector output I/O module. For more information, refer to the *Vector Output I/O Module Manual*.

Program Library - The program library usually contains programs that perform frequently used operations that are not UUT specific. Unlike the programs in individual UUT directories, these programs can be called (invoked) by any other program on the userdisk. Storing these programs in the program library, rather than in a UUT directory, avoids duplicating the same program for every UUT that uses it.

Pod Library - The pod library contains pod descriptions. Each of these descriptions contains a database describing the pod. For some pods, the pod description also contains special TL/1 programs that are used with the pod.

Part Library - The part library consists solely of part descriptions. A "part description" contains a description of a component, such as a 7400 quad NAND gate or a resistor. The part library is shared by all the UUT directories on the same userdisk so that a part description does not have to be duplicated for each UUT that uses that part.

Help Library - The help library contains the help messages associated with fault messages that appear on the operator's display. The help messages are text files. There is one file called INDEX in the help library which maps fault names with help text.

Creating or Changing Directories or Files

You create and change directories and files through the editor. The operations you can perform with the 9100A editor are context sensitive (they depend on what you are editing). When the EDIT key on the operator's keypad is first pressed, the userdisk screen appears on the monitor. If you direct the editor to a particular UUT directory, library, or userdisk text file, the editor commands change to match the type of the item selected: a directory, library, or text file, for example.

When accessing files, the editor follows the userdisk organization shown in Figure 2-1. For example, if you are currently editing the node list of UUT directory #1 and want to perform an operation on a program of UUT directory #2, you can follow these steps:

1. Quit editing the node list.

2. Quit editing UUT directory #1.

3. Now you are at the userdisk level. Type in the name of UUT directory #2 and select its type.

4. Type in the name of the program and select its type.

This type of procedure (and a short-cut method) is described in the heading "Entering and Exiting the Editor" located in Section 2.

PHYSICAL ENVIRONMENT 2.3.

This section describes the physical tools used to edit the contents
of the userdisk. Your 9100A must be equipped with a monitor
and keyboard to enable editing. The connections are described
in *Getting Started*. The monitor displays information from the
editor. The keyboard includes an ASCII keyboard, which you
enter text through, and an editor keypad and softkeys (function
keys), which you enter commands with (see Figure 2-2).
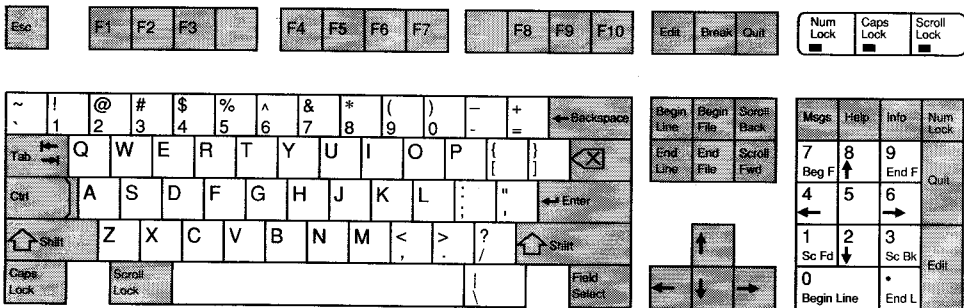
Figure 2-2: Programmer's Keyboard

The monitor's display contains 24 lines by 80 columns. When you first enter the editor by pressing the EDIT key on the operator's keypad, this display is divided into areas called *windows* as shown in Figure 2-3.

The contents of these windows vary according to what you are editing:

- **Information Window:** In Figure 2-3 the editor is operating on a userdisk screen, so the information window contains information such as the name and description of the userdisk, the write-protection status of the disk, and the amount of space available.
- **Edit Window:** The edit window lists the contents of the userdisk, organized by categories.

Commands to manipulate windows are described later in this section under "Window Commands."

When you are editing a directory (such as a UUT directory or the part library), the edit window lists the items in the directory. For example, when you view a userdisk directory, the edit window lists the UUT directories, userdisk text files, the part library, the pod library, the program library, and the help library (press the Scroll Forward key to see this item in the userdisk directory). In this case, you cannot move the cursor into the edit window nor can you turn the information window off.

When you are editing an item that is not a directory (such as a program or a node list), you *can* move the cursor into the edit window, and you *can* turn the information window on and off (by pressing the Info key on the programmer's keyboard).

— Information Window

```
/HDR (USERDISK)
NAME: HDR                                    DISK FREE: 18,398,976 BYTES
DESCRIPTION: ▮
STARTUP UUT:              PROGRAM:           DISK PROTECTED:  NO

                    PRESS A COMMAND KEY OR HELP KEY

                    DIRECTORY OF /HDR  (USERDISK)

Units Under Test (UUT):
     ABC

Text Files (TEXT):

Part Library (LIBRARY):
     PARTLIB

Pod Library (LIBRARY):
     PODLIB

Program Library (LIBRARY):
     PROGLIB
 F1    F2    F3      F4    F5    F6    F7        F8    F9    F10
REMOVE SAVE  FORMAT   COPY  TERM  STYLE
```

— Edit Window

Figure 2-3: Windows in the Userdisk Screen

Three other windows can appear on the display:

- **Help Window:** This window, which is controlled by the Help key on the programmer's keyboard, contains one line of text that describes the type of information required at the cursor location. The window also contains a variable amount of reference information depending on the type of the item being edited. The window appears in the lower portion of the display (see Figure 2-4). When you turn the help window on, the cursor moves into the window allowing you to search or scroll through the help message. To move the cursor to its original position outside the window, press the Help key to turn the window off.

- **Messages Window:** This window displays asynchronous messages that are generated by the 9100A. The messages window covers the entire display area (monitor) and appears as a blank window if there are no messages to display. You can turn the messages window on and off by pressing the Msgs key. The Scroll Lock key stops and starts the addition of new messages.

- **Fault Window:** This window is used to display a fault message that is generated by a TL/1 program, either by a stimulus program or by the debugger. When you turn the window on, the cursor moves into the window allowing you to scroll through the complete fault message.

```
/HDR (USERDISK)
 NAME: HDR                              DISK FREE: 18,398,976 BYTES
 DESCRIPTION:
 STARTUP UUT:           PROGRAM:        DISK PROTECTED:  NO

                  PRESS A COMMAND KEY OR HELP KEY

                  DIRECTORY OF /HDR  (USERDISK)
                            HELP WINDOW
 Description of user disk (opt)



         GETTING STARTED (use down arrow key to read HELP)

 Press the Edit key to create a new UUT directory.

 Try This:
 Press Help to remove this window then:
 Press Edit, type in BOARD and Press Return Twice (selects "UUT" file type)


 F1    F2    F3        F4    F5    F6    F7        F8    F9    F10
                                                      SEARCH
```

Figure 2-4: A Userdisk Screen with Help Window

The top line and two bottom lines of the display are reserved for the following information as shown in Figure 2-5:

- **Status Line:** This displays the name (pathname) and type for the item you are editing. The status line also displays the line number of the cursor location. If you make changes, the status line displays a note to remind you to save the changes before you quit editing.

- **Softkey Numbers Line:** This displays the softkey numbers (F1, F2, . . ., F10) above the softkey labels.

- **Softkey Labels Line:** This displays the labels for the ten softkeys. You press softkeys to perform editor commands. The softkey labels change according to the commands that are available. When fewer than ten commands are available, some of the labels remain blank.

  Some commands require that you enter information (the name of a program, for example). Pressing a key for one of these commands causes a prompt line to replace the softkey labels, and the cursor moves to the line so you can type in whatever is requested.

  The softkey labels line is also used to display messages that pertain to the status of disk operations, such as "SAVING . . ." and "LOADING . . ." messages.

  If you enter inappropriate information, an error message replaces the softkey labels on the softkey labels line. Press the Return key after you read the error message to restore the softkey labels on the screen.

Status Line

NAME: HDR   DISK FREE: 18,398,976 BYTES
DESCRIPTION:
STARTUP UUT: PROGRAM: DISK PROTECTED: NO

PRESS A COMMAND KEY OR HELP KEY

DIRECTORY OF /HDR (USERDISK)

Units Under Test (UUT):
     ABC

Text Files (TEXT):

Part Library (LIBRARY):
     PARTLIB

Pod Library (LIBRARY):
     PODLIB

Program Library (LIBRARY):
     PROGLIB

F1	F2	F3		F4	F5	F6	F7		F8	F9	F10
REMOVE	SAVE	FORMAT		COPY	TERM	STYLE					

Softkey Labels Line

Softkey Numbers Line

Figure 2-5: Status Line, and Softkey Numbers and Labels Lines

The ASCII keyboard (see Figure 2-6) includes keys for all ASCII characters. In addition, the following keys perform special functions:

- **Shift:** When pressed at the same time as another key, the Shift key causes the shifted (upper) value of the key to be typed.

- **Caps Lock:** This key affects only alphabetic keys and causes these keys to type upper-case letters. Press the Caps Lock key again to turn off the feature. The indicator lamp on the key turns on when Caps Lock is active.

- **Ctrl:** When pressed at the same time as another key, the CTRL key causes the corresponding control sequence (CTRL-C, for example) to be typed. The CTRL key is not used during editing.

- **Scroll Lock:** As new messages appear at the bottom of the messages window, previous messages scroll up and off the screen. Pressing the Scroll Lock key stops the scrolling so that messages do not disappear. Pressing the Scroll Lock key again unlocks the display and allows scrolling to resume. The indicator lamp on the key turns on when Scroll Lock is active.

- **Arrow Keys:** These keys move the cursor in the indicated direction. The Back Space key is identical to the left arrow key.

- **Delete:** The Delete key (marked with a large X) removes one character to the left of the cursor and moves the cursor one character to the left.

- **Tab:** This key is used when editing programs, node lists, and text files. When pressed, the Tab key causes spaces to be inserted up to the next tab stop. Tab stops are located every eight columns.

- **Field Select:** This key is active only when the cursor is located at a selectable field. The selectable field will be highlighted. To scroll through the various selections, press
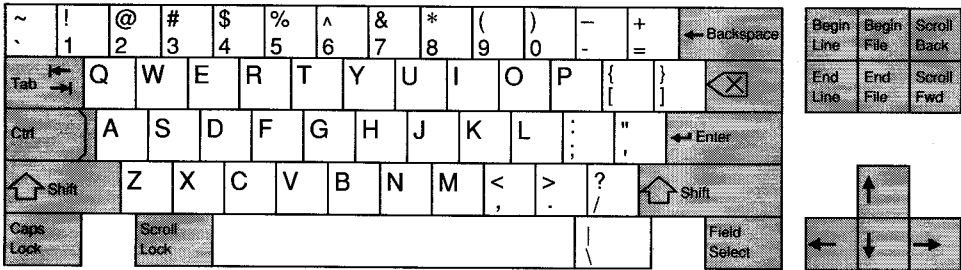
Figure 2-6: ASCII Keyboard

the Field Select key. To scroll backwards through previously viewed selections, hold down the Shift key while pressing the Field Select key.

- **Escape**: This key is not used during editing. If you press this key during editing, you will hear a beep.

- **Break**: This key is not used during editing. If you press this key during editing, you will hear a beep.

All keys except the Return key and the Escape key repeat when held down.

Editor Keypad 2.3.3.

The editor keys shown in Figure 2-7 perform the following commands:

- **Edit**: This key lets you edit a new item. For example, when you are editing a UUT directory and instead want to edit a response file, you press the Edit key. The editor responds by prompting for a file name and a file type.

- **Quit**: This key lets you quit editing at the current level and return to the next higher level. For example, if you are editing a node list and press the Quit key, the editor returns to the UUT directory. You also press the Quit key to cancel a prompt.

NOTE

You may avoid repetitive quitting through higher levels by holding down the Shift key while pressing the Quit key. In this case, control immediately returns to the operator's keypad. When you subsequently press the EDIT key on the operator's keypad, the editor resumes with the file or directory from which the Shift-Quit was issued.
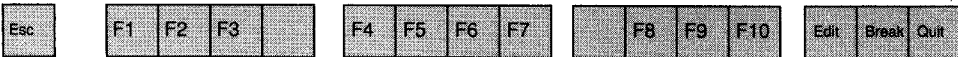
Editor Keypad

Softkeys (Function Keys)

Figure 2-7: Editor Keypad and Softkeys

- **Msgs:** This key turns the messages window on and off.

- **Help:** This key turns the help window on and off.

- **Info:** This key turns the information window on and off. It is active only when you are editing a file.

- **Beg F (Begin File):** This key moves the cursor to the first character or field of the file.

- **End F (End File):** This key moves the cursor to the last character or field of the file.

- **Sc Fd (Scroll Forward):** This key scrolls the display up 20 lines, moving the bottom line to the top of the edit window.

- **Sc Bk (Scroll Backward):** This key scrolls the display down 20 lines, moving the top line to the bottom of the edit window. If fewer than 20 lines exist before the currently displayed lines, the display is scrolled until the first line of the file appears and the cursor will stay on the current line.

- **Begin Line:** This key moves the cursor to the first character or field of the current line. It is active only when the cursor is in the edit window or information window.

- **End L (End Line):** This key moves the cursor to the last character or field of the current line. It is active only when the cursor is in the edit window or information window.

Softkeys (Function Keys) 2.3.4.

Ten keys labeled F1, F2, . . . F10 (see Figure 2-7) are designated softkeys because their functions are determined by the editor software. The labels that appear in the softkey labels line of the display specify the function for each key that is active.

The monitor and programmer's keyboard provide the communications interface to the 9100A editor. When you use the editor, you cannot troubleshoot with the 9100A since the operator's keypad and display are inactive for the duration of your editing session.

To invoke the editor, press the EDIT key on the operator's keypad. The 9100A checks for the presence of the programmer's interface; if the interface is connected, the information window and edit window for the HDR userdisk (on the hard disk) appear on the display. From this point on, you enter commands from the programmer's keyboard.

Initially you are editing the userdisk on the hard disk drive (called the HDR userdisk). Press the Edit key to edit either the DR1 userdisk (floppy disk drive) or any item displayed for the HDR userdisk. A prompt appears to let you enter the name of the items and its type (USERDISK, UUT, etc.). If the name and type match an existing item, the information is retrieved from the userdisk and displayed. Otherwise, the editor creates a new (blank) item with the name and type you have specified.

If you are familiar with the organization of the userdisk, you can direct the editor to a low-level item immediately (rather than by editing successively lower levels). To do this, enter the full pathname of the item you want to edit. For example, to edit the program VIDEO_TEST in the UUT directory MAIN_BOARD on the DR1 userdisk, enter /dr1/main_board/video_test as the name of the item to edit.

If you have made changes to the item you are currently editing and you press the Edit key, the editor will prompt you for the name and type of the next item to be edited. Then the editor prompts you to determine whether you want to save the changes made to the original item.

To finish editing, press the Quit key. If you have made changes, the editor prompts you to determine whether to save the changes. Once you answer this prompt the editor returns to

the next higher level. For example, if you quit editing a node list, the editor returns to the UUT directory. If you quit editing the UUT directory, the editor returns to the userdisk. Finally, if you quit editing the userdisk, the editor returns control to the operator's keypad.

NOTE

You may avoid repetitive quitting through higher levels by holding down the Shift key while pressing the Quit key. In this case, control immediately returns to the operator's keypad. When you subsequently press the EDIT key on the operator's keypad, the editor resumes with the file or directory from which the Shift-Quit was issued.

DISK UTILITIES   2.5.

Utilities operate on whole entities: programs, node lists, or UUT directories, for example. These commands are all invoked by pressing softkeys when the appropriate labels are displayed:

- COPY: This command performs several operations. When you copy a directory, you copy all the files contained in that directory.

  Copy files and directories - You can copy an item to a new item of the same type on either the same userdisk or a different userdisk, with the same name or a new name. If the name and type of the destination you specify matches an existing item, you are prompted about whether to overwrite the existing item.

  Create a backup disk - You can copy an entire userdisk to a different disk. If a userdisk already exists on the destination disk, it is erased before copying.

Convert files to and from text - If you copy a non-text file to a new file of the type TEXT, you create a text document equivalent of the file. If you copy a text document to a new file of any type except TEXT, the text in the original document must meet all the format and syntax requirements for the new type; otherwise, the conversion is not allowed. These operations allow you to transfer files to and from a different system with a different editor.

Print files and directories - If you print a directory, the print format of that directory depends on the setting of the STYLE softkey. If it is set to BRIEF, a list of the directory files is printed. If it is set to LONG, a list of the directory files, file sizes, and file modification dates and times are printed. You specify a portname (either /PORT1 or /PORT2, whichever the printer is connected to) as the name and PORT as the type of the item to copy to. The 9100A automatically converts the item to text and sends it to the printer port. This operation can be used to print any file or directory.

• REMOVE: This command removes the item or items you specify. You cannot remove the item that you are currently editing.

• FORMAT: This command operates on the floppy disk in disk drive /DR1. The hard disk cannot be formatted. FORMAT clears the current contents of the disk, prepares it for storing files, and inspects it for physical defects. You use the FORMAT command either to erase all files from a disk or to prepare an unused disk for storing files. If the specified disk is already formatted, you are prompted as to whether you want to overwrite the information. Otherwise you are prompted to verify that you really want to format the disk. If a physical defect is found on the floppy disk, the FORMAT operation terminates, returning an error message.

• SAVE: This command writes the current state of the item you are editing, including the name and write protection status, to the disk and leaves the cursor at its current position. For example, if you create a new text document and type in some text, you enter the SAVE command to save the text on the disk. If there is not enough free memory on the disk to save the file, an error message

memory on the disk to save the file, an error message appears. To save the file, insert another disk and temporarily save the file on the new disk. Then after making room on the original disk, the file can be copied back onto it.

If the file on the disk is write-protected (indicated by a YES in the WRITE PROTECT field of the information window), you are prompted whether you want to overwrite the disk version of the file. Because the prompt is based on the disk version of the file, if you do not want to be prompted, change the write-protection field of the information window to NO and save the file. Thereafter you will not be prompted.

INFORMATION ENTRY 2.6.

You enter all editor commands by pressing a softkey or an editor keypad key; you do not need to memorize control key (Ctrl) sequences or type command names. When the editor requires more information, it displays a prompt and you type a reply. When the information that you can enter is limited to a small number of choices, the editor provides the choices for you to select from.

Text Entry 2.6.1.

You insert text at the cursor location by typing characters from the ASCII keyboard. When you press the Return key, the line of text is created. If you type beyond the 80 character width of the display, the editor automatically inserts a continuation character (>) at the end of the display line, moves the cursor to the beginning of the next display line, and inserts a second continuation character (<). The continuation characters connect one display line to the next, resulting in one continuous text line containing more than 80 characters.

The Tab key inserts spaces. Tab stops are fixed at every eighth column. Pressing the Tab key advances the cursor to the next tab stop. Tabs are not allowed when filling in a field.

The Delete key moves the cursor left and erases the character in that location. Character deletion wraps to the previous line; if you press the Delete key when the cursor is at the beginning of a line, the carriage return after the previous line is deleted, and the two lines are joined. You cannot delete past the beginning of a field.

You cannot enter control characters in the text. If you type a control character during editing, you will hear a beep, and the input will be ignored by the editor.

Fields   2.6.2.

In programs, text files, and node lists, you can enter characters anywhere within the display. In response files, part descriptions, and other items (such as the information window), cursor movement is limited to specific areas called fields. A simple way to determine the presence of fields is to move the cursor with the right arrow and left arrow keys. If there are no fields, the cursor moves one character at a time; if there are fields, the cursor jumps from one field to the next.

There are two types of fields; an example of each is shown in Figure 2-8:

- **Fill-in:** This field appears as a long blank, similar to those on a paper form. You type information into the field from the keyboard. When you press the Return key, or move the cursor out of the field, or enter the SAVE command, the information you have typed is entered.

    The number of characters you can type in a fill-in field is limited by the size of the field.

- **Selectable:** This field can only be filled by a limited number of choices. When you move the cursor to a selectable field, the cursor disappears and the entire field is highlighted. After you move the cursor to the field, press the Field Select key to see the available choices. You can think of the choices as being attached to a knob that you turn by pressing the Field Select key until the choice you want appears. To review the selections in the opposite direction, hold down the Shift key while you press the Field Select key.

When editing an item in the edit window that contains fields, lines are automatically inserted when you attempt to move down from the last line. Lines can also be inserted above the last line by using the INSERT softkey. When the INSERT softkey is pressed, a new line is inserted beneath the line where the cursor is located. Lines can be deleted by using the DELETE softkey. When the DELETE softkey is pressed, the line where the cursor is located will be deleted.

```
   RELATED INPUT PINS  PIN NAME                    PIN NAME  RELATED INPUT PINS

                                    →  ┌───────┐ PWR
                                    →  │ 1  14 │ ←
                                    →  │ 2  13 │ ←
        1,2                         ═  │ 3  12 │ ←
                                    →  │ 4  11 │ →        12,13
                                    →  │ 5  10 │ ←
        4,5                         ←  │ 6   9 │ ←
                                   GND │ 7   8 │ →        9,10
                                       └───────┘

        PIN      RELATED INPUT PINS
```

```
 F1    F2    F3       F4    F5    F6    F7        F8    F9    F10
GOTO  SAVE           MARK              PASTE
```

— Fill-in Field   — Selectable Field

Figure 2-8: Fields

2-26
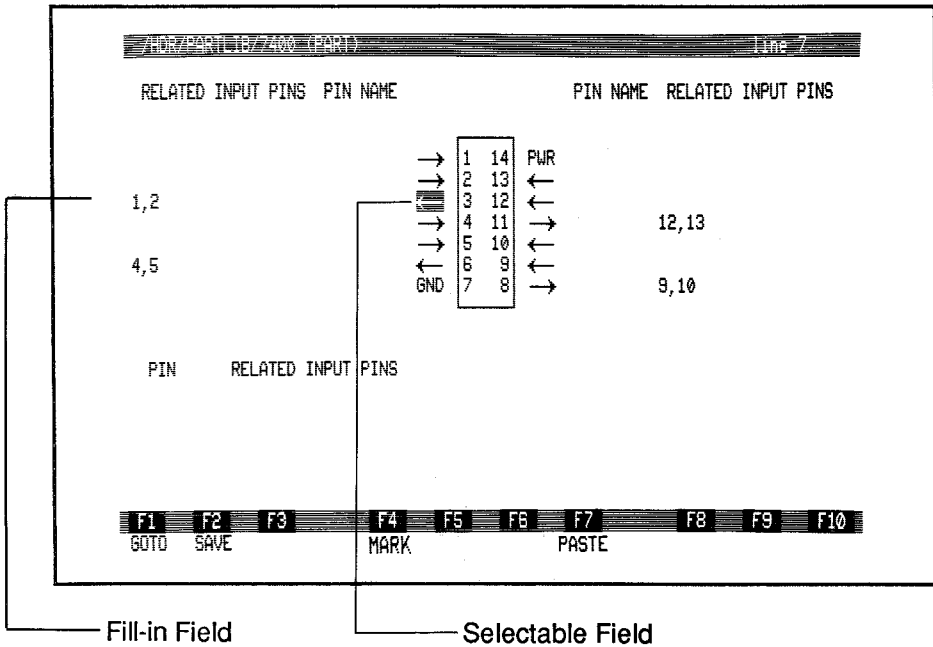
Prompts and Defaults

A prompt is a word or phrase that appears on the prompt line. Whenever a prompt appears, the cursor is moved to the prompt line so that you can type a reply. After you type the reply, press the Return key. The bottom line of Figure 2-9 shows two prompts with appropriate replies.

Frequently, a prompt appears with the reply already provided. The editor retains the reply that you entered the last time it displayed this prompt and offers this reply as a default. To enter the default, press only the Return key. To enter another reply instead, type it in; the default disappears with the first character you type. Once you press the Return key, the command is issued and the new reply becomes the default. To cancel a prompt, press the Quit key. The softkey labels reappear and no operation is performed.

The 9100A editor recognizes the asterisk (*) as a wildcard in your replies to many of its prompts. The most common use of the wildcard is while entering names, either to save typing or to specify several names at once.

For example, if you want to remove all programs that begin with the letter R from a UUT directory, you edit the UUT directory and use the REMOVE command, specifying R* as the name and PROGRAM as the type. In this case, the wildcard provides a way to identify many names at once.

You can include several wildcards in a single name. For example, *TEST* can represent the name BITTEST5. You cannot include a wildcard in any but the last item of a pathname; for example, the pathname /DR1/TK*/EXAMPLE is not valid.

```
/HDR  (USERDISK)
NAME: HDR                                    DISK FREE: 18,398,976 BYTES
DESCRIPTION:
STARTUP UUT:              PROGRAM:           DISK PROTECTED:  NO

                      PRESS A COMMAND KEY OR HELP KEY

                      DIRECTORY OF /HDR  (USERDISK)

Units Under Test (UUT):
    ABC

Text Files (TEXT):

Part Library (LIBRARY):
    PARTLIB

Pod Library (LIBRARY):
    PODLIB

Program Library (LIBRARY):
    PROGLIB
  F1     F2     F3        F4    F5    F6    F7         F8    F9    F10
NAME ABC                                   TYPE UUT
```

┗━ Reply   ┗━ Reply
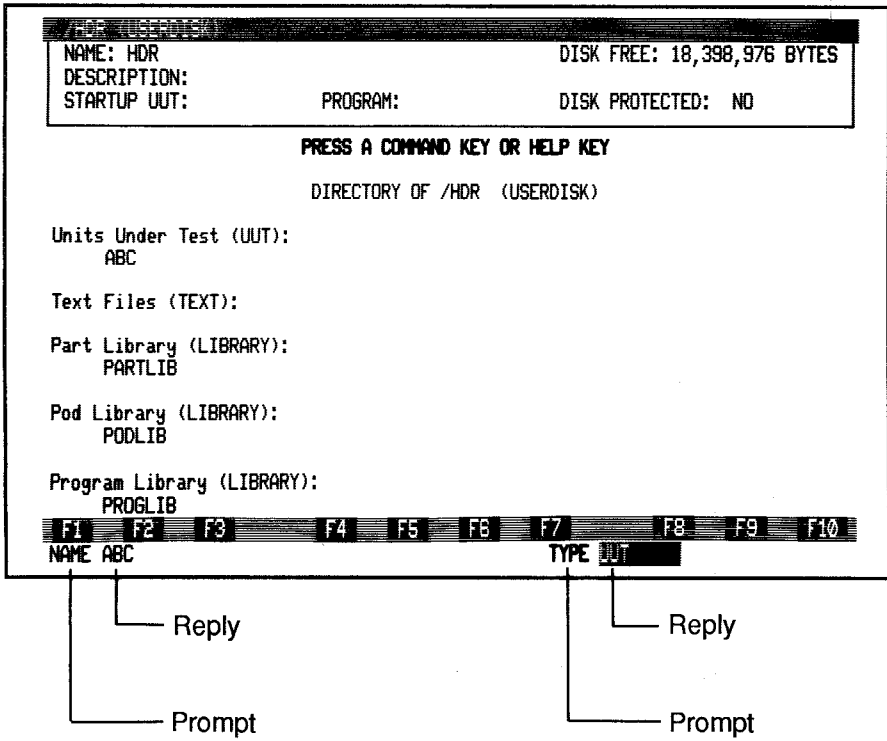
┗━━━━ Prompt   ┗━━━━ Prompt

Figure 2-9: Prompts and Replies

The 9100A editor operates on structured information. A node list, for example, contains only pin names. A program contains only TL/1 statements. Because of this structure, the editor can detect errors in the information you enter.

The types of errors that the editor detects depend on what you are editing. For example, when you are editing a node list, the editor checks that everything you enter has the form of a pin name. For a program, the editor checks that the text you enter conforms to TL/1 language rules.

The editor and debugger check for syntax and run-time errors as follows:

- An editor line-check detects syntax errors as you type. If you try to move the cursor off a line that contains a syntax error, you will see an error message displayed on the message line. You must change the line to correct the error; otherwise, you cannot move the cursor off the line, you can neither save the file nor quit editing.

  If you are editing a TL/1 program or a node list, you can turn the line-checker ON or OFF by pressing the CHECK softkey while holding down the SHIFT key. The current status of the line-checker is shown in the status line. The line checking mode cannot be changed until the current line is correct.

- The CHECK command in the editor performs an overall check by searching for syntax errors that cannot be detected by the line-check, such as missing block delimiters. When it discovers an error, the editor inserts an error message after the erroneous line. If you correct the error and reissue the CHECK command, the message disappears. (You can also delete the message yourself, using regular editor commands.) The CHECK command is applicable only for programs and node lists. A display resulting from a CHECK command is shown in Figure 2-10.

```
/HDR/ABC/DEMO (PROGRM)                                        [MODIFIED] line 1

program demo (num, address)
declare
  numeric num
  numeric address
end declare

  open device "/termi", as "output"

  if num = 0 then
     print The number of iterations is zero"
  else
     loop while num < > 0
         data = read (address)
         print "The data read = ", data
         num = num - "1"
+++CHECK: illegal operand type for '-'
     end loop

end demo
+++CHECK: inappropriate END at end of IF block
+++CHECK: expected END PROGRAM statement

 F1     F2     F3        F4    F5    F6    F7        F8    F9    F10
Encountered error(s) during check .                      (PRESS RETURN)
```

Result of missing *end if* statement following the *end loop* statement.

Result of subtracting a string from a numeric.
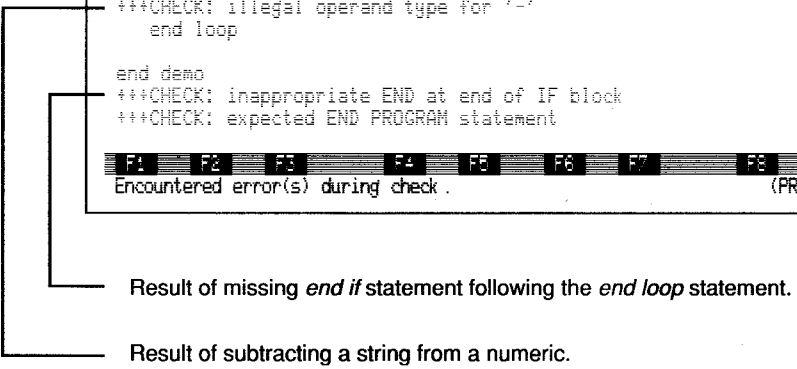
Figure 2-10: CHECK Errors

When the CHECK command is used on TL/1 programs, you are prompted to determine if you want to use the current set of options to the checker. If you select NO, a dialog window listing the various options is displayed. Use the up and down arrow keys to select various items, and the Field Select key to choose the value of the entry.

NOTE

CHECK messages begin with a series of plus signs (+) so that you can locate the messages quickly using the SEARCH command.

- The debugger checks for run-time errors. With the debugger, you can view and alter the values of variables at intermediate stages of program execution. By tracing the values of variables during execution, you can determine if a program performs as intended. See Section 4, "Debugger," for more information.

FILE AND DIRECTORY NAMES   2.8.

Every file and directory has a name. A file or directory name must meet the following requirements:

- It consists only of letters, digits, underscore characters "_", and periods ".".
- Its first character is either a letter or digit.
- It has no more than 10 characters.

File and directory names are not case-sensitive; "TEST1" is the same name as "test1". Two files or directories can have the same name if they have different types. For example, a program named TEST1 is distinct from a text document named TEST1. Two files of the same type can have the same name if they are in different directories. The program DEMO in the program library does not conflict with the program DEMO in a UUT directory.

The names PARTLIB, PROGLIB, PODLIB, and HELPLIB can only be given to a parts library, program library, pod library, and help library, respectively. For example, you cannot name a program PODLIB.

The names of directories that are limited to one per userdisk or files that are limited to one per UUT directory are predetermined. These items and their names are:

Items limited to one per userdisk

Directory	Name	Type
user disk (hard drive)	HDR	USERDISK
user disk (floppy drive 1)	DR1	USERDISK
user disk (floppy drive 2)*	DR2	USERDISK
part library	PARTLIB	LIBRARY
program library	PROGLIB	LIBRARY
pod library	PODLIB	LIBRARY
help library	HELPLIB	LIBRARY

Items limited to one per UUT directory

File	Name	Type
reference designator list	REFLIST	REF
node list	NODELIST	NODE

* On the 9105A only.

EDITING A USERDISK 2.9.

An example of the edit window and information window formats of a userdisk screen is shown in Figure 2-11.

The edit window shows the various directories and files on the userdisk.

The information window includes the following fields:

- NAME: The editor displays the name of the userdisk in this field. The name is one of the following:

    HDR - The hard disk
    DR1 - Floppy disk drive 1
    DR2 - Floppy disk drive 2 (9105A only)

- DESCRIPTION: Enter text describing the contents of the userdisk in this field.
- STARTUP UUT: Enter the name of a UUT on the userdisk. This UUT directory automatically becomes the current UUT directory when the system is powered up with this userdisk.
- PROGRAM: Enter the name of a program in the startup UUT directory. This program is automatically executed when the system is powered up with this userdisk.
- DISK FREE: The amount of disk space that is still available. This field cannot be edited.
- DISK PROTECTED: The editor fills in this field that indicates whether the physical disk is write protected. This field is not the same as the write-protection field for files. Disk write protection is accomplished in hardware (physically), whereas file protection is accomplished in software.

To edit any item in the userdisk, press the Edit key and enter the name of the item. The edit and information windows for any directory are similar to the userdisk screen, except that the sub-directory names are different and no STARTUP UUT nor PROGRAM field appears.

```
┌─────────────────────────────────────────────────────────────────┐
│ ▓▓HDR▓▓(USERDISK)▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓ │
│  NAME: HDR                          DISK FREE: 18,398,976 BYTES  │
│  DESCRIPTION: ▓                                                  │
│  STARTUP UUT:          PROGRAM:          DISK PROTECTED:  NO     │
│ └──────────────────────────────────────────────────────────────│

              PRESS A COMMAND KEY OR HELP KEY

              DIRECTORY OF /HDR  (USERDISK)

  Units Under Test (UUT):
      ABC

  Text Files (TEXT):

  Part Library (LIBRARY):
      PARTLIB

  Pod Library (LIBRARY):
      PODLIB

  Program Library (LIBRARY):
      PROGLIB
  ▓F1▓▓▓▓F2▓▓▓▓F3▓▓▓▓▓▓▓F4▓▓▓F5▓▓▓F6▓▓▓F7▓▓▓▓▓▓▓F8▓▓▓▓F9▓▓▓F10▓
  REMOVE  SAVE  FORMAT      COPY  TERM  STYLE
```
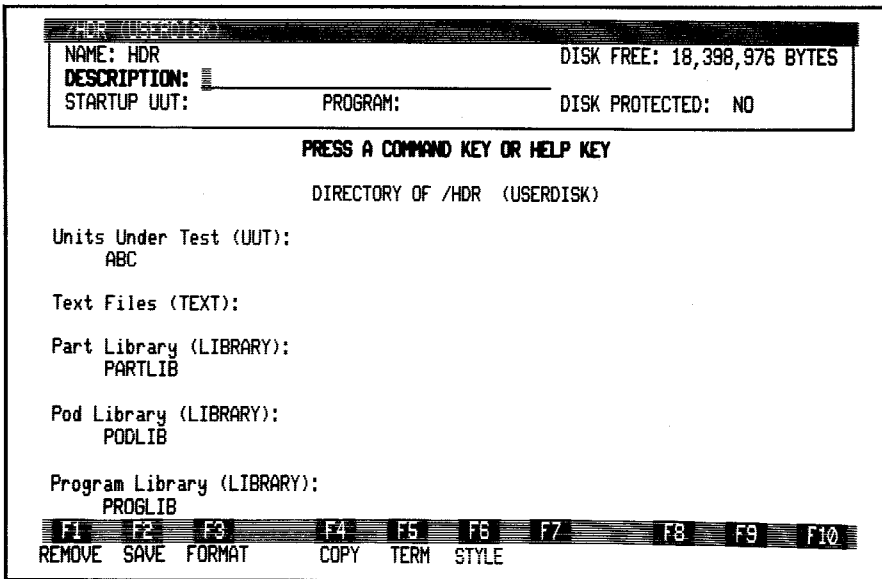
Figure 2-11: Userdisk Screen

CURSOR COMMANDS   2.10.

Four arrow keys, six editor keypad keys, and three softkeys
control the position of the cursor. These keys help you make
changes quickly.

The arrow keys move the cursor a character at a time:

*   Up Arrow and Down Arrow: The up arrow key moves the
    cursor up one line. The down arrow key moves the cursor
    down one line.

    If you move the cursor up when it is on the first line of the
    display, the display scrolls down a line. If you move the
    cursor up when it is on the first line of the file, the 9100A
    beeps. If you move the cursor down when it is on the last
    line of the display, the display scrolls up a line. If you
    attempt to move the cursor down when it is on the last line
    of the file, you will hear a beep and the cursor will remain
    in its current location.

*   Right Arrow and Left Arrow: The right arrow key moves
    the cursor one character to the right or, if the item contains
    fields, one field to the right. The left arrow key moves the
    cursor one character to the left or, if the item contains
    fields, one field to the left. If there are no characters or
    fields in the direction you move the cursor, you will hear a
    beep.

The following editor keypad keys change the position of the
cursor:

*   **Begin Line:** This key moves the cursor to the first
    character of the current line. If the file contains fields, the
    cursor moves to the first character position in the left-most
    editable field of the current line.

*   **End L:** This key moves the cursor to the last character of
    the current line. If the file contains fields, the cursor

moves to the first character position in the right-most editable field of the current line.

- **Sc Fd (Scroll Forward):** This key scrolls the display up 20 lines, moving the bottom line to the top of the edit window. The cursor moves to the first character of the top line.

- **Sc Bk (Scroll Backward):** This key scrolls the display down 20 lines, moving the top line to the bottom of the edit window. The cursor moves to the first character or first field of the bottom line. If fewer than 20 lines exist before the currently displayed lines, the display is scrolled until the first line of the file appears and the cursor will stay on the current line.

- **Beg F (Begin File):** This key moves the cursor to the first character of the file. The text is scrolled back to the first screen if necessary. If the file contains fields, the cursor moves to the first character position in the first editable field.

- **End F (End File):** This key moves the cursor to the last character of the file. The text is scrolled forward to the last screen if necessary. If the file contains fields, the cursor moves to the first character position in the last editable field.

The following softkey commands change the position of the cursor:

- GOTO: This command, which is active when the GOTO softkey label appears, moves the cursor to the beginning of any line. When you press the GOTO softkey, the editor prompts you for the line number:

  ```
  GOTO LINE _____
  ```

  Lines are numbered consecutively from the top of the file. If you enter the number of a line that does not exist or is not editable, the 9100A displays an error message.

- SEARCH: This command, which is active when the SEARCH softkey label appears, moves the cursor to the next occurrence of a character string you specify at the prompt:

SEARCH FOR _____

The character string may be a word, part of a word, or several words, up to 20 characters in length. The search is case sensitive; the upper-case "A", for example, is different from the lower-case "a".

If the editor does not find the character string between the cursor position and the end of the file, the search wraps around to the beginning of the file and continues. If the editor does not find the character string anywhere in the file, it displays an error message. The editor retains the string you enter and offers it as a default the next time you issue the SEARCH command.

The searchstring can contain one or more wildcard characters (*). For example, if you specify MOD*, the editor finds the next occurrence of MOD followed by any character: MOD2, MODULE, or MODE, for example. If you want to search for a literal asterisk (*), enter two asterisks (**) in the search string. For example, to search for the expression 2*3, you would enter the search string 2**3. By entering two asterisks, the editor interprets the character sequence as a literal asterisk rather than as two wildcard characters.

To reissue your last search (and avoid re-typing the search string), press the Shift key and hold it down while pressing the SEARCH softkey.

- REPLACE: This command, which is active when the REPL softkey label appears, moves the cursor to and replaces the next occurrence of a character string you specify at the prompt:

REPLACE _____ WITH _____

The string is replaced with a second string that you specify. The search is performed exactly as for the SEARCH command. The replacement string cannot contain a wildcard.

To reissue your last replace command (and avoid re-typing both the search string and the replace string), press the Shift key and hold it down while pressing the REPL softkey.

NOTE

Programs and node lists may contain check messages. These lines are not editable. When the string to be replaced is found in one of these lines, the cursor will be positioned at the beginning of the string but the replacement will not be carried out.

WINDOW COMMANDS   2.11.

The following keys control the display of windows that cover the edit window:

- **Msgs**: This key turns the messages window on and off.

- **Help**: This key turns the help window on and off.

- **Info**: This key turns the information window on and off. This key is only active when you are editing a file.

- **FAULT** (softkey): This softkey turns the fault window on and off. The FAULT softkey is active when editing a stimulus program response file or when using the debugger.

BLOCK COMMANDS

The MARK, CUT, YANK, and PASTE commands are available for deleting, moving, and copying blocks of text within a file or between two files. The editor maintains two buffers for temporary text storage. By moving text to and from these buffers, you can save time and effort while making changes to existing text. Figure 2-12 illustrates how you delete, move, and copy text.

The following commands are active only when their softkey labels appear; in addition, the CUT and YANK commands are active only when you use the MARK command:

*   MARK: This command identifies a block of text for use with the CUT or YANK command. You can mark a set of contiguous characters or contiguous lines, but not a combination of both (you cannot mark one and a half lines, for example). One end of the block is the cursor position when you press the MARK softkey. The other end of the block is the cursor position when you press the CUT or YANK softkey. You can use the arrow keys or the cursor movement keys on the editor keypad to move the cursor forward or backward. As you move the cursor, the block is highlighted in inverse video, and a message in the status line reminds you that you are in the process of marking a block.

    To cancel the MARK command and turn off the block marking, press the MARK softkey again. The block marking also disappears when you press the CUT softkey or YANK softkey.

*   CUT: This command deletes the marked block from the display and moves it into one of the temporary buffers, replacing the current buffer contents. Pressing the CUT softkey alone moves the block into buffer #1; pressing the Shift key and the CUT softkey at the same time moves the block into buffer #2. After the CUT command is
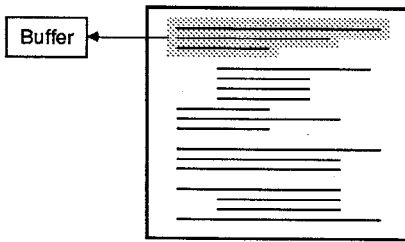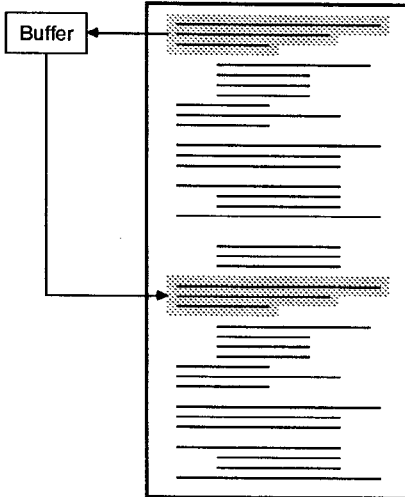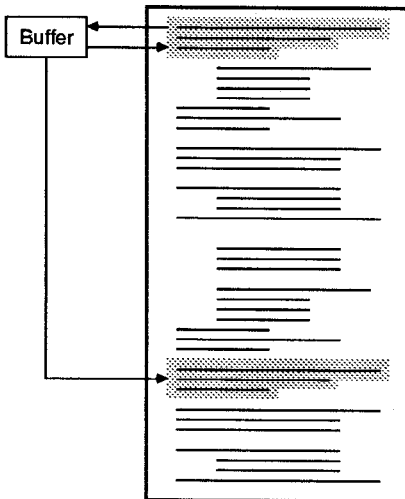
Figure 2-12: Deleting, Moving, and Copying Text

performed, the softkey labels line displays a message indicating how many characters or lines were affected.

- YANK: This command copies the marked block into one of the temporary buffers, replacing the current buffer contents; the text remains on the display. Pressing the YANK softkey alone copies the block into buffer #1; pressing the Shift key and the YANK softkey at the same time copies the block into buffer #2. After the YANK command is performed, the message line displays how many characters or lines were affected.

- PASTE: This command copies the contents of one of the temporary buffers into the display just before the cursor position. Pressing the PASTE softkey alone copies the contents of buffer #1; pressing the Shift key and the PASTE softkey at the same time copies the contents of buffer #2. The PASTE command does not alter the contents of the buffers.

To move text between two files, simply edit the file containing the text to be copied and use the MARK and CUT (or YANK) commands to copy the text block to a buffer. Then edit the destination file and PASTE the text block from the buffer into the desired location.

GUIDED FAULT ISOLATION COMMANDS   2.13.

The following commands are associated with the LEARN operation, which gathers response data from a good UUT and stores it in a stimulus program response file. The LEARN operation involves executing a stimulus program, gathering signatures, recording level history and count data, and storing it for future use. Section 5, "Guided Fault Isolation," describes these commands in detail.

The following commands apply only to stimulus program
response files:

- LEARN: This command gathers a set of node response
  data from a known-good UUT while a stimulus program is
  executed. It generates the necessary operator prompts,
  executes stimulus programs, and measures the response
  data.

- SELECT: This command selects the data in the field at the
  cursor location as data to be saved in the response file.
  You view the learned information and decide what should
  be saved.

- OFFSET: This command determines the offset delay at
  which clocked measurements should be taken with the
  probe and I/O module.

The following commands operate only on UUT directories:

- SUMMARY: This command analyzes the compiled
  database and generates a summary describing the GFI
  coverage of the UUT. For more information, see the
  "Guided Fault Isolation (GFI)" section of this manual.

The following command operates only on UUT directories,
POD directories, and the PROGLIB directory:

- COMPILE: This command selects the TL/1 GFI or UFI
  compiler.

The following command operates only on programs and node lists:

- CHECK: For programs, the CHECK command looks for TL/1 syntax errors that are not detected by the line syntax check. For node lists, the CHECK command detects duplicate pin occurrences. Error messages are inserted in the program or node list after the line in which the error is detected.

NOTE

CHECK messages begin with a series of plus signs (+) so that you can locate the messages quickly using the SEARCH command.

TERMINAL EMULATION COMMANDS 2.14.

The following command performs terminal emulation operations:

- TERM: This command makes the 9100A programmer's interface act as a terminal. You can connect another computer through an RS-232 port on the 9100A mainframe. For more information on this command, see the "Terminal Emulator" section of this manual.

CAD TRANSLATOR COMMANDS 2.15.

The following command performs translations of CAD system output files into a format acceptable by the 9100A/9105A.

- CAD: This command translates a CAD system output file into the format required by the 9100A/9105A for a reference designator list (REFLIST) and a node list (NODELIST). For more information on this command, see the "CAD Translator" section of this manual.

Overview of TL/1

GETTING STARTED WITH TL/1 PROGRAMS   3.1.

You may find it helpful to refer to the *TL/1 Reference Manual* for the specifics of command syntax while reading through this overview of the TL/1 language.

Features of TL/1   3.1.1.

TL/1 is a structured programming language specifically designed for convenient use in developing test and troubleshooting routines. Its BASIC-like statements are easy to learn and use. Command vocabulary is based on the vocabulary of the test environment, minimizing language learning time. For most commands, default entries are available to simplify the process of writing test and troubleshooting programs.

There are pre-programmed functional tests for the bus circuitry, RAM, and ROM. TL/1 also includes stimulus and response-gathering capabilities both at the microprocessor bus and at up to 160 nodes at any place on your UUT. In addition, there are fault-handling provisions which allow you to choose the appropriate action for any expected fault on your UUT.

TL/1 is capable of many other functions. It can efficiently manipulate numeric, string, and floating-point data. It can perform input and output operations on the operator's display and keypad, the monitor and programmer's keyboard, the two RS-232 ports, the IEEE-488 port, and text files on the disk drives.

Locations of TL/1 Programs   3.1.2.

TL/1 programs may exist in three places in the userdisk.

- UUT Directory - In each UUT directory, there are programs that perform testing or stimulus actions for only that type of UUT. The most likely starting point for writing a first program would be in a UUT directory.

- Program Library - General purpose programs which might be useful for more than one UUT may be stored in the program library.

- Pod Library - The pod library contains special TL/1 programs, which are used to support various pod activities.

Figure 3-1 shows each of these locations for TL/1 programs. A diagram and description of the complete 9100A/9105A file structure is contained in "Userdisk Organization" located in Section 2 of this manual.

Bringing Up a Program Screen   3.1.3.

TL/1 programs are entered at the programmer's keyboard using the 9100A editor. The editor understands the structure of TL/1 statements and checks the syntax of each line after you type it in.

To write a TL/1 program in a UUT directory, first enter the editor by pressing the EDIT key on the operator's keypad. This transfers control to the programmer's keyboard and monitor and puts an editor's screen on the monitor. Then, enter the name for the UUT and the name for the program. There are two ways to enter these names (methods A and B appear on the following pages):
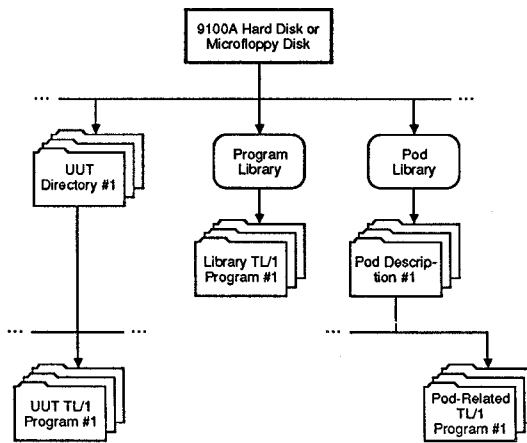
Figure 3-1: Locations of TL/1 Programs

A. Using a complete pathname:

1. Press the Edit key on the programmer's keyboard.

2. Enter a disk name, a UUT directory name, and a program name together as shown in the example below, and then press the Return key:

    ```
    /hdr/abc/test101
    ```

    ("hdr" is the disk name, "abc" is the UUT directory name, and "test101" is the program name.)

3. Select PROGRAM as the TYPE by pressing the Field Select key as many times as necessary to bring the word "PROGRAM" into the TYPE field. Then, press the Return key.

B. Moving down the file tree (see Figure 3-1):

1. The userdisk screen should now be displayed (see Figure 2-3).

    If any other screen is displayed, press the Quit key on the programmer's keyboard and wait for the monitor's display to change. If the new display is not the userdisk screen either, repeat this step until the userdisk screen does appear. If you press Quit too many times, control will return to the operator's interface and you will need to press the EDIT key on the operator's keypad to get back into the editor.

2. Press the Edit key on the programmer's keyboard.

3. Enter a UUT directory name (for example, "abc") and press the Return key.

4. Select UUT as the TYPE by pressing the Field Select key as many times as necessary to bring the word "UUT" into the TYPE field. Then press the Return key. The requested UUT directory will be displayed on the monitor.

5. Press the Edit key again and enter a program name (for example, "test101") followed by a Return.

6. Select PROGRAM as the TYPE by pressing the Field Select key as many times as necessary to bring the word "PROGRAM" into the TYPE field. Then, press the Return key.

Using either method A or method B to enter the UUT name and program name will cause the program screen to appear on the monitor. See Figure 3-2 for an example of a program screen for a new program.

Pressing the Info key shows the information window, which displays the space available on the disk and the size of the program file. In addition, you can enter a short description for the program and change the program's write protection status. Pressing the Info key again will turn the information window off. Section 2 of this manual, "Editor," explains more about this information window.

The first statement of the program must be the *program* statement and it must use the same name (*<program name>*) as the one used for the program file name. The last statement of the program must be an end statement, which is either *end program* or *end <program name>*, where the program name is the same name as the program file name.

```
/HDR/ABC/TEST101 (PROGRAM)                                      line 1
```

```
 F1    F2    F3      F4    F5    F6    F7      F8    F9    F10
GOTO  SAVE  DEBUG   MARK              PASTE   REPL  SEARCH CHECK
```
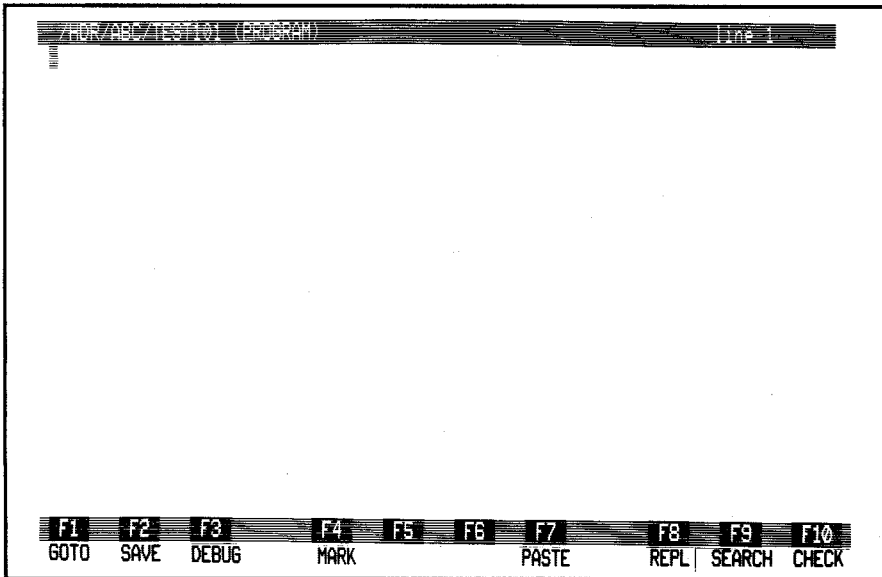
Figure 3-2: Program Screen

You can exit the program by pressing the Quit key, which moves you up one level in the file tree to the UUT directory. Or you can do a quick exit to the operator's keypad and operator's display by pressing the Shift key and the Quit key at the same time. If you do this, when you enter the editor the next time, you will be returned to the same screen from which you did the quick exit.

Structure of a TL/1 Program   3.1.4.

Figure 3-3 shows that TL/1 is a block-structured language in which executable commands are preceded by any necessary definition blocks.

There are four types of definition blocks that may be placed within a TL/1 program block:

- Declaration blocks: Used to define the type and default value of variables. These definition blocks begin with a *declare* statement.

- Function definition blocks: Used to define a function, which may be called from any place within the program block that defines that function. These definition blocks begin with a *function* statement.

- Fault condition handler definition blocks: Used to define a block which is called when a UUT fault condition is detected. These definition blocks begin with a *handle* statement.

- Fault condition exerciser definition blocks: Used to define a TL/1 block which is called when a UUT fault condition is detected and the LOOP key on the operator's keypad is pressed. These definition blocks begin with an *exercise* statement.
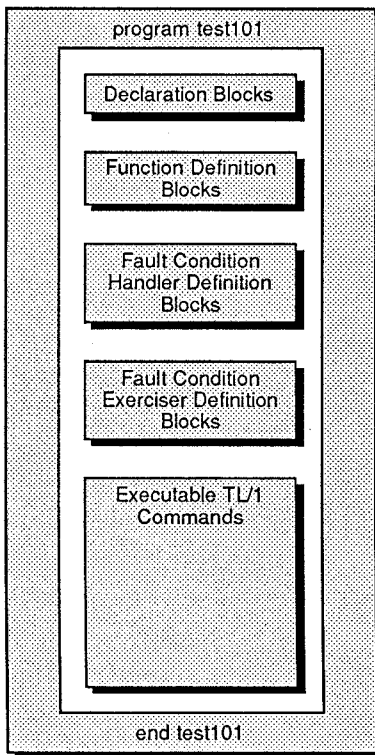
Figure 3-3: Block Structure of TL/1 Programs

Suppose you type in the program shown in Figure 3-4. To make the task easier, you might wish to leave out the exclamation points and any text that follows on a line, since the exclamation point is used to indicate the beginning of a comment.

If you make a mistake, you can move the cursor to the right of the offending characters (or character) and press the delete key once for each character to be deleted. Or, to delete the character under the cursor, you can press the Ctrl key and the Delete key (marked with an X) at the same time. More powerful editing features are described in Section 2, "Editor," of this manual. These features include deleting, moving, and copying either blocks, lines, or parts of lines.

You may have noticed that the 9100A is persistent about syntax errors and will not let you off a line that contains such an error. This feature is for your protection, but it might occasionally lead to frustration if the necessary correction is not immediately obvious. The 9100A provides three solutions for this situation.

The first solution is to press the Help key on the programmer's keyboard to bring up a help window. Pressing the SEARCH softkey and entering the TL/1 command that has given you difficulties will position you in the help file where there are examples of this command. You can use the Scroll Forward, Scroll Backward, up arrow, and down arrow keys, as well as the SEARCH softkey to move around in the help file. When you are finished with the file, press the Help key again and the help window will disappear.

```
program test101                              ! The program name must match
                                             ! the program file name

ch = open device "/term2", as "output"       ! Opens a channel for
                                             ! output to the monitor
k=1
loop while k <= 12                           ! First line of a loop block
                                             ! to loop twelve times.

        print using "The number is #@", k    ! print using allows printing
                                             ! a formatted column of
                                             ! decimal numbers

      wait time 800                          ! Wait 800 milliseconds
                                             ! before displaying a new
                                             ! number
      k=k+1
end loop                                     ! Last line of the loop block

print                                        ! Send a blank line
print "THAT'S ALL FOLKS!"
wait time 5000                               ! Allows time to view the
                                             ! Messages Window

close channel ch                             ! Close the channel ch

end program                                  ! Could also be written as:
                                             ! end test101
```

Figure 3-4: A Practice TL/1 Program

The second solution is to press the Begin Line key to move to the beginning of the line and type in an exclamation point. This makes the whole line a comment, and the syntax checker doesn't care what you have written. This can be of value if you need to save the program and come back to it later.

The third solution is to disable the line syntax checking by simultaneously pressing the Shift key and the CHECK softkey. The status line at the top of the CRT displays the current line checking mode as either '[CHECK is ON]' or '[CHECK is OFF]'. The line checking mode cannot be changed until the current line is correct.

Using the CHECK Function   3.1.6.

As you write each line, the editor checks the line for syntax errors. After you have entered the whole program, you need to check the program for errors that can be detected only by comparing each line to the other lines in the program. The CHECK softkey initiates this action.

CHECK identifies the same errors that the TL/1 compiler finds. Using CHECK has the effect of embedding the compiler error messages into the program. Both CHECK and the compiler have options to control the type of warning messages that are generated. They share a dialog window that controls the option settings. If you change an option for the CHECK function, the compiler options will automatically be changed to match.

This section describes how to check TL/1 programs from the editor.

Check procedure

The following two procedures are used to check a TL/1 source program:
- Checking programs by using the current options.
- Checking programs by changing the current options.

Use the first procedure if the current option settings are correct. Use the second procedure if you want to change the options, or

if you want to see what the options are. Each time the editor is started, the options are set to system default values that report only errors, and no warnings.

Check procedures (using current options)

1. Press the CHECK softkey. Observe the prompt, USE CURRENT TL/1 COMPILER OPTIONS.

2. After the USE CURRENT TL/1 COMPILER OPTIONS prompt, use the Field Select key to select YES. This prompt controls the type of optional warning messages that are generated when a program is checked.

3. Press the Return key to begin checking. Observe the message "CHECKING ..." on the bottom of the monitor screen. When the check operation is complete, a message like "3 errors or warnings detected" is displayed on the bottom of the monitor screen. In addition, the CHECK function inserts error messages into the program text. Each error message appears in bold and begins with the characters "+++". If you cannot see any of these errors because the program is longer than one screen, the editor search command can be used. Press the SEARCH softkey and enter "+++" as the search string as shown below:

   SEARCH FOR +++_____

   If the CHECK function does not find any errors, the message "0 errors detected" is displayed.

Check procedures (by changing current options)

1. Press the CHECK softkey. Observe the prompt, USE CURRENT TL/1 COMPILER OPTIONS.

2. After the USE CURRENT TL/1 COMPILER OPTIONS prompt, use the Field Select key to select NO, then press the Return key.

3.   Observe that the Tl/1 Compiler Options Dialog Window has appeared. This window is shown in Figure 3-5. Observe the prompt, "Generate standard warning messages". Use the Field Select key to select YES or NO. This prompt controls whether the checker will generate optional warning messages. Warning messages are precautionary only, and advise of TL/1 constructs that may be bugs.

Use the Field Select key to select YES. Observe that when YES is selected, five additional prompts appear in the lower half of the dialog window, as shown in Figure 3-6. Press the down-arrow key to move the cursor to the first prompt.

4.   Observe the five prompts, Undeclared formal parameters, Implicit variables, Uninitialized global variables, and Unused global variables. These prompts can be turned on and off independently, and control whether the checker will generate warning messages when it detects these conditions in a program. Use the Field Select key to select YES or NO for each prompt. Use the Return key or arrow keys to move between the prompts. The section entitled "Using the Compiler Options for Diagnostics" further on in Section 3 describes each prompt in detail.

Setting each prompt to YES instructs the checker to check for each of these conditions, and results in the maximum number of warning messages.

```
═══════════════════════════════════════════
        TL/1 Compiler Options
═══════════════════════════════════════════
         PRESS F10 TO BEGIN CHECK
             OR QUIT TO EXIT


  Generate standard warning messages        ▓▓▓▓


                No warning messages
```
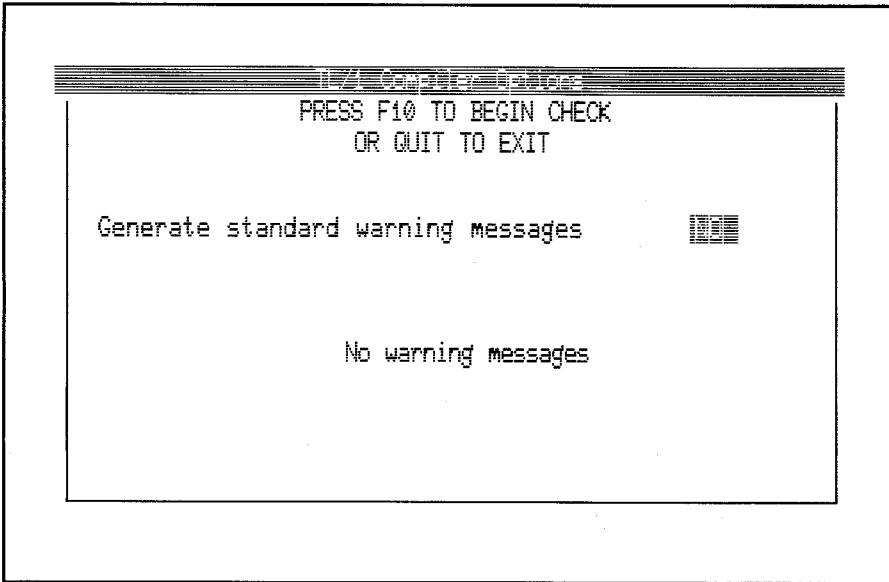
Figure 3-5 : TL/1 Check Dialog Window

```
═══════════════════════════════════════════
        TL/1 Compiler Options
═══════════════════════════════════════════
         PRESS F10 TO BEGIN CHECK
             OR QUIT TO EXIT

  Generate standard warning messages     ▓▓▓▓

  Additional warning messages generated for:
      Undeclared formal parameters:           NO
      Implicit variables:                     NO
      Uninitialized global variables:         NO
      References of unassigned variables:     NO
      Unused global variables:                NO
```
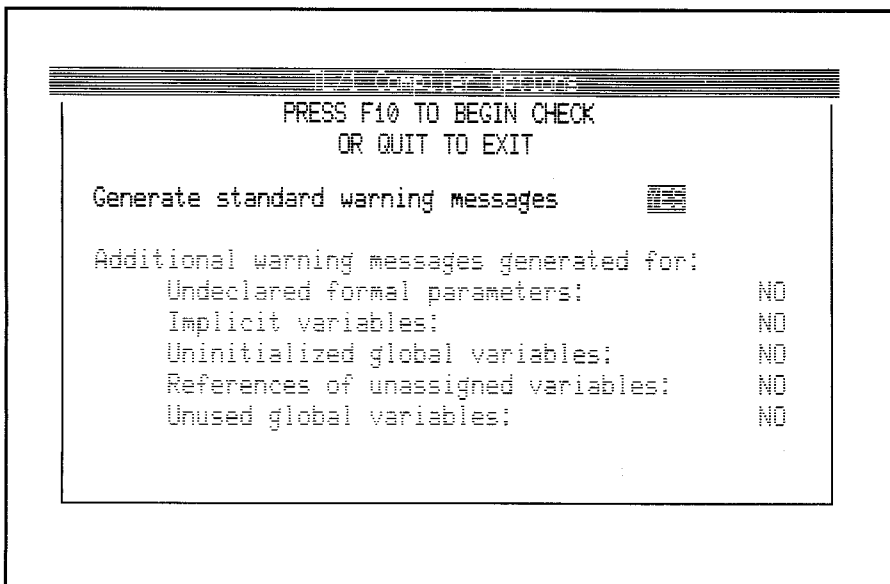
Figure 3-6: Lower Half of TL/1 Check Dialog Window

5. When all five prompts have been responded to, press the CHECK softkey (F10) to begin the check. Observe the message "CHECKING ..." on the bottom of the monitor screen. When the check operation is complete, a message like "3 errors or warnings detected" is displayed on the bottom of the monitor screen. In addition, the CHECK function inserts error messages into the program text. Each error message appears in bold and begins with the characters "+++". If you cannot see any of these errors because the program is longer than one screen, the editor search command can be used. Press the SEARCH softkey and enter "+++" as the search string as shown below:

```
SEARCH FOR +++
```

If the CHECK function does not find any errors, the message "0 errors detected" is displayed.

Using the Shift-CHECK Function   3.1.7.

As you enter each line, the editor checks it for syntax errors. This line syntax checking can be disabled for TL/1 programs and node lists. To disable the line syntax checking, edit a program and simultaneously press the Shift key and the CHECK softkey. This key combination will toggle the line checker ON and OFF. The current mode is displayed in the status line at the top of the monitor as shown below:

```
[CHECK is ON]   or   [CHECK is OFF]
```

When CHECK is ON, the line syntax is checked when you leave a line that you have modified. An error message is displayed if the line contains a syntax error, and you cannot move the cursor off the line until the syntax error has been corrected.

When CHECK is OFF, the line syntax is not checked. In this mode, you can create syntactically incorrect lines. These line syntax errors are not reported until you compile the program or use the CHECK function.

The line checking mode cannot be changed until the current line is correct.

Once the program passes the CHECK function, it is ready to be tested to make sure it does what you intend. The 9100A debugger can be used for this purpose. The debugger allows execution of TL/1 programs from the programmer's keyboard, setting of software break points, single-stepping, and setting or examination of variables.

To use the debugger on the program you have entered, press the DEBUG softkey, then the EXECUTE softkey, and finally the Return key. If no errors are found the program will run to completion and the following message will appear at the bottom of the monitor:

```
Complete, status = PASS          <PRESS RETURN>
```

The results for the program *test101* are displayed in the messages window (see Figure 3-7). After program completion, the messages window disappears; however, you can toggle the messages window on and off with the Msgs key to review the results of your program.

If an error is found, the debugger displays an error message on the bottom line of the monitor display and the debugger will place the cursor at the line where the error was found. As an example, you might want to try editing the next-to-last line of program *test101* by changing the channel name to *ch1*. Then, run the debugger to see how it handles errors.

Pressing the Quit key exits the debugger and returns to the editor to allow you to make any changes necessary in your program.

After you fix any errors that the debugger catches automatically, you return to the debugger for its main use: making sure that your program does what you intend. To single-step your program while in the debugger, move the cursor down to the first executable line and press the BREAK softkey. Figure 3-8 shows what the debugger display will look like for the example program you entered.

```
╔═══════════════════════════ Messages Window ═══════════════════════════╗
The number is  1
The number is  2
The number is  3
The number is  4
The number is  5
The number is  6
The number is  7
The number is  8
The number is  9
The number is 10
The number is 11
The number is 12

THAT'S ALL, FOLKS!
▤
```

Figure 3-7: Results of the Practice Program (test101)

```
        program test101
BRK     ch= open device "/term2", as "output"
        k=1
        loop while k<=12
            print using "The number is #0\nl", k
            wait time 800
            k=k+1
        end loop
        print
        print "THAT'S ALL FOLKS!"
        wait time 5000
        close channel ch
        end program
```

F1	F2	F3	F4	F5	F6	F7	F8	F9	F10
STEP	NEXT	CONT	EXECUTE	INIT	BREAK	SHOW	SET	SEARCH	FAULT

Figure 3-8: Debugger Screen Example

Now, when you press the EXECUTE softkey and press the Return key, the program will stop at the indicated breakpoint and make it possible to step through your program using the STEP softkey. Other debugger actions available when stopped in the middle of a program are explained in Section 4, of this manual.

Compiling a TL/1 Program 3.1.9.

9100A systems with version 6.0 or later software execute compiled TL/1 programs. You can let the 9100A automatically compile the programs when you execute (this is the default system behavior), or you can precompile your programs from the editor.

Precompiling programs has the following advantages:

- Programs which are precompiled will start executing faster.
- Precompiling programs lets you find all the compilation errors at one time during the program development process, rather than finding the errors one-at-a-time during program execution.
- Precompiling programs allows you to distribute object programs rather than TL/1 source programs. An object program is an execute-only program that cannot be modified.

NOTE

It is strongly recommended that you precompile all your TL/1 programs before executing them.

The following paragraphs describe how to precompile TL/1 programs from the editor.

Compiling Procedures

The following procedures allow you to compile a TL/1 source program (type PROGRAM) into an object program (type OBJPROG). Each step includes a discussion of the available options. Two procedures are provided:

* Using the Current Compiler Options.
* Changing the Current Compiler Options.

The TL/1 compiler has two types of options:

* An option to save the compiler error messages to a text file.
* Options to control whether the compiler generates warning messages about certain types of TL/1 constructs.

The option settings are shared by the TL/1 compiler and the TL/1 CHECK function.

Warning messages attempt to identify features of the TL/1 program that are likely to be bugs or to be wasteful. For example, one option causes the compiler to generate a warning message about variables that have not been assigned a value before they are used. Warning messages are precautionary only, and a program which generates warning messages can be executed.

Each time the editor is started, the compiler options are set to system default values that report only errors, and no warnings. If you change the compiler options, your choices become the new default values that will be used throughout the edit session.

Use the first procedure if you want to compile a program using the current compiler options. Use the second procedure if you want to change the compiler options, or if you want to see what the current compiler options are.

Compiling procedure (using current options)

1. Edit the UUT, PROGLIB, or POD that contains the programs that you want to compile. Observe that a directory is displayed, and that the program (or programs) to be compiled are listed in the directory under PROGRAM.

2. Press the COMPILE softkey. Observe the prompt, COMPILER TYPE.

3. After the COMPILER TYPE prompt, use the Field Select key to display TL/1 in the Reply window, then press the Return key. Observe the prompt, COMPILE NAME.

4. After the COMPILE NAME prompt, enter the name of the TL/1 program to be compiled. More than one program can be selected using the * wildcard. For example, F* compiles all the programs that begin with F. Entering just * compiles all the programs. Press the Return key after entering the COMPILE NAME selection. Observe the prompt, USE CURRENT TL/1 COMPILER OPTIONS.

5. After the USE CURRENT TL/1 COMPILER OPTIONS prompt, use the Field Select key to select YES. This prompt controls the type of optional warning messages that are generated when a program is compiled. It also controls whether the compiler messages are saved in a test file or simply written to the monitor display.

6. Press the Return key to begin the compile. Observe that the compilation process begins with the display "TL/1 Compiler" on the monitor display. As the compiler generates error messages, these are displayed on the monitor. Use the Scroll Lock key to stop the messages from scrolling off the monitor screen. Compilation is complete when the message "Press Msgs key to continue" appears. To embed the compiler error and warning messages directly into the program, edit the program and press the CHECK softkey. For more information, refer to "Using the CHECK Function" in Section 3.

Compiling Procedure (by changing current options)

1. Edit the UUT, PROGLIB or POD that contains the programs that you want to compile. Observe that a directory is displayed, and that the program (or programs) to be compiled are listed in the directory under PROGRAM.

2. Press the COMPILE softkey. Observe the prompt, COMPILER TYPE.

3. After the COMPILER TYPE prompt, use the Field Select key to display TL/1 in the Reply window, then press the Return key. Observe the prompt, COMPILE NAME.

4. After the COMPILE NAME prompt, enter the name of the TL/1 program to be compiled. More than one program can be selected using the * wildcard. For example, F* compiles all the programs that begin with F. Entering just * compiles all the programs. Press the Return key after entering the COMPILE NAME selection. Observe the prompt, USE CURRENT TL/1 COMPILER OPTIONS.

5. After the USE CURRENT TL/1 COMPILER OPTIONS prompt, use the Field Select key to select NO.

6. Observe that the TL/1 Compiler Options Dialog Window has appeared. This window is shown in Figure 3-9. Observe the prompt, "Save error messages in text file". During compilation, the compiler error messages are written to the monitor display. These messages can also be saved in a text file for later review.

   If you want the messages to be saved in a text file, enter the name of the text file and press the Return key. If a file by that name already exists, it is deleted.

```
                    TL/1 Compiler Options
              PRESS F3 TO START COMPILING
                    OR QUIT TO EXIT

Save error messages in text file ▓_____
Generate standard warning messages              NO




              No warning messages
```
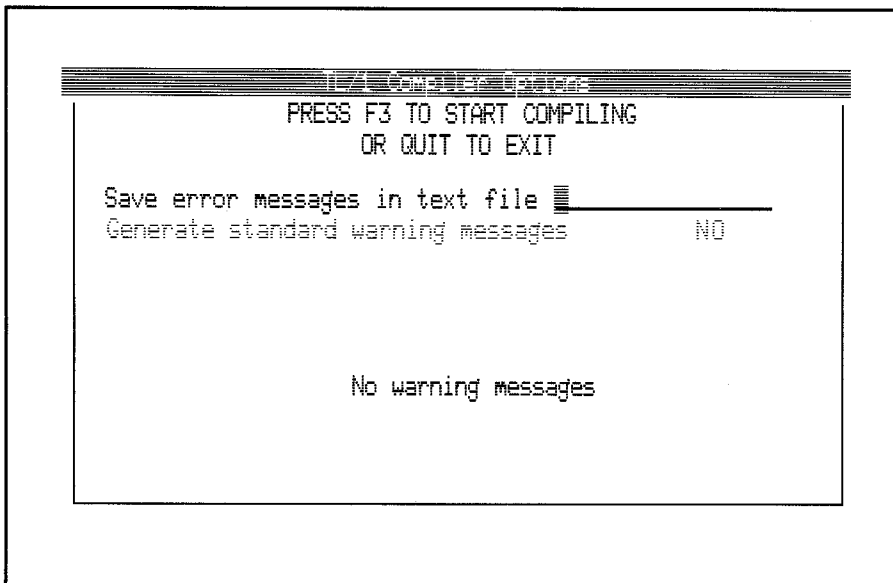
Figure 3-9: TL/1 Compiler Dialog Window

If you do not want to save these messages in a text file, leave the field blank, and press the Return key.

7.   Observe the prompt, "Generate standard warning messages". Use the Field Select key to select YES or NO. This prompt controls whether the TL/1 compiler generates optional warning messages when it compiles a program. Warning messages are precautionary only, and advise of TL/1 constructs that may be bugs.

Use the Field Select key to select YES and press the down-arrow key. Observe that when YES is selected, five additional prompts appear in the lower half of the Dialog Window, as shown in Figure 3-10.

8.   Observe the five prompts, Undeclared formal parameters, Implicit variables, Uninitialized global variables, References of unassigned variables and Unused global variables. These prompts can be turned on and off independently, and control whether the compiler generates warning messages when it detects these conditions in a program. Use the Field Select key to select YES or NO for each prompt. Use the Return key or arrow keys to move between the prompts. "Using the Compiler Options for Diagnostics" further on in Section 3 describes each prompt in detail.

Setting each prompt to YES instructs the compiler to check for each of these conditions, and results in the maximum number of warning messages. When all five prompts have been responded to, press the COMPILE softkey (F3).

9.   The compilation process begins with the display "TL/1 Compiler" followed by a list of status and error messages. Use the Scroll Lock key to stop the messages from scrolling off the monitor screen. Compilation is complete when the message "Press Msgs key to continue" appears.

```
                     TL/1 Compiler Options
                PRESS F3 TO START COMPILING
                      OR QUIT TO EXIT

  Save error messages in text file
  Generate standard warning messages          ▓▓▓

  Additional warning messages generated for:
     Undeclared formal parameters:                 NO
     Implicit variables:                           NO
     Uninitialized global variables:               NO
     References of Unassigned variables:           NO
     Unused global variables:                      NO
```
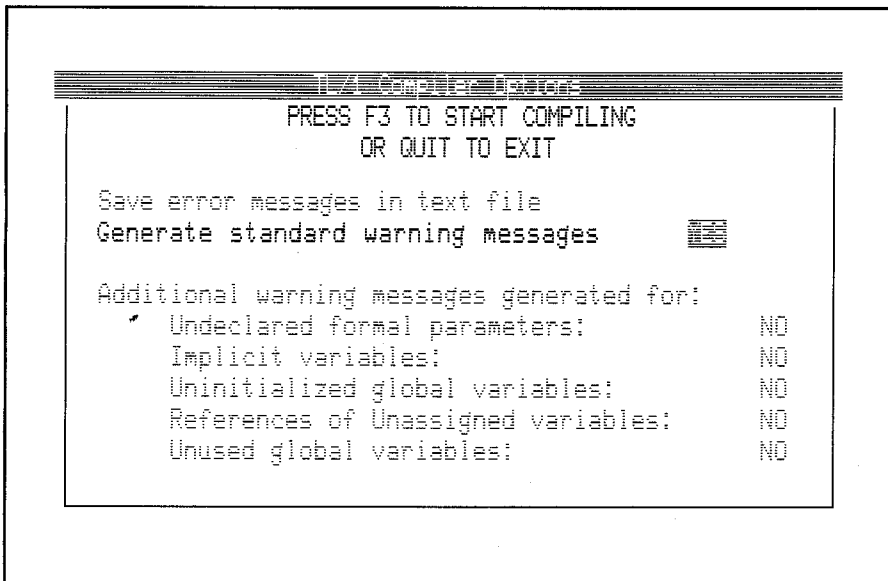
Figure 3-10: Lower Half of TL/1 Compiler Dialog Window

To embed the compiler error and warning messages directly into the program, edit the program and press the CHECK softkey. To review the compilation messages, edit the text file specified in step 6 (if applicable).

Using the Compiler Options for Diagnostics

When compiling a TL/1 program, part of the procedure includes changing the TL/1 compiler options. The compiler has five options which control the warning messages which can be issued. These options are:

- **Undeclared formal parameters:** Warns about undeclared formal parameters to programs, functions, exercisers or handlers.
- **Implicit variables:** Warns about variables declared implicitly by assigning to them.
- **Uninitialized global variables:** Warns about uninitialized global variables.
- **References of unassigned variables:** Warns about variables that were not assigned a value before being referenced (this is the default only for string variables).
- **Unused global variables:** Warns about global variables that are declared but not used.

Diagnostics emitted during compilation have the following format:

```
podtest(5): warning: variable 'y' unused
```

The parts are:

File Name: podtest

Line Number of the Offender: (5)

(Optional) Non-fatal Error: warning:

Error Description: variable 'y' unused

The following TL/1 program contains some illustrative errors:

```
program foo (arg1, arg2)
     numeric arg1
     string s1

     y = = 3
     s1 = 3
     s1 = arg2
end bar
```

The messages without any option causes the following output:

```
foo(5): syntax error
foo(6): operands of '=' have incompatible types
foo(8): warning: END name 'bar' does not match
  program name 'foo'
foo(3): warning: variable 's1' set but not used
foo(2): warning: variable 'arg1' unused
```

A few of the diagnostics above deserve special mention. First, note that line 6 attempts an assignment of a numeric constant to a string variable. Where possible, the compiler options verify the legality of types in all expressions, assignments, etc.

Next, notice that variable s1 is declared as not used. This warning is issued even though s1 has had a value assigned to it, in the belief that a variable is not really used unless it is referenced somewhere in an expression. This is only a warning.

Since the compiler options check for a large number of possible errors, it is not feasible to illustrate all of them here. It is important to note that the compiler options cannot find errors that occur at run-time; for example, expressions that evaluate to out-of-range values will not be detected. The remainder of these examples discuss certain key features of the compiler options, particularly optional diagnostics controlled by the selected option.

Example of an Undeclared Format Parameter Warning

This option warns about undeclared formal parameters to programs, functions, exercisers, or handlers. With this flag, the test program above, foo, would have an additional diagnostic generated:

```
foo(7): warning: type of argument 'arg2'
    undeclared
```

Example of an Implicit Variable Warning

This option turns on diagnostics about variables that are declared by assigning to them, as opposed to a formal declaration. Most users will have little use for this particular diagnostic, as such implicit declarations are supported by the TL/1 language specification. However, some users prefer to declare all variables explicitly, and may find this diagnostic useful. For example, the compiler option generates no diagnostic output for the following program:

```
program test2
    i = 3
    test3 (i)
end program
```

However, with this option, the compiler generates the following diagnostic:

```
test2(2): warning: 'i' implicitly defined
```

Another reason to avoid implicit type declarations is that the compiler option may not always be able to discern the type of a variable declared this way. The compiler option does not track the types of values returned by functions and programs, since there is no explicit support by the language syntax for declaring this information. Therefore, the following implicit declaration of the variable foo does not give the compiler option any type information:

```
foo = func1 (3)
```

Suppose the actual execution of the above assignment results in foo having type numeric. Since the compiler option was not able to determine foo's type, it will not warn about the subsequent type mismatch in the following statement:

```
foo = "foo"
```

If foo had been declared explicitly, the compiler option would have caught this error.

Example of an Uninitialized Global Variable Warning

This option turns on diagnostics about uninitialized global variables. It is incorrect to use a global variable without first assigning it a value, either via a default value specification in the declaration or via an *assignment* statement. TL/1 assigns reasonable values to uninitialized global variables, which is the reason why this feature is optional.

If this option is not selected, the compiler generates no diagnostics for the following TL/1 program:

```
program test3
    function foo1
       declare
              global numeric glob1
              global numeric glob2
              global numeric glob3 = 5
              global numeric glob4
       end declare
       glob1 = 3
    end foo1
    function foo2
       declare
              global numeric glob1
              global numeric glob2 = 4
              global numeric glob3
              global numeric glob4
              numeric n1
              numeric n2
       end declare
       n1 = glob1
       n1 = glob2
       n1 = glob3
       n1 = glob4
       n1 = n2
       n2 = n1
    end foo2
    foo1 ()
    foo2 ()
end program
```

With the option selected, the compiler generates the following diagnostic:

```
test3(7): warning: global variable 'glob4'
   never assigned a value
```

The compiler does not analyze the control flow of the program to verify that global variables are actually initialized before they are used, since this information cannot be determined by the kinds of static analyses performed by the compiler option. All the compiler can check is that global variables have been initialized.

Example of a Reference of Unassigned Variables Warning

This option turns on warnings about variables that were not assigned a value before being used. This behavior is the default for string variables, so this option actually affects only numeric and floating variables. For example:

```
program test4
    declare numeric n
    declare floating f
    declare string s
    foo1 (n)
    foo2 (f)
    foo3 (s)
end program
```

When analyzed by the compiler without this option, the compiler produces the following diagnostic:

```
test4(7): 's' has not been assigned a value
```

With the option selected, two additional diagnostic lines appear:

```
test4(5): 'n' has not been assigned a value
test4(6): 'f' has not been assigned a value
test4(7): 's' has not been assigned a value
```

Example of an Unused Global Variable Warning

This option turns on diagnostics about unused global variables. For example:

```
program test5
    function foo1
        declare
            global numeric glob1
            global numeric glob2
        end declare
    end foo1
    function foo2
        declare
            global numeric glob1
            global numeric glob3
            numeric n1
```

3-31

```
            end declare
            n1 = glob1
            glob1 = n1
         end foo2
         foo1 ()
         foo2 ()
end program
```

If this option is not selected, no diagnostics are generated by the
compiler for the above program. If the option is selected, the
following two diagnostic lines appear:

```
test5(5): warning: global variable 'glob2' unused
test5(11): warning: global variable 'glob3' unused
```

Example of Built-In Function Checking

The compiler options is capable of checking the use of built-in
TL/1 functions (for example, gfi, podinfo, etc.). With this
information, the compiler is capable of checking semantic
constraints on calls of built-in functions in either positional or
keyword notation.

For positional notation function calls, the compiler checks the
quantity and types of the arguments. The compiler also checks
that it is legal to call the function in positional notation; for
example, gfi may be called only in keyword notation, also
known as slot notation. For example, for the following
program:

```
program test6
    gfi (3)
    read (3,3)
    write ("foo", 3)
end program
```

The compiler generates the following diagnostics:

```
test6(2): 'gfi' must be called in keyword
  notation
test6(3): too many arguments to 'read'
test6(4): argument number 1 to 'write' is the
  wrong type
```

For keyword notation function calls, the compiler checks the keyword names, also known as slot names, and types, and ensures that the grouping of keyword arguments is legal. For example, for the following program:

```
program test7
    gfi fail "foo", status "bar"
    gfi accuse 3
    clip ref "foo"
end program
```

The compiler generates the following diagnostics:

```
test7(2): illegal combination of arguments to
  'gfi'
test7(3): argument 'accuse' to 'gfi' does not
  take a value
test7(4): required argument 'pins' to 'clip' is
  missing
```

Examples of Return Value Checking

The compiler also monitors *return* statements to check that functions and programs do not return more than one type, and to verify that functions and programs that return a value do not also "fall off the end" without returning a value. For example, for the following program:

```
program test8 (arg)
    declare numeric arg
    if (arg) then
        return (3)
    end if
    return ("foo")
end test8
```

The compiler generates the following diagnostics:

```
test8(6): warning: 'test8' returns more than
  one type
```

For the following program:

```
program test9 (arg)
    declare numeric arg
    if (arg) then
        return (3)
    end if
end program
```

The compiler generates the following diagnostics:

```
test9(6): warning: 'test9' has RETURN
    (expression), and RETURN
```

This indicates that test9 is in danger of "falling off the end," when it has been established that test9 is expected to return a value (at least in some context).

Examples of Control Flow Checking

The compiler also analyzes control flow to verify that statements are reachable. For example:

```
program test10
    return (3)
    foo (3)
end program
```

generates the diagnostic:

```
test10(3): warning: statement not reached
```

Analyses of control flow and return type often interact in ways that the user should be aware of, especially since return type diagnostics are sometimes difficult to get rid of if you tend to return values from inside loops that are guaranteed to terminate for reasons that are not obvious to the compiler. For example, the compiler does not warn about "falling off the end" of the following program:

```
program test11
    if (foo () ) then
        return (1)
    else
        return (3)
    end if
end program
```

This is because control flow analyses revealed that "falling off the end" was not possible. However, the compiler warns about "falling off the end" of the following program:

```
program test12
    declare numeric i
    for i = 0 to 100
       if (foo (i)) then
             return (0)
       else if (bar (i)) then
             return (1)
       end if
    next
end program
```

The compiler generates the following diagnostics:

```
test12(10): warning: 'test12' has RETURN
   (expression), and RETURN
```

In the above example, it is not obvious to the compiler that the loop will not terminate without returning (assuming that it does). The simplest way to prevent the compiler warnings in this situation is to add a *return* statement at the end.

The compiler does recognize that infinite loops of the format loop .. end loop cannot be exited except via *goto* and *return* statements. Thus the compiler does not warn about this program:

```
program test13
    loop
       if (foo() ) then
             return (3)
       end if
    end loop
end program
```

However, the compiler warns about the equivalent program:

```
program test14
    loop while 1
        if (foo ()) then
                return (3)
        end if
    end loop
end program
```

The compiler generates the following diagnostic:

```
test14(7): warning: 'test14' has RETURN
    (expression), and RETURN
```

The compiler does not examine the expression supplied to *while* statements for the possibility of ever being false. Therefore, the way to get the compiler to ignore return types for constructs that depend on eventually returning from inside an infinite loop is to use a loop .. end loop block to implement the loop.

Executing a TL/1 Program   3.1.10.

After being debugged, TL/1 programs are usually run from the operator's keypad by pressing the EXEC key and entering the UUT name and program name. Even though the program name might be in lower-case within the program, the operator's interface allows you to enter it in upper-case. When you press the ENTER key, the 9100A/9105A searches for your program. First it looks in the currently selected UUT directory. If the program is not there and if there is a pod plugged into the 9100A/9105A, the pod description for that pod is searched for the program. If the program still isn't found, the program library is searched.

A TL/1 program can be executed by calling it from another TL/1 program. When one program calls another, the same three disk locations are searched to find the called program.

If the TL/1 program has been pre-compiled, the compiled form of the program will be executed; otherwise, the source program will be compiled and executed (the compiled form is not saved on the disk). In addition, pre-compiled programs that are out-of-date with respect to their source programs are automatically recompiled before being executed.

TL/1 Syntax   3.1.11.

Each line of a TL/1 program has up to three pieces, which must be placed in the following order: label, statement, and comment. A line in a TL/1 program may consist of a label only, a statement only, a comment only, or any combination of these. It may also include none of them (a blank line).

The label is a character string that meets the requirements for a variable name (see Section 2.1, "Name Conventions," in the *TL/1 Reference Manual*) and that ends with a colon. A comment begins with an exclamation point (!); none of the text after this exclamation point is seen or executed by TL/1.

The *TL/1 Reference Manual* gives a complete explanation of each TL/1 command, including syntax in typed form (metasyntax) and picture form (syntax diagram). The first pages of the "TL/1 Alphabetical Reference" section of the *TL/1 Reference Manual* show the conventions used in the syntax of TL/1 commands.

Many TL/1 commands can be written in two forms: keyword notation and positional notation.

With positional notation, only argument values are entered and they must be entered in the correct order. No argument values may be omitted. A TL/1 command written in positional notation might look like this:

```
x = getromsig (0,$7FF,$FFFFFFFF,2)
```

With keyword notation, each argument value is preceded by a keyword associated with that argument. For example, the command above could be written as follows:

```
x = getromsig addr 0, upto $7FF, addrstep 2
```

In keyword notation, it is much clearer that this command would gather the signature from ROM starting at address 0 and ending at hexadecimal address 7FF, using an address step of 2. The *mask* argument was not needed in the keyword notation because the default value of FFFFFFFF was used instead. Keyword notation is usually preferred because it is easier to read and it provides better documentation for users who have not written the programs. Keyword notation also allows default values to be used with optional arguments. And, keyword notation reduces the chances of making errors, which could be caused by mixing up the order of argument values required by positional notation.

DATA TYPES, VARIABLES, AND EXPRESSIONS 3.2.

This section discusses the kinds of data that TL/1 can manipulate, the operators and functions that operate on data, and how simple variables and arrays are both declared and used. This section summarizes information presented in Section 2 of the *TL/1 Reference Manual*.

Data Types   3.2.1.

TL/1 supports three kinds of data: integer numbers (data type: numeric), floating-point numbers (data type: floating), and strings (data type: string). Integer numbers in TL/1 are 32-bit positive integers which have values from 0 through 4,294,967,295 (base 10) or from 0 through FFFFFFFF (base 16). Floating-point numbers use the IEEE standard for double-precision floating-point numbers, except that Infinity and NaN (Not a Number) are not supported. Strings in TL/1 are sequences which contain from 0 through 255 ASCII (8-bit) characters.

Numeric values may be written in either hexadecimal or decimal notation. Hexadecimal numeric constants must be prefixed with a "$" character. The hexadecimal characters A through F must always be written in upper-case. For example, the decimal number 43 could be expressed as any of the following:

$2B   (hexadecimal)

43   (decimal)

String constants are written as a sequence of characters placed between double quotes:

""   is a string containing no characters.

"abc"   is a three-character string.

ASCII characters which do not have a printable representation in strings can be placed in strings by using backslash escapes:

\"   is the string quote character.

\nl   is the newline character (defines the end of a line and does not necessarily include a line feed).

\\   is the backslash character itself.

HH\   is the ASCII character corresponding to the two-digit hex number *HH*.

Variables   3.2.2.

A TL/1 variable is characterized by:

- **Name** - an identifier by which the variable's value is known. See Section 2 of the *TL/1 Reference Manual* for information on legal variable names.

- **Type** - numeric, string, or floating.

- **Value** - an initial value, either a number (floating or numeric) or string, depending upon the variable's type. A variable has no value until one has been assigned, which is discussed in a subsequent paragraph.

- **Scope** - local, global, or persistent. A variable is defined to be either valid only within an invocation block (local scope) or valid both within an invocation and outside it (global or persistent scope). A local variable is accessible only in the block in which it is defined. For more information on blocks, see Section 3.3 of this manual and the sections in the *TL/1 Reference Manual* covering the *program, function, handle,* and *exercise* commands. A global variable is accessible in any block that contains a *declare global* statement for that variable. Likewise, a persistent variable is accessible in any block that contains a *declare persistent* statement for that variable.

Once a variable is created, its name and type are fixed—only the value may be changed. The value of a variable is accessed by mentioning the variable in an expression such as:

```
print beta
```

A variable is assigned a new value by writing the variable's name to the left of an equals sign in an assignment statement:

```
beta = 4
```

The variable above, *beta,* is called a simple variable: it holds only one value at a time. TL/1 also allows arrays of a single data type. Arrays may have any number of subscripts, limited only by available memory. Individual array elements are referred to by comma-separated subscripts enclosed in square brackets:

```
testvec[1] = $FE00

uutloc[i,j]

signame[testvec[i], uutloc[i,j]]
```

Only individual array elements, like those shown in the examples above, may be used in TL/1 assignment statements, expressions, or as function arguments.

Variable Declarations

The *declare* statement declares the name, scope, array, type, and default value for a single variable. For example, the following declaration uses only one statement:

```
declare global string uut_name
```

TL/1 also allows multiple declarations with a declaration block, as shown with the following example:

```
declare
    global numeric array [0:9] intr_vec
    string 'error message' = "no error"
end declare
```

Variable declarations using the *declare* statement may occur in program, function, handler, and exerciser blocks.

A local variable that is not an array does not have to be declared; simply assigning a value to it is enough to implicitly declare the variable. The type of the variable is taken to be the type of the expression assigned to it.

The table below shows how the different variable attributes affect valid variable declarations:

Variable	Scope	Implicit declaration allowed?	Default value allowed?
simple	local	yes	yes
	global	no	yes
	persistent	no	yes
array	local	no	no
	global	no	no
	persistent	N/A*	N/A*

* persistent variables may not be arrays.

Assigning Default Values to Variables

A default value may be given to a simple variable in a declare statement by following the variable name with an equals sign and a value:

```
declare numeric nr_of_pins = 24
declare
    string start_msg = "Beginning test"
    numeric pin_mask = $DFE0
    floating Vcc = 5.0
end declare
```

Remember that default values may not be assigned to arrays.

Persistent Variables

TL/1 variable declarations can use an optional persistent attribute, using the same syntax applicable to global variables. For example, the following are allowable persistent variable declarations:

```
declare persistent numeric n

declare persistent string s1

declare
  persistent floating f
end declare
```

Persistent variables allow TL/1 programs to preserve the values of certain variables across the execution of several programs. This is especially useful for programs executed during GFI, which otherwise would have no convenient way of communicating variable values among various stimulus programs.

Persistent TL/1 variables are similar to TL/1 global variables, except that their values survive execution. However, they are not in the same "name space" as global variables, so that in the following TL/1 program, the two instances of the foo variable actually refer to distinct variables:

```
program prog1

  declare persistent numeric foo

  function xyzzy

     declare global numeric foo

  end function

end program
```

Persistent variables have the same property as global variables in that synonymous declarations in different functions and programs refer to the same variable; for example, in the program:

```
program prog2

  declare persistent numeric foo

  function Xyzzy

          declare persistent numeric foo

  end function

end program
```

Both declarations of foo refer to the same variable.

Persistent variables may have type numeric, string, or floating. They may not be declared as arrays. Also, arguments to the enclosing block (for example, to a function or program) may not be declared as persistent variables.

The model for implementation of persistent variables consists of two entities: the persistent variable set and a set of local copies in the currently running TL/1 program, and a number of operations for propagating elements of one set to the other. This model is is shown in Figure 3-11.

CURRENTLY RUNNING TL/1 PROGRAM
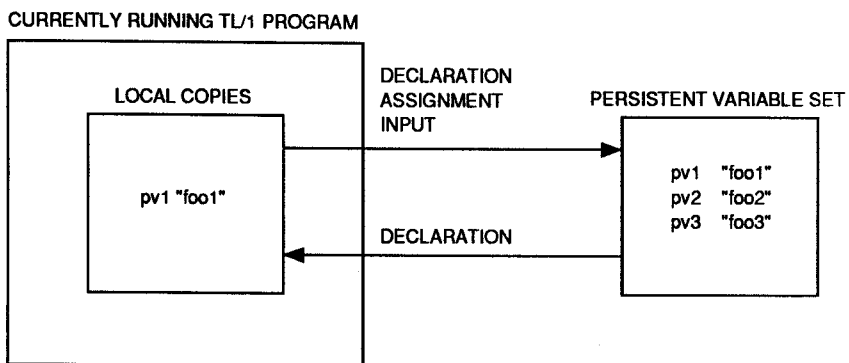
Figure 3-11: Persistent Variables Model

Certain TL/1 operations retrieve persistent variables from the persistent variable set, while other operations write them to the persistent variable set. The set of local copies known by the TL/1 program is also called the currently active set of persistent variables.

When a TL/1 declaration for a persistent variable is processed, the persistent variable set is first checked to see if it contains the declared variable. If it does, the value is retrieved and used to set the local copy. If not, the value held by any previously declared local copy is used. If there is no local copy, then a default value is assigned. Finally, after the value of the local copy is established, the variable and its value are written back to the persistent variable set. If the set did not previously contain this variable, the write operation adds it to the set.

When a TL/1 operation occurs that assigns a value to a persistent variable (e.g., input, assignment, for loop variable updating), the local copy is first updated to the new value. Next, the variable and its value are written to the persistent variable set. If the set did not previously contain this variable, the write operation adds it to the set. In general, the persistent variable set already contains the variable, since it must have been declared prior to use in the TL/1 program. However, the persistent variable set contents may have been reset in the interim as described in the following paragraphs.

When a TL/1 operation occurs that references the value of a persistent variable (for example, usage of the value in an expression), only the local copy is checked. The contents of the persistent variable set are not used. In particular, if the persistent variable set does not contain the referenced variable, the reference does not add the variable back to the set.

The values of all currently active persistent variables can be set to zero values with the *clearpersvars* command. *clearpersvars* sets all currently active numeric persistent variables to 0, all currently active floating persistent variables to 0, and all currently active string persistent variables to "" (the empty string). Currently active means the set of persistent variables known so far by the TL/1 program executing the *clearpersvars* command (the set of local copies).

The currently active set does not include persistent variables with declarations in the current program that have not been processed, nor does it include persistent variables installed in the persistent variable set by some previously executed TL/1 program, but unknown by the current program. This feature of the *clearpersvars* command is essential to hide non-volatile information in the persistent variable set which is not relevant to the current application.

For example:

```
program prog3
  declare persistent numeric pn1
  declare persistent floating pf1
  declare persistent string ps1

  clearpersvars()

  ! pn1 is now 0, pf1 is now 0.0, and ps1 is
  ! now "" in both the local copy set and the
  ! persistent variable set.
  ! all other variables in the persistent
  ! variable set are unchanged.

end program
```

The set of persistent variables always starts out empty each time the 9100A/9105A is turned on, and accumulates from that point. Resetting the 9100A/9105A with the front panel RESET key does not affect the set.

The persistent variable set is explicitly emptied with the TL/1 *resetpersvars* command. *resetpersvars* resets the persistent variable set to its initial empty condition. An important feature of this command is that even though the persistent variable set is emptied, the set of local copies active in a currently running TL/1 program retain their values locally to the execution of the program. Persistent variables which receive a value or are declared after resetting the set are added to the persistent variable set as shown in Figure 3-12.

```
program prog4

    !
    ! func1 - establishes persistent variables
    ! pv1...pv4, and assigns them initial
    ! values
    !
    function func1
        declare
          persistent string pv1
          persistent string pv2
          persistent string pv3
          persistent string pv4 = "foo4"
        end declare

        pv1 = "foo1"
        pv2 = "foo2"
        pv3 = "foo3"
    end function


  !
  ! func2 - resets the persistent variable set,
  ! then assigns a new value to pv2 and
  ! accesses the value of pv3.
  ! the former operation will add pv2 back to
  ! the persistent variable set, while the
  ! latter operation will not add pv3.
  !
  function func2
    declare
      persistent string pv1
      persistent string pv2
      persistent string pv3
      persistent string pv4
    end declare

    resetpersvars()   ! reset the set of
                      ! persistent variables

   pv2 = pv3
  end function
```

(Continued on next page)

Figure 3-12: Persistent Variable Set Program Example

```
!
! func3 - declares pv1 again for the first
! time since resetting the persistent
! variable set, thus adding it back to
! the set.
!
function func3
  declare
    persistent string pv1
  end declare
end function

func1()
func2()
func3()

end program
```

Figure 3-12: Persistent Variable Set Program Example *(cont)*

For the above program, the following describes the sequence of operations on the set of persistent variables during execution of the program (assuming that the set of persistent variables is initially empty):

1. During the processing of the declarations for function func1, the persistent variables pv1, pv2, pv3, and pv4 are added to the persistent variable set. By the time func1 returns, assignment statements to pv1, pv2, and pv3 have set their values to "foo1", "foo2", and "foo3", respectively, and pv4 has been set to "foo4" by the default initializer in its declaration.

2. After executing the *resetpersvars* command in func2, the persistent variable set is empty. Note that even though pv1 through pv4 are declared in func2, the declarations do not have the effect of adding them back to the persistent variable set, since their effect occurs before the *resetpersvars* command is executed.

    Again, note that even though pv1 through pv4 are removed from the persistent variable set, they retain their values in the currently running program. Thus, the value of pv1 is still "foo1", pv2 is "foo2", etc. If no further operations are performed on these variables, they disappear on termination of the program.

3. After executing the assignment statement:

    pv2=pv3

    the variable pv2 is added back to the persistent variable set, since a value was assigned to it. pv3 was not added to the set, since referencing the value of a variable formerly declared persistent is not sufficient to add it to the set. The value of pv2 is now "foo3".

4.   Since pv1 is declared in func3, it is added back to the set when the declaration is processed. The value associated with pv1 is still "foo1"

Thus, when the above program terminates, the persistent variable set contains two variables, pv1 and pv2, with values "foo1" and "foo3", respectively.

It is important to note that the *clearpersvars* function writes new values to the currently active set of persistent variables. The effect of the following call to *resetpersvars* is negated by the subsequent call to *clearpersvars* for all persistent variables known to the currently active program (persistent variables pv1 through pv3):

```
program prog5
  declare
     persistent string pv1
     persistent string pv2
     persistent string pv3
  end declare

  resetpersvars ()
  clearpersvars ()
end program
```

The previous program has the effect of discarding all other persistent variables, then setting the values of pv1, pv2, and pv3 to the empty string and installing them as the entire contents of the persistent variable set.

The Assignment Statement

An assignment statement can have either of the forms below:

```
variable = expression

array[subscript1, subscript2, ..., subscriptn]
  = expression
```

Values can be assigned only to individual array elements, not to entire arrays.

Passing Arguments

Passing an argument to a program or function is another way of assigning a value to a variable. The function block *uuttest* is defined below followed by an invocation (*execute* command) of *uuttest:*

```
function uuttest (addr)
    declare numeric addr
            .
            .
            .
end function

execute uuttest addr $100
```

The *addr* argument for *uuttest* will be assigned the hexadecimal value of 100 as part of the invocation process.

Operators   3.2.3.

The TL/1 operators for string, numeric, and floating-point values are explained in Section 2.4 of the *TL/1 Reference Manual;* the order in which operators are applied (precedence) is explained in Section 2.5 of the *TL/1 Reference Manual*. Many of these operators will be familiar to those with experience using any high-level language.

TL/1 also provides logical string operators. Certain functions for the pod and the I/O modules require long sequences of bits which may take on the values 0 (low), 1 (high), or X (undefined). TL/1 has provided operators which compute the and, or, xor, and complement (cpl) of these strings. The least-significant bit of such strings is defined to occupy the right-most place in the string. If two strings of unequal length are combined using a logical operator, the shorter string is extended to the left (high-order bit positions) with zeros. The operators themselves produce values as shown in the following table:

A	B	A and B	A or B	A xor B	cpl A
0	0	0	0	0	1
0	1	0	1	1	1
0	X	0	X	X	1
1	0	0	1	1	0
1	1	1	1	0	0
1	X	X	1	X	0
X	0	0	X	X	X
X	1	X	1	X	X
X	X	X	X	X	X

Expressions   3.2.4.

TL/1 uses expressions to combine data values into new values.
These new values are usually assigned to variables or passed as
arguments to functions or programs. Expressions in TL/1 are
composed of simple variables, array elements, constants,
function invocations, and operators. Some examples of
expressions are:

```
pinnr                   a simple variable
maskval [j]             an array element
read ($F0) + 4          a function invocation
                         and addition
read addr $F0 - $10     a function invocation
                         and subtraction
300                     a numeric constant
j * 4                   a multiplication
                         operation
2.5/1E-3                a floating-point division
label + "abc"           a string concatenation
                         which adds the characters
                         abc to the end of the
                         string named label.
```

Math Functions   3.2.5.

TL/1 provides a number of math functions for floating-point
numbers:

Math Function	TL/1 Function
absolute value	fabs
square root	sqrt
exponential	pow
logarithm (base may be specified)	log
sine	sin
inverse sine	asin
cosine	cos
inverse cosine	acos
tangent	tan
inverse tangent	atan

In addition, the *natural* command provides built-in constants for
the transcendental numbers π and e.

System Functions   3.2.6.

The *systime* Function

The *systime* function returns the number of seconds since
00:00:00 on the arbitrary date of January 1, 1980. The *readdate*
function converts the number returned by *systime* into a usable
string for the current date. The *readtime* function converts the
number returned by *systime* into a usable string for the current
time.

The *systime* function is also useful in timing the duration of events; the difference between the values returned by two invocations of *systime* is the number of seconds required to perform an action:

```
start = systime ()
execute test23 ()
print using "the test took #####@
 seconds" systime() - start
```

The *sysdata* and *sysaddr* Functions

These functions are primarily for use with exercisers and handlers invoked from the operator's keypad. Operations directed to the pod (such as *read, write,* and *writefill*) set the values returned by *sysaddr* and *sysdata*. The exact values will depend upon the circumstances; *sysdata* and *sysaddr* together are meant to provide an easy-to-use feature that reduces the data entry load on the operator.

PROGRAM STRUCTURE AND FLOW CONTROL   3.3.

Block Structure of TL/1   3.3.1.

Every TL/1 program begins with a *program* statement and terminates with an *end program* statement. These two statements delimit the program definition block, which encloses all the declarations and executable statements of the program. A program is the smallest unit of TL/1 code that may be executed (or invoked) using the EXEC key on the operator's keypad. Here is a simple TL/1 program:

```
program echo(message)
    declare string message = "Hello, world!"
    n = open device "/term2", as "output"
    print message
    close channel n
end program
```

The program consists of the name, *echo,* a declaration and default value for the argument message, and three additional statements. The first opens a channel to the monitor, the second prints the message on the channel, and the third closes the channel.

TL/1 is a block-structured language. Blocks serve to group:

- Statements - In the *echo* program, only the statements defined in the *echo* program block will be executed.

- Variables - The variables that are declared inside *echo* are only known to that program (except for global variables, which will be covered later). When *echo* finishes execution, the storage used for variables is reused for other purposes.

Every block consists of a statement that starts the block (such as *program, declare,* or *if*) and another statement that ends the block (such as *end program, end declare,* or *end if*). A summary of the characteristics of TL/1 blocks is shown in Figure 3-13.

Block Type	Name	Contain Function Definitions?	How Is It Invoked?	Can Variables Be Local to Block Only?
program	1-10 characters, valid file name	yes	EXEC key, *execute* statement, or invocation in expression	yes
function	1-255 characters	no	*execute* statement or invocation in expression	yes
handler	fault name 1-255 characters	no	*fault* statement	yes
exerciser	fault name 1-255 characters	no	LOOP key	yes
declare	none	no	*declare* statement	N/A
if	none	no	*if* statement	no
loop for	none	no	*loop* statement	loop index only
loop while	none	no	*loop* statement	no
loop until	none	no	*loop* statement	no

Figure 3-13: TL/1 Block Types

The program, function, handler, and exerciser blocks share the following characteristics:

- A name - Names for programs can be 1 to 10 characters long. Names for function, handler, and exerciser blocks can be 1 to 255 characters long.
- An argument list - The argument list tells which values each block expects to receive when it begins execution. The actual declaration of the argument variables gives the type and may specify a default value, if it is appropriate.
- Variable declarations - Any variables used by a block should be declared before use. While this is optional for local variables (variables known only inside the block being defined), it is required for global and persistent variables.
- TL/1 statements - The statements define the actions to be performed when the program is executed.

The program block is the principal building block for TL/1. It contains:

- A *program* statement, which gives a name to the program and lists the program arguments. The name of the program must match the name of the file containing the TL/1 program.
- Declarations for any local or global variables.
- Any function definition blocks, fault condition handler definition blocks, or fault condition exerciser definition blocks.
- TL/1 statements that make up the program.
- An *end program* statement, which defines the end of the program block.

Figure 3-14 shows a skeleton of a program, which contains a function definition block, a fault condition handler definition block and a fault condition exerciser definition block. The *example* program shows that a program's variables are declared

```
program example (start, finish)
declare
    numeric start = 0
    numeric finish = 0
    floating frequency = 60.0
    global string active_space
    numeric my_variable
end declare
function do_test (addr, range)
    declare numeric addr
    declare numeric range
    declare global string active_space
    !  executable statements for function

        .
        .
        .
end do_test
handle some_fault (addr)
    declare numeric addr
    declare global string active_space
    !  executable statements for handler

        .
        .
        .
end some_fault
exercise a_fault (addr)
    declare numeric addr
    declare global string active_space
    !  executable statements for exerciser

        .
        .
        .
end a_fault
!  Executable statements for program

        .
        .
        .
end program
```

Figure 3-14: Program Structure Example

first, then functions are defined, then fault condition handlers or fault condition exercisers are defined, and finally the executable statements for the program are listed. The executable statements for each function, handler, or exerciser are included within each respective definition block.

How Programs and Functions Are Invoked　　　3.3.2.

TL/1 programs can be invoked from the operator's keypad or they can be called from another TL/1 program:

- A program is invoked from the operator's keypad by pressing the EXEC key and then providing the name of the UUT directory and the name of the program.

- A program is called from another program by using the *execute* statement:

```
execute example (0, 1000)
```

The statement above calls the program named *example* using *0* and *1000* as arguments.

- A program will be called when its name appears in an expression:

```
total = times (plus (a, b), minus (c, d))
```

The statement above calls the user-defined programs *times, plus,* and *minus.*

Functions are also called using the *execute* statement or by placing the function name in an expression.

When a program is invoked, it is first loaded from a disk file if it is not already in memory from a previous invocation; from then on, the 9100A/9105A processes programs and functions in the same way, as is shown on the next page.

1. Storage is allocated for the local variables of the newly created program or function.

2. Argument values from the calling statement are copied to the corresponding newly created local variables, overriding the default values of the called block. Any variables that do not appear in the calling statement are assigned default values.

3. Execution of the program or function begins with the first executable statement in the block.

Scope Rules for Programs and Functions 3.3.3.

A TL/1 program name appears both in the *program* statement and as the name of the file in which the program is stored (this file name appears in a 9100A/9105A directory). When the EXEC key is pressed or when an *execute* statement is performed, the search for the file containing the named TL/1 program follows this order:

1. USERDISK current UUT directory
2. USERDISK current pod directory
3. USERDISK program library directory

- If E-disk is loaded:

1. E-disk current UUT directory
2. E-disk current pod directory
3. USERDISK current UUT directory
4. USERDISK current pod directory
5. USERDISK program library directory

If the program isn't found in any of these directories, an error occurs.

The scope of a program name depends upon the directory in which the specified program is placed:

- A program file in the program library has a scope extending over all pod descriptions and UUT directories.

But if a program of the same name is placed in a pod description or UUT directory and the EXEC key or an execute statement is performed while testing that UUT or using that pod, the program in the program library will not be executed; instead, the program in the pod description or UUT directory will be executed.

- A program file in a pod description will be found only when that particular pod is being used. But a program of the same name in a UUT directory will be found before the program in the pod description when that particular UUT directory is the current directory.

- A program file in a UUT directory will be found only when that particular UUT directory is the current directory.

- The scope of a function name extends throughout the program block in which the function is defined. The function cannot be called from another program block.

Passing Arguments   3.3.4.

TL/1 provides a convenient method for passing variable values into program, function, handler, and exerciser blocks. These variables are called arguments; any necessary argument names and argument values appear in the statement that invokes the block. For example, the block below requires two numeric arguments, *start* and *finish:*

```
program test_uut (start, finish)
    declare
        numeric start = 0
        numeric finish
    end declare
```

The program *test_uut* could be called using any of the following statements:

```
execute test_uut finish 54, start 10
```

```
execute test_uut finish 15

execute test_uut (40, 50)
```

The first two *execute* statements use keyword notation; the name
of the argument is followed by the value to be assigned to it.
The third *execute* statement uses positional notation; in this
case, all arguments must be supplied and they must be supplied
in the order given by the *program* statement.

The arguments supplied to a program, function, handler, or
exerciser must be simple numeric or string values—arrays
cannot be passed as arguments. When the block is invoked, the
values of the arguments from the calling statement are copied to
the area of memory set aside for local variables in the block
being invoked.

Returning Values from Programs and Functions   3.3.5.

A program or function completes its work by executing either an
implied or an explicit return. At this point, all local variables
disappear along with all other information about the terminated
program or function. The return may be implicit due to an
implied *return* statement (which returns nothing) at the *end*
statement that terminates every TL/1 program and function. An
explicit return uses a *return* statement. It can either return no
value or a single numeric, string, or floating-point value (a
return cannot return an array of values). A program or function
may contain more than one *return* statement, but all *return*
statements in a program or function must return the same type of
object: no value, a numeric value, a floating-point value, or a
string value.

To use a returned value, you must use it in an expression. For
example, if *a_function* returns a value, it can be assigned to the
variable *x:*

```
x = a_function (3, 4)
```

When a program or function is invoked from an expression, an error will be reported if the program or function does not return a value or if it returns a value of the wrong type. For example, if *a_function* does not return a value, the following statement would cause an error:

```
print a_function (3, 4) + 5
```

Scope Rules for Variables   3.3.6.

Figure 3-10 also shows how information can be passed through both arguments and global or persistent variables. The scope rules of TL/1 define which variables are known inside programs, functions, handlers, and exercisers. The scope rules are simple:

*   A variable that is an argument to a block is visible only inside the block. The variables *start* and *finish* in Figure 3-10 are arguments in the *example* program, as are the *addr* and *range* arguments in the *do_test* function.

*   A variable that is not explicitly declared as global or persistent is a local variable. This means that *my_variable* is accessible only to the executable statements for the program *example,* not to the function *do_test* nor to the handler *some_fault* nor to the exerciser *a_fault.* A local variable is accessible only to the block in which it is declared, and not to any nested blocks.

*   A variable that is used in a block, but does not appear in a *declare* statement, is a local variable and is visible only inside that block.

*   A variable that is declared global in a *declare* statement is visible in every block that also contains a global declaration for the same variable.

*   A variable that is declared persistent in a *declare* statement is visible in every block that also contains a persistent declaration for the same variable.

Global variables exist from the time the EXEC key on the operator keypad is pressed until the time a different TL/1 program is run. Pressing the REPEAT key, which re-runs the program, permits global variables to retain their values.

If some function *a* calls another function *b,* the local variables in function *a* retain their values when *b* terminates, so that *a* can continue its work. But the local variables in *b* disappear when *b* terminates, just as the local variables in *a* disappear when it returns to its caller.

Conditional Flow of Control   3.3.7.

TL/1 uses block structuring to organize *if* and *loop* statements. Both are conditional statements; refer to Section 2.4, "Conditional Expressions," of the *TL/1 Reference Manual* for more information on formats for conditional expressions.

If Blocks

The *if* statement is used to select alternative courses of action based upon one or more conditions. The action of the *if* statement is to try all of the alternative conditions *(condition₁, condition₂, . . ., conditionₙ)* in order until one of the conditions evaluates to true (a non-zero value). If one of the conditions is true, then the corresponding action is executed. If none of the conditions is true and an *else* clause is present, the else action is executed. The actions themselves can be any list of TL/1 statements, including other *if* statements.

```
if <condition₁> then
    <action₁>
else if <condition₂> then
    <action₂>
         .
         .
         .
```

```
else if <condition_n> then
    <action_n>
else
    <action_else>
end if
```

The block terminator *end if* can also be written as *endif* for compatibility with BASIC, but this usage is not the preferred TL/1 form.

Simple If Statements

Where you need only a single condition, with no *else* clause, a simpler form of the *if* statement can be used. This statement has the form:

```
if <condition> then <statement_1>\<statement_2>\
    . . .
```

The statements, *statement_1, statement_2,* and so on, are executed only if the condition evaluates to true.

A Word about Compound Conditions

TL/1 always evaluates conditional expressions completely. This is usually not important but does make a difference in the following example:

```
if (a <> 0) and (b/a <> 3) then ...
```

The problem is that the subexpression *b/a* will be evaluated whether or not *a* is equal to zero, which can result in a divide-by-zero error. This kind of test must be converted into two *if* statements as shown below:

```
if a <> 0 then if b/a <> 3 then ...
```

Loop Blocks

The loop block is used to group statements which are to be executed repeatedly. This looping can proceed while some condition is true, or until some condition becomes true, or for some sequence of numeric values, or indefinitely.
The *loop while* block has the form:

```
loop while <condition>
    <action>
end loop
```

The effect is to repeatedly execute the action within the block delimited by the *loop while* and *end loop* statements as long as the condition is true. The condition is tested before any statements in the block are executed. The action may be any list of TL/1 statements including other loop blocks.

The *loop until* block has the form:

```
loop until <condition>
    <action>
end loop
```

The effect is to repeatedly execute the action within the block delimited by the *loop until* and *end loop* statements as long as the condition is false. The condition is tested before any statements in the block are executed.

The *loop for* block has the form:

```
loop for <var_index> = <expr_1> to <expr_2>
  [step <expr_3>]
    <action>
end loop
```

The *step* expression is optional and assumed to be 1 if omitted. The loop index variable (var_{index}) begins with the initial value, $expr_1$; while the index variable is less than $expr_2$, the statements inside the *loop* block are executed. Following each iteration of the loop, the value of $expr_3$ (which defaults to 1) is added to the index variable. The *end loop* statement can be replaced by the keyword *next,* which is compatible with BASIC; however, this is not the preferred TL/1 form.

Omitting a controlling condition such as *while, until,* or *for* creates an endless loop:

```
loop
    <action>
end loop
```

The action between *loop* and *end loop* is executed until some external event (such as a fault condition) causes control to be transferred outside the loop.

INPUT, OUTPUT, AND FILE COMMANDS   3.4.

This section provides an overview of how to use the TL/1 input and output commands.

1. The *open* command performs the initialization required to allow your program to read from or write to the device or file named in the device argument of the command. The *open* command returns a channel number which may be assigned to any numeric variable you specify. This channel number is used in subsequent *print* and *input* commands to identify the device to be used. Up to 16 channels may be open at any given time.

2. The *print* and *input* commands transfer information between the TL/1 program and the device. The channel number returned by *open* is used to identify the device to be used. Several devices can be open at once.

3. The *close* command breaks the association between the channel number and the device. Once a *close* has been performed, no further operations can be performed on the device unless another *open* command is performed for it.

File and Device Types

There are several kinds of devices which may be used from TL/1. These following devices produce and accept the printable ASCII data that the I/O commands are designed to handle:

- Operator's Interface - named "/term1". The name refers to the operator's display on output and the operator's keypad on input.

- Programmer's Interface - named "/term2". The name refers to the monitor on output and the programmer's keyboard on input.

- Windows - named "/term1/win" for a window on the operator's display and "/term2/win" (or "/win") for a window on the monitor. Additional arguments are provided in the *open* command to set window parameters.

- RS-232 Ports - named "/port1" and "/port2". Additional arguments are provided in the *open* command to control the communication through these ports.

- Disk Text Files - named using the system's file naming conventions (see Sections 2.8 and 3.4.6 of this manual).

- IEEE-488 Interface and Devices - named "/ieee" or "/ieee/*address list*". The names refer to either the IEEE-488 interface, or to a group of devices attached to the interface.

Opening Devices and Files

The *open* command makes a device or disk file accessible to your TL/1 program. The *open* command accepts several arguments, all optional, giving information about the device or file to be used; *open* returns a channel number which serves to identify the device or file in subsequent I/O function calls.

The *device* argument gives the name of the device or file to be opened. The *as* argument ("input", "output", "update", or "append") gives the direction of the I/O transfers. The *mode* argument tells whether I/O is done a line at a time ("buffered") or a character at a time ("unbuffered").

All arguments to the *open* command are optional. The applicable defaults provided are documented in the *open* command section of the *TL/1 Reference Manual*.

Buffered and Unbuffered Channels   3.4.3.

The *mode* argument of the *open* command governs several aspects of how the device attached to the channel is treated. Most of the argument's values apply to the operator's interface, programmer's interface, and the RS-232 ports.

Buffered Channels

Buffered mode is appropriate for operator input—the editing features are almost essential for manual data entry. Buffered mode input is also useful with disk files that have fixed data formats compatible with the *input using* command (which cannot be used in unbuffered mode). Buffered channels have the following characteristics:

- Record size: Input and output occur in units of lines rather than characters. An *input* command, for example, will wait for input until a line-terminating character (such as Return on the keyboard, or a newline character on the RS-232 port) is entered.

- Input data types: The *input* command reads numbers and multi-character strings in buffered mode, rather than the single-character strings read in unbuffered mode.

- Newline character: Newline characters printed on a buffered RS-232 channel are converted either to a single carriage return, or to a carriage-return/linefeed sequence, as selected by the SETUP MENU key.

Unbuffered Channels

Unbuffered channels are most useful when dealing with inter-machine communications, special operator interface functions (in which the format of the operator's or programmer's display must be very carefully controlled), or when the input format is too complex for *input using*.

Unbuffered channels have the following characteristics:

- Record size: Input and output occur in units of characters. An *input* command will read data as it is entered rather than waiting for a line-terminating character.

- Input data types: The *input* command places a one-character string into each of the string variables in its argument list; numeric and floating-point variables are not permitted. The *input using* command is illegal for unbuffered channels.

- Newline character: The newline character on an unbuffered channel may only be a carriage return.

Printing Newlines on Output Channels

The *print* command will always print a termination character (the default is a carriage return (CR)) after the last expression in its argument list. The *print using* command prints a carriage return only when a newline character appears in its format string. (See the *print using* command in the *TL/1 Reference Manual* for more information on format strings.)

A carriage return may be transformed into a carriage-return/linefeed (CRLF) sequence when printing to a buffered RS-232 port if the newline character in the SETUP MENU has been set to CRLF. The possible combinations are summarized in the following table:

Command	Channel Mode	Newline Character	
		CR	CRLF
print	buffered	CR	CRLF
	unbuffered	CR	CR
print using (with	buffered	CR	CRLF
\nl in the format)	unbuffered	CR	CR
print using (with	buffered	(none)	(none)
no \nl in the format)	unbuffered	(none)	(none)

I/O Commands   3.4.4.

The *input* command is used to read all data in TL/1. The simple form of *input* reads only decimal numeric data, floating-point data, and unformatted string data. The format strings used with *input using* are designed to handle the most common numeric, floating, and string input requirements, and format strings are particularly suited for machine-generated, fixed-format data files (such as CAD databases). When this isn't enough, the *input* command can read entire lines of input into string variables, which may be processed using the string functions of TL/1.

The *print* command is used to output data in TL/1. The simple form of *print* prints strings "as is", numeric numbers in decimal format, and floating-point numbers in scientific format. The *print using* command uses a format string to print numeric numbers in hexadecimal, decimal, or binary format; floating-point numbers in scientific or fixed-point format; and strings using fixed-width fields.

See the *print using* and the *input using* commands in the *TL/1 Reference Manual* for more information on format strings.

When no *on* clause appears, the *print* and *input* commands use the first channel that was opened with the appropriate direction ("input" or "update" for *input;* "update", "append", or "output" for *print*). For example, the following program reads a line from the operator's keypad and echoes it on the operator's display:

```
program example
    declare string line
    chan = open device "/term1"
    input line
    print line
    close channel chan
end example
```

The *poll* command returns I/O status information about devices or files accessible to a TL/1 program via an open channel. Most of this information is useful only for devices, but the "input" condition will also tell your program whether or not the end of a disk input file has been reached. To avoid an I/O error, your programs should make sure that a poll for an "input" condition on the input channel returns a non-zero value before using the *input* command.

The *delete* command is used to remove text files from a disk. The *delete* command cannot be used to remove directories or non-text files.

Windows   3.4.5.

The 9100A/9105A uses a window manager to manage all displays on both the operator's display and the monitor. A window manager allows the screen to be thought of a series of rectangular "screens" on the physical screen. For example, when using the programming interface, there is an info window that can be placed on top of the display window. The types of things you might want to do in a window are to display textual information, to display UUT pictures, to prompt for operator action, or to build menu-driven interfaces.

Each window is opened and closed using the TL/1 *open* and *close* commands. This allows normal print and input to be done on windows just as it is done on any other display device. Windows are permitted to overlap each other. What is displayed is determined by the order in which the windows were created. A new window is always on top of all the other windows. An existing window may be moved to the front or the back using the *winctl* command. The *winctl* command also permits making a window invisible by "hiding" it, and making an invisible window visible by "unhiding" it.

The location of the upper, left-hand corner of a window is specified by xorg and yorg (see Figure 3-15). The size of a window is specified by xdim and ydim. The size of the object to be displayed in the window is controlled by xscale and yscale. All references to locations inside a window and sizes of objects displayed in a window are made relative to the full-scale coordinates specified. For example, if xscale and yscale are both 1000, the center of the window is (500,500). If the object size is larger than the window, only part of the object will be visible at any given time.

The 9100A/9105A monitor displays 24 rows with 80 characters per row. The operator's screen displays 3 rows with 42 characters per row. Therefore, a window that exactly covered the monitor would use (0,0) for xorg-yorg and (80,24) for xdim-ydim. If you wish to specify locations in a window using character offsets from the origin of the window, you can set xscale = xdim and yscale = ydim. In this case, since the window is of size (80,24), the center of the window would be (40,12).
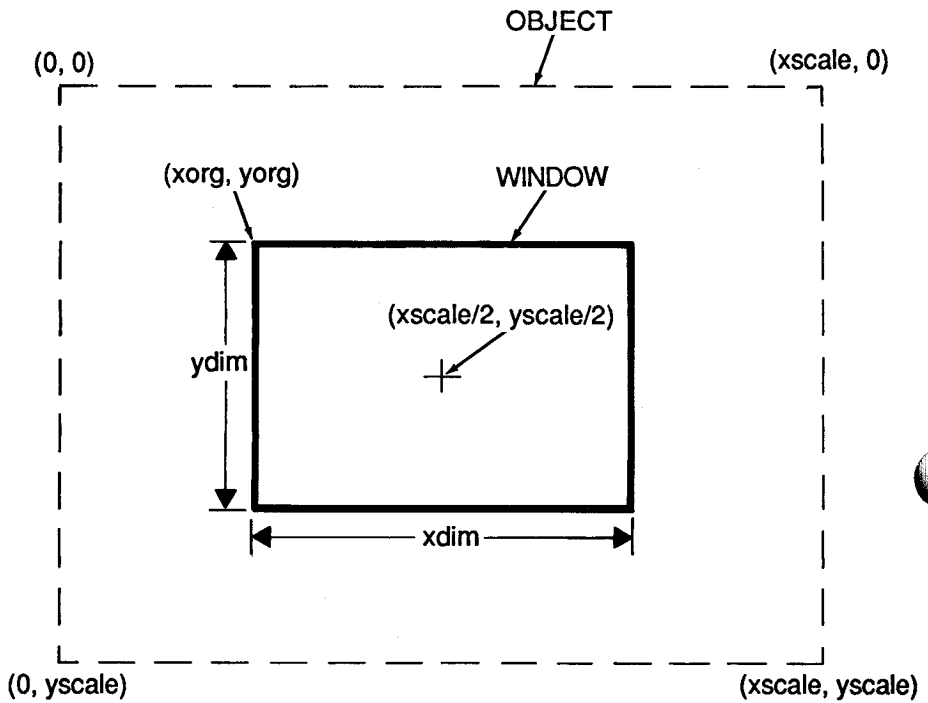
Figure 3-15: Window Coordinate Systems

All normal *print* and *input* commands can operate on a window.
Using an *input* command with a window device open in update
or read mode will take input from the programmer's keyboard
(in the case of a window on the monitor) or from the operator's
keypad (in the case of a window on the operator's display).
Each window is an ANSI terminal with all of the escape
sequences and control codes active as defined in Appendix B.

The TL/1 command below shows an example of an *open*
command used to create a window on the monitor at origin
(20,6) with a dimension of 40 horizontal characters by 12 rows.
This window is to be centered on the monitor display. In
addition, the window is to have the title RESULTS centered in
the border at the top of the window. The full-scale coordinates
of objects to be displayed in the window are to be (1000,1000).

```
channel = open device "/win", xorg 20, yorg 6,
  xdim 40, ydim 12, xscale 1000, yscale 1000,
  border "RESULTS"
```

Disk Pathnames in TL/1   3.4.6.

Pathnames are used to specify text files and I/O devices. Serial
I/O devices have only a device name, such as "/term1". But disk
devices allow directories and files to be embedded within them.
Files exist within directories, and each disk device contains at
least one directory.

A full pathname, which begins with a "/" character, is a disk
device name, followed by zero or more directory names,
followed by the file name. The different components of the
pathname are separated by "/" characters. For example:

```
/hdr/abc/test4
```

A relative pathname begins without a "/" character and consists
of zero or more directory names and ends with the file name.
The different components of the pathname are separated by "/"
characters. For example:

```
abc/test4
```

A full pathname uniquely identifies a file, but a relative pathname refers to the current directory. For most TL/1 programs started from the operator's interface, the current directory is a UUT directory. But this may not be true when a TL/1 program is run under the TL/1 debugger. Therefore, use relative pathnames in *open* and *delete* functions with care. In most cases these commands will not cause problems, but using the debugger on programs that use relative pathnames may cause inappropriate files to be placed in or removed from 9100A/9105A directories.

POD-RELATED COMMANDS 3.5.

The commands described in this section are used to communicate between a TL/1 program and a pod attached to a 9100A/9105A system. Figure 3-16 provides a summary of the available commands and a classification of their normal use. The following sections provide an overview of how these commands are used in TL/1 programs.

Category	Commands	Use
Pod Setup	setspace, getspace, sysspace	Selects UUT address space.
	podsetup	Select pod error reporting.
	sync	Select pod sync generation.
	readspecial, writespecial	Interface to special pod operations.
Read or Write Memory or I/O	read	Read from current UUT address space.
	write, writefill	Write to current UUT address space.
	readblock, writeblock	Copy data between UUT memory and disk file.
Read or Write Microprocessor Interface Signals	readstatus	Read microprocessor status inputs.
	writecontrol	Write microprocessor control outputs.
Stimulus Functions for Signature Analysis	rotate, rampdata, toggledata	Wiggle data signals.
	rampaddr, toggleaddr	Wiggle address signals.
	togglecontrol	Wiggle microprocessor control signals.
Built-in Functional Test Commands	testbus	Test address, control, and data buses for drivability and tied lines.
	pretestram	Very fast pretest of RAM.
	testramfast, testramfull	RAM memory tests.
	diagnoseram	Post-process fault analysis for custom RAM tests.
	getromsig, testromfull	ROM memory tests.
RUN UUT Mode	haltuut, runuut, waituut	Control RUN UUT mode.
	polluut	Determine if pod is in RUN UUT mode.

Figure 3-16: Pod-Related Commands

UUT Address Space Selection

Complex microprocessors provide several memory access
methods; these methods may include different data widths (8,
16, and 32 bits), different program privilege levels (user,
executive, supervisor, and kernel), and different memory
segments (code, data, stack, and so on). Some microprocessors
also support parallel I/O and memory address spaces, both
addressed by the microprocessor's address signals. The
microprocessor's memory-interface signals are set to values
which depend upon the access method desired for each cycle.
Each possible setting of these control signals is called an address
space. The *getspace, setspace,* and *sysspace* commands tell the
9100A/9105A and the pod which address space should be used
during subsequent UUT read and write cycles.

Each microprocessor manufacturer uses different terminology
for the different address spaces supported by its
microprocessors, and microprocessors differ in the number and
kinds of address spaces that each provides. The documentation
for each pod and Appendix I of the *TL/1 Reference Manual* list
the address spaces supported for each microprocessor. The
getspace command converts a readable description of the address
space you want to use into a number used internally by
9100A/9105A software to describe the address space. Using the
80186 as an example, we can create two address space
descriptors as follows:

```
normalmemword = getspace mode "normal", space
 "memory", size "word"

dmaiobyte = getspace mode "dma", space "i/o",
 size "byte"
```

The variable, *normalmemword,* will be assigned a number corresponding to the 9100A/9105A internal encoding for a normal (non-DMA), memory word access, while *dmaiobyte* will be assigned a number corresponding to a DMA-mode, I/O byte access.

The numbers returned by *getspace* are used as arguments to the *setspace* command, which actually tells the pod which address space to use. For example, to read a word from memory and then a byte from the DMA I/O space of an 80186:

```
setspace space normalmemword
firstmemword = read addr $1F0A
secondmemword = read addr $1F0C
setspace space dmaiobyte
iobyte = read addr $F
```

Notice that you only need to call *setspace* when you want to change address spaces; once set, an address space selection remains in force until another *setspace* command is used.

The number of the microprocessor address space currently in use can be determined by using the *sysspace* command. This command is normally used in functions, handlers, or exercisers which need to temporarily change address spaces. For example, a handler which needs to read a word vector at location 0 in 68000 code space, without disturbing the current address space selection, could use the following steps:

```
oldspace = sysspace ()
newspace = getspace space "supporg",
 size "word"
setspace space newspace
```

Once the operations requiring *newspace* have been completed, the old address space can be restored by executing:

```
setspace space oldspace
```

Setting Pod Error Reporting and Sync Mode

The pod can detect and report a number of exception and error conditions. The *podsetup* command allows you to selectively enable and disable different classes of error reporting. For example, pods will normally report an error if a forcing line (such as the RESET input of a microprocessor) is asserted during a UUT access cycle. But this error can be masked by using *podsetup* as follows:

```
podsetup 'report forcing' "off"
```

Notice that the *report forcing* argument contains a space character, therefore it must be surrounded by single quote characters.

The pod sync mode is selected using the *sync* command. The available sync modes are documented in the relevant pod manual; all pods support at least ADDR and DATA sync modes:

```
sync device "/pod", mode "addr"

sync device "/pod", mode "data"
```

Interface to Special Pod Operations

Certain pods have functions which cannot be accessed using the normal functions of the 9100A/9105A. These special functions are accessed by reading and writing to locations, called special addresses, outside the normal UUT microprocessor address range. The *readspecial* and *writespecial* commands access these special addresses so that the full functionality of each pod is available.

Each pod manual fully documents any special addresses, and the side effects of reading from and writing to them. Use *readspecial* and *writespecial* only when you know that the normal 9100A/9105A commands cannot perform the special operation required. Incorrect use of *readspecial* and *writespecial* can get the pod and the 9100A/9105A into inconsistent states, requiring that the pod be reset to recover.

You need to select an address space with the *getspace* and *setspace* commands before accessing the UUT memory and I/O spaces. All UUT memory accesses take place in the context of some address space, which must be selected before the UUT memory access is attempted.

Reading and Writing a Single Location

The *read* and *write* commands provide the basic interface to the UUT's memory and I/O. The *read* command simply tells the pod to read the data at the location you specify and to return the result. The *write* command tells the pod to write the data you want at the address you specify. As noted in Section 3.5.1, the *setspace* command selects the address space, after which *read* and *write* actually perform the UUT accesses. To increment a byte at location 100 (hex) in UUT memory using the Z80 pod:

```
setspace space (getspace space "memory")
val = read addr $100
write addr $100, data val + 1
```

Filling a Block of Memory

The *writefill* command can quickly fill a block of memory with a single value. The interpretation of the width of the data value (for example, byte or word) is taken from the address space currently in use. The following statements will set each 16-bit word in the first 4K bytes (2K words) of a 68000-based UUT user data memory to the value 8:

```
setspace space (getspace space "usrdata", size
  "word")
writefill addr 0, upto $FFE, data 8
```

Likewise, the following statements will write a zero to each of the I/O ports of an 8085:

```
setspace space (getspace space "i/o")
writefill addr 0, upto $FF, data 0
```

Saving and Restoring UUT Memory Data

TL/1's *readblock* command reads a block of memory and creates a Motorola-format (S-record) text file containing a copy of the memory data. This file contains the memory data, the starting address of the region copied, and the region's size. The *writeblock* command reads a Motorola-format text file (as created by *readblock*) and fills the UUT memory with the data taken from the file. The starting address and size of the region filled by *writeblock* are the same as when the file was created by *readblock*.

One way to copy the first 16K bytes of an 8088-based UUT's code space to disk is:

```
setspace space (getspace space "code")
readblock file "codestuff", format "motorola"
 addr 0, upto 16383
```

This data could later be restored by executing:

```
setspace space (getspace space "code")
writeblock file "codestuff", format "motorola"
```

Notice that *writeblock* omits the starting and ending addresses since this information is recorded in the file named *codestuff*.

The *readstatus* command returns a pod-dependent number where the bit settings reflect the current status of the pod and UUT microprocessor. The status returned by *readstatus* is usually a mixture of information regarding the microprocessor's status lines and other information about the pod itself. For example, the 80286 pod returns information about the UUT power and ground integrity, the health of the 80286 substrate bias capacitor, along with the values of the eight microprocessor status input lines. The status of PEREQ, for example, is indicated in bit 5 (the 6th bit over from the right) of the number returned by *readstatus*. To check whether or not the PEREQ input is asserted, you could use the statements:

```
status = readstatus()
if (status and $20) then . . .
```

Because microprocessors differ greatly, each pod defines a different set of *readstatus* bit values.

The purpose of the *writecontrol* command is to briefly set user-writable control lines to specific logic levels, to check for drivability, and for use in troubleshooting. Control lines are defined as writable by the specific pod documentation. Again, using the 80286 as an example, the writable control lines are:

Bit	Control Line
2	PEACK-
1	HLDA
0	LOCK-

To briefly drive the LOCK line high, you could use:

```
writecontrol ctl 1
```

The exact length of time that any line is driven differs, depending upon the kind of pod in use.

The TL/1 commands described in this section are intended to be used with signature analysis. These commands produce a repeatable stimulus on a microprocessor's address, data, and control buses.

The ramp commands generate patterns which resemble binary counting (or "ramping up"). For example, the *rampaddr* command, defined as:

```
rampaddr addr a, mask m
```

performs reads beginning at address *a* and not *m*, where the bits in *a* selected by corresponding one-bits in *m* are first set to zero. Reads are then performed at successively higher addresses; the addresses are ramped up, which is simply binary counting in which only the bits in *a* selected by one-bits in *m* are changed. The result is that 2^n *reads* are performed when *n* one-bits occur in the mask *m*. The operation of *rampdata* is similar.

The toggle commands also use a *mask* argument which selects bits in the *addr* argument. For example:

```
toggleaddr addr a, mask m
```

But the toggle commands simply invert the bits in *a* selected by one-bits in *m*. The *toggleaddr* procedure is fairly simple:

```
loop for bit = (each one-bit in mask from
 least- to most-significant)
    read addr a
    read addr (a xor bit)
end loop
```

Notice that two *reads* are performed for each one-bit in the mask. The *toggledata* and *togglecontrol* commands operate in a similar fashion.

NOTE

It is not a good idea to use the built in functional test commands (testbus, testramfast, testramfull, and testromfull) to provide stimulus where signature measurements will be made. These functional test implementations may change, resulting in changes in signatures obtained with these stimuli.

Data Bus Stimulus Commands

The *rotate, rampdata,* and *toggledata* commands produce repeatable bit patterns to stimulate microprocessor data bus lines by writing data patterns to an address specified by your TL/1 program.

- *rotate* writes a data pattern to the data bus, rotates the pattern right by one bit position, and then writes it again. The last data pattern written is the original data pattern rotated left by one bit position.

- *rampdata* writes a data pattern you specify to the data bus, and then ramps up only the data bits you've specified in the *mask* argument. The number of *writes* performed is 2^n, where n is the number of bits set to one in the *mask* argument.

- *toggledata* writes a data pattern you specify but individually toggles each bit written as specified in the *mask* argument. Two *writes* are performed for each bit set to one in the *mask* argument.

Address Bus Stimulus Commands

The *rampaddr* and *toggleaddr* commands produce repeatable patterns on the microprocessor's address bus by performing a series of *reads* at different addresses as specified by your TL/1 program.

- **rampaddr** reads beginning at the address specified, and ramps up only the address bits specified in the *mask* argument. The *rampaddr* command performs 2^n *reads,* where n is the number of bits set to one in *mask.*

- **toggleaddr** reads from the address specified and from the address formed by complementing each address bit corresponding to a one-bit in the *mask* argument. The *toggleaddr* commandperforms two *reads* for each bit set to one in *mask.*

Control Line Stimulus Commands

The *togglecontrol* command performs a series of *writecontrols* (setting and resetting the microprocessor's writable control lines) in a repeatable fashion. For each bit set to one in the *mask* argument, two *writecontrol* operations will be performed, so that each *ctl*-bit value matching a one-bit in the *mask* is driven both high and low.

Built-in Functional Tests   3.5.5.

The 9100A/9105A built-in functional tests are designed to provide fast, reliable implementations of commonly required tests. These tests cover the major microprocessor buses, RAM tests, and ROM tests.

Two common features of the functional tests are:

- They return only a termination status (pass or fail) so that these tests commonly appear in TL/1 programs as:

```
if testbus addr $1000 fails then . . .
```

- They may raise a number of fault conditions. The fault conditions raised are listed in Appendix G of the *TL/1 Reference Manual.*

NOTE

These functional tests are not intended to provide stimulus for fault isolation techniques, such as GFI, which depend upon signature analysis. Test implementations may change, which would change learned UUT signatures as well. Use the commands described in Section 3.5.4 for signature analysis.

Testing the Microprocessor Buses

The *testbus* command performs a comprehensive test of the microprocessor address, data, and control buses. All three buses are tested for drivability, and the address and data buses are tested to ensure that no two address or data lines are tied together. The *addr* argument for *testbus* should be a RAM memory location that can be written to and read from without causing bus access faults.

Testing RAM Memory

There are three RAM tests available to TL/1 programs: *pretestram, testramfast* and *testramfull*. The table below and Figure 3-17 compare their features. Refer to the *pretestram, testramfast,* and *testramfull* commands in the *TL/1 Reference Manual* for more information on these tests.

Test Type	Coupling	RAM Width	Accesses per Cell
pretestram	N/A	any	Only some addresses are checked
testramfast	N/A	any	5
testramfull	disabled	any	17
	enabled	8	29
	enabled	16	33
	enabled	32	37

- *pretestram* is a very fast pretest of RAM to find any simple faults such as a totally dead memory chip, stuck address lines, or stuck data lines.

- *testramfast* is a fast RAM test that performs only five passes through memory. The *testramfast* command writes pseudo-random data to memory. Since this data is random, faults found in one invocation of *testramfast* may not be found in another invocation.

- *testramfull* is a deterministic RAM test; if the UUT operates in the same way, every invocation of *testramfull* will find the same faults.

Fault Condition	testramfast	testramfull
Stuck cells.	Always found.	Always found.
Aliased cells.	"	"
Stuck data lines.	"	"
Stuck address lines.	"	"
Shorted address lines.	"	"
Multiple selection decoder.	May be found.	Always found.
Dynamic coupling.	"	"
Shorted data lines.	May be found.	Always found for coupling enabled; may be found for coupling disabled.
Aliasing between bits in same word.	"	"
Pattern-sensitive faults.	May be found.	Not found.
Refresh problems.	Always found if delay is sufficiently long and standby reads do not mask the problem.	

Figure 3-17: Fault Detection for RAM Tests

- *testramfast* will always find stuck or aliased cells, stuck address or data lines, and shorted address lines, but *testramfast* may not find problems like multiple selection decoding, dynamic coupling, or aliasing between bits in a single memory word.

- *testramfast* is more likely to find some pattern-sensitive faults than is *testramfull: testramfast* writes pseudo-random data to memory, and all possible data patterns are equally likely.

- *testramfast* is not as likely as *testramfull* to find problems due to electrical transients related to writing all ones or all zeros to memory.

If you create your own customized RAM tests, *pretestram* can be used as a quick pretest. In addition, *diagnoseram* can be used to provide diagnostics and fault messages which are consistent with those of *testramfast* and *testramfull*.

Testing ROM Memory

The *getromsig* and *testromfull* commands are used together to perform ROM memory tests. To read 64K bytes of ROM in a 16-bit wide address space use the following command:

```
sigval = getromsig addr $FF0000, upto $FFFFFE,
   addrstep 2
```

The signature is returned and assigned to *sigval*. A *mask* argument to *getromsig* can mask any bits of the ROM which shouldn't be used.

The *testromfull* command first generates a ROM signature and then compares it with the signature from a known-good ROM (generated by *getromsig*). For example, the ROM signature generated by the *getromsig* command in the previous paragraph could then be used in the following command:

```
testromfull addr $FF0000, upto $FFFFFE,
   addrstep 2, sig sigval
```

If the calculated and expected signatures don't match, *testromfull* will report a fault message on the operator's display. As with all signature-based schemes, there is a small probability (in this case, not more than 1 in 2^{16}) that a ROM containing incorrect data will not be detected as faulty.

RUN UUT Mode   3.5.6.

The RUN UUT mode of 9100 pods allows the pod microprocessor to emulate the target microprocessor, executing programs on the UUT. This mode is useful for executing pre-written tests stored in a UUT ROM or to perform initialization of UUT peripherals prior to a test. The commands related to RUN UUT mode are:

- *runuut,* which puts the pod into RUN UUT mode.

- *haltuut,* which brings the pod out of RUN UUT mode.

- *waituut,* which suspends the TL/1 program until either a time limit expires or the pod leaves RUN UUT mode.

- *polluut,* which returns 1 if the pod is in RUN UUT mode, and 0 otherwise.

Placing a Pod in RUN UUT Mode

Before placing a pod in RUN UUT mode, make sure it is properly set up. Your TL/1 program should:

- Perform any *podsetup* operations required to initialize the pod. In particular, using overlay RAM will require special initialization.

- If the *compare* command will be used, instruct the operator to clip any I/O modules to the UUT, and then, use the *compare* command to set up any I/O modules.

The *runuut* command places a pod in RUN UUT mode. This command has the following form in which *start* is the memory address at which execution should begin and *stop* is a breakpoint address:

```
runuut addr start, break stop
```

Not all pods support breakpoints: check the pod manual for your microprocessor. For microprocessors that provide several modes of operation (for example, the 80286), check the pod manual to find out how the microprocessor is initialized by *runuut*.

Once the *runuut* command has been executed, the TL/1 program can proceed to perform other tasks, but the only pod-related commands which may be executed are *waituut, haltuut,* and *polluut*.

Removing a Pod from RUN UUT Mode

The pod itself will remain in RUN UUT mode until one of the following occurs:

- The pod encounters a breakpoint.
- An I/O modulereports a data-compare-equal (DCE) condition.
- A *haltuut* command is performed.
- A *waituut* command times out.
- The operator enters RESET or RUN UUT HALT from the operator's keypad.

If the pod is brought out of RUN UUT mode by a DCE condition, the pod microprocessor will probably have executed a number of instructions since the DCE condition was actually detected.

You can expect that fault conditions may be generated by *haltuut* or *waituut*. Any fault conditions encountered by the pod after the *runuut* command is completed will be retained by the pod and returned on the next command to the pod.

I/O MODULE AND PROBE COMMANDS 3.6.

This section describes the TL/1 commands which control the use of the I/O modules and the probe. An I/O module is normally used with a clip module that fits over an integrated circuit; this permits measurement and stimulus access to all pins of the IC component at once. The probe is a single-point device, manipulated either by a machine (autoprober) or an operator, to measure or stimulate any single node on a UUT. Figures 3-18 and 3-19 summarize the commands used to control the I/O modules and probe.

Naming UUT Components and Pins 3.6.1.

A reference designator is a UUT component name. A reference designator begins with a letter (A to Z) or digit (0 to 9), and consists of from one to six letters, digits, underscores (_), or periods (.). Some valid reference designators are:

Reference Designator	Part
U21	IC 21
R1	Resistor 1
TP5	Test point 5

Reference designators are case-insensitive; "u43" and "U43" refer to the same component.

Category	Command	Use
Configure I/O module or probe for measurement	counter	Set counter mode.
	edge	Set active edges for external sync.
	enable	Set enable mode for external sync.
	reset	Reset to default mode.
	stopcount	Set number of enabled clock pulses for measurement.
	sync	Set synchronization mode.
	threshold	Set input threshold levels.
Attach probe or I/O module to UUT	assign	Resets connection data.
	assoc	Associates a UUT part with an I/O module.
	clip	Prompt operator to clip I/O module.
	connect	Prompt operator to connect external sync lines.
	probe	Prompt operator to place probe.
Perform measurement with I/O module or probe	arm	Arm measurement hardware.
	checkstatus	Check if measurement complete.
	readout	Get data from measurement.
	strobeclock	Strobe internal clock for probe or I/O module.
Read measurement taken for one UUT component pin	count	Read count or frequency.
	level	Read level history.
	sig	Read signature.
Probe stimulus	pulser	Set probe pulser mode.
I/O module stimulus	clearoutputs	Turn off output drivers.
	clearpatt	Discard output patterns.
	storepatt	Set output patterns to be written.
	writepatt	Write output patterns to UUT.
	writepin	Latch or pulse level on single pin.
I/O module word recognition	compare	Set bit pattern to be compared with I/O module input.
Get or set delay for I/O module or probe	getoffset	Return current delay offset.
	setoffset	Set new delay offset.

Figure 3-18: I/O Module and Probe Commands, by Category

Command	I/O Module	Probe	Sync	Measurement	Stimulus
arm	•	•	any	•	
assign	•		any	•	•
assoc	•		any	•	•
checkstatus	•	•	"ext"	•	
clearoutputs	•				•
clearpatt	•				•
clip	•		any	•	•
compare	•		any	•	
connect		•	"ext"	•	•
		•	"ext"	•	
count	•	•		•	
counter	•	•	any	•	
edge		•	"ext"	•	•
		•	"ext"	•	
enable		•	"ext"	•	•
	•		"ext"	•	
getoffset	•	•	any		
level	•	•	any	•	
probe		•	any	•	•
pulser		•	any		•
readout	•	•	any	•	
reset	•	•	any	•	•
setoffset	•	•	any		
sig	•	•	any	•	

Figure 3-19: I/O Module and Probe Commands, Alphabetized

Command	I/O Module	Probe	Sync	Measurement	Stimulus
stopcount		•	"ext"	•	•
	•		"ext"	•	
storepatt	•				•
strobeclock		•	"int"	•	•
	•		"int"	•	
sync	•	•	any	•	•
threshold	•	•	any	•	
writepatt	•		"int"		•
writepin	•				•

Figure 3-19: I/O Module and Probe Commands, Alphabetized (cont)

A reference pin argument tells which pin on a UUT component should be used by a command. Legal reference pin values are between 1 and 255. Different commands require that reference pins be specified in one of two ways:

- Commands such as *probe* and *connect,* which tell the operator to connect a 9100A/9105A device to a UUT component, require that reference pins be specified as xxxxxx-nnn; that is, as a reference designator followed by a hyphen and one to three characters.
- Commands such as *count, level,* or *sig* may refer either to UUT component pins or 9100A/9105A device pins:

Example 1:

```
! uses a 9100A/9105A device pin
x = count device "/mod1B", pin 22
```

Example 2:

```
! Uses a UUT component pin.
! The pin number is specified as a separate
! numeric argument i.e., u17-2 is not correct
! syntax.
x = count device "u17", pin 2
```

Naming 9100A/9105A Devices   3.6.2.

Each 9100A/9105A system may have one pod, one probe, and up to four I/O modules attached.

The name of the pod is the string:

"/pod"

The name of the probe is the string:

"/probe"

The I/O modules are named by the strings:

"/mod1"
"/mod2"
"/mod3"
"/mod4"

Each I/O module can have either one or two clip modules installed. The clip modules are named by the strings:

"/mod1A"
"/mod1B"
"/mod2A"
"/mod2B"
"/mod3A"
"/mod3B"
"/mod4A"
"/mod4B"

The name of each clip module names the I/O module (1-4), and the side (A or B) to which the clip is attached. Appendix E of the *TL/1 Reference Manual* contains tables which describe how I/O module pin positions and clip pin positions are related.

Kinds of Measurements that Can Be Made   3.6.3.

The measurements gathered by the probe and the I/O module include both synchronous and asynchronous data. Synchronous data is sampled at some fixed offset from a clock signal edge; clock signals are specified using the *sync* command. Asynchronous data is gathered continuously, without respect to any clock edge. The asynchronous measurements made by the 9100A/9105A system are asynchronous level histories, transition counts, and frequencies. All other measurements require a clock.

Signatures

A signature is a number which represents (or summarizes) the sequence of data values seen at some circuit node (or pin) in a UUT. The *sync* command tells which clock to use in order to get valid data at the node. The *arm* and *readout* commands are used to begin and end the signature measurement. Finally, the *sig* command returns a number which represents the (16-bit) signature taken at a single UUT pin. Both the probe and I/O modules are capable of taking signature measurements.

Level Histories—Synchronous and Asynchronous

A level history is a record of whether or not a signal has taken on one of the values low, high, or invalid during the execution of an *arm . . . readout* block. The exact input voltages that are considered low, high, or invalid are set by the *threshold* command.

A synchronous level history simply examines the level at a pin whenever a clock edge occurs, and the fact that the value was low, high, or invalid is recorded. An asynchronous level history examines the level continuously; as such, the asynchronous history is useful as a "glitch catcher."

A level history measurement is in no way equivalent to the kind of information provided by a logic analyzer. The level history simply tells whether or not a particular level was ever seen at a UUT node. The actual level measurement is made within an *arm . . . readout* block. The *level* command returns a number that tells which levels were seen at that pin:

Value	Levels Recorded
0	none
1	low only
2	invalid
3	low and invalid
4	high only
5	high and low
6	high and invalid
7	high, low, and invalid

Transition Counting and Frequency Measurement

Transition counting and frequency measurements do not depend on the synchronization method used. The transition count is simply the number of active rising edges (transitions from the invalid to high state) measured at a pin between the *arm* and *readout* commands. The frequency measurement is the frequency measured at a pin during the *arm . . . readout* block.

Each device (I/O module or probe) can perform either a transition count or a frequency measurement during a single *arm . . . readout* block; the *counter* command selects which of these measurements will be made. The *count* command is used following *readout* to return either the transition count or frequency measured at a UUT pin. If a counter overflows, the result returned has bit 31 (the high-order bit) set. Frequency measurements are returned in Hz.

Synchronization Modes   3.6.4.

The *sync* command sets the synchronization mode (or clock source) for the probe and I/O modules. For the I/O modules, the sync mode affects only measurements: any stimulus generated by an I/O module uses timing generated internally by the 9100A/9105A (for the *writepatt* command) or by a TL/1 program (for the *writepin* command). The probe's output, however, can be synchronized to any of the available clock sources.

Each of the I/O modules and the probe can use a different synchronization, defined by the *sync* command. The available synchronization types are described in the following sections.

Pod Synchronization

The 9100A/9105A pods are designed to provide timing signals, which indicate the beginning and end of UUT access cycles. The falling edge of the ~PodSync signal indicates the beginning of a UUT access. All pods support at least ADDR and DATA sync modes; see the *sync* command description and the Fluke pod manual for the microprocessor you are using.

Internal Synchronization

Internal sync is used in conjunction with the *strobeclock* and *writepatt* commands. The *strobeclock* command is used when measurements (using the probe or an I/O module) or stimulus (using the probe) are performed under the control of a TL/1 program. Internal sync is also used when gathering signatures with *writepatt*. Internal sync can also be used whenever no sync signal is desired, as might be true with an asynchronous level history, transition count, or frequency measurement.

External Synchronization

External sync uses the external control leads on the clock module (for the probe) or on an I/O module. These leads provide edge-triggered Start, Stop, and Clock signals, and a level-sensitive Enable signal. Refer to Figure 3-20 for a diagram of how to set up external sync in a TL/1 program.

External sync first requires a Start signal edge. Once this has been received, each active external clock edge during which the Enable input is asserted true will trigger a measurement until either a Stop signal edge occurs or the number of enabled clock edges specified by the stopcount command has occurred.

For the probe, external sync can also be used in conjunction with the *pulser* command. In this case, a pulser output transition occurs for each enabled clock edge.
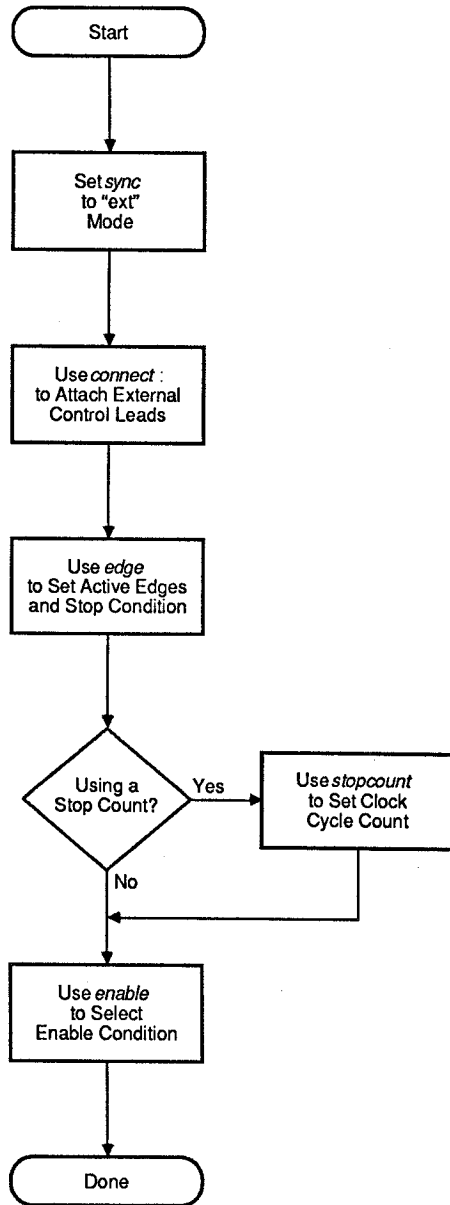
Figure 3-20: Setup for External Synchronization

Freerun Synchronization

Freerun sync is used with the probe when the probe is used to provide a low-frequency stimulus for troubleshooting. It is driven by an internal 1K-Hz oscillator. Signature and other measurements should not be made using freerun sync—any such measurements will be meaningless.

Making Measurements with the Probe and I/O Module   3.6.5.

Every time a measurement is made with the probe or I/O module, the same four steps must be performed:

1. Place probe or I/O module - Prompt the operator to connect the probe or I/O module to the UUT, or direct an autoprober.

2. Configure hardware - Set the counter mode, sync mode, and input thresholds. If using external synchronization, set the active sync signal edges and clock enable level.

3. Perform measurement - Apply stimulus, using the pod, probe, or I/O module, to the UUT within an *arm . . . readout* block. If using internal sync, use *strobeclock* to trigger measurements. If using external sync, use *checkstatus* to make sure the measurement is complete.

4. Read data for each component pin - Use the *sig, count,* and/or *level* commands to read the measurement data collected for either the single pin measured by the probe, or by all input pins clipped by an I/O module.

A stimulus program that will be invoked by GFI should not perform step 1 and step 4 listed above. GFI will already have chosen and placed a measurement device (see the *gfi device* command), and GFI will use the *sig, count,* or *level* commands itself, as required. To design a TL/1 stimulus program that may be run alone or under GFI control, see the *gfi control* command, and "GFI Commands" in Section 3 of this manual.

Selecting and Placing an I/O Module

Normally the 9100A/9105A software itself is used to prompt an operator to select an I/O module and to clip the leads of an I/O module adapter to the UUT. The *clip* command will ask the operator to select a clip module and to attach it to the UUT component specified. For example, the following commands would prompt the operator to clip onto U22 (a component with 24 pins):

```
module = clip ref "U22", pins 24
```

The value returned by *clip* is the name of the one or more clip modules selected by the operator to clip to the UUT component.

When using fixturing, the placement of the I/O module clips is preset so it is unnecessary and undesirable to press the ready button on each of the clips (as required by the *clip* command). In this case, the *assoc* command should be used. For example, the following command would associate the "B" side of I/O module 1 with the reference designator U22 (a component with 24 pins):

```
assoc ref "U22", pins 24, device "/mod1B"
```

The *assoc* command is functionally equivalent to the *clip* command except that the device list is set in the TL/1 program by the programmer rather than being determined by the I/O module button that is pressed.

The *assign* command resets the connection data for a specified I/O module so that it no longer associates that module with a particular reference designator. This command is not required in most programs.

Placing the Probe

The *probe* command generates a message to the operator to probe a particular pin on the UUT before a measurement is taken.

Connecting External Sync Leads

If you are using the *"ext"* mode in the *sync* command, the START, STOP, CLOCK, and ENABLE leads must be connected. The *connect* command prompts the operator to hook up the leads from the clock module (for the probe) or from the I/O module used:

```
connect device "/probe", start "u1-4", stop
"u4-12", clock "u4-3", common "tp4"
```

Any leads left unspecified in the *connect* command are considered to be not used, but the COMMON lead should always be connected.

Configuring Measurement Hardware

Before performing any measurements, the response-gathering hardware must be configured to conform to the test requirements.

- **Logic threshold levels-** The *threshold* command selects one of four possible logic threshold values for the probe: TTL, CMOS, RS232, or ECL; and one of two possible thresholds for the I/O module: TTL or CMOS. ECL is only valid if the ECL capability is installed.

- **Synchronization mode** - The *sync* command selects one of the four available sync modes: external ("ext"), internal ("int"), "pod", or "freerun". Each device (probe and I/O module) can use a different sync source. As noted before, freerun sync should be used only with the probe pulser.

- **Counter mode** - If measuring transitions or frequencies, the *counter* command must be used.

```
counter device "/probe", mode "freq"

mod = clip ref "u5", pins 16
counter device mod, mode "transition"
```

Using Pod Synchronization

When using *"pod"* mode for the *sync* command, you must tell the pod what kind of pod sync signal should be generated. This is done with an additional *sync* command:

```
sync device "/probe", mode "pod"
sync device "/pod", mode "data"
```

All Fluke pods support at least "addr" and "data" sync; other modes are supported for pods that require these additional modes.

Remember that a pod can only generate one form of sync signal at a time: it is not possible to use "addr" sync for one I/O module and "data" sync for another while performing a single measurement.

Using External Synchronization

Because of its flexibility, external sync requires more information than other sync modes. This additional information is explained in the following paragraphs.

1. The active edges of the edge-sensitive Start, Stop, and Clock signals must be selected. This is done with the *edge* command:

```
edge device "/probe", start "+", stop "-",
 clock "-"
```

2. The clock-enable condition must be chosen. This condition can be either "always", "high", or "low", or it can be some combination of the PodSync line and the external enable line as detailed in the explanation of the *sync* command in the *TL/1 Reference Manual*.

3. The stop condition must be specified. It may either be a transition of the Stop signal, or a predetermined number of enabled clock pulses. The *edge* and *stopcount* commands are used to select these options. Using a Stop edge requires that the *stop* argument for *edge* be one of the strings *"+"* or *"-"* as shown above. But stopping after a certain number of enabled Clock signal transitions requires using both *edge* and *stopcount:*

```
edge device "/probe", stop "count"
stopcount device "/probe", count 100

mod = clip ref "u43", pins 22
stopcount device mod, count a * 4
edge device mod, start "-", clock "+",
 stop "count"
```

Performing a Measurement

Making measurements requires the *arm* and *readout* commands to enclose a group of TL/1 statements that provide stimulus to the UUT. When no stimulus is required (for example, when measuring the frequency of an oscillator), the *arm . . . readout* block will contain no statements. But some stimulus from the pod, the probe, or an I/O module is usually required.

arm and *readout*

Once the measurement hardware has been configured, it is possible to make a measurement. All measurements are made within an *arm . . . readout* block—the *arm* command signals the hardware to begin taking a new measurement, and the *readout* commandterminates the measurement. For example, to measure the frequency output of a clock generator using the probe, you might use:

```
probe ref "u31-5"
counter device "/probe", mode "freq"
arm device "/probe"
readout device "/probe"
    ! No stimulus is required in
    ! the arm . . . readout block
clock_freq = count device "/probe"
```

Since the clock generator is a free-running component (it doesn't need a stimulus), simply probing its output pin while power is applied to the UUT will give a valid frequency measurement.

When stimulus is required (which is normal when taking signatures), either the pod, the probe, or an I/O module can be used to apply the stimulus. The example on the following page shows how signatures could be gathered from an 8-bit bus data buffer using the pod.

```
iomod = clip ref "u23", pins 20
sync device iomod, mode "pod"
sync device "/pod", mode "data"
arm device iomod
    rampdata addr 0, data 0, mask $F
    rampdata addr 0, data 0, mask $F0
readout device iomod
sigbit1 = sig device "u23", pin 18
sigbit2 = sig device "u23", pin 17
            .
            .
            .
sigbit8 = sig device "u23", pin 11
```

The *rampdata* commands inside the *arm . . . readout* block are used to generate repeatable data patterns for signature analysis. Since the the data buffers are driven from the microprocessor, the pod is used to provide the stimulus. The pod is also the source of timing information: the I/O module is synchronized to the pod data timing since the bus data buffers are being tested.

The *checkstatus* Command

When using external sync, the *checkstatus* command is used to determine whether or not the measurement is complete. A *checkstatus* command has the form:

```
status = checkstatus device "/probe"
```

It returns a 4-bit numeric result, which is interpreted as:

Bit	Signal	Value
4-31	none	always 0
3	Stop received	1=yes, 0=no
2	Start received	1=yes, 0=no
1	Enable received	1=yes, 0=no
0	Data clocked	1=yes, 0=no

When using external sync, the *checkstatus* command may be used in a *loop while* block to check for completion of a measurement prior to executing a *readout* command. As illustrated in the example below, the loop checks for a complete measurement every 50 milliseconds (approximate) until it has checked eight times. If the measurement isn't complete by that time, an error is reported: either the clock module lines were not connected properly by the operator, or the stimulus circuit isn't working.

```
probe ref "u43-16"
connect device "/probe", start "TP1", stop
 "TP2", clock "u3-25"
sync device "/probe", mode "ext"
enable device "/probe", mode "always"
```

```
edge device "/probe", clock "-"
arm device "/probe"
    loops = 0
    loop while ((checkstatus device "/probe")
    <> $F) and (loops < 8)
        wait time 50
        loops = loops + 1
    end loop
readout device "/probe"
if loops = 8 then
    !
    ! take action for incomplete measurement
    !
end if
```

When the TL/1 program itself is providing the stimulus, *checkstatus* is usually invoked following the *arm . . . readout* block to ensure that the external sync leads were connected properly and that the circuitry generating the sync signals was operating correctly. The example below illustrates using *checkstatus* when stimulus is provided by a TL/1 program.

```
iomod = clip ref "u40", pins 28
connect device iomod, start "u23-4",
 stop "u2-15", clock "u15-5", common "u23-7"
sync device iomod, mode "ext"
sync device "/pod", mode "data"
edge device iomod, start "-", stop "-",
 clock "+"
enable device iomod, mode "pod"
arm device iomod
    rotate addr $1000, data $9669
    loops = 0
    loop while ((checkstatus device iomod)
    <> $F) and (loops < 8)
        wait time 10
        loops = loops +1
    end loop
readout device iomod
if loops = 8 then
    !
    ! take corrective action
    !
end if
```

Reading Data for Each Component Pin

Once a measurement has been made, the data gathered by the measurement device is returned to the TL/1 program by using the *count, level,* and *sig* commands. These commands return the data associated with a single pin on a UUT component. Although an I/O module measures counts, level histories, and signatures for all pins on a UUT component at once, the *count, level,* and *sig* commands return the measurements for single pins.

Appendix E of the *TL/1 Reference Manual* shows how clip pins and I/O module pins are related.

When using the probe, the device name "/probe" is used:

```
probe ref "u3-13"
arm device "/probe"
          ! perform stimulus
readout device "/probe"
sig2 = sig device "/probe"
```

The data collected by an *arm . . . readout* block remains valid until the probe or I/O module is probed or clipped again, and a new measurement is made. Be sure to use the *count, level,* or *sig* command before performing new measurements, so that it is clear that the data being read is from the measurement just made and that later measurements don't overwrite data that should have been saved.

Data Comparison with the I/O Module   3.6.6.

The *compare* command causes an I/O module to continuously (asynchronously) compare its inputs with a specified data word; whenever a match occurs, a *iomod_dce* condition is signalled. Up to 40 bits of comparison information may be specified, consisting of 0's, 1's, and X's (don't care values). Any invalid levels measured are considered to be low when making comparisons. The example below shows how a data buffer could be used to generate a DCE (data-compare-equal) condition whenever the pattern 11XXXX00 (for inputs A_1 to A_8) occurs on a 74245 octal bus transceiver:

```
handle iomod_dce
    !
    !   code to handle iomod_dce condition
    !
end iomod_dce
module = clip ref "u2", pins 20
compare device module, patt
  "111XXXX00XXXXXXXXXX0X", state "enable"
```

The *patt* argument indicates that the direction control, pin 1, should be "A to B" and that the active-low G control line, pin 19, should be false, in addition to the data pattern to be matched.

Once a DCE condition has been raised, the *compare* command must be used again before another comparison will be performed.

Pattern Driving with the I/O Module   3.6.7.

The I/O modules can overdrive signals to a clipped component in order to write patterns to a UUT component. The patterns are written by the 9100A/9105A without regard for the synchronization mode programmed for the I/O module. To prevent damage to UUT circuitry, patterns are overdriven for a maximum of 10 milliseconds.

The *clearoutputs* command places all I/O module outputs in the high-impedance state, *clearpatt* removes any previously programmed patterns, and *storepatt* stores new patterns to be written to the UUT with the *writepatt* command. Figure 3-21 illustrates how signature analysis of both gates in a 7420 dual 4-input NAND gate would be performed in parallel using an I/O module clip. Each NAND gate is driven with the patterns *0000, 0001, . . . , 1111*. Generating a signature when using *writepatt* requires using internal sync. The example gathers a signature so that the output of the component under test can be compared with the response of a known-good 7420 device.

```
iomod = clip ref "u12", pins 14
reset device iomod
sync device iomod, mode "int"
clearpatt device "u12"                           ! gate 1 inputs
storepatt device "u12", pin  1 , patt "0000000011111111"
storepatt device "u12", pin  2 , patt "0000111100001111"
storepatt device "u12", pin  4 , patt "0011001100110011"
storepatt device "u12", pin  5 , patt "0101010101010101"
                                                 ! gate 2 inputs
storepatt device "u12", pin 13 , patt "0000000011111111"
storepatt device "u12", pin 12 , patt "0000111100001111"
storepatt device "u12", pin 10 , patt "0011001100110011"
storepatt device "u12", pin  9 , patt "0101010101010101"
arm device iomod
    writepatt device "u12", mode "pulse"
readout device iomod
gate_1_sig = sig device "u12", pin 6     !  gate 1 output
gate_2_sig = sig device "u12", pin 8     !  gate 2 output
```

Figure 3-21: Pattern Driving Example

The maximum pattern depth for *writepatt* depends on the number of I/O modules used:

Number of I/O modules	Maximum Pattern Depth
1	255
2	128
3	85
4	64

This restriction assures that *writepatt* will drive pins for no longer than 10 milliseconds.

Probe Stimulus   3.6.8.

The *sync* and *pulser* commands are used together to generate stimulus using the probe. The probe pulser will generate a string of high, low, or alternating pulses synchronized to any available timing source. For example, the following statements will produce a low pulse at the trailing (rising) edge of the ~PodSync signal:

```
probe ref "u14-5"
sync device "/probe", mode "pod"
sync mode "addr"
pulser mode "low"
```

The probe pulser can also be synchronized to the 1 kHz freerun clock, to internal sync that generates a pulse whenever the *strobeclock* command is invoked, or to an external sync source. When external sync is used, the Start, Stop, and Enable signals control when the pulser operates, as does any stop count in effect. Signature gathering with the probe is possible while the pulser is used.

Changing the Calibration Delay Offset for the I/O Module or Probe

Both the probe and I/O modules have hardware delay lines that can adjust the relative timing between clock and data signals. These delay lines are calibrated by using the MAIN MENU key and then the CAL softkey on the operator's keypad. There is a different offset value stored for each sync mode. See the discussion in Appendix I of the *TL/1 Reference Manual* for more information.

When the calibrated offset delay for an I/O module or the probe is not appropriate for a measurement, the *setoffset* command may be used to change the delay. For example, even though a pod manual may indicate that data should be sampled at 30 nanoseconds after the rising edge of a signal, *setoffset* could be used to sample at other times to check for marginal UUT performance.

The *setoffset* command takes an argument for the desired offset value. This offset has a bias of 1000000. So if you want to program an offset for the probe of -30 nanoseconds in external sync mode, do the following:

```
sync device "/probe", mode "ext"
offset_in_range = setoffset device "/probe",
 offset 1000000 - 30
```

The *setoffset* command returns either *0* or *1*. A *1* is returned if the delay could be programmed successfully. A *0* is returned if the delay requested is outside the range of the hardware, in which case, the delay lines will be set as close as possible to the requested delay (that is, to the maximum or minimum delay setting).

Delays can be varied in approximately 4-nanosecond steps for the probe and 15-nanosecond steps for the I/O modules. After the *setoffset* example (above), the current offset value might equal 999972 (100000 - 28, or -28 nanoseconds). This would indicate that -28 was the closest possible setting to the desired -30.

The *getoffset* command is valuable for accessing the current offset value for a sync mode in order to view or to save it:

```
current_offset = getoffset device "/probe"
```

NOTE

Both the setoffset and getoffset commands reflect the offset for the current sync mode only.

FAULT CONDITIONS AND FAULT HANDLING   3.7.

When a fault is detected in a UUT, the normal action of 9100A/9105A software is to "raise a fault condition." The program raising the fault condition is suspended until some corrective action is taken. The corrective action may be performed automatically either by a fault condition handler or manually by the operator using the operator's keypad.

TL/1 is designed to permit you to fully utilize the 9100A/9105A fault condition handling and reporting mechanisms in your own programs. A test may:

- Raise fault conditions in a TL/1 program, whenever the TL/1 program detects a fault in a UUT.
- Handle any fault condition, whether it is reported by a built-in test, or a test written in TL/1.
- Provide an exerciser for any fault condition, in response to an operator pressing the LOOP key to cause a repeated generation of the fault condition. This is also called exercising a fault.

A test (whether built-in or written in TL/1) may either pass or fail; this status may be tested by using TL/1 conditional statements.

You may write TL/1 procedures to override or supplement the 9100A's normal fault-handling behavior whenever necessary.

The following paragraphs will give you the information needed to write such procedures.

Raising a Fault Condition 3.7.1.

A fault condition is simply a notification by a program or function (whether built-in or written in TL/1) that a fault has been detected in a UUT. In TL/1, a fault condition is raised by using the *fault* command, for example:

```
fault bus_data_tied addr a, data d, mask m
```

This *fault* command would raise a *bus_data_tied* fault condition for the bits given in the 64-bit mask *m* detected at address *a* with data *d*.

A fault condition consists of:

- **Fault condition name:** In the example, the name of the fault condition is *bus_data_tied*. A number of fault condition names are predefined by the 9100A/9105A software (see Appendix G of the *TL/1 Reference Manual*). The fault condition name describes the kind of fault detected.

- **Fault condition arguments:** In the example, the address and data which made the fault appear are reported along with the fault name. The fault condition arguments are just like the arguments for any other TL/1 program or function.

Raising a fault condition is like calling a TL/1 function. The *fault* command starts a search for a fault condition handler designed to handle that particular fault condition. If your program doesn't supply a handler for the fault condition, the 9100A/9105A will print a message describing the fault condition on the operator's display and then wait for the operator to choose a course of action.

Figure 3-22 shows how the 9100A/9105A acts when a program raises a fault condition. The following sections will fill in the details of this process.

Built-in Test

TL/1 *fault* Command

Raises a
Fault
Condition

Handler
Available? — No → Display Fault Message on
Operator's Display

Yes

Execute
Handler

Resume Test
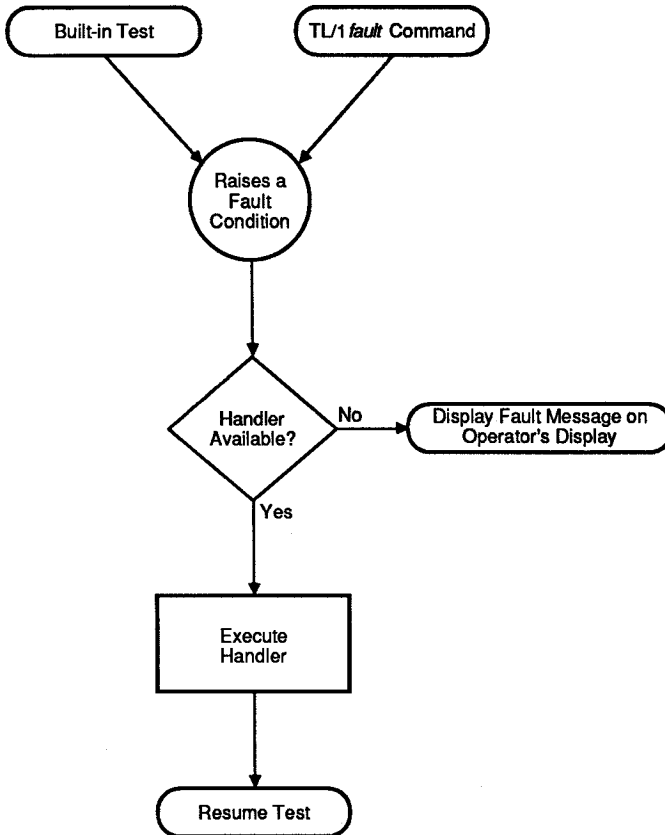
Figure 3-22: Raising and Handling a Fault Condition

Fault Condition Names   3.7.2.

Fault condition names take the form of any legal TL/1 name. A number of fault condition names are predefined by the 9100A/9105A software; these fault condition names and their arguments are listed in Appendix G of the *TL/1 Reference Manual.* When one of these fault conditions is reported on the operator's display, the 9100A/9105A uses special message formats appropriate to the fault condition. These messages are listed in Appendix H of the *TL/1 Reference Manual* and Appendix F of the *Technical User's Manual.*

You are free to choose any fault condition name you like for faults unique to your UUT and testing requirements. But giving non-standard meanings to the standard 9100A/9105A fault condition names is probably not a good idea.

Creating a Fault Condition Handler   3.7.3.

Any TL/1 program or function can contain definitions for one or more fault condition handlers. The name of the handler is the name of the fault condition it is meant to handle: this name can be one of the 9100A/9105A built-in fault condition names, or the name of a fault condition you have created for your own use. Figure 3-23 shows a program that includes handlers for the *pod_addr_tied* and *pod_forcing_active* built-in fault conditions (in the program *uut_test*), and a handler for the *pod_addr_tied* built-in fault condition (in the function *mem_test*).

A diagram of how the 9100A/9105A invokes *uut_test*, which in turn invokes *mem_test,* which includes a *read* command is illustrated in Figure 3-24.

A fault condition handler is a block of code, like a program or function, possessing an optional argument list and local variables. When the program *uut_test* is invoked, its fault condition handlers for *pod_addr_tied* and *bus_data_tied* will become active. Likewise, when *mem_test* is called by *uut_test,* its own fault condition handler for *pod_addr_tied* will become active as well.

```
program uut_test

function mem_test (start, last)
    declare numeric start = 0
    declare numeric last
    handle pod_addr_tied (addr, access_attempted, mask)
            .
            .
            .
    end pod_addr_tied
        .
        .
        .
    first = read addr start     ! Here is a read command
        .
        .
        .
end mem_test

handle pod_addr_tied (addr, access_attempted, mask)
        .
        .
        .
end pod_addr_tied

handle pod_forcing_active (addr, ctl, mask)
        .
        .
        .
end pod_forcing_active

if mem_test start 0, last $FFFF fails then     ! First command after
                                               ! definition blocks
        .
        .
        .
end if
        .
        .
        .
end uut_test
```

Figure 3-23: Example of a Program with Handlers

A fault condition handler defined within some program or
function, *p,* is active from the time *p* is called until *p* returns to
its caller.

How a Fault Condition Handler Is Chosen 3.7.4.

When a built-in test or TL/1 program raises a fault condition, the
9100A/9105A software searches for an active handler to deal
with the fault condition. This search begins in the program or
function that raised the fault condition, and then, if a handler is
not found, the search continues in the software block that called
the program or function, and so on. Figure 3-24 shows that at
each successive level of invocation, a new set of fault condition
handlers may be made available. And, when returning from
each program or function, the fault condition handlers for that
invocation become unavailable.

- The *read* command may raise *pod_addr_tied,*
  pod_forcing_active, or *pod_uut_power* fault conditions.
  Since *read* is a built-in function, it contains no fault
  condition handlers.

- The *mem_test* function contains a fault condition handler
  for the *pod_addr_tied* fault condition.

- The *uut_test* program contains fault condition handlers for
  pod_addr_tied and *pod_forcing_active* fault conditions.

- The 9100A/9105A prints a message on the operator's
  display for any unhandled fault conditions.

If the *read* command raises a fault condition when it has been
invoked as shown in Figure 3-24, which handler will take
control?

The search for a fault condition handler begins in the test which
raised the fault condition. If no fault condition handler for the
fault condition is found in that software block, the search
continues in the block that called the test that raised the fault
condition, and so on.

| 9100A/9105A | - - - - { | Takes Care of All Unhandled Fault Conditions |

Invokes | Returns to

| TL/1 Program "uut_test" | - - - - { | Fault Condition Handlers for pod_addr_tied and pod_forcing_active |

Invokes | Returns to

| TL/1 Function "mem_test" | - - - - { | Fault Condition Handlers for pod_addr_tied |

Invokes | Returns to

| *read* Command | - - - - { | No Fault Condition Handlers May Raise Fault Conditions: pod_addr_tied or pod_forcing_active or pod_uut_power |

Figure 3-24: Locations of Fault Condition Handlers

- If *read* raised a *pod_addr_tied* fault condition, the search for an active *pod_addr_tied* fault condition handler would be unsuccessful within *read* itself (the built-in function, *read,* contains no fault condition handlers), but would succeed in the next software block searched, *mem_test,* the software block that invoked *read.* The fault condition handler *pod_addr_tied* within *mem_test* will be used instead of the *pod_addr_tied* fault condition handler within *uut_test.*

- If *read* raised a *pod_forcing_active* fault condition, the search for an active fault condition handler would be unsuccessful first in *read* (the built-in function *read* contains no fault condition handlers) and also in *mem_test,* but would finally succeed in *uut_test.*

- If *read* raised a *pod_uut_power* fault condition, the search for an active fault condition handler would be unsuccessful first in *read* (the built-in function *read* contains no fault condition handlers), then in *mem_test,* and also in *uut_test.* In this case, the fault condition name and any arguments used would be displayed on the operator's display.

So the rule for finding a fault condition handler is: the search begins in the software block that raised the fault condition and continues back through all of the calling programs and functions until a fault condition handler that has the same name as the fault condition being raised is found. If this search process is unsuccessful, the fault condition name and arguments are displayed on the operator's display.

How a TL/1 Fault Condition Handler Is Invoked 3.7.5.

The job of a fault condition handler is to perform some action appropriate to the fault condition, and then dispose of the fault by performing a return.

Once an active fault condition handler matching the current fault condition is found, it is called, just like any other TL/1 block, using the arguments supplied in the *fault* command. A fault condition handler may do anything another TL/1 program or

condition handler may do anything another TL/1 program or
function block may do, including raising fault conditions and
invoking tests.

If the fault condition handler itself executes a *fault* command, a
new fault condition is raised. As with any fault condition, the
search for a handler begins in the current software block (in this
case, the handler itself) and then continues to the calling block.
Therefore, a fault condition handler must not raise the same fault
condition it handles or an infinite recursion will result. In this
case, the 9100A/9105A would generate the following error
message:

```
Stack overflow or infinite recursion
```

A fault condition handler terminates its execution by performing
a return, which discards the fault condition and allows the test
program to proceed.

Unhandled Fault Conditions   3.7.6.

If you don't supply a handler for a fault condition, the
9100A/9105A does the following:

- Displays a predefined message describing the fault
  condition for built-in fault conditions (see Appendix H of
  the *TL/1 Reference Manual*).

- Displays a message giving the name of the fault condition,
  and the fault condition argument names and values for
  non-built-in fault conditions.

The 9100A/9105A will then wait for the operator to select an
action from the operator's keypad as shown in Figure 3-25. The
operator may:

- **Press CONT:** This continues the test, but the test is
  considered to have failed (see the "Termination Status"
  section).

- **Press REPEAT:** The test is restarted by reinitiating
  execution of the top-level program.
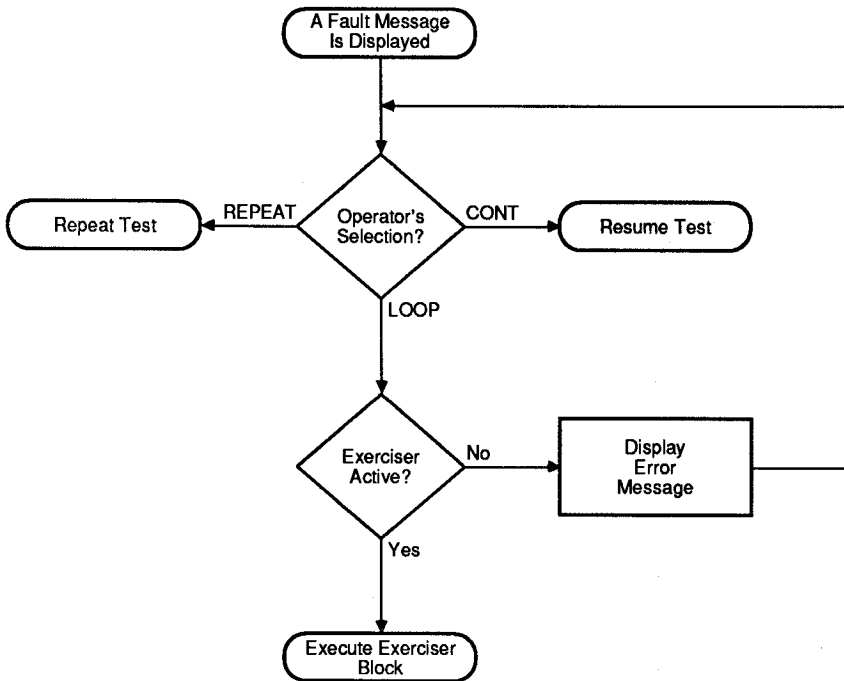
Figure 3-25: Alternative Actions for Unhandled Faults

- **Press LOOP:** If an exerciser for the fault can be found, it is invoked to try to re-create the UUT fault so that the operator can diagnose the problem.
- **Press HELP:** Displays a user-defined message from the HELP library if a message was defined for this fault name.

Creating a Fault Condition Exerciser 3.7.7.

A fault condition exerciser is a software block designed specifically to reproduce a fault condition in a UUT. A fault condition exerciser resembles a fault condition handler in that:

- A fault condition exerciser is defined within a program or function.
- A fault condition exerciser is active from the time that the program or function that defines the exerciser is called, until that program or function returns.
- A fault condition exerciser has a name that is the name of the fault condition that invokes the exerciser.
- The search for an active fault condition exerciser proceeds from the software block that raised the fault condition.

For example, the function *test18* could have an exerciser designed to re-create an *sw_short* fault condition simply by adding an exerciser block as shown below. The exerciser might have arguments such as the *position* argument in the example below:

```
function test18 (start, last)
            .
            .
            .
    exercise sw_short (position)
            .
            .
            .
    end sw_short
        .
        .
        .
end test18
```

A fault condition exerciser has an argument list, just like a fault condition handler, that is comprised of the fault condition arguments.

A fault condition exerciser is invoked when the operator presses the LOOP key on the operator's keypad (see Figure 3-25). The exerciser is invoked continually until the operator presses the STOP key on the operator's keypad. When an exerciser raises a fault condition, the exerciser is considered to have failed. No fault condition handler is invoked, and a message appears on the operator's display.

Termination Status (Passes or Fails)   3.7.8.

TL/1 programs and functions may either pass or fail. The TL/1 functions *passes* and *fails* are used to test this status in *if* statements:

```
if testbus addr $8000 passes then y = 1

if write($1000, $1FFF) fails then return
```

Every time a program, function, or built-in test is called, its status is initially set to "passes." But once a fault condition is raised, the termination status may change as illustrated in Figure 3-26 (when handling fault conditions) and Figure 3-27 (when exercising fault conditions).

- If the fault condition is handled by a TL/1 fault condition handler that returns to the calling block, the test status is not changed since the fault handler is assumed to have fixed the problem.

- If the fault condition is not handled and the operator presses the CONT key to continue the test before running an exerciser, the test status is set to "fails".

- If the operator presses the LOOP key to run an exerciser, the status of the last full invocation of the exerciser is retained: if the last iteration of the exerciser raised a fault condition, the retained status is "fails", otherwise it is "passes". Once the operator presses the CONT key to continue the test, the status of the test is set to "fails" if the exerciser's status was "fails".
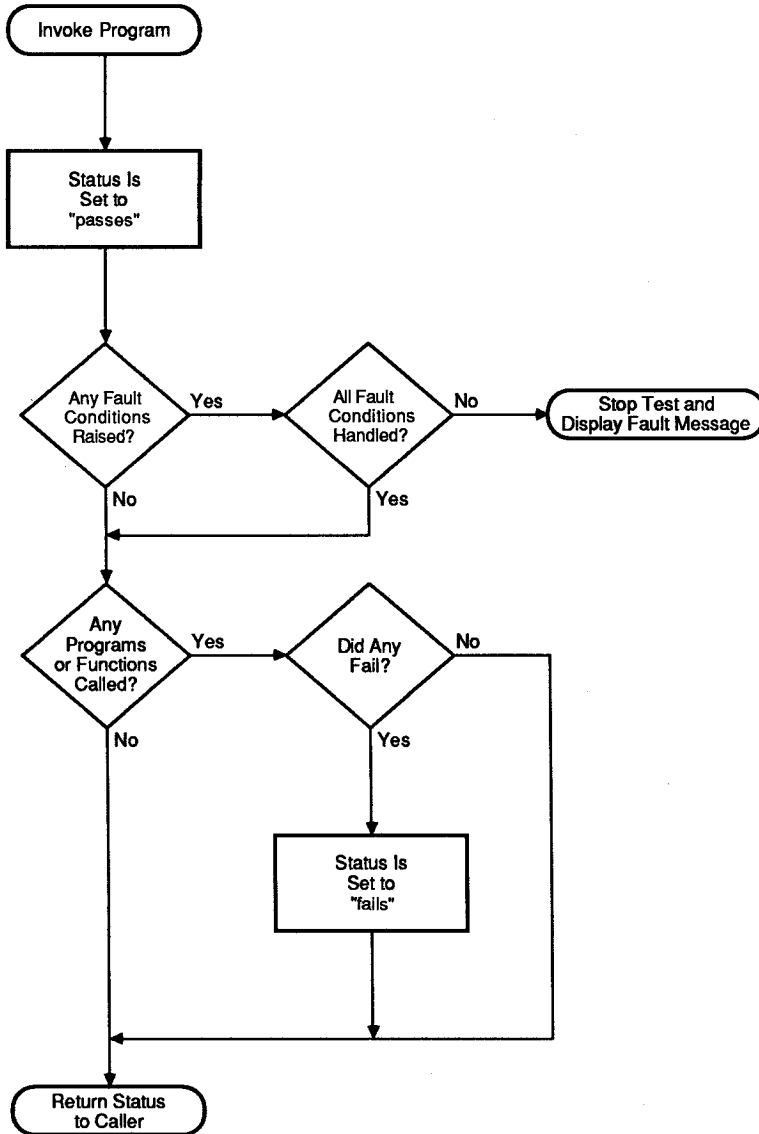
Figure 3-26: Termination Status when Handling Fault Conditions
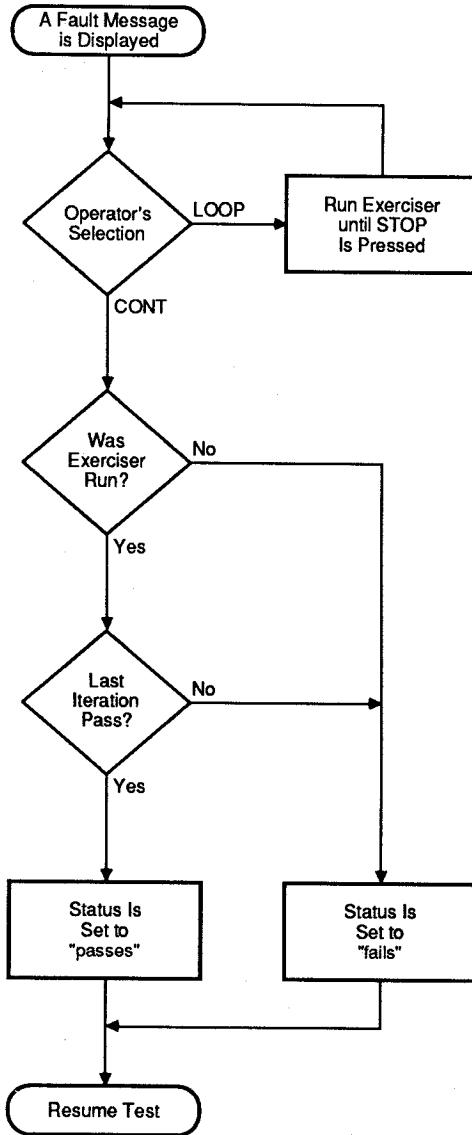
Figure 3-27: Termination Status when Exercising Fault Conditions

- If any program or function called by a software block fails, the status of the calling software block is set to "fails".

HELP LIBRARY   3.8.

The HELP library allows you to associate a help message with each fault name. Help messages may contain any text, but are most often used to display UUT-specific troubleshooting hints or procedures to follow.

HELP messages are stored together as text files in the HELP library. The 9100A editor edits the HELP library as object HELPLIB type LIBRARY. Refer to Figure 3-28 showing the editor display of a HELP library.

INDEX File   3.8.1.

The text file INDEX in the HELP library is special. The INDEX file contains zero or more lines of the form:

<fault name> <text file name>

<text file name> is an object name naming one of the text files in the HELP library. Refer to Figure 3-29 showing a typical INDEX file.

HELP Messages   3.8.2.

When an unhandled fault is displayed on the front panel of the mainframe, press the HELP key on the operator's keypad. The fault name is compared with each entry in the HELP library INDEX file for the current USERDISK. If a match is found, and the named text file exists, it is loaded from the disk and displayed. Otherwise, the 9100A/9105A beeps to indicate that no help is available for the current fault.

NAME: HELPLIB   DISK FREE: 4,589,056 BYTES
DESCRIPTION: Fault Specific Help Library

                        PRESS A COMMAND KEY OR HELP KEY

                        DIRECTORY OF HELPLIB (LIBRARY)

Help Files (TEXT):
     BUS1   BUS2   BUS3   BUS4   BUS5   BUS6
     CLKMOD1 . INDEX   IOMOD1   IOMOD2   IOMOD3   POD1
     POD10   POD11   POD12   POD13   POD14   POD15
     POD16   POD2   POD3   POD4   POD5   POD6
     POD7   POD8   POD9   PROBE1   RAM1   RAM10
     RAM11   RAM12   RAM2   RAM3   RAM4   RAM5
     RAM6   RAM7   RAM8   RAM9   ROM1   ROM2
     ROM3   ROM4   ROM5   ROM6   ROM7   UNKNOWN1

 F1   F2   F3   F4   F5   F6   F7   F8   F9   F10
REMOVE SAVE   COPY   STYLE

Figure 3-28: Editor Display of the HELP Library

```
/HDR/HELP_FILE/INDEX  (TEXT)                                    line 1
bus_data_tied bus1
bus_data_low_tied bus2
bus_data_high_tied bus3
bus_addr_low_tied bus4
bus_addr_high_tied bus5
bus_addr_tied bus6
clkmod_fuse_blown clkmod1
generic_fault gen1
iomod_dce iomod1
iomod_current_fault iomod2
iomod_fuse_blown iomod3
m_pod_bus_addr_high mpod1
m_pod_bus_addr_low mpod2
m_pod_romi_cs mpod3
m_pod_no_reset mpod4
m_pod_slow_clock mpod5
m_pod_buscycle_clock mpod6
m_pod_reset_addr mpod7
m_pod_reset_data mpod8
m_bus_kernel mpod9
m_pod_stopped mpod10
```

F1	F2	F3	F4	F5	F6	F7	F8	F9	F10
GOTO	SAVE		MARK			PASTE	REPL	SEARCH	

Figure 3-29: A Typical INDEX File

For the HELP facility to work, the fault must be displayed on the application display. Thus, the HELP facility works only when the 9100A/9105A is controlled from the operator's front panel, and is not available when running programs from the Debugger. Also, the fault must be an "unhandled" fault. This means either that there is no active handler for the fault, or that inside the handler is a *refault* or *fault* statement.

The name of the fault must exactly match the <fault name> in the HELP library's INDEX file. This name is the name appearing in the most recently executed *fault* statement. For example, if the program executes "fault bad_DMA", and this fault is not handled, and there is a line in "/hdr/helplib/INDEX" with the form:

```
bad_DMA dma_msg1
```

Then the text file "dma_msg1" is displayed when the HELP key is pressed.

If there is a handler for bad_DMA which does further diagnosis and executes the statement "fault DMA_no_handshake", then the HELP file associated with DMA_no_handshake (if any) is displayed, not the file associated with bad_DMA.

HELP files may be written for both built-in and user-written faults. A number of HELP files for built-in faults are provided on the Master User Disk. For a list of built-in fault names, see Appendix G of the *TL/1 Reference Manual*.

GFI COMMANDS   3.9.

TL/1 programs and the Guided Fault Isolation (GFI) software of the 9100A/9105A are designed to work together. TL/1 programs can call upon GFI to perform functional tests on selected UUT circuit nodes; GFI invokes TL/1 stimulus programs (also called stimulus routines) to initialize the UUT, to initialize 9100A/9105A hardware, and to apply the stimulus to the UUT.

The job of GFI is to interpret the UUT database stored on disk. Section 5 of this manual gives full information on how GFI databases are created. The GFI software is designed to use the information in each UUT database to:

- Decide which node of the UUT should be tested next.
- Execute TL/1 stimulus programs to exercise the node.
- Compare the actual and expected results of the stimulus.
- Either make an accusation about a faulty part or connection, or test another node.

TL/1 programs provide the customization required to gather responses from a particular UUT in a form usable by GFI. This section will show:

- How TL/1 stimulus programs, run under GFI control, should retrieve information from GFI.
- How GFI can be used by TL/1 programs to automatically run tests on UUT nodes.

Figure 3-30 summarizes the commands used to communicate between TL/1 and GFI.

Used When a Program Is	Command	Purpose
Invoked by GFI	gfi control	Tells if program was invoked by GFI.
	gfi device	Name of measurement device chosen by GFI.
	gfi ref	Get name of node being tested by GFI.
	gfi fail	Forces GFI to fail a pin.
	gfi pass	Forces GFI to pass a pin.
Invoking GFI	gfi accuse	Get GFI diagnosis of problem.
	gfi autostart	Enable or disable automatic startyp of GFI.
	gfi clear	Reset GFI for new UUT.
	gfi hint	Add node to end of GFI's suggestion list.
	gfi status	Return status of test on node.
	gfi suggest	Get next node in GFI's hint list.
	gfi test	Invoke GFI on a node.

Figure 3-30: Commands Used to Communicate Between TL/1 and GFI

Stimulus Programs Called from GFI 3.9.1.

Whenever GFI decides that a UUT node needs to be tested, it looks in the UUT database to find the names of one or more TL/1 programs. A stimulus program should perform reads and writes to the UUT that exercise a node in a repeatable way. Sections 5.5.8. and 5.5.9. of this manual contain information on how UUT nodes and stimulus programs are related.

A stimulus program is an independent program that must:

- Initialize the UUT as required in order for the stimulus to be applied.

- Initialize the pod and the measurement device (I/O module or probe) GFI has chosen to measure the response.

- Apply the stimulus to the UUT in an *arm . . . readout* block. If any faults are detected, stimulus programs may raise fault conditions.

- Read the results of the stimulus by using the *readout* command.

Stimulus programs do not compare the results of the stimulus with the results learned by GFI; the GFI software itself will do the comparison. An outline of a GFI stimulus program is shown in Figure 3-31. The only GFI command used is *gfi device,* which gives the name of the device (probe or I/O module) to be used in the setup actions and in the *arm* and *readout* commands. The *gfi device* command will report an error if it is used in a program not called from GFI.

```
program example1
    !
    ! Find out name of measurement device
devlist = gfi device
    !
    ! Initialize measurement device and pod.  Include connect
    ! statements to attach external control lines.
    !
        .
        .
        .
    !
    ! Perform stimulus in an arm . . . readout block
arm device devlist
        .
        .
        .
        ! Provide stimulus to make nodes wiggle
        !
        ! Make sure signatures are complete by calling
        ! checkstatus if using external sync on I/O module.
        !
        .
        .
        .
readout device devlist
end example1
```

Figure 3-31: Stimulus Program Called from GFI

The *gfi control* command returns the string "yes" when used in a stimulus program called from GFI; otherwise it returns the string "no"   The *gfi control* command (as shown below) allows you to write stimulus programs that may be called either by the operator, using the operator's keypad, or from GFI.

```
program example2
!
! If called from GFI, use measurement device
! chosen by GFI, otherwise use "/mod1".
if (gfi control) = "yes" then
    devlist = gfi device
    ref = gfi ref
else
    devlist = "/mod1"
end if
!
! Remainder of processing is identical whether
! or not this program was called by GFI.
!
    .
    .
    .
end example2
```

Figure 3-32 shows the typical steps that most stimulus programs should use.
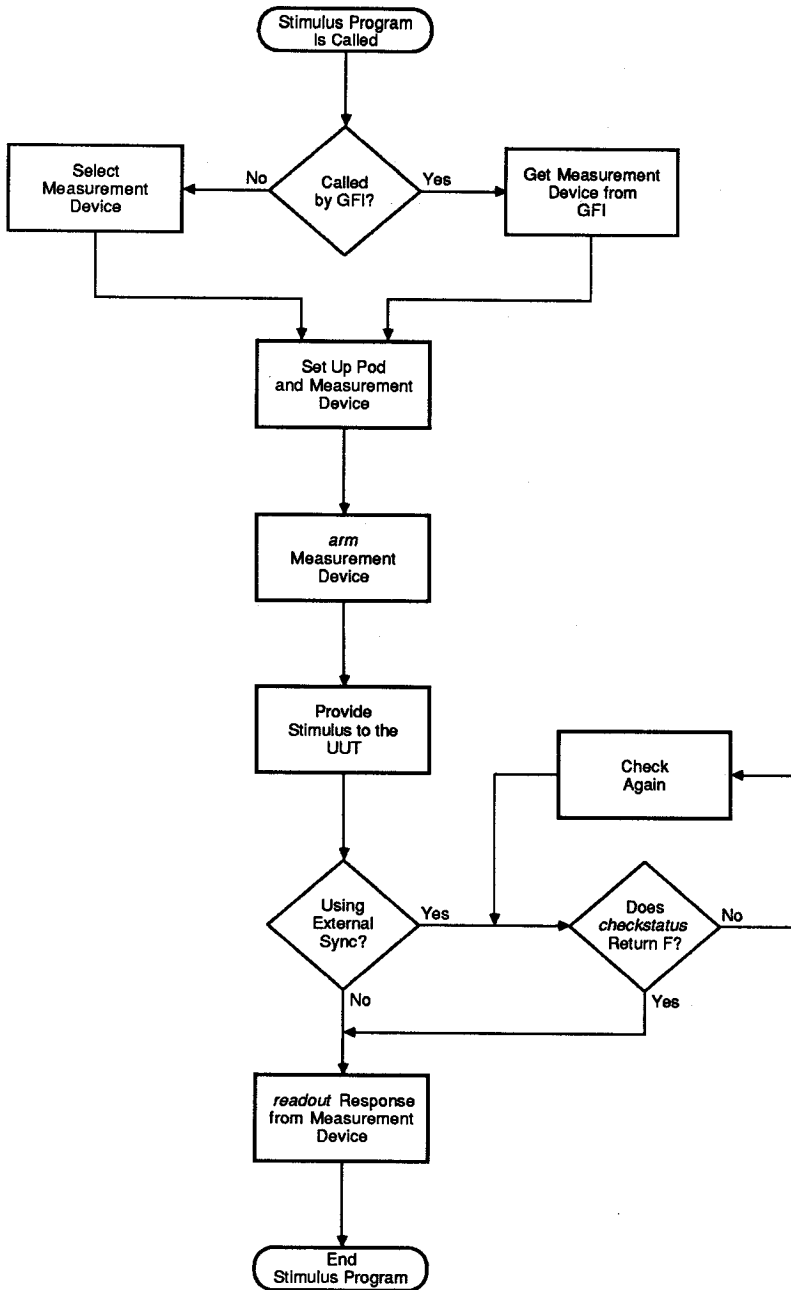
```
                    ┌─────────────────────┐
                    │  Stimulus Program   │
                    │     Is Called       │
                    └─────────────────────┘
                               │
                               ▼
┌──────────────┐          ◇─────────◇          ┌──────────────────┐
│    Select    │   No    ◇  Called   ◇   Yes   │ Get Measurement  │
│ Measurement  │◄────────◇  by GFI?  ◇────────►│  Device from     │
│    Device    │          ◇─────────◇          │      GFI         │
└──────────────┘                               └──────────────────┘
       │                                                 │
       └──────────────────────┐   ┌────────────────────┘
                              ▼   ▼
                        ┌───────────────┐
                        │  Set Up Pod   │
                        │ and Measurement│
                        │    Device     │
                        └───────────────┘
                               │
                               ▼
                        ┌───────────────┐
                        │      arm      │
                        │  Measurement  │
                        │    Device     │
                        └───────────────┘
                               │
                               ▼
                        ┌───────────────┐
                        │    Provide    │              ┌───────────────┐
                        │ Stimulus to the│             │    Check      │
                        │     UUT       │              │    Again      │◄───┐
                        └───────────────┘              └───────────────┘    │
                               │                              │             │
                               ▼                              ▼             │
                          ◇─────────◇              ◇──────────────◇         │
                          ◇  Using   ◇   Yes       ◇    Does      ◇   No    │
                          ◇ External ◇────────────►◇ checkstatus  ◇─────────┘
                          ◇  Sync?   ◇             ◇  Return F?   ◇
                          ◇─────────◇              ◇──────────────◇
                               │ No                      │ Yes
                               ◄─────────────────────────┘
                               ▼
                        ┌───────────────┐
                        │ readout Response│
                        │from Measurement│
                        │    Device     │
                        └───────────────┘
                               │
                               ▼
                        ┌───────────────┐
                        │     End       │
                        │Stimulus Program│
                        └───────────────┘
```

Figure 3-32: Typical Steps for Stimulus Programs

Invoking GFI from a TL/1 Program 3.9.3.

GFI is designed to be run either from the operator's keypad or from a TL/1 program. You may want to invoke GFI from a program in order to:

- Generate commands for an autoprober - The 9100A/9105A does not directly support automatic probing of components on a UUT. But a TL/1 program can generate commands for an autoprober, based upon suggestions generated by GFI.

- Generate a report based on the results of GFI - Automatic generation of a report of nodes tested, passed, failed, and accused requires a TL/1 program that creates a record of the GFI results.

- Provide hints to GFI - Functional tests for the functional blocks of a UUT can use GFI to test just the outputs of the functional blocks. When failures are discovered, the functional tests can provide hints to GFI to identify the nodes most likely related to any particular failure.

A TL/1 program that simulates the action of the GFI key on the operator's keypad would use the *gfi clear, gfi test, gfi accuse,* and *gfi suggest* commands. An example of such a program, which begins operation at its argument, *refdes,* and drives an autoprober, is illustrated in Figure 3-33. The commands used are:

- *gfi clear* -initializes GFI for a new UUT.

- *gfi hint* - adds a pin name to GFI's suggestion list. It is used in this example to give GFI a place to start on a new UUT.

- *gfi accuse* - is a string that identifies the problem which GFI has found.

- *gfi suggest* - is a string that is the name of the next node on GFI's list of nodes to be tested.

- *gfi test* - invokes GFI to perform all stimulus programs identified in the UUT database for that node. The *autoprompt "no"* argument prevents GFI from telling the operator where to place the probe, since the example uses an automatic probe.

- *gfi status* - returns one of the strings "good", "bad", or "untested". In this example, it is used to generate messages sent to a log file which records the results of testing each pin on each UUT.

```
program gficontrol (refdes)
declare string refdes = "U41-3"
        !
        ! Initialize GFI operation
gfi clear
        !
        ! Tell GFI to start at "refdes"
gfi hint refdes
        !
        ! Test as long as GFI has no accusation but
        ! has another suggestion to offer.
loop while (gfi accuse) = "" and (gfi suggest) <> ""
        ! Probe to next pin, test it, and log test results.
    nextpin = gfi suggest
        ! autoprobe is a user-defined program to move an
        ! autoprober arm to the specified pin
    autoprobe moveto nextpin              ! moveto is an argument name
    gfi test nextpin, autoprompt "no"
        ! logtest is a user-defined program to log failure data
    logtest pin nextpin, status (gfi status nextpin)
end loop
        !
        ! Record test results in log file.
if (gfi accuse) <> "" then
        ! loguut is a user-defined program to log failure
        ! messages
    loguut message (gfi accuse)
else
    loguut message "GFI failed"
end if
end gficontrol
```

Figure 3-33: GFI Called from a TL/1 Program

Section 4
Debugger

The debugger is an interactive tool for finding logical problems in TL/1 programs. Using this tool, you can initiate and control the execution of the TL/1 programs and functions. You can also view and alter the values of variables at intermediate stages of program execution. By following the path of execution and examining the values of variables during execution, you can determine if a program performs as intended.

The debugger requires compiled programs for execution and setting breakpoints. If you have not compiled your programs before entering the debugger, the debugger automatically compiles them as they are executed. If a program cannot be compiled due to errors, an error message is displayed and debugging cannot be continued. Exit the debugger, edit the program, and correct the error.

It is recommended that you use the COMPILE softkey and compile all the programs that you will be executing before you enter the debugger. This allows you to find all the compilation errors at one time, instead of coming across them one at a time during your debugging session.

ENTERING AND EXITING THE DEBUGGER 4.1.

You access the debugger by pressing the DEBUG softkey while you are editing a program. To return to the editor, press the Quit key. The cursor position is maintained when you move from the editor to the debugger and vice versa.

If you enter the debugger from a program that has not been compiled, the program is compiled automatically to ensure that the program is free from syntax errors and that it can be executed. If the compiler detects errors, an error message is displayed. Exit the debugger, edit the program and correct the error.

DEBUGGER SCREEN 4.2.

The debugger screen, shown in Figure 4-1, contains the same windows as the editor screen. It also includes an additional window, the execution window, which contains:

* Breakpoint indicators (BRK): Breakpoints may be set at specific lines in the program; a breakpoint causes the debugger to stop execution just before executing the statement(s) on the line containing the breakpoint.

* Execution pointer (→): This symbol is located at the line containing the next statement to be executed. The only exception is if execution is stopped due to a fault or error; then the execution pointer is located at the line that caused the fault or error.

  The execution pointer appears in boldface if it covers a BRK.

Once you start execution of a program with the debugger, execution can stop for any of these reasons:

- The end of the program is reached.

- A breakpoint is reached.

- An error occurs.

- A fault is detected on the UUT.

- The Quit key on the programmer's keyboard is pressed.

```
/HDR/ABC/DEMO (PROGRAM)                            STOPPED    Line 9
    program demo (num, address)
    declare
        numeric num
        numeric address
    end declare

BRK open device "/term1", as "output"

 →  if num = 0 then
        print "The number of iterations is zero"
    else
        loop while num <> 0
            data = read (address)
            print "The data read = ", data
            num = num -1
        end loop
    end if

    end demo


 F1    F2    F3       F4   F5   F6   F7       F8   F9   F10
Stopped at line 9                              (PRESS RETURN)
```

Figure 4-1: Debugger Screen Example

When execution stops because of an error or because a breakpoint is reached before the end of the program, the word "STOPPED" appears on the status line. If a fault is detected, the word "FAULTED" appears on the status line. If the end of the program is reached, the word "COMPLETE" appears on the status line. The debugger screen is updated to show the statement at which execution is stopped; the execution pointer is located at the line containing the next statement to be executed, unless a fault or error has occurred, then the execution pointer is located at the line that caused the fault or error. If a fault or error has occurred, a message is displayed. While the program or function is stopped, you can examine and/or change the value of variables. You can then continue execution of the program or function, or restart execution from the beginning.

Executions can be nested. For example, if program execution is stopped at a breakpoint, you can start executing another program or function by pressing the EXEC key. When that program or function completes execution, the debugger returns to the point at which you were originally stopped, and you can continue debugging the original program.

When a program completes execution, the return value from that program (if there is one) is displayed on the prompt line. Normally, the screen is updated to display line 1 of the program that finished executing. However, if the program that completed was a nested execution, the display is updated to show where execution was stopped on the original program.

DEBUGGER KEYBOARD   4.4.

The debugger keyboard is the same as for the editor. In particular, the Msgs key, the Help key, the down arrow key, and the up arrow key perform the same or similar functions. The Info key is inactive because the display contains no information window. The Edit key is inactive because you cannot edit other files from the debugger; to return to the editor, press the Quit key.

The following softkey commands are available only through the debugger:

- EXECUTE: Starts execution of a TL/1 program or a function defined in the current program. You enter the name of the program or function in response to the prompt or use the default name provided. If the program or function requires arguments, you are prompted for their values.

  If the argument being prompted for is numeric, you can enter it with either a decimal or hexadecimal radix. The default is decimal; to enter a hexadecimal number, place a "$" character in front of the number.

  You can EXEC any program that is within the standard TL/1 search path. When you enter the name of a program to be EXECed, the debugger first looks for the program in the currently selected UUT directory. If the program is not there and a pod is plugged into the 9100A, the pod directory is searched for the program. If the program still is not found, the program library is searched.

  The debugger can only execute programs that have been compiled. If you try to EXEC a program that has not been compiled, the debugger automatically compiles it before attempting to execute it. If the program cannot be compiled, a compiler error message is displayed. At this point you need to exit the debugger, edit the program, and fix the error.

  If the debugger stops in a function or handler that declares a variable with the same name as another function, EXECUTE will not allow you to execute that other function.

  During program execution, the messages window is activated when necessary to display TL/1 output on the monitor. The messages window remains active until execution is completed. After program completion, the

messages window is replaced by the debugger screen. Press the Msgs key to review the TL/1 output last displayed.

If you want to stop program execution prior to completion, press the Quit key on the programmer's keyboard or set a breakpoint. The STEP, NEXT, CONT, SHOW, and SET softkeys are valid only when the program is stopped prior to completion.

- VIEW: Displays an alternate TL/1 program. Enter the name of the desired program in response to the prompt. The program is loaded off the disk and displayed. Once a program is displayed, you can scroll through it, set and clear breakpoints, and execute it. This is a convenient way to examine called programs from within the debugger.

When you enter the name of a program to be VIEWed, the debugger uses the standard TL/1 program search path to find the program. First it looks in the currently selected UUT directory. If the program is not there and a pod is plugged into the 9100A, the pod directory is searched for the program. If the program still is not found, the program library is searched.

The VIEW softkey does not affect program execution. You can still CONT, STEP, or NEXT a stopped program after using the VIEW softkey.

- BREAK: Toggles the breakpoint status of the line at which the cursor is located. If the line does not contain a breakpoint, pressing the BREAK softkey sets a breakpoint; an indicator appears in the execution window. If the line already contains a breakpoint, pressing the BREAK softkey clears the breakpoint; the indicator disappears.

A breakpoint can be set for any program line that performs an action. A blank line, the lines of a declaration block, the first line of a definition block, and a line containing only a label or a comment cannot contain a breakpoint.

If you try to set a breakpoint in a program that has not been compiled, the debugger automatically compiles it before setting the breakpoint. If the program cannot be compiled, a compiler error message is displayed and the breakpoint is not set. At this point you need to exit the debugger, edit the program, and fix the error.

Setting a breakpoint is a convenient way to activate the CONT, STEP, NEXT, SHOW, and SET softkeys. Setting a breakpoint at the first executable statement of a program allows you to gain control so you can step through the program while following the path of execution or examining (or setting) the value of variables. Setting a breakpoint at the last executable statement of a program allows you to examine variable values that exist at the end of the program. Setting a breakpoint at intermediate points in a program allows you to stop execution at these points and then to single-step through the program after one of these breakpoints is encountered.

- CONT (CONTINUE): Continues the execution of a stopped program from the statement at which it was stopped. In some cases, it may not be possible to continue execution. In these cases, the following message will appear on the status line:

```
Execution cannot be continued.    <PRESS RETURN>
```

Press the Return key followed by the EXECUTE softkey to start execution again.

- STEP: Executes the next TL/1 line; the execution pointer is moved to the next line to be executed. If a line contains multiple TL/1 statements, all the statements are executed.

If the executed line is a program or function invocation, the execution pointer moves to the beginning of the function or program, and the screen is updated as necessary. There may be a slight delay while the program to be displayed is loaded off the disk.

- NEXT: Executes the next TL/1 line; the execution pointer is moved to the next line to be executed. If a line contains multiple TL/1 statements, all the statements are executed.

  If the executed statement is a function or program invocation, the function or program is executed completely; execution does not pause inside the function or program.

- SHOW: Displays the current value of a variable. If is is a numeric variable, it will be displayed as a decimal value. To display the equivalent hexadecimal number, press the Shift key and the SHOW softkey. You enter the variable name in response to the prompt. If the variable name is valid, the value of the variable is displayed. If the variable name you specify is not valid, an error message is displayed.

- SET (SET VARIABLE): Sets the value of a variable. You enter the variable name and a value in response to the prompts. If the variable does not exist or the value is not valid for the specified variable, an error message is displayed.

  If the variable being set is numeric, you can enter it with either a decimal or hexadecimal radix. The default is decimal; to enter a hexadecimal number, place a "$" character in front of the number.

- INIT (INITIALIZE): Clears all breakpoints, variable values, and other execution information so a program may be run (or re-run) from a known initial state. Also discards all nested executions. The display is updated to show the original program that was being edited when the debugger was entered.

- SEARCH: Moves the cursor to the next occurrence of a character string you specify at the prompt:

  SEARCH FOR _____

The character string may be a word, part of a word, or several words, up to 20 characters in length. The search is case sensitive; the upper-case "A", for example, is different from the lower-case "a".

If the debugger does not find the character string between the cursor position and the end of the file, the search wraps around to the beginning of the file and continues. If the debugger does not find the character string anywhere in the file, it displays an error message. The debugger retains the string you enter and offers it as a default the next time you issue the SEARCH command.

The search string can contain one or more wildcard characters (*). For example, if you specify MOD*, the debugger finds the next occurrence of MOD followed by any set of characters: MOD2, MODULE, or MODE, for example. If you want to search for a literal asterisk (*), enter two asterisks (**) in the search string. For example, to search for the expression 2*3, you would enter the search string 2**3. By entering two asterisks, the debugger interprets the character sequence as a literal asterisk rather than as two wildcard characters.

To reissue your last search (and avoid re-typing the search string), press the Shift key and hold it down while pressing the SEARCH softkey.

- FAULT: Turns the fault window on and off.

USING THE DEBUGGER   4.6.

This section shows how to use the debugger when:

- Execution errors occur.
- Debugging programs.
- Debugging blocks within programs.
- Debugging chained programs.

In addition, since some debugger commands are only valid at particular times during the execution of a program, the section below discusses when debugger commands are valid.

Availability of Debugger Commands   4.6.1.

Before Execution Begins

When the debugger is first started, program execution has not yet begun. The debugger knows nothing about the contents of the program, and program variables have not been created yet. Therefore, trying to use the SET softkey or the SHOW softkey prior to program execution will cause an error message to be displayed.

Likewise, the STEP, NEXT, and CONT softkeys cannot be used; they may be used only after execution of a program has begun. Before execution begins, you may set and clear breakpoints or initialize the debugger (which clears all breakpoints).

The debugger is in this state when INIT is pressed.

After Execution Ends

When program execution is complete (execution has not been stopped by a breakpoint or by pressing the Quit key), all local variables are discarded, but the breakpoints are not cleared. Trying to use SET or SHOW after program execution has ended will only work for global and persistent variables.

Likewise, the STEP, NEXT, and CONT softkeys cannot be used.

As described below, program executions can be nested. If a nested program completes execution, debugging can continue with the original program.

When Execution Is Stopped

Program execution can be stopped by pressing the Quit key or when a breakpoint is encountered. When execution is stopped, it is possible to show and set variables, to execute the program or one of the functions defined inside the program, to set and clear breakpoints, or to initialize the debugger. The debugger also stops execution when a fault is reported to the user. When the fault window is displayed, only the FAULT softkey (F10) is available. Pressing the FAULT softkey toggles the fault window on and off and leaves the program stopped.

When execution is stopped, the debugger sees variables from the perspective of the block containing the statement marked by the execution pointer. If execution is stopped at a statement inside a function or handler block, the debugger can set and show values only for the variables that are declared within that block, not the enclosing block. Other functions defined inside the program may be executed if they are not masked by a local variable declaration with the same name.

Execution may be resumed by pressing the CONT softkey, or the next statement may be executed with NEXT or STEP.

Start a nested execution by pressing the EXEC key and entering the name of a program or function to be executed. When that program or function completes execution, the debugger returns to the point at which you were originally stopped, and you can continue debugging the original program.

When an Error Occurs   4.6.2.

If an error occurs in the executing program, execution is interrupted and the line containing the error is marked by the execution pointer. Because an error occurred, execution cannot be resumed with STEP, NEXT, or CONT. However, you can usually set and show variables or execute the program or one of the functions defined inside the program. Some errors result in more serious trouble for the debugger. These errors are called fatal errors and cause execution to end. After a fatal error, the following message will be displayed if you attempt an illegal operation:

```
Cannot run program after a fatal error.
```

If the program that is being debugged calls another TL/1 program, and an error occurs in the called program, the display is updated to show the called program.

Another type of error that can occur when a program is called is a TL/1 compiler error. If the called program has not been compiled, the debugger compiles it before attempting to execute it. If a compilation error occurs, the display is updated to show the point of the program call and the compiler error message is displayed. At this point you should exit the debugger, edit the called program, and fix the error.

Setting Breakpoints

A breakpoint may be set on any executable statement. Declarations, comment lines, and blank lines are not executable statements. The first line of a program, function, handler, or exerciser block is not executable either.

Gaining Control of Program Execution

Normally, if you start the debugger and begin execution with the EXECUTE softkey, the program is run without pause, as it would run from the operator's interface. This does not provide much assistance in debugging. However, it is easy to get control of the program by setting a breakpoint early in the program, either at the first executable statement or after initializing functions are performed (but before the statements you wish to examine). You could also watch either display for output that indicates the progress of a program. When execution seems to have progressed far enough, you can press the Quit key to stop the program. This is less precise than setting breakpoints but can be effective on long programs that frequently send output to the displays.

Multiple Statements

If a breakpoint appears on a line containing multiple statements, the breakpoint is encountered only once, before the first statement is executed. If you press the STEP or NEXT softkey, execution will continue through the rest of the line without pause.

Setting and Showing Variables

The execution pointer indicates the next statement to be executed. This is important to remember when examining variables in an assignment statement. Suppose you wish to examine the effect of this statement:

```
n = val (stringvar)
```

If you set a breakpoint on that line, when execution stops you see:

```
|→|    n = val (stringvar)
```

The debugger is ready to execute this statement. If you show the value of n, it will be the value existing before the assignment takes place. Then if you press STEP or NEXT, the assignment will occur and the execution pointer will point to the next statement to be executed. At that point you can show n to see the effect of the assignment statement.

Debugging Blocks Within Programs   4.6.4.

Debugging If Blocks

When TL/1 encounters the *if* command, it evaluates the condition for the first block of controlled statements. If the condition is false, the second condition (supplied by an *else if* command if any) is evaluated, and so forth. As soon as TL/1 finds a condition that is true, the first statement in that controlled block is executed.

When TL/1 encounters an *else if* command or an *else* command
after executing the statements in a controlled block, TL/1 knows
the end of the controlled block of statements has been reached.
Execution continues with the first statement past the *end if*
command.

A breakpoint at an *if* command will stop execution before the
condition is evaluated. To determine whether a given branch of
the *if* has been taken, a breakpoint should be set on the first
statement of the controlled block, not at the *if*, *else if*, or *else*
command.

Debugging Loop Blocks

A breakpoint at the *loop* command will stop execution before the
loop is entered. A breakpoint at the first controlled statement
will stop execution at the beginning of each iteration of the loop,
and a breakpoint at the *end loop* command will stop execution at
the end of each iteration.

Debugging Functions

To get control of the debugger inside a function, a breakpoint
can be placed on the first executable statement of the function.
Or, you can set a breakpoint at the statement where the function
is invoked. Then, when execution is stopped, press the STEP
softkey to single-step through the function. Pressing the NEXT
softkey would execute the function completely, without pause
(unless a breakpoint is encountered while executing the
function).

If an error is found when debugging a function, execution of the
function cannot be continued. You can still set and show
variables or call another function defined inside the program.

Debugging Handlers

To get control of the debugger inside a handler, a breakpoint can be placed on the first executable statement of the handler. Or, if the handler is to be invoked through a TL/1 *fault* statement, you can set a breakpoint on the line containing the *fault* statement. Then, when execution is stopped, press the STEP softkey to single-step through the handler. Remember, since handlers for a particular fault condition can be defined within different program and function blocks, more than one handler may be available when a *fault* statement is executed.

It is not possible to debug fault condition exercisers using the debugger. However, they may be partially tested by temporarily changing them to functions and adding a call to the function either in the fault handler or following the statement that raised the fault.

Debugging Chained Programs   4.6.5.

In a fully developed system of test and troubleshooting programs, one program often calls another, creating chains that can grow quite complex.

The VIEW softkey is useful for displaying called programs and setting breakpoints in called programs. The STEP softkey is useful for following execution into and out of called programs. The SET and SHOW softkeys are useful for examining variables when execution is stopped in a called program.

It is recommended that you first compile all the TL/1 programs before attempting to debug a large set of programs. This allows you to find and fix all the compilation errors before beginning the debugging session. It also ensures that your debugging session is not interrupted by compilation errors. Using the debugger to find compilation errors is not recommended. It is much easier to compile all the programs in advance, find, and fix all the compilation errors before starting the debugging session. To compile all the programs in a UUT, edit the UUT, and press the COMPILE softkey.

Section 5
Guided Fault Isolation
(GFI)

INTRODUCTION   5.1.

This section introduces the 9100A/9105A Guided Fault Isolation (GFI)troubleshooting utility. The material assumes that you are familiar with the 9100A editor and TL/1 programming concepts.

The following features are described:

- The basic GFI algorithm.

- 9100A/9105A enhancements to GFI.

- GFI database and stimulus program reference.

- UFI (Unguided Fault Isolation).

- How GFI differs from UFI.

- Using GFI at the operator interface.

Functional tests determine whether a UUT performs as intended and therefore indicate whether or not it is functional. If a UUT fails a functional test, the test results generally cannot tell you how to repair the UUT. If you wish to find out why the UUT failed, you must troubleshoot it.

GFI is a troubleshooting procedure, implemented in the 9100A/9105A by a built-in program that directs the operator through a series of steps to locate the cause of UUT failure. The program uses a GFI algorithm to backtrace from a bad output to the responsible fault.

The GFI program is general enough to troubleshoot any digital circuit. To apply GFI to a particular UUT, you must supply UUT-specific information to the GFI database. Once you have stored the database in the UUT directory, an operator can use GFI to troubleshoot the UUT without much knowledge of its functionality. The operator has only to follow the directions displayed by the GFI program.

Some examples in this section are designed for an 80286 pod and the Demo/Trainer UUT (available as an option from Fluke). Even if you do not have this option you will find it useful to study the examples; they can be applied to other UUTs.

NOTE

In this section, "components" refers to parts such as ICs on the UUT. "Devices" refers to 9100A/9105A attachments such as the probe or an I/O module.

GFI locates UUT faults by backtracing from a bad output until it finds the fault. GFI considers a fault located when it finds a componentaccepting good input but producing bad output. The component could be bad, or its outputs loaded. Loading is often due to a bad connection that is (incorrectly) stuck at one level or tied to another signal.

GFI also considers a fault located when it finds an open circuit: a connection where the measured response is good at one end but bad at the other.

The following example demonstrates the GFI backtracing process. The circuit of Figure 5-1 represents a portion of a UUT with a fault at point A, a short to ground. When you perform the functional test for this portion of the UUT, the test should fail with a bad output at point B.

To begin backtracing, first verify that the output at point B is bad. You execute a stimulus (typically a combination of read and write commands) from the operator's keypad and observe the response at point B using the probe. With knowledge of the UUT logic, you can decide whether the response indicates that the circuit is performing correctly. A correct response contradicts the result of the functional test, and you must question whether the stimulus adequately reproduces conditions that caused the circuit to fail.
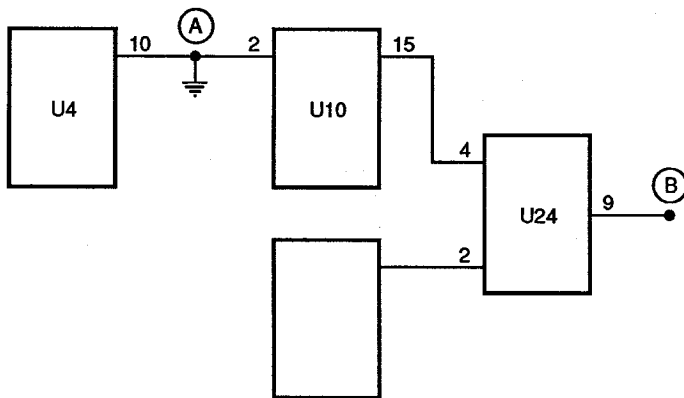
Figure 5-1: Example UUT Circuit with Fault

Once you verify that the response at point B is incorrect, you follow these steps to locate the fault:

1. Verify that the signal is also bad at the output pin, U24-9.

2. Check each input to U24 by applying a stimulus for each input and observing its response. U24 has two inputs, at pins 2 and 4.

3. Assuming that you first check the input at U24-2 and find it good, you should then check U24-4.

4. The input signal at U24-4 is bad; according to Figure 5-1 the input originated at U10-15.

5. When you probe U10-15, you will find that the signal is bad. You have therefore eliminated the chance of an open connection between U10-15 and U24-4.

6. The input signal at U10-2 is bad; according to Figure 5-1 the input originated at U4-10.

7. Check the inputs to U4. They are all good so at this point backtracing stops, having found that U4 accepts good inputs but produces bad output. The result suggests either that U4 is bad or that its output is loaded.

If U4 is defective, it can be replaced. If its outputs are loaded, a little thought is necessary. Loading may be caused by a short (as in this case), a bad component connected to U4-10, or a bad control line on a component connected to U4-10. At this point it should take little time to check all possibilities until you find the short at A.

This backtracing method is the basis of the GFI algorithm illustrated in Figure 5-2. GFI starts with a bad signal and locates the immediate source of the signal. GFI then checks each input of the source for more bad signals. As long as it
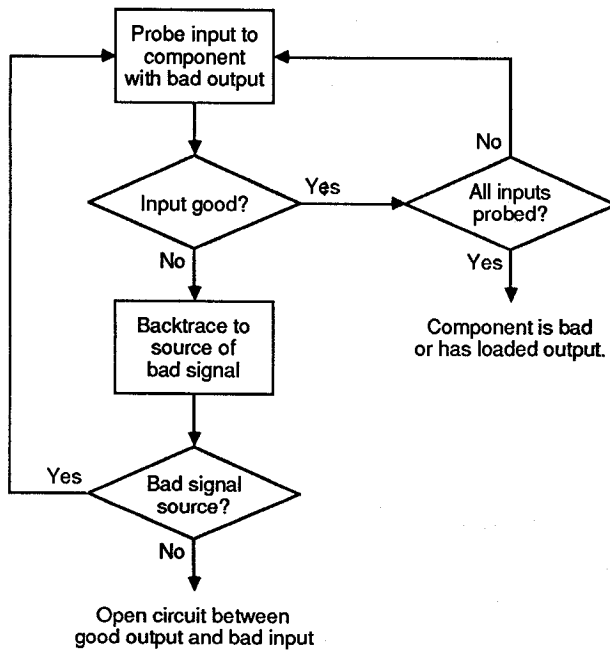
```
                ┌─────────────────┐
        ┌──────→│ Probe input to  │←──────────────┐
        │       │   component     │               │
        │       │ with bad output │               │
        │       └─────────────────┘               │
        │                │                         │
        │                ▼                      No │
        │            ╱───────╲          ╱───────────╲
        │           ╱         ╲   Yes  ╱  All inputs  ╲
        │          ╱ Input good?╲─────→╲   probed?    ╱
        │           ╲         ╱         ╲            ╱
        │            ╲───────╱           ╲──────────╱
        │             No │                   Yes │
        │                ▼                       ▼
        │       ┌─────────────────┐    Component is bad
        │       │  Backtrace to   │    or has loaded output.
        │       │   source of     │
        │       │   bad signal    │
        │       └─────────────────┘
        │                │
        │                ▼
        │  Yes       ╱───────╲
        └───────────╱ Bad signal╲
                    ╲  source?  ╱
                     ╲───────╱
                         No │
                            ▼
                   Open circuit between
                   good output and bad input
```

Figure 5-2: The Basic GFI Algorithm

finds bad input signals, it will backtrace to the source of the signal and check the source for bad inputs.

ADDITIONAL GFI FEATURES   5.3.

GFI is a very effective algorithm which locates faults in almost any digital electronic circuit. The 9100A/9105A uses the enhancements discussed in the following sections to reduce the time needed by GFI to troubleshoot a circuit.

The I/O Modules   5.3.1.

GFI algorithm efficiency is significantly increased if all pins on a UUT component can be probed simultaneously. The I/O modules were designed for this purpose. They are connected to the ICs by adapters of various sizes. The operator, when prompted by GFI, uses the adapter to clip over the IC to be tested.

Using I/O modules reduces the chances of probing the wrong IC pin, and avoids the need to probe the same IC more than once. Since many pins of an LSI chip may have to be probed during one backtracing operation, the time saved can be substantial.

More information on the I/O modules and the use of the I/O MOD operator's keypad command can be found in the *Technical User's Manual*.

Experience has shown that relatively few faults are caused by bad connections. We can therefore usually assume that if an input is bad, the output driving it is also bad. One of the easiest ways of reducing backtracing time is to use this assumption and initially probe only IC inputs.

Figure 5-3 shows how initially probing only inputs can save time. In the example, $(2n + 3)$ probes would be needed to diagnose the bad node by probing outputs and inputs. By probing only inputs, we reduce the number of probes needed to $(n + 4)$.

Once a fault has been tentatively diagnosed, you must verify the initial assumption that there was no bad connection. In this example, a final probing of pin U1-12 would verify the assumption that U1-12 and U2-3 are connected.
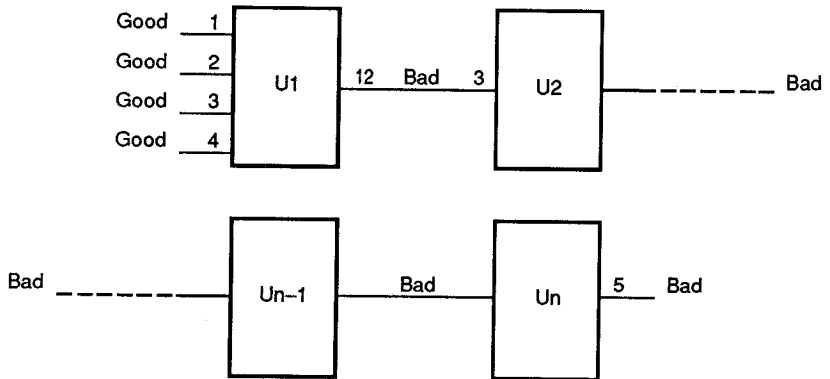
Figure 5-3: Benefits of Probing Inputs before Outputs

Related Inputs   5.3.3.

Related inputs are the pins that should be examined if an output pin is bad. These pins affect an output pin or a bidirectional pin when it is acting as an output. Power and ground connections are not related inputs, but they can cause an output pin to fail, so they are always tested when an output is bad.

In Figure 5-4, related inputs to U7-3 are U7-1 and U7-2. If U7-3 is bad, GFI probes only U7-1, U7-2, and power and ground connections, ignoring other inputs.

U16 in Figure 5-4 is a component whose inputs have been prioritized. According to the table, if output 02 fails, inputs are probed in the following order: Enable, Select, A2, B2, Vcc, and ground. It is not necessary to list Vcc or ground; they will be checked automatically as the lowest priority pins. Backtracing continues from the first input found to be bad. If they are all good, U16 is bad or has loaded outputs.

The related input pins are specified in a list. You can control the order in which they are probed by listing them in the desired order. The highest priority pins should be listed first and the lowest priority pins last.

If no related inputs are specified for a pin that has a bad output, GFI will begin probing the component's inputs in the order of their pin numbers.

U7

U16
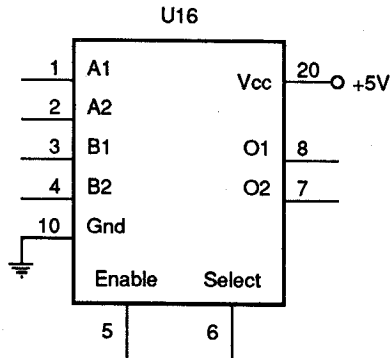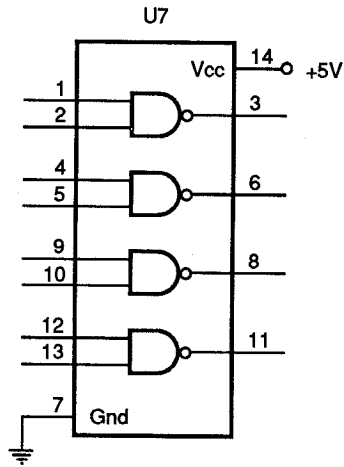
U16 Output	Related inputs, highest priority first
O2	5, 6, 2, 4, 20, 10 (Enable, Select, A2, B2, Vcc, Gnd)
O1	5, 6, 2, 4, 20, 10 (Enable, Select, A2, B2, Vcc, Gnd)

U16 Related Input Priorities

Figure 5-4: Related Inputs and Their Priorities

Leapfrogging

In Figure 5-5, if U6-10 is bad, a programmer may know from experience that the bad output originates several components away on the backtracing path at U3-2. Considerable time can be saved by jumping directly to the suspect component. This capability, called leapfrogging, is accomplished by specifying priority pins in the stimulus program response files. In Figure 5-5, U3-2 might be specified as a priority pin.

If GFI finds that U6-10 is bad, it will jump to U3-2, avoiding intermediate components. If U3-2 is bad, then the fault must lie even further back on the path; backtracing will therefore resume from U3-2.

If U3-2 is good, GFI will return to U6, where it will test related inputs of U6-10. The related inputs are tested in the order of their priorities. If U6-10 has no related inputs, all inputs to U6 are tested in the order of their pin numbers.
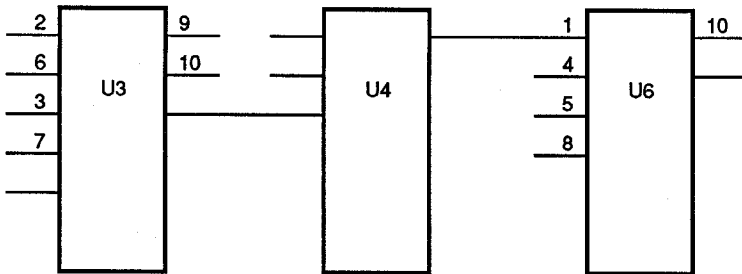
If defective output is:	Priority pin is:	Related inputs, in order of priority (highest first), of U6–10
U6-10	U3-2	U6-1, U6-4, U6-5, U6-8

Figure 5-5: Priority Pins

Feedback Loops

The GFI backtracing algorithm is successful for most digital circuitry where logic paths are straight lines. However, some logic paths are circular rather than straight. Such paths are called feedback loops.

Figure 5-6 shows a feedback loop. U6 receives input from U4, and U4 receives input from U8; this much of the path is straight. However, U8 receives input from U6, creating a loop.

If a loop is defective, none of its components will receive input that is all good. A component can be called "bad" only if it accepts good input but produces bad output. Therefore, a component cannot be considered bad while forming part of a feedback loop.

If a bad output is found at C, GFI will backtrace from U6 to U4 to U8 and then encounter U6 again. At this stage GFI will realize that it has found a loop and will try to establish that the cause of the fault lies outside the loop. It does so by testing all inputs to the loop from components outside the loop (inputs A and B). If GFI finds that one of those inputs is bad, it will continue backtracing from the bad input to components outside the loop.

If all inputs to the loop from outside are good, GFI will display a message indicating that there is a bad feedback loop and will list the output pins comprising the loop. For Figure 5-6, this list would be U6-3, U8-3, and U4-2.
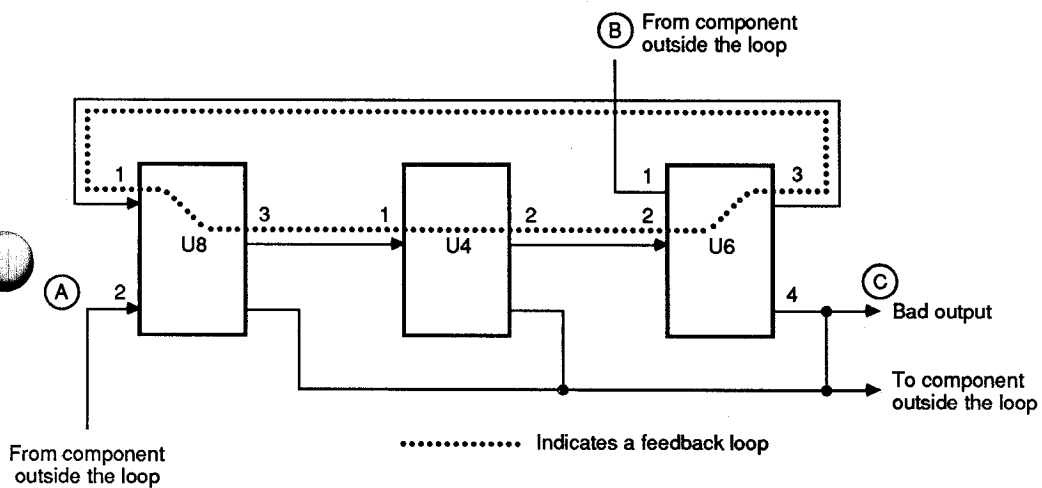
Figure 5-6: Feedback Loops

GFI DATABASE OVERVIEW   5.4.

With GFI, troubleshooting becomes a routine matter of moving a probe or an I/O module adapter to locations on the UUT as prompted. An inexperienced technician can troubleshoot a UUT without knowing how it works because you, the programmer, have previously stored UUT-specific information in the GFI database.

The Database and Stimulus Programs   5.4.1.

The compiled GFI database contains the following types of items:

- Part descriptions.
- Reference designator list (REFLIST).
- Node list (NODELIST).
- Stimulus program responses.

In addition to items in the database, GFI requires a set of TL/1 stimulus programs. The programs are used to generate the stimulus program responses stored in the database. Stimulus programs are stored along with other programs. Each stimulus program should have a corresponding stimulus program response file.

The stimulus programs and all items in the GFI database (except for the part descriptions) are stored in the UUT directory. Part descriptions are stored in a part library (PARTLIB) and can be used for any UUT.

Items associated with the GFI database are described on the next page and (in more detail) in the "GFI Database Reference" further on in Section 5.

- **Part description:** A UUT component description that identifies the package type, number of pins, and functions of each pin (such as input or output). The related input pins are identified for each output.

  Descriptions are stored in a UUT, or in a part library (PARTLIB). Descriptions in the PARTLIB can be used for any UUT. Thus, you do not need to enter the same description into the database for every UUT that uses the part.

- **Reference designator list (REFLIST):** A pairing of the name (reference designator) of each component on the UUT with a part description from the part library. For example, U5 may be a designator for the part 7400, whose description is stored in the part library. The device needed to test the part is also specified.

- **Node list (NODELIST):** A description of all UUT nodes. A node is a group of pins connected to each other. All pins forming a node must be identified.

- **Stimulus programs:** TL/1 programs that exercise UUT nodes. For example, a data line stimulus is a sequence of *read* and *write* commands, which cause the UUT to transmit signals over the line. The responses caused at a node by a stimulus can be measured and analyzed.

  GFI uses a stimulus to check a suspect node, comparing its response to that previously obtained from a good node. A stimulus must be available for each node. You need enough knowledge of the UUT logic to design stimuli that thoroughly and accurately exercise all nodes on the UUT.

- **Stimulus program responses:** The responses
  characterizing a known-good UUT. Responses have an
  important role in GFI, linking a stimulus program to the
  nodes that the program tests. A response file identifies the
  nodes exercised by a stimulus program, and contains data
  characterizing each node. A node is characterized by using
  GFI LEARN to collect response data from that node on a
  known-good UUT. Response data can be CRC
  signatures, asynchronous level histories, clocked level
  histories, and transition counts or frequency data. GFI
  compares response data from a tested node to data stored
  in the response file to determine if the node is good.

  In GFI, each stimulus program is paired with an identically
  named stimulus program response file: for example, the
  response file *dma_circ* contains responses to the stimulus
  program *dma_circ*. A stimulus program may exercise
  several UUT nodes. Each node should be described by a
  line in the corresponding response file. The line should
  identify the node being exercised, specify its priority pin
  (if any), and display the response data chosen to
  characterize the node.

How GFI Uses the Database and Stimuli   5.4.2.

The table in Figure 5-7 summarizes how GFI uses the database
and stimuli to test a component and generate probing
suggestions if the component has bad output.

Consider the example of Figure 5-5; if we specified that pin
U6-10 was to be tested, GFI would:

1. Look in REFLIST to determine the device (probe or
   I/O module) to test U6 with. GFI then prompts the
   operator to probe or clip U6.

2. Look in NODELIST to see what other pins are on the
   same node as U6-10.

3. Determine all suitable stimuli. GFI searches for stimulus program response files specifying (as a node signal source) U6-10 or a pin on the same node as U6-10. Suppose the response file named *addr_out* lists responses for node signal source U34-1. If U34-1 is on the same node as U6-10, then the stimulus program named *addr_out* is suitable for testing U6-10 as an input.

4. Apply all suitable stimuli by executing all relevant stimulus programs. In step 3, if it was found that stimulus programs *addr_out* and *micro_data* exercised U6-10, both are executed.

5. Determine whether the node is good or bad. In step 4, as each stimulus program is executed, responses at U6-10 are compared to those stored in the corresponding response file.

6. If step 5 shows U6-10 was bad, and if the stimulus program response file specifies a priority pin, then GFI would recommend probing at the priority pin.

    If step 5 shows U6-10 to be bad and if no priority pin is specified, then GFI would look at the part description and recommend probing related inputs in the order that they are listed.

When	Look at:	(TYPE)	In order to:
Testing pins at a specified location	REFLIST	REF	Determine the testing device for the pin; prompt the operator.
	NODELIST	NODE	Determine suitable stimulus programs: • Check which pins are on the same node.
	Stimulus Program Responses	RESPONSE	• Find all the stimulus programs which use, as signal sources, the pin under test, or any other pin on the same node.
	Stimulus Programs	PROGRAM	Execute all suitable stimulus programs.
	Stimulus Program Responses	RESPONSE	Determine whether the pin is good or bad by comparing responses to those stored in program response files.
Generating suggestions	Stimulus Program Responses	RESPONSE	Suggest priority pin, if specified.
	REFLIST Part Descriptions	REF PART	Suggest related inputs at the component of steps 1–5, if priority pin is unspecified or was not a useful hint.
	Stimulus Program Responses	RESPONSE	Suggest backtracking to source of bad signal.

Figure 5-7: How GFI Uses the Database and Stimuli

This section is a reference for the following items, some of which were described in the "GFI Database Overview" located in Section 5.

- Part library (PARTLIB).
- Part descriptions.
- Reference designator list (REFLIST).
- Node list (NODELIST).
- Stimulus programs.
- Stimulus program responses.

The compiled GFI databaseconsists of part descriptions, a reference designator list, a node list, and stimulus program response files. You can create or modify each of these files by using the editor. GFI also uses stimulus programs, which are not included in the database itself.

When you first create a UUT directory, it is empty. The editor provides a framework for entering or editing each type of UUT information.

Developing stimulus programs and stimulus program response files requires a thorough knowledge of the UUT logic. However, creating part descriptions, reference designator lists, and node lists mainly entails data entry and can usually be performed by a less skilled user.

Creating a GFI database involves:

- Describing the circuit by creating REFLIST, NODELIST, and updating PARTLIB if necessary.
- Writing stimulus programs and storing them.
- Learning responses from a known-good UUT and storing them in stimulus program response files.
- Compiling the GFI database.

- Generating a summary of the GFI database, which analyzes the GFI test coverage.

Part Library   5.5.1.

Figure 5-8 shows a screen from the 9100A's standard part library. Each item in the part library is a part description, which can be accessed by any UUT's GFI database. You can modify the existing descriptions or add new ones as described in "Entering a Part Description." further on in Section 5.

The information window includes the following fields:

- NAME: The name PARTLIB. This field cannot be edited.
- DISK FREE: The amount of disk space that is still available. This field cannot be edited.
- DESCRIPTION: An optional one-line description of the part library.

Below the information window is a listing of the names of all part descriptions contained in the part library.

```
/HDR/PARTLIB (LIBRARY)
NAME: PARTLIB                              DISK FREE: 18,399,232 BYTES
DESCRIPTION: ▓_____

             PRESS A COMMAND KEY OR HELP KEY

             DIRECTORY OF PARTLIB (LIBRARY)

Parts (PART):
     2016     2674     2675     2681     27128    27256
     4000     4164     7400     7401     7402     7403
     7404     7405     7406     7407     7408     7409
     7410     74107    74109    7411     74111    74112
     74113    74114    7412     74121    74122    74123
     74125    74126    74128    7413     74131    74132
     74133    74134    74138    74139    7414     74148
     7415     74150    74151    74153    74157    7416
     74160    74161    74162    74163    74164    74165
     74166    7417     74175    7420     7421     7422
     7423     7424     74244    74245    7425     74257
     7426     7427     7428     7430     7431     7432
 F1    F2    F3       F4    F5    F6    F7          F8    F9    F10
REMOVE  SAVE         COPY        EDIT  QUIT
```

Figure 5-8: Standard Part Library

5-23

Part descriptions for SIPand DIP packages, are shown in Figures 5-9 and 5-10, respectively. Each part description will contain data about one type of component, such as a 2114 IC, or an 74LS4148 IC, or a resistor. The part description fields that are active depend upon whether the information window is on or off.

Information Window on:

- NAME: The part identification. This field cannot be edited.

- NO. PINS: The number of pins on the part. The number must be in the range 1 through 255.

- PACKAGE: The package type for the part. Use the Field Select key to set this field to either SIP (Single In-line Package) or DIP (Dual In-line Package). Any part that is not a Dual In-line Package should be specified as SIP.

- DISK FREE: The amount of disk space that is still available. This field cannot be edited.

- WRITE PROTECT: The write-protection status of the file. Use the Field Select key to set this field to YES to specify write protection for the file. If the file is write protected, the editor prompts you when the file is saved to ensure that changes are intentional. If the file is not write protected, you will not be prompted. A change in write-protection status does not become effective until after you save your current edits.

The NO. PINS and PACKAGE fields define how DIP component pins are mapped to I/O module pins. The mapping is performed during GFI. For SIP components, the probe should be used for GFI.

```
/DRV/PARTLIB/JACK16 (PART)
NAME: JACK16                              DISK FREE:    521,984 BYTES
NO. PINS: 16
PACKAGE: SIP                              WRITE PROTECT: NO

   1 | →
   2 | →
   3 | →
   4 | →
   5 | →
   6 | ←
   7 | ←
   8 | →
   9 | ←
  10 | →
  11 | →
  12 | →
  13 | →
  14 | →
  15 | →
  16 | →

F1   F2      F3       F4   F5   F6   F7       F8   F9   F10
     SAVE
```

Information Window On

```
/DRV/PARTLIB/JACK16 (PART)                              Line 5

          PIN NAME  RELATED INPUT PINS

   1 | ≣
   2 | →
   3 | →
   4 | →
   5 | →
   6 | ←
   7 | ←
   8 | →
   9 | ←
  10 | →
  11 | →
  12 | →
  13 | →
  14 | →
  15 | →
  16 | →

F1   F2      F3       F4   F5   F6   F7       F8   F9   F10
GOTO SAVE             MARK           PASTE
```
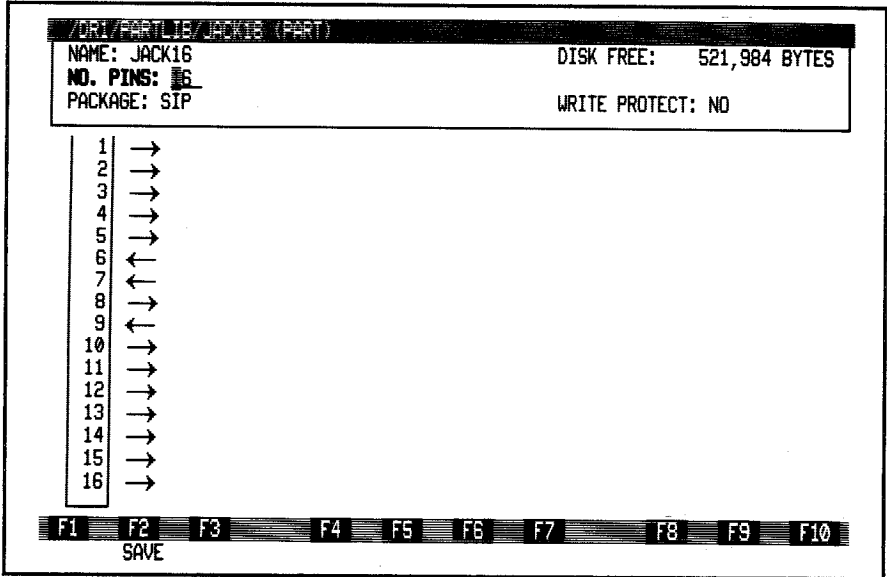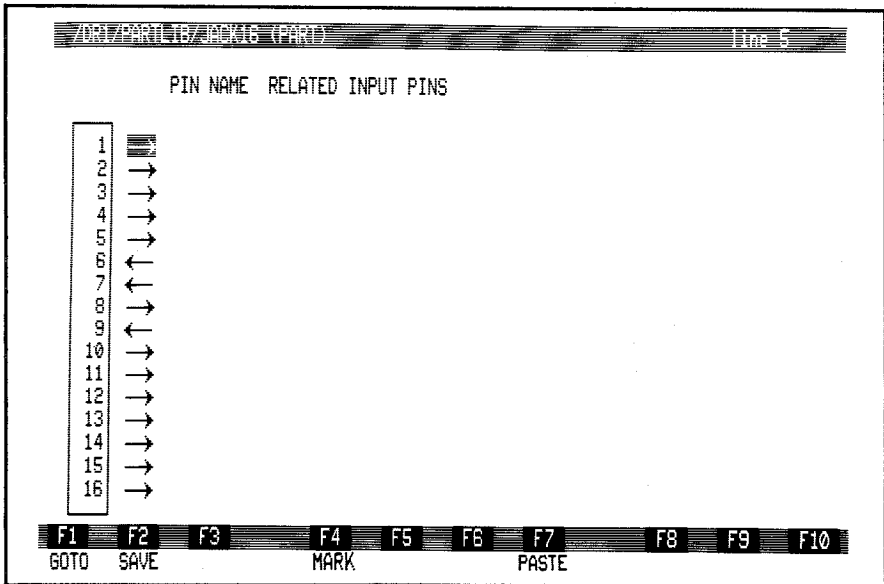
Information Window Off

Figure 5-9: SIP Part Description

NAME: 7420   DISK FREE: 628,224 BYTES
NO. PINS: 14
PACKAGE: DIP   WRITE PROTECT: YES

```
              →  |1  14|  PWR
              →  |2  13|  ←
          UNU    |3  12|  ←
              →  |4  11|  UNU
              →  |5  10|  ←
1,2,4,5       ←  |6   9|  ←
          GND    |7   8|  →        9,10,12,13
```

    PIN   RELATED INPUT PINS

```
 F1   F2   F3       F4   F5   F6   F7       F8   F9   F10
      SAVE
```

Information Window On

RELATED INPUT PINS PIN NAME   PIN NAME RELATED INPUT PINS

```
              →  |1  14|  PWR
              →  |2  13|  ←
          UNU    |3  12|  ←
              →  |4  11|  UNU
              →  |5  10|  ←
1,2,4,5       ←  |6   9|  ←
          GND    |7   8|  →        9,10,12,13
```

    PIN   RELATED INPUT PINS

```
 F1   F2   F3       F4   F5   F6   F7       F8   F9   F10
GOTO  SAVE           MARK           PASTE
```
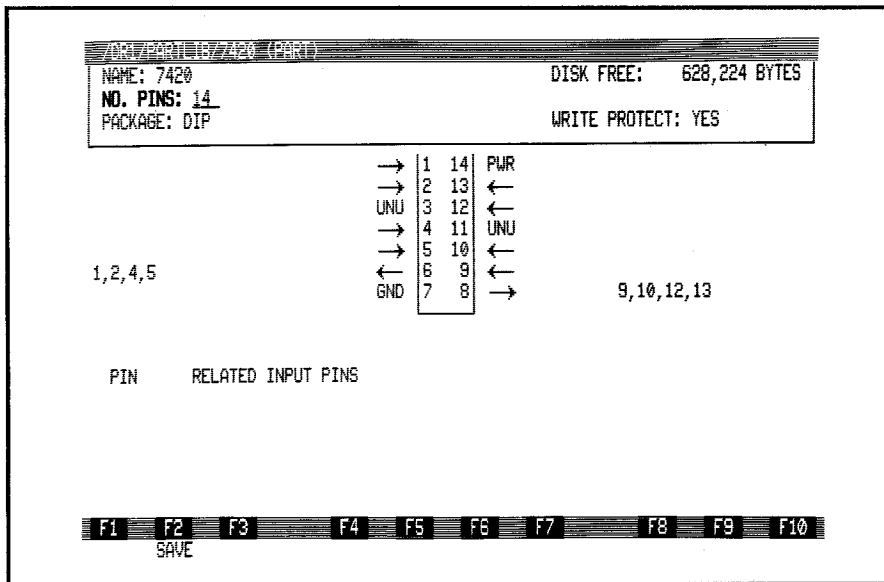
Information Window Off

Figure 5-10: DIP Part Description

Information Window off:

- Pin type field: Specifies the function of each IC pin according to the guidelines laid out in Figure 5-11. The fields are initially set to a default state. To change a pin field, move the cursor next to a pin number on the IC figure and use the Field Select key to change the field.

- PIN NAME: The name of the pin. This field is typically left blank and is only used for components that use names rather than numbers to identify pins. For example, a connector may label one row of pins "a1" through "a12", and the other row of pins "b1" through "b12".

  If a component uses pin names, a name must be specified for every pin. The 9100A/9105A will not recognize pin numbers for these components. Use the pin name in the node list, response files and with GFI.

- RELATED INPUT PINS: Related inputs are the pins that should be examined if an output pin is bad. These pins logically affect an output (or a bidirectional pin when it is acting as an output). When GFI finds a bad output pin, it uses this list to determine where to probe next. Related inputs are specified for each output, including status lines and bidirectional lines.

  As a default, the 9100A/9105A assumes that all the input pins and bidirectional pins are related inputs. It further assumes that they should be probed in ascending order, based on pin number. If these assumptions are correct for a particular output pin, this field can be left blank.

  However, for some pins this assumption is incorrect. For example, on a 74LS00, the list of related input pins for pin 3 should be limited to pins 1 and 2. If you wish to restrict the set of related input pins to some subset of the default list, or if you want to change the order in which the pins are probed, you should enter a list of related input pins. List the pins in the order that they should be examined (i.e., the highest priority pins first, and the lowest priority last). It is not necessary to list power and ground pins as related inputs; the 9100A/9105A assumes this.

If an output (such as a microprocessor pin or edge connector pin) has no related inputs, you should type a zero into this field. This will force GFI to immediately make an accusation involving the pin when it finds that it is bad.

To specify related inputs, move the cursor horizontally away from the pin field and type in the pin numbers of related inputs, separated by commas. If there are too many related inputs for one line, you should move the cursor below the IC figure to the PIN field and type the pin name or number. Then move the cursor to the RELATED INPUT PINS field and type in the list of related input pin numbers. The extra related pins appearing below the IC figure are treated as a continuation of the related pins listed at the side of the IC. If desired, all related input pins may be listed below the IC figure.

```
                    Pin Field          Pin Function
                 (In Part Description)   (from Data Book)

    ┌─────────────┐  1  ◄──────        Input

    │             │  2  ◄──────        Control

    │             │  3  ──────►        Output
    │ Typical IC  │
    │Outline in Part│ 4  ──────►        Status
    │ Description │
    │   Screen    │  5  ◄──────►       Bidirectional

    │             │  6   GND          Ground

    │             │  7   PWR          Power

    │             │  8   UNU          Not Connected

    │             │  9   UNU          No Function
    └─────────────┘
```
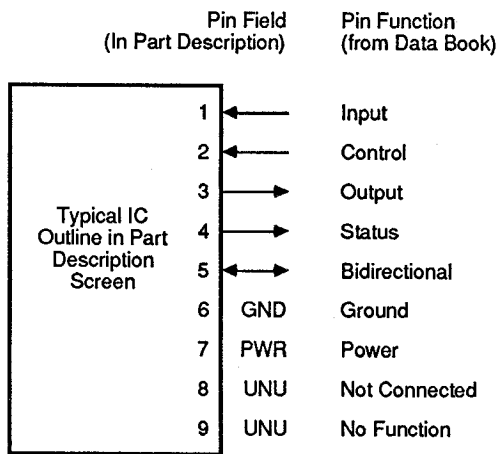
Figure 5-11: Specifying Pin Functions in a Part Description

2114 Example

The following figures show the block diagram, pin layout, and logic symbol for a 2114 RAM chip. Each memory location is addressed by lines A_0 through A_9, which are therefore inputs. Data is written to or read from an address on data lines I/O_1 through I/O_4, which are therefore bidirectional pins. The CS (chip select) and WE (write enable) control lines are also considered inputs.
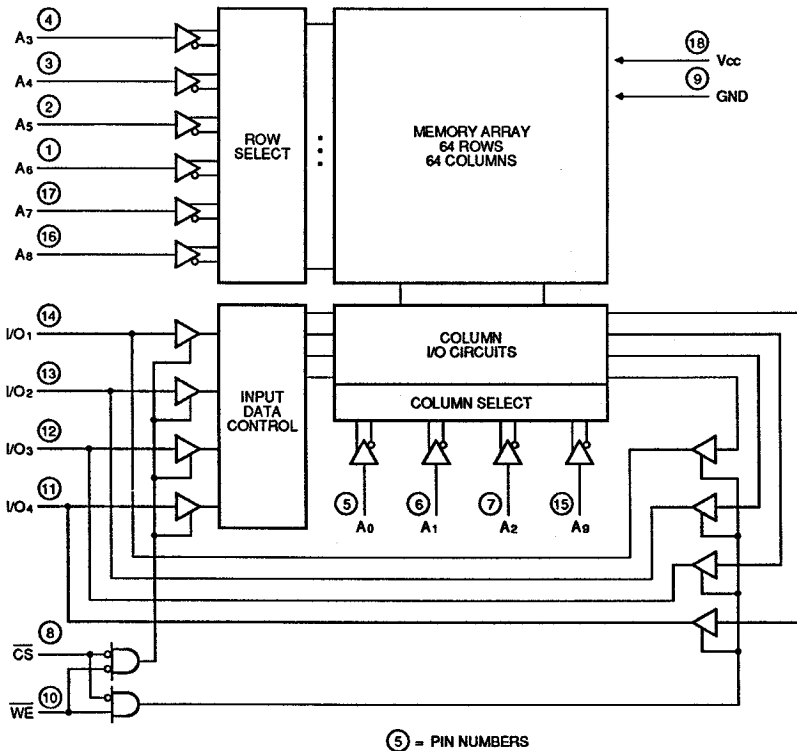
2114 Block Diagram

(This page is intentionally blank.)

A0
A1   I/O1
A2
A3
A4   I/O2
A5
A6   I/O3
A7
A8
A9   I/O4
WE   CS

2114 Logic Symbol

A6   1   18   Vcc
A5   2   17   A7
A4   3   16   A8
A3   4   15   A9
A0   5   2114A   14   I/O1
A1   6   13   I/O2
A2   7   12   I/O3
CS   8   11   I/O4
GND 9   10   WE
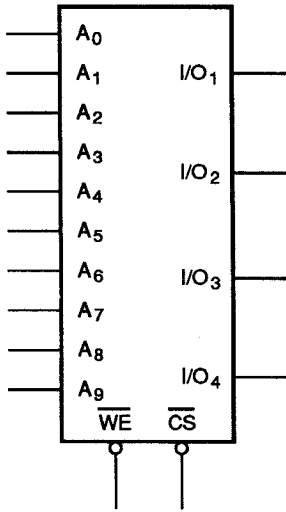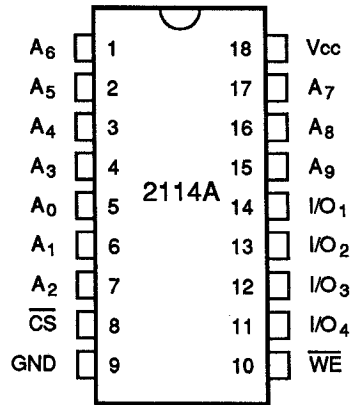
2114 Pin Configuration

Figure 5-12 shows what the 2114 part description should look like. Since bidirectional pins I/O1 through I/O4 sometimes act as outputs, related inputs should be entered for them.

For example, the related inputs for I/O4 (pin 11) are determined by first locating all inputs that could affect the data at pin 11. The previous block diagram shows these inputs to be A0 through A9, CS, and WE. Next, arrange the related inputs in the order you would check them if you found bad data on pin 11. CS and WE are the most important lines: if either is bad, these lines should be checked out regardless of the state of the address line.

Thus, the pins are listed in order: CS, WE, and then the address lines. In Figure 5-12, the related inputs were too long to fit on one line so the line was continued below the symbol of the IC.

```
/DR1/PARTLIB/2114 (PART)                        [MODIFIED]
NAME: 2114                          DISK FREE:    628,224 BYTES
NO. PINS: 18
PACKAGE: DIP                        WRITE PROTECT: NO

                    →  |1  18| PWR
                    →  |2  17| ←
                    →  |3  16| ←
                    →  |4  15| ←
                    →  |5  14| ↔        8,10,15,16,17,1,2,3
                    →  |6  13| ↔        8,10,15,16,17,1,2,3
                    →  |7  12| ↔        8,10,15,16,17,1,2,3
                    →  |8  11| ↔        8,10,15,16,17,1,2,3
                  GND  |9  10| ←


     PIN      RELATED INPUT PINS
     14       4,7,6,5
     13       4,7,6,5
     12       4,7,6,5
     11       4,7,6,5
 F1   F2   F3         F4   F5   F6   F7        F8   F9   F10
      SAVE
```

Information Window On

```
/DR1/PARTLIB/2114 (PART)                    [MODIFIED]  line 5
RELATED INPUT PINS  PIN NAME         PIN NAME  RELATED INPUT PINS


 _____          →  |1  18| PWR
                      →  |2  17| ←
                      →  |3  16| ←
                      →  |4  15| ←
                      →  |5  14| ↔        8,10,15,16,17,1,2,3
                      →  |6  13| ↔        8,10,15,16,17,1,2,3
                      →  |7  12| ↔        8,10,15,16,17,1,2,3
                      →  |8  11| ↔        8,10,15,16,17,1,2,3
                    GND  |9  10| ←


     PIN      RELATED INPUT PINS
     14       4,7,6,5
     13       4,7,6,5
     12       4,7,6,5
     11       4,7,6,5
 F1   F2   F3         F4   F5   F6   F7        F8   F9   F10
 GOTO SAVE            MARK           PASTE
```

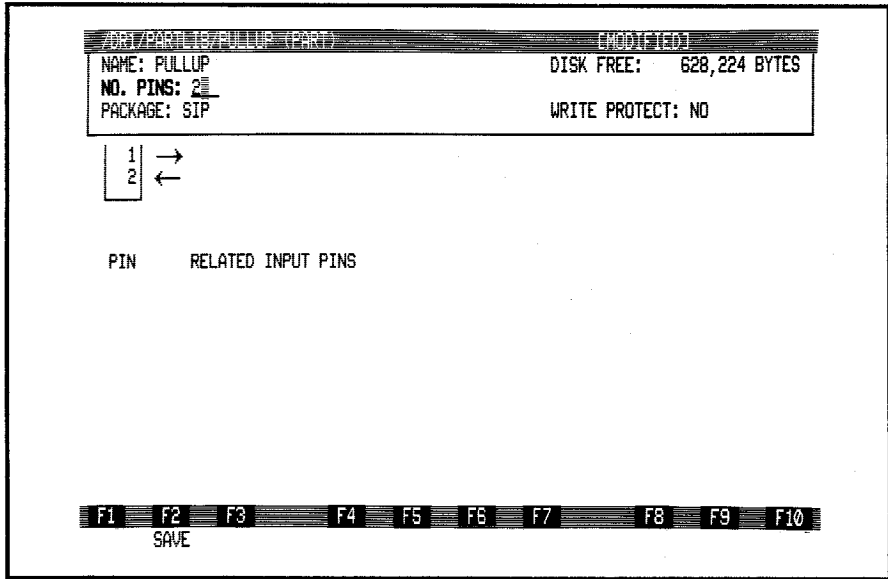Information Window Off

Figure 5-12: 2114 Part Description

4034 Example

The following figures show some typical applications, the pin layout, and logic symbols for a 4034. The 4034 is an eight-stage bidirectional, parallel/serial, input/output, bus register. If necessary, refer to a CMOS data book for details on its pin functions.
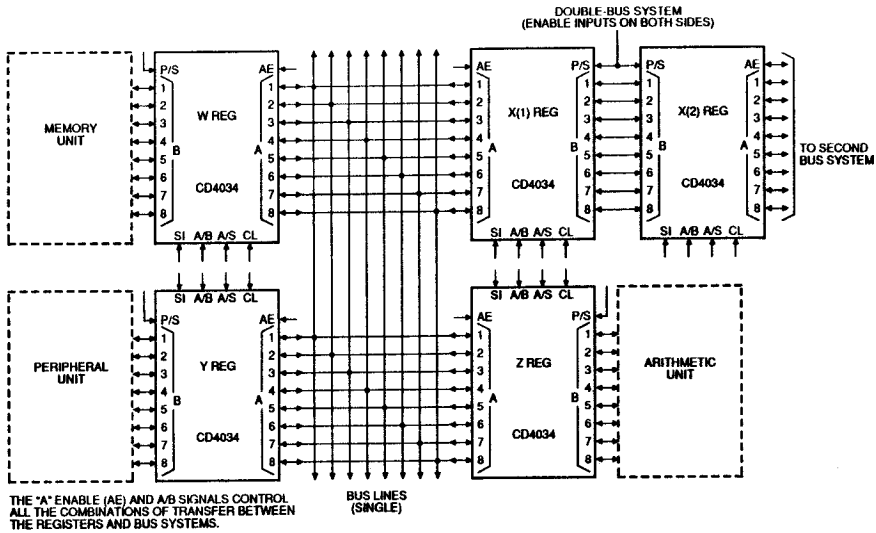
Single- and Double-Bus Application
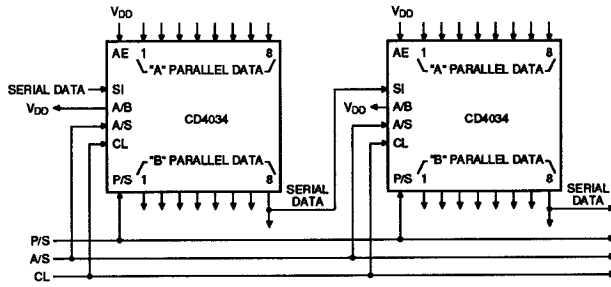
16-bit Register:
- parallel in/parallel out.
- parallel in/serial out.
- serial in/parallel out.
- serial in/serial out.
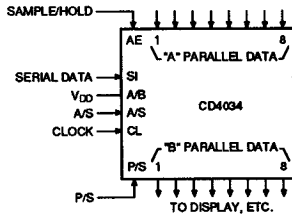
Sample and Hold Register:
- serial-parallel in/ parallel out.
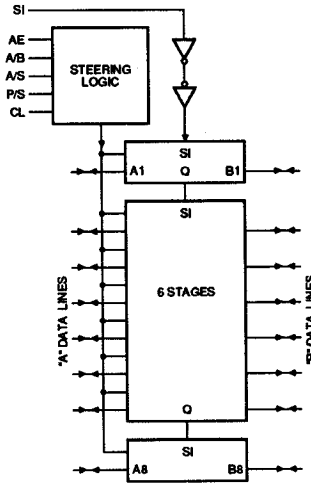
Functional Diagram
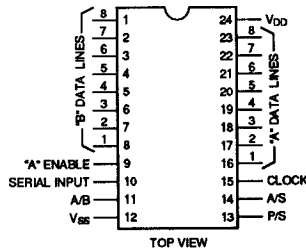
Pin Layout

The 4034 part description should be designed and entered like the 2114 part description of the previous section. Figure 5-13 shows what the 4034 part description should look like.

```
/DR1/PARTLIB/4034 (PART)                    [MODIFIED]
NAME: 4034                          DISK FREE:    628,224 BYTES
NO. PINS: 24
PACKAGE: DIP                        WRITE PROTECT: NO

                        ↔ │ 1  24│ PWR
                        ↔ │ 2  23│ ↔
                        ↔ │ 3  22│ ↔
                        ↔ │ 4  21│ ↔
                        ↔ │ 5  20│ ↔
                        ↔ │ 6  19│ ↔
                        ↔ │ 7  18│ ↔
                        ↔ │ 8  17│ ↔
                        → │ 9  16│ ↔
                        → │10  15│ ←
                        → │11  14│ ←
                      GND │12  13│ ←

       PIN     RELATED INPUT PINS
       16      13,15,10,14,11,9,8,7,6,5,4,3,2,1,9,16,17,18,19,20,21,22,23
  F1     F2     F3        F4    F5    F6    F7        F8    F9    F10
         SAVE
```

Information Window On

```
/DR1/PARTLIB/4034 (PART)                    [MODIFIED]  line 5

RELATED INPUT PINS  PIN NAME              PIN NAME  RELATED INPUT PINS

                        ↔ │ 1  24│ PWR
                        ↔ │ 2  23│ ↔
                        ↔ │ 3  22│ ↔
                        ↔ │ 4  21│ ↔
                        ↔ │ 5  20│ ↔
                        ↔ │ 6  19│ ↔
                        ↔ │ 7  18│ ↔
                        ↔ │ 8  17│ ↔
                        → │ 9  16│ ↔
                        → │10  15│ ←
                        → │11  14│ ←
                      GND │12  13│ ←

       PIN     RELATED INPUT PINS
       16      13,15,10,14,11,9,8,7,6,5,4,3,2,1,9,16,17,18,19,20,21,22,23
  F1     F2     F3        F4    F5    F6    F7        F8    F9    F10
  GOTO   SAVE             MARK              PASTE
```

Information Window Off

Figure 5-13: 4034 Part Description

(This page is intentionally blank.)

Pull-Up Resistor Example

The figure below shows a pull-up resistor whose leads have been assigned pin numbers. Pin 2 is connected to +5 volts so that pin 1 can provide a pull-up voltage at the output of a semiconductor component.
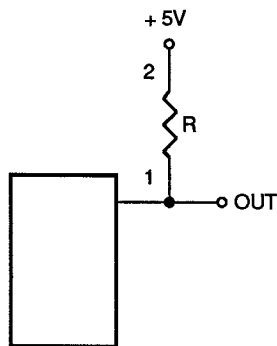
Pull-Up Resistor (R) at Component Output

Figure 5-14 is the part description of the above resistor. Resistors can be described in the same way as ICs: they are two-pin SIP components. Each pin can be input, output, or bidirectional, depending on current flow. Since pin numbers must be assigned arbitrarily, the test or troubleshooting operator should be provided with information describing the numbering scheme.

```
NAME: PULLUP                          DISK FREE:     628,224 BYTES
NO. PINS: 2_
PACKAGE: SIP                          WRITE PROTECT: NO
```

```
1 | →
2 | ←
```

```
PIN     RELATED INPUT PINS
```

```
F1    F2    F3    F4    F5    F6    F7    F8    F9    F10
      SAVE
```

Information Window On

```
        PIN NAME   RELATED INPUT PINS
```

```
1 | →
2 | ≣
```

```
PIN     RELATED INPUT PINS
```

```
F1    F2    F3    F4    F5    F6    F7    F8    F9    F10
GOTO  SAVE          MARK          PASTE
```
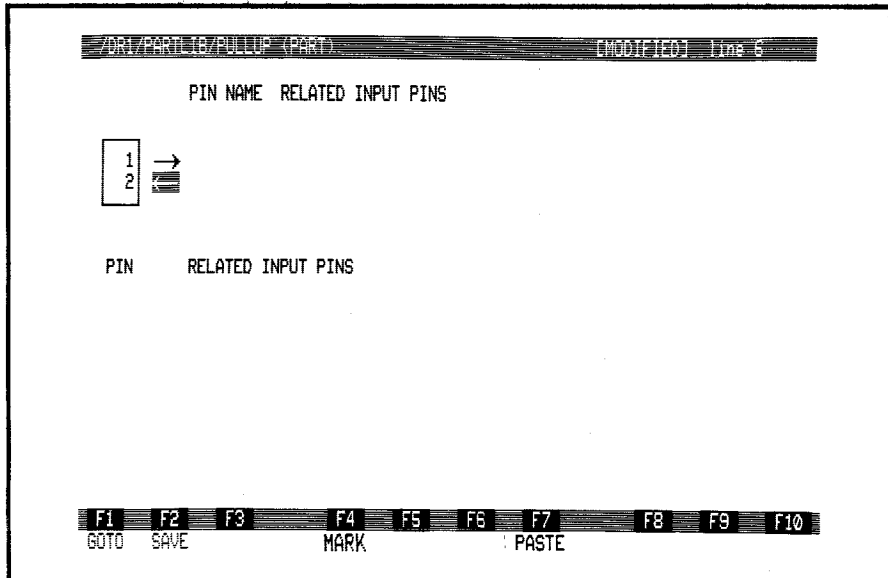
Information Window Off

Figure 5-14: Pull-Up Resistor Part Description

Entering a Part Description   5.5.3.

The part library (PARTLIB) contains component descriptions
that can be accessed by any UUT. A UUT can also contain part
descriptions. If REFLIST specifies a part that is not in the UUT
or library, that part must be added to the UUT or PARTLIB.
For example, to enter a 7420 description:

1. Consult a TTL data book to see what each pin does.
   A 7420 is a dual, four-input NAND gate. You need
   to know if each pin is input, output, bidirectional,
   ground, power, or unused. You also need to know
   the related inputs for each output or bidirectional pin.

2. Press the Edit key and type:

   /hdr/partlib/7420

3. Press Return, select PART as the TYPE field, and
   press Return again to view the description, which
   should appear with the information window on.

   If this part description has already been created, the
   information window won't be displayed unless you
   press the Info key. Pressing the Info key a second
   time turns the information window off.

4. Type the number of pins (14) on the IC and specify
   the package type (DIP) at the information window;
   then turn the information window off by pressing the
   Info key.

5. Type the related inputs for each output, and use the
   Field Select key to specify each pin's function
   according to the information obtained in step 1.
   Figure 5-15 shows the 7420 part description in its
   final form.

6. Press Quit and use Field Select to specify whether or
   not to save your changes.

74LS20
DUAL 4-INPUT
POSITIVE NAND GATES

Vcc	2D	2C	NC	2B	2A	2Y
14	13	12	11	10	9	8

1	2	3	4	5	6	7
1A	1B	NC	1C	1D	1Y	GND

POSITIVE LOGIC: $Y = \overline{ABCD}$

```
/HDR/PARTLIB/7420 (PART)                                    Line 5

RELATED INPUT PINS  PIN NAME              PIN NAME  RELATED INPUT PINS

                              → | 1   14 | PWR
                              → | 2   13 | ←
                            UNU | 3   12 | ←
                              → | 4   11 | UNU
                              → | 5   10 | ←
                              ← | 6    9 | ←
  1,2,4,5                   GND | 7    8 | →          9,10,12,13


    PIN    RELATED INPUT PINS




  F1    F2    F3         F4    F5    F6    F7         F8    F9    F10
 GOTO  SAVE              MARK              PASTE
```

After Edits

Figure 5-15: 7420 Part Description

Reference Designator List   5.5.4.

Figure 5-16 shows an example reference designator list (REFLIST), which pairs the names (reference designators) of all UUT components with part descriptions from the part library and with the device needed to test each component.

Information Window on:

* NAME: The name REFLIST. This field cannot be edited.

* DESCRIPTION: An optional one-line description of REFLIST.

* DISK FREE, SIZE: The amount of disk space that is still available and the size of the reference designator list. These fields cannot be edited.

* WRITE PROTECT: The write-protection status of the file. Use the Field Select key to set this field to YES to specify write protection for the file. If the file is write protected, the editor prompts you when the file is saved to ensure that changes are intentional. If the file is not write protected, you will not be prompted. A change in write-protection status does not become effective until after you save your current edits.

Information Window off:

* REF: Enter a reference designator (such as U4 or J8) for the UUT component referred to. A reference designator can be from one to six characters long. It may include only alphanumeric characters, underscores "_", and periods ".". Also it must begin with an alphanumeric character.

* PART: Enter the name of the part (such as 7400) that corresponds to the reference designator. The part library will need to contain a description for the part with this name.

* TESTING DEVICE: The device (probe or I/O module) to be used to test the component during GFI. Press the Field Select key to change this field.

```
/DR1/JTK81/REFLIST (REF)                          [MODIFIED]
NAME: REFLIST                           DISK FREE:    397,696 BYTES
DESCRIPTION: _▌                         SIZE:             258 BYTES
                                        WRITE PROTECT: NO

              j1          jack16        PROBE
              u24         8187          I/O MODULE
              u34         8228          I/O MODULE
              u22         74139         PROBE
              u11         2114          I/O MODULE
              u13         2114          I/O MODULE
              u30         8255          I/O MODULE
              u25         8080          PROBE
              r1          resistor      PROBE
              kmem        key           PROBE




  F1     F2     F3        F4    F5    F6    F7         F8     F9     F10
         SAVE
```

Information Window On

```
/DR1/JTK81/REFLIST (REF)                          [MODIFIED]  line 5

                                        TESTING
              REF         PART          DEVICE

              j1          jack16        PROBE
              u24         8187          I/O MODULE
              u34         8228          I/O MODULE
              u22         74139         PROBE
              u11         2114          I/O MODULE
              u13         2114          I/O MODULE
              u30         8255          I/O MODULE
              u25         8080          PROBE
              r1          resistor      PROBE
              kmem        key           PROBE




  F1     F2     F3        F4    F5    F6    F7         F8     F9     F10
 GOTO   SAVE              MARK              PASTE
```

Information Window Off

Figure 5-16 Reference Designator List (REFLIST)

SIP components must be tested with the probe, but DIP components can be tested with either the probe or an I/O module. If the I/O module is used, GFI will test all the pins on the component whenever the component is clipped.

Editing the Reference Designator List   5.5.5.

The reference designator list (REFLIST) contains a list of reference designators, the component each designator identifies, and the device needed to test each component of a UUT. For example, assume that U99 designates a TTL 74LS20 IC. To add U99 to REFLIST for the UUT *abc:*

1. Press the Edit key and type:

    /hdr/abc/reflist

2. Press Return, select REF as the TYPE field, and press Return again. REFLIST should appear as shown in Figure 5-17 "Before Edits."

3. Move the cursor to the bottom (blank) line and type U99 into the REF field.

4. Press Return and type 7420 into the PART field.

5. Press Return and use the Field Select key to select the TESTING DEVICE as the probe or I/O module.

6. Repeat steps 3 through 5 for subsequent entries; when done, press Quit and use Field Select to specify whether or not to save your changes. Figure 5-17 "After Edits" shows the results of steps 3 through 5 on REFLIST.

Steps 3 through 5 identify U99 as a 7420 PART. If the UUT and PARTLIB (the part library) does not contain a 7420 description, the UUT or library must be updated by entering the necessary part description.

| | | TESTING |
REF	PART	DEVICE
U23	74245	I/O MODULE
U16	74373	I/O MODULE
U2	74373	I/O MODULE
U22	74373	I/O MODULE
U15	82288	I/O MODULE
J5	conn1	I/O MODULE
U14	80286	I/O MODULE
J1C	conn1	PROBE
SW3	switch1	PROBE
J1A	conn1	PROBE
SW2	switch1	PROBE
U27	27256	I/O MODULE
U3	74245	PROBE

F1	F2	F3	F4	F5	F6	F7	F8	F9	F10
GOTO	SAVE		MARK			PASTE			

Before Edits

| | | TESTING |
REF	PART	DEVICE
U23	74245	I/O MODULE
U16	74373	I/O MODULE
U2	74373	I/O MODULE
U22	74373	I/O MODULE
U15	82288	I/O MODULE
J5	conn1	I/O MODULE
U14	80286	I/O MODULE
J1C	conn1	PROBE
SW3	switch1	PROBE
J1A	conn1	PROBE
SW2	switch1	PROBE
U27	27256	I/O MODULE
U3	74245	PROBE
U99	7420	PROBE

F1	F2	F3	F4	F5	F6	F7	F8	F9	F10
GOTO	SAVE		MARK			PASTE			

After Edits

Figure 5-17: Editing the Reference Designator List

Node List 5.5.6.

Figure 5-18 shows an example node list, which describes all UUT interconnections.

Information Window on:

- NAME: The name NODELIST appears in this field, which cannot be edited.

- DISK FREE, SIZE: The amount of disk space that is still available and the size of the node list. These fields cannot be edited.

- WRITE PROTECT: The write-protection status of the file. Use the Field Select key to set this field to YES to specify write protection for the file. If the file is write protected, the editor prompts you when the file is saved to ensure that changes are intentional. If the file is not write protected, you will not be prompted. A change in write-protection status does not become effective until after you save your current edits.

Information Window off:

- Each node field is one or more text lines, each listing the pins connected together to form one node. Pin names are separated by spaces or tab characters. Comments can be inserted in node fields. As in TL/1 programs, precede a comment with a "!" character. When you finish specifying a node, press Return. When you do so, the editor checks the line for errors.

```
NAME: NODELIST                          DISK FREE:    628,224 BYTES
                                        SIZE:           1,017 BYTES
                                        WRITE PROTECT: YES

  u25-1  u22-14                                         !A10

  u25-35 u11-15 u13-15                                  !A9
  u25-34 u11-16 u13-16                                  !A8
  u25-33 u11-17 u13-17                                  !A7
  u25-32 u11-1 u13-1                                    !A6
  u25-31  u11-2   u13-2                                 !A5
  u25-30 u11-3 u13-3                                    !A4
  u25-29 u11-4 u13-4                                    !A3
  u25-27 u24-18                                         !A2
  u25-26 u24-16                                         !A1
  u25-25 u24-14                                         !A0

  *masters
  u25-35 u25-34 u25-33 u25-32
  u25-31 u25-30 u25-29
```

```
 F1   F2   F3      F4   F5   F6   F7      F8   F9   F10
      SAVE
```

Information Window On

```
  u25-38 u30-6                                          !A13
  u25-37 u22-15                                         !A12
  u25-40 u22-13                                         !A11
  u25-1  u22-14                                         !A10

  u25-35 u11-15 u13-15                                  !A9
  u25-34 u11-16 u13-16                                  !A8
  u25-33 u11-17 u13-17                                  !A7
  u25-32 u11-1 u13-1                                    !A6
  u25-31  u11-2   u13-2                                 !A5
  u25-30 u11-3 u13-3                                    !A4
  u25-29 u11-4 u13-4                                    !A3
  u25-27 u24-18                                         !A2
  u25-26 u24-16                                         !A1
  u25-25 u24-14                                         !A0

  *masters
  u25-35 u25-34 u25-33 u25-32
  u25-31 u25-30 u25-29
```

```
 F1   F2   F3      F4   F5   F6   F7      F8   F9   F10
 GOTO SAVE         MARK           PASTE   REPL SEARCH CHECK
```

Information Window Off

Figure 5-18: Node List (NODELIST)

To use more than one line for a node, you can:

- Type a backslash (\\) character to end the first line, and then type on the next line.

- Continue typing at the end of the first line; the editor inserts a continuation marker (>) at the end of the first line moves the cursor to the next line and inserts another marker (<).

CHECK Command

The editor's CHECK command looks for errors in the node list. CHECK indicates if a pin appears in more than one node.

Naming Bus-Master (*master) Pins

The screen's last lines in Figure 5-18 show a *masters (star masters) entry. The entry lists all pins in the node list that are "bus-masters." A bus-master is a pin which can send data to every other pin on the same node or receive data from every other pin on the same node.

Figure 5-19 shows why U3-14 is a bus-master. U3-14, U12-1, U13-1, U10-12, and U56-12 form a node. Information can flow through the pins as indicated by the arrows. U3-14 is the bus-master (the only one on the node) because it communicates with all the other pins on the node.

Why is *masters necessary? Sometimes components that are connected together do not communicate: the most common examples are bus components, as in Figure 5-19. GFI determines data flow from the node list and assumes that data can be sent from a pin to any other pin on that same node. In Figure 5-19 the assumption is incorrect because the RAM, ROM, and I/O communicate only with U3. The *master entry allows GFI to decide which pins actually send data to other pins.

Micro-Processor

14

1

1

12

12

RAM 1

RAM 2

ROM

I/O

U3

U12

U13

U10

U56

Arrows indicate possible signal directions; RAM 1,
RAM 2, and I/O only send signals to the microprocessor
and not to each other. U3-14 is the bus-master (*master) pin.
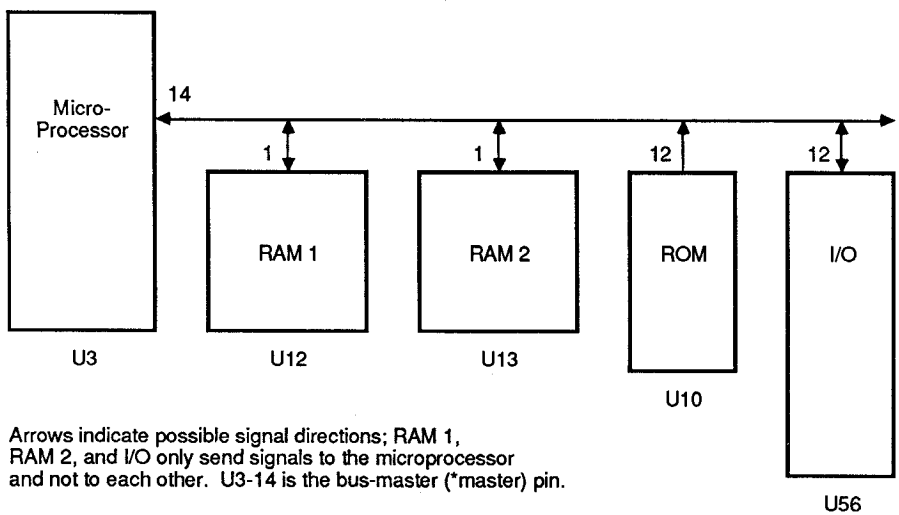
Figure 5-19: Bus-Master (*master) Example

The *masters entry is optional; it is better to make the entry after compiling the database once without it. The entry is helpful in cases of nodes that:

- Are formed by three or more pins.

- Have two or more signal sources.

Editing the Node List   5.5.7.

The node list (NODELIST)contains a list of nodes on the UUT. A node is formed by the connection of two or more pins. For example, in Figure 5-20, node A is formed by U23-3, U14-49, and connector pins J1a-25 and J5-49. To enter node A into the node list for UUT *abc:*

1. Press the Edit key and type:

    `/hdr/abc/nodelist`

2. Press Return, select NODE as the TYPE field, and press Return again.

3. Move the cursor to the bottom (blank) line, and type the pins forming node A. Press the Return key at the end of each line. The line should read:

    `u23-3   j1a-25   u14-49   j5-49`

4. Repeat step 3 until all nodes have been entered.

5. Use the CHECK command to ensure that a pin does not appear in more than one node.

6. Press Quit, and use Field Select to specify whether or not to save any changes you have made.
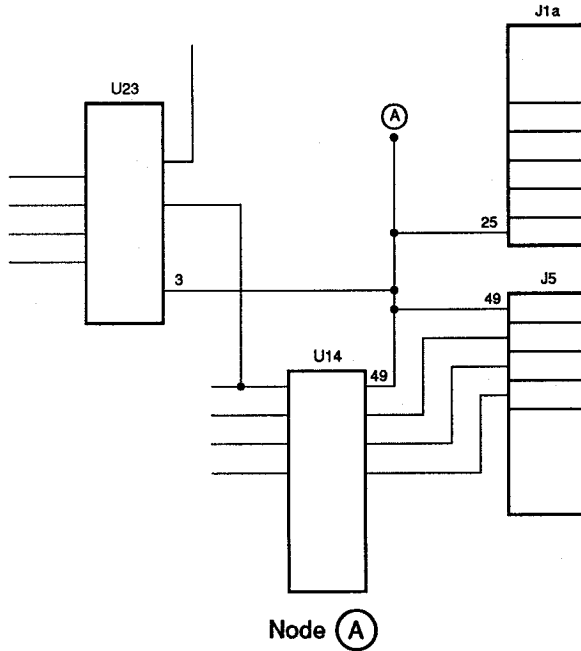
```
u23-2 j1a-26 u14-51 j5-51

u23-6 j1a-22 u14-43 j5-43
u23-8 j1a-20 sw2-8 u14-39 j5-39
u23-7 j1a-21 u14-41 j5-41
u23-9 j1a-19 sw2-9 sw2-11 u14-37 j5-37

u23-5 j1a-23 sw2-7 u14-45 j5-45


u3-11 u27-11
u3-12 u27-12
u3-13 u27-13
u3-14 u27-15
u3-15 u27-16
u3-16 u27-17
u3-17 u27-18
u3-18 u27-19

u23-3 j1a-25 u14-49 j5-49
```

F1	F2	F3	F4	F5	F6	F7	F8	F9	F10
GOTO	SAVE		MARK			PASTE	REPL	SEARCH	CHECK

The Node A, Entered into the Node List, on the Bottom Line

Figure 5-20: Editing the Node List

Stimulus programs are TL/1 programs used by GFI to exercise UUT nodes in such a way that the responses of nodes can be analyzed and compared to responses of nodes on a known-good UUT. The responses include CRC signatures, asynchronous level histories, clocked level histories, and either a signal frequency or the number of signal transitions.

Response data is gathered during program execution in an *arm .
. . readout* block. The stimulus should contain a sequence of TL/1 commands that configures and synchronizes 9100A/9105A hardware for response gathering.

Figure 5-21 shows an example stimulus program; note the TL/1 commands used to configure the response-gathering hardware.

Information Window on:

• NAME: The name of the stimulus program. This field cannot be edited.

• DESCRIPTION: An optional one-line program description.

• DISK FREE, SIZE: The amount of disk space that is still available and the size of the current program. These fields cannot be edited.

• WRITE PROTECT: The write-protection status of the file. Use the Field Select key to set this field to YES to specify write protection for the file. If the file is write protected, the editor prompts you when the file is saved to ensure that changes are intentional. If the file is not write protected, you will not be prompted. A change in write-protection status does not become effective until after you save your current edits.

```
! This program is a simplified example of a GFI stimulus program.
! The stimulus program is designed to stimulate a node, while
! the node's response is captured with the probe or I/O module.
!
! This program has two main parts.  First, the response-gathering
! hardware on the testing device is configured with the TL/1
reset,
! sync, threshold, and counter commands.  Then the response-
! gathering hardware is activated, and the stimulus is applied.
!
! This program shows the TL/1 commands that are used to configure
! the probe or I/O module to collect responses using external
sync.
!
program ext_sync

devlist = gfi device                          ! get the device from GFI
reset device devlist                          ! reset device to a known
                                              ! state
threshold device devlist, level "ttl"         ! set threshold levels
counter device devlist, mode "transition"     ! set counter mode
sync device devlist, mode "ext"               ! sync device to external
edge device devlist, start "+", stop "+", clock "+"
connect device devlist, start "u3-1", stop "U7-8", clock "U4-8"
enable device devlist, mode "always"
arm device devlist                            ! start the response
                                              ! capture
    rampdata addr $F0000, data 0, mask $F     ! apply the stimulus
    rampdata addr $F0000, data 0, mask $F0
    rampdata addr $F0000, data 0, mask $F00
    rampdata addr $F0000, data 0, mask $F000
    ! Check that signatures are complete.  Raise a fault if they
    ! aren't.
    status = checkstatus device devlist
    if status <> $F then
        if (status and 4) = 0 then
            reason = "no valid start seen"
        else if (status and 2) = 0 then
            reason = "no valid enable seen"
        else if (status and 1) = 0 then
            reason = "no valid clock seen"
        else if (status and 8) = 0 then
            reason = "no valid stop seen"
        end if
         fault signatures_incomplete because reason
    end if
readout device devlist                        ! terminate the response
                                              ! capture

end ext_sync
```

Figure 5-21: Stimulus Program (ext_sync)

Information Window off:

A stimulus program is simply a program written for the specific purpose of providing a stimulus to exercise a UUT node. Editing a stimulus program is identical to editing any other program. When writing or changing a line, the editor checks the line for correct TL/1 syntax before allowing the cursor to move off the line. The CHECK command checks for syntax errors which cannot be detected by the line check. The debugger can be used to check the program's logical operation.

Stimulus programs written for GFI should not use the *assign, clip,* or *probe* commands. GFI automatically prompts the operator with the name of the reference designator or pin being measured by the I/O module or probe. The stimulus does not need to store or check the resulting response data, since GFI makes troubleshooting decisions based on data stored in response files. The stimulus program must do the necessary UUT initialization and the setup and control of measurement hardware.

A stimulus program has two main parts. First, the measurement hardware must be configured:

•   For a program using pod sync, *threshold* and *counter* commands should be used.

•   For a program using external sync, the *sync, threshold, counter, enable, edge,* and *connect* commands should be used.

After measurement hardware is configured, it should be activated and the stimulus applied:

•   The *arm* and *readout* commands should be used.

•   For a program using external sync, the *checkstatus* command should be used before *readout,* to ensure that signatures are complete.

- The stimulus is typically a sequence of *read* and *write* commands.

Stimulus programs should satisfy two very important criteria:

- The program must be independent, initializing the UUT as required. This is because GFI can begin backtracing at any node, and the state of the UUT, prior to running the stimulus, is unknown.
- During stimulus execution, only one pin should drive a node: that is, during the period between the *arm* and *readout* commands, one and only one pin should be a node signal source. There are two reasons for this that are explained in the following examples:

Example 1: Node A (below) is bidirectional: either U1-1 or U3-1 can be signal sources. To exercise the node, two stimuli are needed, one naming U1-1 as a source and the other naming U3-1 as a source. The reason is, that if either pin is found to be bad, GFI needs to know whether the pin is an output or an input. If the pin is an output, GFI recommends probing that IC; if the pin is an input, GFI recommends probing the source of that input.
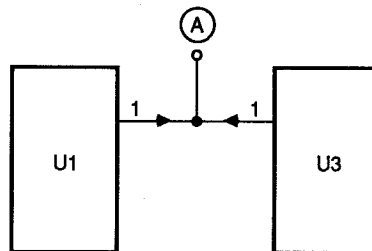
(A) Is a bidirectional node

Example 2: U10-4 in Figure 5-22 receives input from one data line at a time. The source pin depends on the IC addressed, according to Table 1. If a stimulus program were to be written (to test U10-4) that reads data from locations 8000 through 88FF, each pin (U1-4, U2-4, and U3-4) would function as an input to U10-4 at some time in the stimulus program. For this reason, if U10-4 was found to be bad, GFI would not be able to identify a unique source to backtrace toward. The solution is to write three stimulus programs as shown in Table 2.

Writing Stimulus Programs   5.5.9.

Stimulus programs are TL/1 programs that exercise UUT nodes in such a way that their responses can be characterized by a signature, a level, a count, or a frequency. Figure 5-23 shows *pod_sync,* an example stimulus program.To edit the *pod_sync* program:

1.   Press the Edit key and type:

       /hdr/abc/pod_sync

2.   Press Return, select PROGRAM as the TYPE field, and press Return again. The *pod_sync* program listing should appear on the screen, where it can be edited.

Typically when writing a stimulus program, you:

1.   Study the UUT logic to decide what commands will exercise the node(s) to be tested.

2.   Enter commands from the operator's keypad and check the node's activity with the probe.

3.   Repeat step 2 until you have accumulated a sequence of commands that exercises the node(s) thoroughly. All inputs associated with the node should be in all possible states during the test.
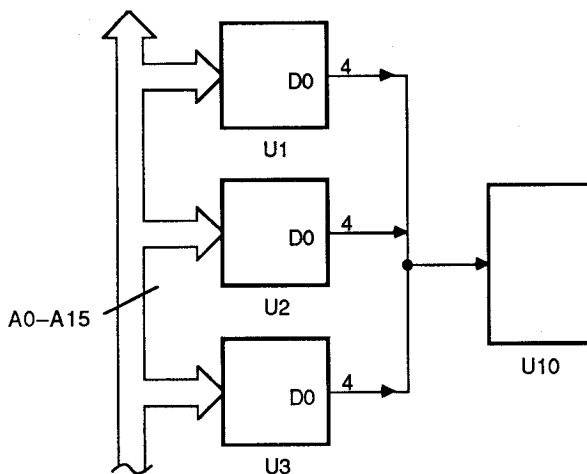
Hexadecimal value of address A0–A15	IC addressed	U10-4 gets input from
8000–83FF	U1	U1-4
8400–87FF	U2	U2-4
8800–88FF	U3	U3-4

Table 1: Multiple node signal sources

Hexadecimal value of address A0–A15	Example stimulus name	Node signal name
8000–83FF	stim_one	U1-4
8400–87FF	stim_two	U2-4
8800–88FF	stim_three	U3-4

Table 2: One node signal source per stimulus

Figure 5-22: Multiple Signal Sources for One Node

```
! This program is a simplified example of a GFI stimulus program.
! The stimulus program is designed to stimulate a node, while
! the node's response is captured with the probe or I/O module.
!
! This program has two main parts.  First, the response-gathering
! hardware on the testing device is configured with the TL/1 reset,
! sync, threshold, and counter commands.  Then the response-
! gathering hardware is activated, and the stimulus is applied.
!
! This program shows the TL/1 commands that are used to configure
! the probe or I/O module to collect responses using external sync.
!
program pod_sync

devlist = gfi device                          ! get the testing device
                                              ! from GFI

reset device devlist                          ! reset device to a known
                                              ! state
sync device devlist, mode "pod"               ! sync the device to pod
                                              ! ADDR
sync device "/pod", mode "addr"
threshold device devlist, level "ttl"         ! specify TTL logic
                                              ! levels
counter device devlist, mode "transition"     ! select the counter mode

arm device devlist                            ! start the response
                                              ! capture
    rampdata addr $F0000, data 0, mask $F     ! apply the stimulus
    rampdata addr $F0000, data 0, mask $F0
    rampdata addr $F0000, data 0, mask $F00
    rampdata addr $F0000, data 0, mask $F000
readout device devlist                        ! terminate the response
                                              ! capture

end pod_sync
```

Figure 5-23: Stimulus Program (pod_sync)

4.   Consult the *TL/1 Reference Manual* to convert the keypad commands into TL/1 statements. Include statements to configure the response-gathering hardware as necessary.

5.   Construct the program from TL/1 statements.

Once you write a stimulus program, you must verify that it works as expected. The debugger can help you in this process; see Section 4 "Debugger" for details.

When you write a stimulus program, its responses must be stored in a stimulus program response file using the same name as the program: in this case, *pod_sync*. When the *pod_sync* program is executed, GFI measures responses at the node and compares them to those stored in the response file named *pod_sync*.

Figure 5-24 shows a stimulus program response file named *addr_out,* which is a set of responses generated at various nodes by the stimulus program named *addr_out*. GFI will use the response file to link stimulus program *addr_out* with the nodes that the stimulus exercises. A response file should contain responses measured at nodes on a known-good UUT.

For example, if GFI tests node U27-11, GFI would run all stimulus programs that exercise U27-11. Responses generated at U27-11 by stimulus program *addr_out* are compared to the responses stored in the response file *addr_out*. GFI uses the comparison to decide whether U27-11 is good or bad.

Stimulus program *addr_out,* like most others, exercises several nodes. Each node is identified by the pin that is that node's signal source. When a stimulus program exercises a node, only one pin can be specified as a signal source for that node.

Information Window on:

- STIMULUS PROGRAM NAME: The name of the stimulus program response file. This field cannot be edited and must match the corresponding stimulus program name exactly.

- DESCRIPTION: An optional one-line description.

- DISK FREE, SIZE: The amount of disk space that is still available and the size of the current stimulus program response file. These fields cannot be edited.

```
STIMULUS PROGRAM NAME: ADDR_OUT                DISK FREE:  8,417,024 BYTES
DESCRIPTION:                                   SIZE:            204 BYTES
                                               WRITE PROTECT: YES

u27-11          I/O MODULE    3C3F    1 0    1 0    TRANS    32
u27-12          I/O MODULE    E735                  TRANS
u27-13          I/O MODULE    8DB2                  TRANS
u27-15          I/O MODULE    8CC4                  TRANS
u27-16          I/O MODULE    3479                  TRANS
u27-17          I/O MODULE    6691                  TRANS
u27-18          I/O MODULE    2D9E                  TRANS
u27-19          I/O MODULE    12E5                  TRANS
```

F1	F2	F3	F4	F5	F6	F7	F8	F9	F10
	SAVE								

Information Window On

```
                         ------------------- Response Data -------------------
    Node          Learned              Async   Clk   Counter
  Signal Src       With        SIG     LVL     LVL   Mode      Counter Range

  u27-11           I/O MODULE   3C3F    1 0    1 0    TRANS    32
  u27-12           I/O MODULE   E735                  TRANS
  u27-13           I/O MODULE   8DB2                  TRANS
  u27-15           I/O MODULE   8CC4                  TRANS
  u27-16           I/O MODULE   3479                  TRANS
  u27-17           I/O MODULE   6691                  TRANS
  u27-18           I/O MODULE   2D9E                  TRANS
  u27-19           I/O MODULE   12E5                  TRANS
```

F1	F2	F3	F4	F5	F6	F7	F8	F9	F10
GOTO	SAVE	LEARN		SELECT	DELETE	INSERT	MORE		FAULT

Information Window Off

Figure 5-24: Stimulus Program Response File (addr_out)

- WRITE PROTECT: The write-protection status of the file. Use the Field Select key to set this field to YES to specify write protection for the file. If the file is write protected, the editor prompts you when the file is saved to ensure that changes are intentional. If the file is not write protected, you will not be prompted. A change in write-protection status does not become effective until after you save your current edits.

Information Window off:

- Node Signal Src: Identifies a node by its signal source pin. The node is exercised by the stimulus program named in the information window. It is essential that while the stimulus program is run, signal activity in the node originates only from the pin specified in this field.

- Learned With: Identifies the device (probe or I/O module) that the response data was learned with. This field cannot be edited and instead is filled in by the 9100A during LEARN.

- Response Data: Characterizes the node's response (described below).

The Response Data fields characterize how nodes on a known-good UUT responded to the stimulus program named in the information window. When GFI tests a node, it compares node responses to the Response Data, which is learned from a known-good UUT using the LEARN command. If GFI is to test the node, at least one of these fields must contain data.

- SIG: A hex CRC signature gathered at the node over the duration of the stimulus program. The signature may be modified by one of the following symbols: "*" to show instability, or "+" or "-" to show that the signature is marginal as explained in the "LEARN Command" section.

- Async LVL: The asynchronous level history gathered at the node over the duration of the stimulus program. The level history can include up to three characters, depending on the levels detected at the node during the stimulus program. The history can include a "1" (high), "0" (low), and "x" or "X", where either character signifies an invalid level. An "*" signifies unstable levels.

  It is also possible to specify one or more *don't care* states in an expected level history by using the "?" character. A *don't care* state is ignored during the comparison of the expected and measured level histories. This allows you to specify level histories that require certain states, but *don't care* whether other states are present. For example, you can specify a level history that requires a high and low to be present, but doesn't care whether a tristate is present.

The "?" character is used to represent a *don't care* state. The meaning of the "?" character is interpreted based on its position in the field. The Async LVL field is three columns wide. The columns represent (from left to right) high, tristate and low. A "?" in the leftmost column means *don't care* on high, in the middle column means *don't care* on tristate, and in the rightmost column means *don't care* on low.

For example, "1?0" means high is required, *don't care* on tristate, and low is required. This pattern would match every level history that contained a high and low state (10 and 1X0 would both match). As another example, "??0" means *don't care* on high, *don't care* on tristate, and low is required. This pattern would match every level history that contained a low (0, X0, 10 and 1X0 would match).

You can edit the learned level histories and insert "?" characters as desired.

- Clk LVL: The clocked level history gathered at the node over the duration of the stimulus program. The level history can include up to three characters, depending on the levels detected at the node during the stimulus program. The history can include a "1" (high), a "0" (low), and "x" or "X", where either character signifies an invalid level. An "*" signifies unstable levels. A "?" represents a *don't care* state.

- Counter Mode: A field that specifies whether the Counter Range is a transition count (TRANS) or a frequency (FREQ). This field reflects the way the counter was used by the stimulus program. The counter mode is set in the program by the *counter* command. This field cannot be edited.

- Counter Range: The frequency or the count measured at the node over the duration of the stimulus program. A stable count is shown by a single decimal number. An unstable count is shown by a range of observed (lowest and highest) values. Overflow is indicated by OVFL.

MORE Command

This command displays additional fields in the edit window. It toggles between two screens. The first screen contains fields for the response data and the second screen contains a priority pin field. The node signal source field appears in both screens for continuity. Figure 5-25 shows a response file before and after the MORE command.

- Priority Pin: Identifies the pin that should be checked next if the response data measured at the current node does not match the expected data.

   This field is normally empty because GFI automatically selects the next pin using related input information from the part description. However, you may know that a particular node failure is often caused by a bad output several components away on the backtracing path. Considerable time can be saved by jumping directly to this suspect pin. This capability is called "leapfrogging", and the suspect pin is called a "priority pin".

   If a bad output has a priority pin, GFI will jump to that pin and test it. If the priority pin is also bad, backtracing resumes from there. If the priority pin is good, GFI returns to the original bad output pin and continues backtracing from there, just as if there had not been a priority pin.

DELETE Command

This command deletes the line where the cursor is located.

---------------------- Response Data ----------------------

Node Signal Src	Learned With	SIG	Async LVL	Clk LVL	Counter Mode	Counter Range
u27-11	I/O MODULE	3C3F	1 0	1 0	TRANS	32
u27-12	I/O MODULE	E735			TRANS	
u27-13	I/O MODULE	8DB2			TRANS	
u27-15	I/O MODULE	8CC4			TRANS	
u27-16	I/O MODULE	3479			TRANS	
u27-17	I/O MODULE	6691			TRANS	
u27-18	I/O MODULE	2D9E			TRANS	
u27-19	I/O MODULE	12E5			TRANS	

F1	F2	F3	F4	F5	F6	F7	F8	F9	F10
GOTO	SAVE	LEARN		SELECT	DELETE	INSERT	MORE		FAULT

Response File Before A MORE Command

Node Signal Src	Priority Pin
u27-11	
u27-12	
u27-13	u14-12
u27-15	u14-12
u27-16	
u27-17	
u27-18	
u27-19	

F1	F2	F3	F4	F5	F6	F7	F8	F9	F10
GOTO	SAVE	LEARN		SELECT	DELETE	INSERT	MORE		FAULT

Response File After A MORE Command

Figure 5-25: MORE Command Response File

INSERT Command

This command inserts a new line below the line where the cursor
is located.

LEARN Command

This command, invoked while editing a stimulus program
response file, should be used to gather data from a known-good
UUT. The LEARN command gathers a set of response data for
one node while a stimulus is being executed. Learned responses
are then written to the stimulus program response file, where the
programmer can review and modify selected items before saving
them. The manual alternative to the LEARN command is to type
data known to be correct into the Response Data fields.

LEARN requires that the GFI data base is successfully compiled
for GFI or UFI LEARN, and the stimulus program is written.
The Info Window fields must be set for the desired LEARN
level, number of repetitions, and coverage.

To initiate LEARN, you must be editing a stimulus program
response file. To learn a node response, position the cursor
anywhere on the line for that node. After pressing the LEARN
softkey the following takes place:

1.   LEARN will prompt you with USE CURRENT
      LEARN OPTIONS. Use the Field Select key to
      select YES and press the Return key to begin the
      learn operation. Select NO if you want to change the
      learn options (refer to "Changing LEARN Options").

2.   The GFI database is loaded; LEARN looks here to
      determine the testing device to be used.

3.   Depending on the results of step 1, you will be
      prompted to clip or probe a component, and to then
      press the Ready button on the testing device.

4.   The stimulus program is executed. If the program
      needs the external control lines on the I/O module or

5-67

clock module, you will be told where the lines are to be connected.

5. The stimulus generates responses, which are collected by LEARN, during the period defined by the TL/1 *arm* and *readout* commands.

6. If more than one signal source is being learned, or LEARN requires both measurement devices, steps 2 through 4 may be repeated several times. All pins measured with the probe are learned first, followed by all pins measured by the I/O module.

7. The learned responses can be reviewed and modified by the programmer.

8. Select responses to be saved with the SELECT command. Only data thus selected will be saved when the response file is saved. LEARN automatically selects stable CRC signatures.

Changing LEARN Options

The LEARN command can gather data in several different ways. To change the way that LEARN gathers data, the following steps should be taken:

1. In a response file, press the LEARN softkey (F3). The editor will then prompt USE CURRENT LEARN OPTIONS. Use the Field Select key to select NO, then press the Return key.

2. A dialog window appears with the current LEARN options. There are three options which can be changed:

   Learn using: Indicates if the next LEARN is for UFI or GFI. A UFI LEARN gathers responses using only the measurement device specified for the signal source pin. A GFI LEARN examines all other pins on the same node and gathers responses using both measurement devices if needed. Before executing a

GFI LEARN, a node list must be entered and the GFI database must be compiled for GFI. Use the Field Select key to select GFI or UFI.

Learn for: Indicates the number of pins covered by LEARN. The following are three options that may be entered using the Field Select key:

- ONE NODE: The line of the Edit Window containing the cursor is examined. The signal source pin is learned.
- ONE REF: The line of the Edit Window containing the cursor is examined. Every line of the response file whose signal source is on the same reference designator is learned.
- ALL REFS: Every signal source in the response file is learned.

Repeat stimulus: the LEARN operation is performed several times to insure that marginal timing situations are detected. This value controls how many times the LEARN operation is repeated. Each LEARN operation runs the stimulus program three times (refer to "Standard LEARN Cycle Timing"). Enter a numeric value between 1 and 99.

Use the cursor keys to move between the three options.

3.   Press the LEARN softkey (F3) to LEARN using the options just explained. These options remain in effect until you change them or exit the editor. You can press the QUIT key to abort the LEARN command, and return to editing the response file.

Standard LEARN Cycle Timing

The LEARN command normally executes a stimulus program three times; each time, the clock edge (used to gather data at the node) is varied slightly. The multiple executions can be used to tell whether the response is:

- Stable.

- Unstable.

- Marginally stable.

Before each type of response is displayed, the results of the three executions are merged into one reading using methods described in the next section.

The clock edges of Figures 5-26 and 5-27 can be described as follows:

- Edge B: The synchronized edge, which occurs at the same instant as an event specified by the *sync* command (which each stimulus should have).

- Edge C: The delayed edge, which occurs a fixed time interval after edge B.

- Edge A: The advanced edge, which occurs a fixed time interval before edge B.

Unstable Response: In Figure 5-26 Example 1, the synchronized edge occurs when data changes from high to low or vice versa. Therefore, the level recorded at the clock edges A, B, and C will differ; the response is considered unstable.

Stable Response: In Figure 5-26 Example 2, the synchronized clock edge occurs when data is always stable. The level recorded at the clock edges A, B, and C will not vary between stimulus program executions and is considered stable.
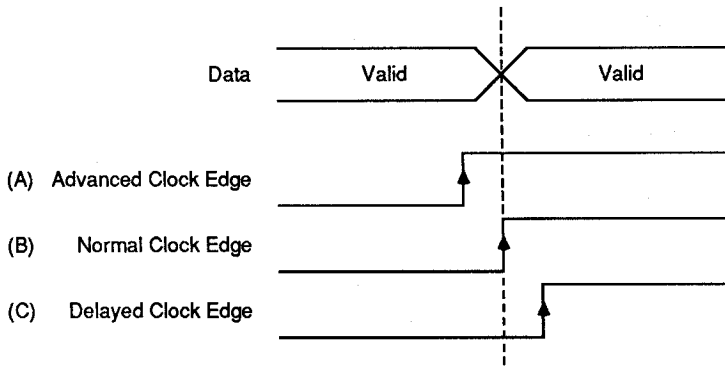
Example 1: Unstable Response
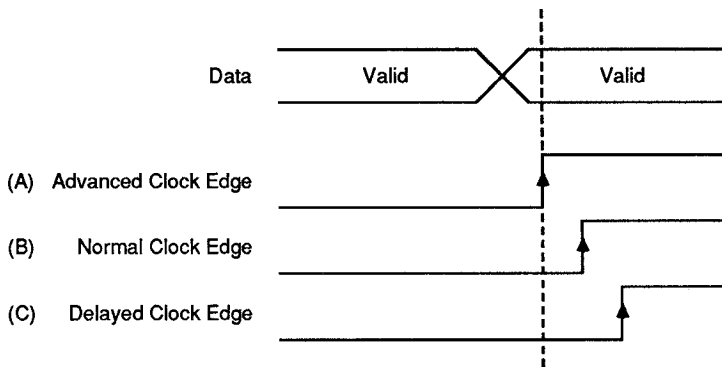
Example 2: Stable Response

Figure 5-26: Stable and Unstable Response Timing

Marginal Response: Figure 5-27 illustrates the marginal timing case where data observed at edge B agrees with either data at edge A or data at edge C. If the data at all three edges agrees, the response would be stable and if the data at all three edges differed, the response would be unstable. In cases where two adjacent data points agree, the response is considered marginally stable. If identical responses at A and C are separated by a different response at B, the response is considered unstable.

The marginal case is important, because the response learned on a known-good UUT may not be the same as that on an unknown UUT, if a slightly different clock edge is used with the unknown UUT. This does not mean that the unknown UUT is defective. On the contrary, if the response from the known-good UUT was reported to be marginal, we may see a different response from almost every good UUT.
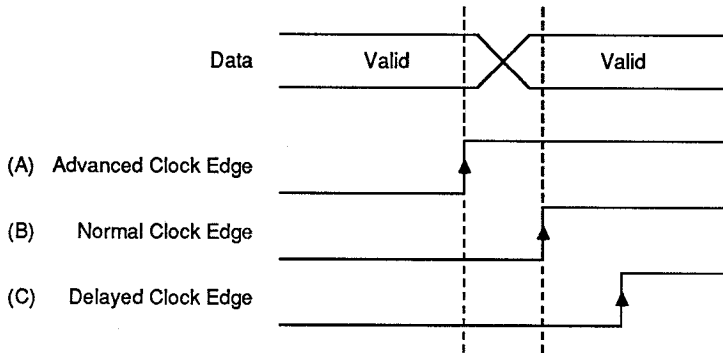
Marginally Unstable (−) Response
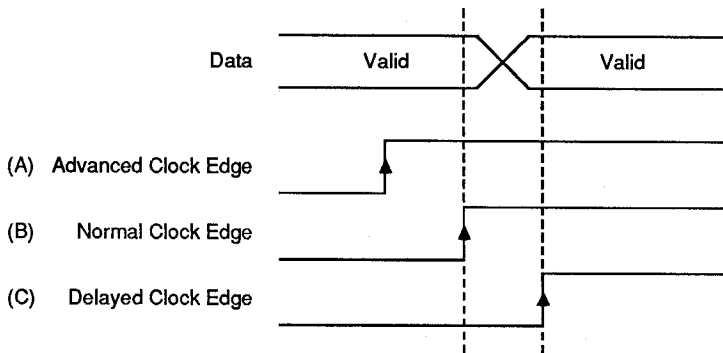
Marginally Unstable (+) Response

Figure 5-27: Marginal Response Timing

5-73

Merging Responses

The stimulus program is run three times each LEARN cycle with synchronized, delayed, and advanced clock edges, as described in the previous section. At each execution, responses are recorded. LEARN merges each type of data from all executions into one set according to the following rules:

- **Signature Merging:** If the same signature is measured three times, this signature is recorded unchanged. This is a stable signature.

  If only the delayed and synchronized edge signatures match, this signature is recorded with a "-" to indicate that the advanced clock signature was different. This is a marginal signature.

  If only the advanced-clock and normal-clock signatures match, this signature is recorded with a "+" to indicate that the delayed-clock signature was different. This is a marginal signature.

  If different signatures are recorded each time, an "*" is displayed to indicate an unstable signature. Figure 5-28 shows examples of how LEARN merges signatures from three different stimulus program executions.

- **Asynchronous Level History Merging:** If the same level history is measured each time LEARN executes the stimulus, that history is recorded; otherwise, the history is reported as unstable (*).

- **Clock Level History Merging:** If the same level historyis measured each time LEARN executes the stimulus, that history is recorded; otherwise, the history is reported unstable (*).

Delayed-Clock Edge Signature (C)	Normal-Clock Edge Signature (B)	Advanced-Clock Edge Signature (A)	Recorded Signature
14EA	14EA	14EA	14EA
14EA	14EA	225C	14EA–
800F	14EA	14EA	14EA+
14EA	907C	24E0	*
14EA	907C	14EA	*

Figure 5-28: Merging Signatures Example

- **Count or Frequency:** If the same count or frequency is measured each time LEARN executes the stimulus program, that countor frequencyis recorded; otherwise, a range (highest and lowest values) is recorded. In case of an overflow, an "OVFL" message is displayed.

SELECT Command

Once the data has been recorded in the Response Data fields, you can select the data you want to save by moving the cursor to the appropriate field and pressing the SELECT softkey. Selected data values are displayed highlighted. A stable signature (one with no + or -) is automatically selected, but you can de-select it using the SELECT softkey. Unstable data cannot be selected. When you quit the editor, the response file is saved containing only the selected data.

Pressing Shift and SELECT simultaneously selects or de-selects an entire column of response data.

Editing a Stimulus Program Response File   5.5.11.

Response files contain data characterizing how nodes on a known-good UUT responded to a stimulus program. To edit the response file *pod_sync,* in the UUT directory *abc:*

1. Press the Edit key and type:

    /hdr/abc/pod_sync

    (*pod_sync* is also the name of an existing stimulus program.)

2. Press the Return key, specify the TYPE field as RESPONSE, and press Return again.

3. Move the cursor to the bottom line and position it in the Node Signal Src field. This field is used to identify the node exercised by the program. The node is identified by typing the name of the pin (on that

node) that acts as the node signal source during the stimulus program.

4. To use GFI's optional leapfrogging capability, press the MORE softkey, move the cursor to the Priority Pin field, and enter the name of a pin. This field can be blank.

5. Repeat steps 3 and 4 until one line has been entered for each node that is exercised by the stimulus program.

6. Press Quit, and use Field Select to specify whether or not to save your edits. The LEARN command can be used later to fill in the Response Data fields.

Example LEARN Session   5.5.12.

Stimulus program response files are paired with stimulus programs. Response files contain the response data that characterizes how nodes on a known-good UUT responded to the stimulus program.

Figures 5-29 through 5-31 show example screens (1 - 5) that you would see when following this example. The steps are:

1. Press the Edit key and type:

   /hdr/abc/addr_out

2. Press Return, select RESPONSE as the TYPE field, and press the Return key again. Screen 1 shows the response file named *addr_out*.

3. Move the cursor down to a node signal source for which responses are to be learned (in this example it will be U27-16).

4. Press the LEARN softkey. You will be prompted USE CURRENT LEARN OPTIONS. Use the Field Select key to select NO and then press the Return key.

5-77

5.   The dialog window for the learn options will appear. Change the settings to the following:

Learn Using: GFI
Learn for: ONE NODE
Repeat Stimulus: 1 time(s)

Use the cursor keys to move between the options. Use the Field Select key for the "Learn Using:" and "Learn for" fields. Use the numeric keys for the "Repeat Stimulus" field. Refer to screen 2 to see how the LEARN options should be set up.

6.   Press the LEARN softkey; the GFI database is loaded, and a GFI LEARN of the node stimulated by U27-16 begins. If many pins are to be learned, measurements are made first with the probe, followed by measurements made with I/O module clips.

7.   You are now prompted to probe U3-15. U27-16 is on the same circuit node as U3-15. Refer to screen 3 in Figure 5-30 for the message prompting you to probe U3-15.

```
/HDR/ABC/ADDR_OUT (RESPONSE)                              [MODIFIED]  Line 3
------------------------- Response Data --------------------
     Node          Learned          Async  Clk   Counter
   Signal Src       With      SIG    LVL    LVL    Mode    Counter Range

   u27-11         I/O MODULE  3C3F   1 0    1 0   TRANS   32
   u27-12         I/O MODULE  E735                TRANS
   u27-13         I/O MODULE  8DB2                TRANS
   u27-15         I/O MODULE  8CC4                TRANS
   u27-16▓_____ I/O MODULE                      TRANS
   u27-17         I/O MODULE                      TRANS
   u27-18         I/O MODULE                      TRANS
   u27-19         I/O MODULE                      TRANS




   F1    F2    F3      F4    F5    F6    F7      F8    F9    F10
  GOTO  SAVE  LEARN        SELECT DELETE INSERT  MORE        FAULT
```

Screen 1

Figure 5-29: Example LEARN Session (Screen 1)
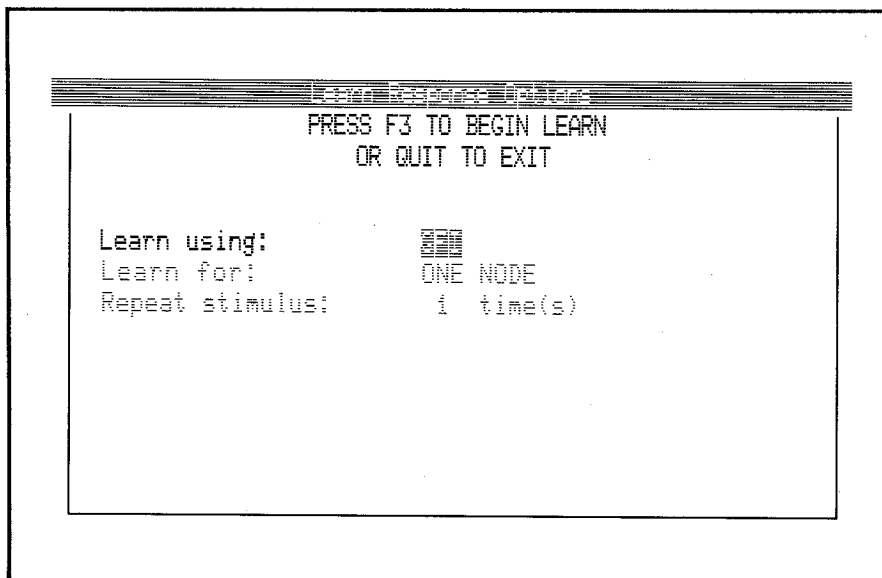
Screen 2
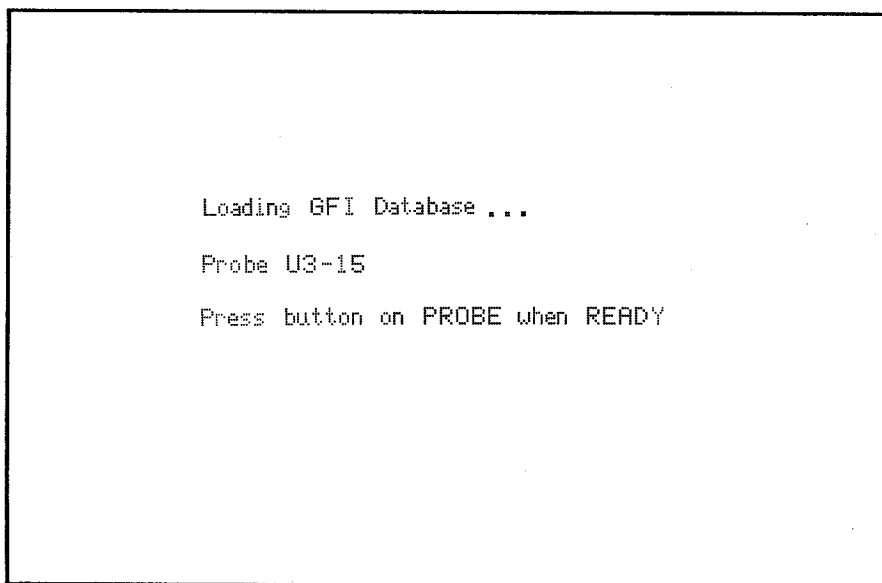
Screen 3

Figure 5-30: Example LEARN Session (Screens 2 and 3)

8.   Connect the probe common clip to common, probe
     U3-15, and press the Ready button. You should
     hear a beep indicating that the LEARN operation is
     beginning. Do not move the probe until you hear a
     second beep indicating the end of the LEARN
     operation.

9.   You are now asked to clip U27, refer to screen 4 of
     Figure 5-31. Using the correct I/O module adaptor
     clip, clip onto U27 and press the I/O module Ready
     button. You should hear a beep as LEARN executes
     the stimulus program.

10. Screen 5 shows the learned responses. There are
     two sets for U27-16: one each from steps 5 and 7.
     Stable signatures are highlighted (selected).

     To select or de-select a field, move the cursor to the
     field and press the SELECT softkey.

11. Quit, saving changes if necessary. Only data
     selected using SELECT will be saved.

```
Clip u27

Press button on I/O MODULE when READY
```

Screen 4

```
------------------- Response Data -------------------
    Node        Learned        Async  Clk  Counter
  Signal Src     With     SIG  LVL    LVL   Mode    Counter Range

  u27-11      I/O MODULE  3C3F  1 0   1 0   TRANS   32
  u27-12      I/O MODULE  E735              TRANS
  u27-13      I/O MODULE  8DB2              TRANS
  u27-15      I/O MODULE  8CC4              TRANS
  u27-16      PROBE       3479  1 0   1 0   TRANS   32
  u27-16      I/O MODULE  3479  1 0   1 0   TRANS   32
  u27-17      I/O MODULE                    TRANS
  u27-18      I/O MODULE                    TRANS
  u27-19      I/O MODULE                    TRANS
```

F1	F2	F3	F4	F5	F6	F7	F8	F9	F10
GOTO	SAVE	LEARN		SELECT	DELETE	INSERT	MORE		FAULT

Screen 5

Figure 5-31: Example LEARN Session (Screens 4 and 5)

OFFSET Command

The probe and I/O module have hardware delay lines that adjust the relative timing between clock and data signals. During probe and I/O module calibration, these delay lines are adjusted and a calibrated offset delay value is stored for each sync mode. The offset value controls when the probe and I/O module latch data.

When the calibrated offset delay is not appropriate for a measurement, the TL/1 *setoffset* command is used inside a TL/1 program to change the delay.

Offsets may need to be used when testing UUTs with slight timing variations on one board, or if there are board-to-board timing variations.

For example, assume that a properly-functioning board has a slight timing variation from board-to-board and also that the timing of the signal of interest can vary by several nanoseconds as shown in Figure 5-32.

Note the variation in the signal state during time periods T3, T4, T5, T9 and T10. If this signal was being clocked by CRC signatures and the signal state was latched at these time periods, different CRC signatures could result. Even though a board is functioning properly, it will occasionally fail during functional test because the measured CRC does not match the expected CRC.

The solution is to control the time at which the signal state is latched for a CRC signature. In the previous example, the signal should not be latched at time periods T3, T4, T5, T9, or T10 because the signal state can vary at these times. Instead, the signal should be latched at time periods T6, T7, or T8 when the signal is known to be at a consistent state. The TL/1 *setoffset* command is used to control when the probe and I/O module latch data.
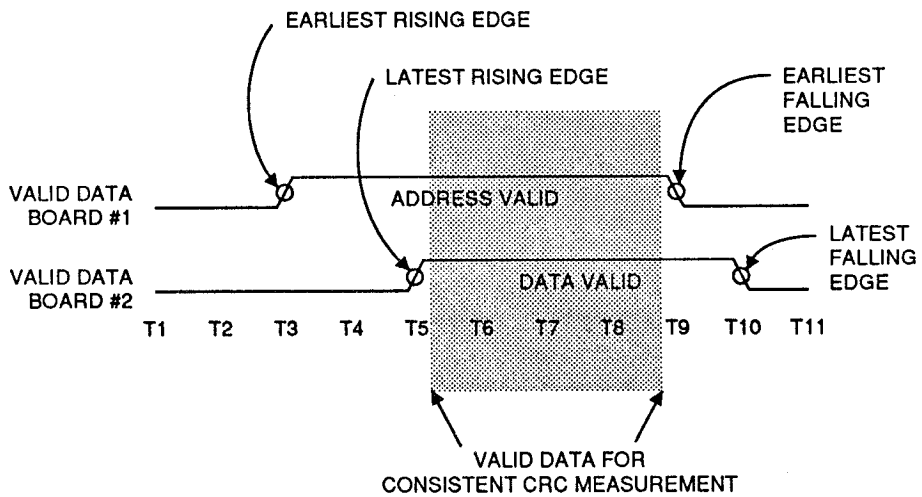
Figure 5-32: A Signal with Timing Variation

The GFI Offset procedure permits you to easily determine the appropriate offset delay for a measurement. Once the appropriate offset delay is determined, the TL/1 program that applies the measurement stimulus can be modified to set the offset.

The Offset Window EXEC softkey executes a TL/1 program over a range of offsets. Samples are taken at various offsets within the range and the response data gathered during each sample is displayed.

To take a sample, the offset value is set, the TL/1 stimulus program is executed, and the response data gathered during that program's *arm...readout* block is displayed. Graphical waveforms show the clocked level and CRC signature throughout the offset range.

In addition, the LOOP softkey repeatedly loops through the offset range, executing the TL/1 stimulus program until the loop is terminated. Each iteration through the offset range is called a sweep. Looping allows multiple samples to be taken at each offset value. This allows the response stability at each offset value to be checked.

Briefly, to use offsets with GFI, follow these steps:

1. Calibrate the probe or I/O module, or restore caldata settings. This establishes the calibrated offset delay to which additional offsets are added.

2. If the desired offset is known, go to step 7.

3. Compile the UUT database. The database is required by the offset procedure.

4. Begin editing a response file. Position the cursor on the line in which the offsets are to be checked.

5. Press the OFFSET softkey.

6. Press the EXEC or LOOP softkey to determine the best offset.

7. Edit the TL/1 stimulus program and add a setoffset command to set the offset to the desired value. "Setting the Offset in a Stimulus Program" explains how to set the offset.

The Offset procedure is designed to be used with GFI stimulus programs and requires that a compiled UUT database exists. Refer to "Compiling the GFI Database for a UUT" in this section.

Probe and I/O module calibration are described in the *Automated Operations Manual, Technical User's Manual,* and the *TL/1 Reference Manual.* The *setoffset* and *getoffset* commands are described in the *TL/1 Reference Manual.* The Offset procedure is described in the following paragraphs.

Description of the GFI Offset Window

The GFI Offset Window (shown in Figure 5-33) overlays the existing editor screen. It has top and bottom status lines, tables on the top left and right, waveforms for both clocked level and CRC signature, a legend that describes symbols used on the waveforms, and softkey labels.

Press the OFFSET softkey to toggle the GFI Offset Window on and off. When selected, the GFI Offset Window overlays the existing screen. Information displayed on this window disappears when it is toggled off. The GFI Offset Window (Figure 5-33) is described in the following paragraphs.

UPPER LEFT TABLE   UPPER RIGHT TABLE

STATUS LINE

```
/HDR/TMP/HDR STIM
-------------------- Response Data --------------------
     Node          Learned        Async  Clk   Counter
   Signal Src       With     SIG   LVL   LVL    Mode      Counter Range

   U1-1
   U2-2

========================= OFFSET WINDOW =========================
  Offset:                       Offset Range:
  SIG:                          Sample Resolution:
  Async LVL:                    Number of Loops Through Range:
  Clk LVL:                      Node Signal Src:
  Count:

  Clk LVL

  SIG

  LEGEND: ▐  means the Clk LVL was 1X, X0 or 1X0
          *  means the SIG or Clk LVL for one offset changed while looping

 F1    F2    F3        F4    F5    F6    F7        F8    F9    F10
                      OFFSET      EXEC  LOOP      FAULT
```

SOFTKEY NUMBER   SOFTKEY LABEL LINE

LEGEND

WAVEFORMS

The Offset Window Before Execution (EXEC or LOOP)

```
/HDR/TMP/HDR STIM                              EXECUTING   Line 3
-------------------- Response Data --------------------
     Node          Learned        Async  Clk   Counter
   Signal Src       With     SIG   LVL   LVL    Mode      Counter Range

   U1-1
   U2-2

========================= OFFSET WINDOW =========================
  Offset:     1000016           Offset Range: 999876 to 1000058 ns
  SIG:        00ED              Sample Resolution: 1
  Async LVL:  1X0               Number of Loops Through Range: 1
  Clk LVL:    1                 Node Signal Src: U1-1
  Count:      594 (580-628)

  Clk LVL     ____/‾‾‾‾<‾‾‾‾‾‾‾‾‾‾>‾‾‾‾‾\____

  SIG         __[___]__[____]_____[___]__[____]_
                                               ↑

  LEGEND: ▐  means the Clk LVL was 1X, X0 or 1X0
          *  means the SIG or Clk LVL for one offset changed while looping

 F1    F2    F3        F4    F5    F6    F7        F8    F9    F10
EXECUTING ...
```

The Offset Window During Execution (EXEC)

Figure 5-33: The GFI Offset Window

Upper Left Table:

The table at the top left of the window displays response data gathered in the current sample. This table is updated continuously as samples are taken at various offsets in the offset range. Data in the table corresponds to the place in the waveform to which the arrow points.

The upper left table contains the following fields:

Offset:	The offset for the current sample.
SIG:	The CRC signature for the current sample.
Async LVL:	The asynchronous level for the current sample.
CLK LVL:	The clocked level for the current sample.
Count:	The count for the current sample. The range of counts seen in all samples is also displayed.

Upper Right Table:

The table at the top right of the window displays information about the execution. It contains responses to the EXEC and LOOP softkey prompts. This table has the following fields:

Offset Range:	The range of offsets that are being sampled.
Sample Resolution:	The resolution between samples in the offset range (1=most resolution, 9 = least resolution).
Number of Loops through Range:	The number of times the offset range has been swept. This is used when the LOOP softkey is active.
Node Signal Src:	The pin being sampled.

Waveforms

Waveforms for the clocked level and CRC signature are displayed. Each character of the waveform represents the response data gathered at a single offset. The characters combine to form a waveform that graphically represents the response data for the entire offset range. The waveform is updated or extended to the right as each sample is taken.

Clk LVL: The clocked level waveform is drawn to display the three states: high, low, tri-state and combinations of these states. See the description of "legend" in "The Bottom of the GFI Offset Window" for more information.

SIG: The CRC signature waveform uses two symbols:

‾‾‾‾‾ Parallel Lines

[Left Bracket

The parallel lines indicate that the CRC signature at a sample is the same as the previous sample.

The left bracket indicates that the CRC signature at a sample is different than the CRC signature at the previous sample. This indicates that the sample was taken near a signal edge.

◆: The diamond shows the original offset before any samples are taken.

↑: The arrow points at the current sample in the SIG waveform. The arrow position is automatically updated during EXEC or LOOP. After the EXEC or LOOP operation, use the left-arrow and right-arrow keys to move the arrow along the waveform. As the arrow is moved, the upper table is updated to display the response data for the sample to which the arrow points. This allows you to examine the response data for all samples in the offset range.

The Bottom of the GFI Offset Window:

The bottom of the GFI Offset Window contains a legend to explain the waveform symbols and the softkey labels.

The graphic character set on the 9100A does not contain symbols that draw the following combined states:

- Tri-state and high level.
- Tri-state and low level.
- Tri-state and high and low level.

The block symbol (▓) represents these three states. The upper left table identifies which of these three states actually occurred.

The asterisk (*) appears in a waveform during looping and indicates an unstable CRC signature or an unstable clocked level. Looping allows multiple samples to be taken at an offset value (once each time the range is swept).

The asterisk appears in the Clk LVL waveform when the clocked level for one offset value changes between sweeps. The asterisk appears in the SIG waveform when the CRC signature for one offset value changes between sweeps.

Status Line:

The status line displays the execution status: EXECUTING, LOOPING, OR COMPLETE.

Softkeys on the GFI Offset Window:

Softkey numbers and their labels appear at the bottom of the window. Error messages and prompts also appear on this line.

Sofkeys used with the GFI Offset Window are:

OFFSET: Toggles the Offset Window on and off. Information displayed in the Offset Window disappears when it is toggled off.

EXEC: Starts one sweep of the offset range. The TL/1 program will be executed at various offsets within the range.

LOOP: Starts sweeping through the offset range. The TL/1 program is executed at various offsets within the range. The range is swept repeatedly, resulting in multiple samples at each offset value. Press the QUIT key to terminate the looping.

FAULT: Toggles the Fault Window on and off. The Fault Window displays fault messages which are raised by the TL/1 program.

The EXEC Softkey

The Offset Window EXEC softkey is used to execute a TL/1 program over a range of offsets. Samples are taken at various offsets within the range and the response data gathered during each sample is displayed.

To take a sample, the offset value is set, the TL/1 stimulus program is executed, and the response data gathered during that program's *arm...readout* block is displayed. One graphical waveform shows the clocked level throughout the offset range. Another waveform shows changes in the CRC signatures throughout the offset range.

To begin execution, use the following procedure:

1. The Offset procedure requires that a compiled UUT database exist. Refer to "Compiling the GFI Database for a UUT" for instructions on compiling a database.

2. Edit the response file in which you want to check the offsets.

3. Determine which node to check for offsets. If the response file already contains a line for that node, position the cursor on that line. If the response file does not contain a line for the node, add it. Leave the cursor on the line for the desired node.

4. Press the OFFSET softkey. The UUT database is loaded and the Offset Window appears.

5. Press the EXEC softkey. The following prompt appears:

    ```
    EXECUTE PROGRAM_____
    ```

    Enter the name of the TL/1 program to be executed throughout the offset range. Typically, you will execute the stimulus program that is paired with this response file (this name appears as the default). However, you may specify any program in the UUT. Press RETURN. See Figure 3-27 and 5-34 in this manual for stimulus program guidelines.

    ### NOTE

    The stimulus program cannot contain a setoffset command when it is executed from the Offset Window.

6. The following prompt appears:

    ```
    OFFSET RANGE (ns) FROM_____
    ```

    Enter the offset to be used as the beginning of the offset range. The value entered represents nanoseconds and is biased by a value of 1000000 (decimal). For example, if the first sample is to be taken 100 nanoseconds before the calibration point, enter 999900 (1000000-100). If the first sample is to be taken 25 nanoseconds before the calibration point, enter 999975 (1000000-25).

Enter the value 999000 (this appears as the initial default) to ensure that the entire offset range is sampled. Press RETURN.

7.   The following prompt appears:

    TO_____

    Enter the offset to be used as the end of the offset range. The value entered represents nanoseconds and is biased by a value of 1000000 (decimal). For example, if the last sample is to be taken 50 nanoseconds after the calibration point, enter 1000050 (1000000 + 50).

    Enter the value 1001000 (this appears as the initial default) to make sure that the entire offset range is sampled. Press the RETURN key.

8.   The following prompt appears:

    SAMPLE RESOLUTION (1-9) __

    Enter a number representing how often samples should be taken in the offset range. Enter 1 for the most resolution and 9 for the least resolution. For example, enter 1 to take a sample at every offset value in the offset range. Enter 2 to take a sample at every other offset value in the offset range. Enter 3 to take a sample at every third offset value in the offset range.

    Enter the value 1 (this appears as the initial default) to make sure that samples are taken at every possible offset value in the offset range.

9.   A prompt to probe a pin or clip a component appears. An example is:

    Probe U3-14 and press button on PROBE
    when READY

Follow the instructions in the message. When the button is pressed, the message "EXECUTING..." appears at the bottom of the screen and executions of the TL/1 program begin. Do not move the probe or I/O module until this message disappears and the softkey labels reappear.

As each sample in the offset range is taken, the upper left table and the waveforms are updated to display the response data for that sample.

10. When the execution is complete, press the left-arrow and right-arrow keys to move through the waveforms and examine the response data at a particular offset.

11. Position the waveform arrow at the point with the desired offset. The actual offset at that point is listed in the upper left table. Make a note of the offset value. Modify the TL/1 stimulus program to set the offset to that value. Refer to "Setting the Offset in a Stimulus Program" for instructions.

Press the QUIT key to abort from the prompts or to stop execution of the TL/1 program.

An Example of Selecting the Desired Offset

Figure 5-34 shows when address and data are valid for a simple microprocessor write cycle. Assume that a data line is tested by using the probe to gather a CRC signature while a series of UUT writes is performed. The signal state should be latched during the data valid period. At other times, the signal on the data line changes, resulting in changing CRC's. The SIG waveform indicates that the CRC is changing during the data invalid period.

In Figure 5-34, the signal state should be latched in the middle of the data valid period.

A0 - A15   ADDRESS VALID

D0 - D7   DATA VALID

DATA SIG
WAVEFORM
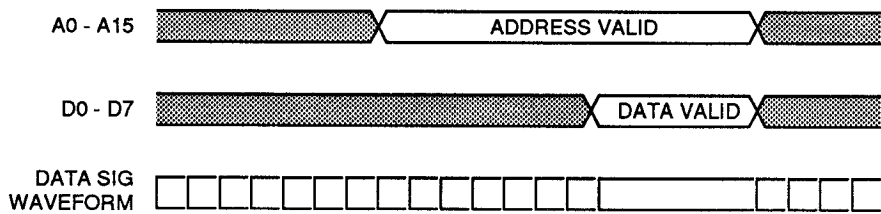
Figure 5-34: Selecting an Offset

The LOOP Softkey

Use the LOOP softkey to repeatedly loop through the offset
range, executing the TL/1 stimulus program until the loop is
terminated. Each iteration through the offset range is called a
"sweep". Looping allows multiple samples to be taken at each
offset value. This allows the response stability at each offset
value to be checked.

To begin looping, use the following procedure:

1. Press the LOOP softkey. The following prompt
   appears:

   LOOP and EXECUTE PROGRAM_____

2. You are prompted for the program name, offset range,
   and sample resolution and instructed to probe or clip a
   component. These steps are described in "The EXEC
   Softkey" in this section.

The waveform drawn on the screen is the same as if you had
pressed the EXEC key. Looping continues until you press the
QUIT key.

NOTE

*Unstable CRC signatures and unstable clocked levels
appear as asterisks. Looping allows multiple samples
to be taken at an offset value (once each time the range
is swept).*

The field entitled "Number of Loops Through Range" in the
upper right table tells you how many loops are completed.

The FAULT Softkey

When both the EXEC and the LOOP functions are executing a
TL/1 program, faults can occur in the program. When a fault is
raised in a TL/1 program, it is reported to the user. The program
stops and the Fault Window pops up. When the Fault Window
is displayed, press the FAULT softkey to remove it.

Setting the Offset in a Stimulus Program

Once the desired offset value is determined, the TL/1 stimulus program must be modified to set the offset to that value. The program should also restore the original offset value when it terminates.

A program containing an offset cannot be used for EXEC and LOOP. The offset in the program overrides any offsets entered in the EXEC and LOOP procedures and an error message appears.

To set the offset in a TL/1 program, the program should do the following:

- Initialize the UUT as required for the stimulus to be applied.
- Configure the response-gathering hardware on the probe or I/O module. This includes setting the sync mode, threshold, counter mode, etc.
- Get the original offset value using *getoffset* and save it.
- Set the offset to the desired value using *setoffset*.
- Apply the stimulus within an *arm...readout* block. Read the results of the stimulus by using the *readout* command.
- Restore the original offset value using *setoffset*.

The TL/1 *setoffset* and *getoffset* commands apply to the current sync mode. Therefore, the sync mode must be set (with the TL/1 *sync* command) before these commands are used. If a fault occurs during execution of the program, the original offset must be restored. The program example includes a universal fault handler which restores the offset and reraises the original fault.

Figure 5-35 is a program example that shows a GFI stimulus program that sets an offset.

```
program pod_sync

declare
    ! global variables shared with fault handler
    global numeric orig_offset
    global string dev
end declare

! This fault handler ensures that the original offset gets restored
! if the program exits because a fault is raised. It restores the
! offset and then reraises the original fault.
handle
   declare
       ! global variables shared with main program
       global numeric orig_offset
       global string dev
   end declare

   setoffset device dev, offset orig_offset
   refault
end handle

! ------------main program starts here --------------------

! get the measurement device name from GFI
dev = gfi device

! configure the measurement hardware on the probe I/) module
reset device dev                 ! reset device to a known state
threshold device dev, level "ttl"    ! set threshold levels
counter device dev, mode "transition" ! set counter mode
sync device dev, mode "pod"           ! sync device to pod
sync device "/pod", mode "addr"       ! sync pod to address

! save the original offset (must be done after the sync mode is set)
orig_offset = getoffset device dev

! set the offset to 10 nanoseconds before the calibrated offset delay
setoffset device dev, offset 999990
```

(continued on the next page)

Figure 5-35: GFI Stimulus Program that Sets an Offset

```
! apply the stimulus
arm device dev                        ! start the response capture
    rampdata addr $F0000, data 0, mask $F
    rampdata addr $F0000, data 0, mask $F0
    rampdata addr $F0000, data 0, mask $F00
    rampdata addr $F0000, data 0, mask $F000
readout device dev                    ! terminate response capture

! restore the offset to the original value
setoffset device dev, offset orig_offset

end program
```

Figure 5-35: GFI Stimulus Program that Sets an Offset *(cont)*

Compiling the GFI Database for a UUT   5.5.13.

To learn responses use the Response Offset Window, or
perform GFI or UFI, information from the UUT (REFLIST,
parts, NODELIST, and responses) must be compiled into a
binary form (the database). GFI, UFI, response LEARN, and
the Offset Window use the database rather than the individual
files.

Compilation is typically an iterative process with at least two
cycles. After the UUT topology information is entered
(NODELIST, RELIST, and parts), the UUT is compiled so that
responses are learned; then the UUT is compiled again so that
GFI or UFI can be performed.

Depending on the intended use of the compiled database,
different UUT files are compiled as shown in Figure 5-36.

Only one data base can be compiled for each UUT. Each time
you compile the UUT, the new database writes over the old one.
A database can be copied or removed using the COPY or
REMOVE softkey respectively.

When the UUT is compiled, the REFLIST, parts, NODELIST,
and responses are compiled into a binary database. GFI, UFI,
and LEARN use the database rather than the individual files. If
you change any of the UUT files (REFLIST, parts,
NODELIST, or responses), the UUT must be recompiled so that
the database includes the UUT file changes.

UUT FILES COMPILED					
DATABASE USE	REFLIST	PARTS	NODE LIST	RESPONSES	PROGRAMS
TROUBLESHOOT UUT for GFI	YES	YES	YES	YES	NO
TROUBLESHOOT UUT for UFI	YES	YES	NO	YES	NO
LEARN RESPONSES for GFI	YES	YES	YES	NO	NO
LEARN RESPONSES for UFI	YES	YES	NO	NO	NO

Figure 5-36: Compiled UUT Files

Compilation is a two pass process using the following seven steps. The first pass allows the response to be learned. In step 4, select LEARN RESPONSES.

Repeat steps 1 through 7 for the second pass (after learn). The second pass readies the system for troubleshooting. In step 4, select TROUBLESHOOTING UUT.

To compile a UUT called abc, refer to the following steps:

1. Enter the UUT directory by pressing the EDIT key and typing:

    /hdr/abc

2. Press the Return key, select UUT as the TYPE field, and press the Return key again.

3. Press the COMPILE softkey (F3), and the 9100A issues the the following prompt:

    COMPILE database to TROUBLESHOOT UUT
                          LEARN RESPONSES

4. Press the Field Select key to select TROUBLESHOOT UUT or LEARN RESPONSES.

5. Press the Return key; an additional prompt is appended to the original prompt shown below:

```
COMPILE database to TROUBLESHOOT UUT for GFI
                                          UFI
```

or

```
COMPILE database to LEARN RESPONSES for GFI
                                        UFI
```

6. Press the FIELD SELECT key to select GFI or UFI.

If the database will be used to perform GFI, select GFI. Make this selection even if you are only learning responses at this time. If the database will be used to perform UFI, select UFI.

7. Press the Return key to compile the database.

If the UUT is successfully compiled with 0 errors, the resulting compiled database is written to the disk.

If the compiler detects a problem, the Messages Window displays an error message. Correct the error and repeat steps 1 through 7.

Status messages, error messages, and warnings are displayed by the UUT compiler on the Messages Window shown in Figure 5-37. Screen 1 shows the information after a successful compile. Screen 2 shows the information after a compile containing errors.

```
                    UUT Compiler (GFI)


Processing UUT reflist and parts...
Processing UUT nodelist...
Processing UUT responses...


0 errors


Writing string table...
Writing part table...
Writing ref table...
Writing response table...
Writing node table...


              Press Msgs key to continue
```

Screen 1

```
                    UUT Compiler (GFI)


Processing UUT reflist and parts...
REFLIST(9):      undefined part '74244'
REFLIST(15):     duplicate ref 'U16'
conn32:    more than one pin is named 'a12'
Processing UUT nodelist...
NODELIST(23):     undefined pin 'u23-6'
NODELIST(54):     'u14-12' appears in multiple nodes
Processing UUT responses...
ADDR_OUT(7):     warning: no response data on line
ADDR_OUT(16):    missing PROBE responses for 'U27-1


6 errors


              Press Msgs key to continue
```

Screen 2

Figure 5-37: Information Displayed After a Successful and Unsuccessful Compile

Error messages are displayed in the following format:

```
File Name (Line Number):   Error Message
```

A database is not created by the compiler until all errors are corrected.

Warnings are displayed in a similar format with one exception; the message is preceded by the word "warning" as shown in the following format:

```
File Name (Line Number):   warning: Warning Message
```

Warnings do not stop the compiler from creating a database. You should investigate the warning; it may indicate that there is a mistake in the named file.

Error Messages

The following are possible error messages issued by the compiler. An explanation of each message is provided with instructions to remove the error.

`'ref-pin'` appears in multiple nodes

> The named pin is listed in the NODELIST more than once. Remove all extra references to the pin from the NODELIST.

`'ref-pin'` has already appeared as a signal source for this node

> According to the NODELIST, the named pin and the pin listed in the response file are on the same node. GFI requires that only one pin drive the node during a stimulus program. This pin should be listed as the signal source. Remove all references to the pin that is not the driver from the response file.

duplicate ref `ref'

> The named reference designator has already appeared in the REFLIST. Remove the duplicate entry from the REFLIST.

incomplete list of pin names beside IC picture

> Some of the pins on the named part have been given pin names, but other pins do not have names. Add pin names to the part for every pin on the IC.

missing I/O MODULE responses for `ref-pin'

> The node requires responses learned with the I/O Module. Use the response LEARN command to characterize the node again.

missing PROBE responses for `ref-pin'

> The node requires responses learned with the Probe. Use the response LEARN command to characterize the node again.

more than one pin is named `pin'

> The named part description has multiple pins with the same name. Rename one of the pins.

no part listed for ref `ref'

> The RELIST contains the named reference designator, but the part field is blank. Fill in the part name in the RELIST.

`pin 'pin' has more than 255 related input pins`

Too many related input pins have been listed for the named pin. Look for duplicate entries in the lines below the IC in the part description.

`signal source 'ref-pin' is an input pin`

The response file specifies that the named pin is driving the node, but the part description identifies the pin as an input pin. The error could be in any of three files. In the response file, the wrong pin on the node has been named as the signal source. In the REFLIST file, the wrong part has been listed. In the part description file, the pin-type is wrong.

`too many name strings in UUT (max 65,534 characters)`

The name table has overflowed. This table contains reference designator names, pin names, and stimulus program names. Shorten the names.

`undefined part 'part'`

The named part is listed in the REFLIST, but does not exist in the part library. Create the part in the part library.

`undefined pin 'ref-pin'`

This message has multiple meanings, depending on what file is being compiled when it is issued.

If the RELIST or part descriptions are being compiled, the error message indicates the part description does not have a matching pin name. Add the pin name to the part description.

If the responses or NODELIST are being compiled, either the reference designator did not appear in the REFLIST, or there is no matching pin name in the part description. Add the reference designator to the REFLIST or add the pin name to the part description. This message is also issued if the named pin appears in the "*MASTERS" section of the NODELIST, but did not appear previously in the NODELIST. Add the pin to the NODELIST.

`unknown number of pins`

A part is referenced, but the part library description of that part is incomplete. Enter the number of pins in the part description.

Warning Messages

Warnings indicate a possible error that the compiler is ignoring. These messages should be investigated to make sure that nothing is wrong on the indicated line. The following are possible warning messages issued by the compiler. Included with the message is an explanation of the message and instructions on how to avoid future similar warnings.

`warning: no response data on line`

A signal source pin is listed, but the response data was not learned. Use the response LEARN command to characterize the node.

`warning: non-empty line ignored (I/O MODULE responses not required)`

The node does not require responses learned with the I/O module. Delete the line from the response file.

```
warning: non-empty line ignored (PROBE responses
 not required)
```

> The node does not require responses learned with the
> Probe. Delete the line from the response file.

```
warning: non-empty line ignored (no pin number
 listed)
```

> The output pin has not been entered in front of the list of
> related input pins appearing below the IC. Enter the
> output pin in the 'pin' field.

```
warning: non-empty line ignored (no ref listed)
```

> A part is listed on this line, but the 'ref' field is empty.
> Enter the name of the reference designator.

```
warning: non-empty line ignored (no signal source
 listed)
```

> This line contains learned response data or a priority pin,
> but the signal source field is empty. Enter the name of
> the signal source pin.

```
warning: signal source 'ref-pin' is an input pin
```

> The response file specifies that the named pin is driving
> the node, but the part description identifies the pin as an
> input pin. Ignore this message for UFI.

Generating a Summary of the GFI Database   5.5.14.

The 9100A provides a convenient means to check the completeness of the information you have compiled into the GFI database. When viewing the UUT directory display, you can press the SUMMARY softkey to request generation of a summary of GFI coverage for that particular UUT. The compiled database (GFIDATA or UFIDATA) will be examined and a summary will be generated, displayed on the monitor, and stored in a UUT text file that you specify. If you press the Shift key on the programmer's keyboard and the SUMMARY softkey, the summary will appear on the monitor without sending a copy to a text file.

The summary can only be generated for databases that are compiled to TROUBLESHOOT UUT. If the database was compiled to LEARN RESPONSES, an error message is displayed when an attempt is made to generate the summary.

Creating a Summary of GFI Coverage

The following procedure is used to generate a Summary of GFI Coverage for a UUT:

1.   Press the EDIT key on the operator's keypad to enter the Editor (unless you are already in the Editor).

2.   Use the Edit key on the Programmer's Keyboard to enter the name of the UUT so that the UUT directory for this UUT is displayed on the monitor. The UUT directory you have selected must contain a compiled database (either GFIDATA or UFIDATA).

3.   Press the SUMMARY Softkey (F8) and the 9100A will issue the prompt shown below to ask for a text file name:

        Generate GFI Summary to TEXT file ____

    The Summary of GFI Coverage to be generated will be stored in this text file.

4. Type in the text file name you wish and press the Return key. The 9100A will then begin generating the Summary of GFI Coverage for the UUT and will display the results on the monitor.

When the generation is complete, the following message will appear on the monitor:

```
Press Msgs key to continue
```

When you press the Msgs key on the programmer's keyboard, the UUT directory display will reappear on the monitor. You can use the Edit key on the programmer's keyboard to access the text file you generated.

Statistical Summary

The first part of the Summary of GFI Coverage is a statistical summary of the UUT, based on the GFI database you have provided. Figure 5-38 shows a typical example of such a summary. Each entry in the summary is described below:

- **Summary for /<disk drive>/<UUT>:** In Figure 5-31, HDR is the disk drive and the UUT directory name is EXAMPLE.
- **Parts:** The number of unique part types in the UUT, based on the reference designator list.
- **Reference Designators:** The number of reference designators in the UUT, based on the node list.
- **Connected Pins:** The number of UUT pins that are connected to other pins on the UUT, based on the node list.
- **Unconnected Pins:** The number of UUT pins that are not connected to any other UUT pins, based on the node list.
- **Total Pins:** The total number of pins on the UUT.

5-110

```
Summary for /HDR/EXAMPLE:

    53 Parts
   167 Reference Designators
  1694 Connected Pins
   225 Unconnected Pins
  1919 Total Pins
    42 Programs

  1688 Testable Connected Pins
    16 Testable Unconnected Pins
  1704 Total Testable Pins

     6 Untestable Connected Pins
   209 Untestable Unconnected Pins
   215 Total Untestable Pins

   99% Test Coverage of Connected Pins
   88% Test Coverage of Total Pins
```

Figure 5-38: Statistical Summary Display for a UUT

- **Programs:** The number of TL/1 programs that can be used by GFI as stimulus programs. This number is equal to the number of response files.

- **Testable Connected Pins:** The number of connected pins that can be tested by GFI. Testable pins have either been characterized with LEARN, or are a member of a node that has been characterized with LEARN.

- **Testable Unconnected Pins:** The number of unconnected pins that can be tested by GFI. Testable unconnected pins have been characterized by LEARN and appear in a response file.

- **Total Testable Pins:** The total number of UUT pins that can be tested with GFI, given the database you have entered.

- **Untestable Connected Pins:** The number of connected pins that cannot be tested with GFI, due to an incomplete database.

- **Untestable Unconnected Pins:** The number of unconnected pins that cannot be tested with GFI, due to an incomplete database.

- **Total Untestable Pins:** The total number of UUT pins that cannot be tested with GFI, given the database you have entered.

- **Test Coverage of Connected Pins:** The percentage of connected pins on the UUT that can be tested with GFI, given the database you have entered. A figure of less than 100% indicates an incomplete database.

- **Test Coverage of Total Pins:** The percentage of UUT pins that can be tested with GFI, given the database you have entered. This figure is typically less than 100% because a UUT often has unused pins.

Pin Coverage Matrix

The second part of the GFI Summary of Coverage display is a matrix showing how component pins are tested with the database you have provided. Figure 5-34 shows a partial example of a pin coverage matrix. The matrix is organized with the reference designators listed vertically (in the left-most column) and with component pin numbers listed horizontally. The number of pins per line will be the number required by the largest component in the list. If more than 35 pins are required, the display will produce a second list of reference designators following the first list and this second set will have pin numbers starting with 36 and continuing up from there.

Each component pin has a one-character symbol that shows what how GFI looks at the pin, given the database you have provided. The table at the bottom of Figure 5-39 shows the meaning of each symbol that is possible.

UNGUIDED FAULT ISOLATION (UFI)   5.6.

UFI is designed for a situation where the user wishes to use GFI's pin-testing capability but does not need probing suggestions. A UFI operator may, for example, use a combination of functional test programs, keypad commands, and UFI to troubleshoot a UUT. The UFI operator is normally someone who is familiar with the UUT, who has a good idea why it failed, and who can save time accordingly.

Differences between UFI and GFI 5.6.1.

GFI tests a pin and determines whether or not it is good. GFI
then evaluates the status (good or bad) of all other UUT pins,
and accordingly makes a probing recommendation. The process
is repeated until GFIcan accuse a faulty component.

UFI only tests output pins; it will not make recommendations on
where to probe next. UFI is used to verify whether a pin is
good or bad. The choice of where to probe next is left to the
operator.

UFI does not require a node list to be entered; GFI uses the list
to make probing recommendations. Since UFI makes no
recommendations, the list is unnecessary.

The UFI User Interface 5.6.2.

At the operator's interface, UFI is invoked by using the GFI key
as described in the following section. Since the database was
compiled using TROUBLESHOOT UUT for UFI, the node list
was not included. When UFI probes a pin, it will not suggest
the next probing point as GFI may do. Instead, the message
"UNGUIDED MODE" appears on the display.

Pin Coverage:

```
                              1 1 1 1 1 1 1 1 1 1 2 2 2 2 2 2 2 2 2 2 3 3 3 3 3 3
              1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5

     C15  I O . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
     C16  I I . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
     C17  I O . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
     J5   I * * I I * * * I * * I I I I I I I I I I I I I I I I I I I I I I I I I I
     J6   I I I I I . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
     Q1   O I I . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
     Q2   O I I . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
     R10  O I . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
     R11  I O . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
     R12  I O . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
     S1   I I . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
     S2   I I . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
     U10  I I I I I B I I B I I B I I B B I I B I . . . . . . . . . . . . . . . . .
     U11  * I * I I I I * I I I * O O O * O B B B B I * O B B B B * * * * O * I
     U12  O O I O I O O I O I I O I O I I . . . . . . . . . . . . . . . . . . . . .
     U13  I O I O I O G O I O I O I P . . . . . . . . . . . . . . . . . . . . . . .
     U14  O * * O O * * * I * * O O O O O O O O O O O O O O O O O O O I I I O O O I
```

Symbol	Meaning
I	The pin is testable as an input only.
O	The pin is testable as an output only.
B	The pin is testable as both an input and an output.
P	The pin is testable as a power pin.
G	The pin is testable as a ground pin.
*	The pin is not testable (because it has no associated stimulus program or no known-good response stored for this pin).
.	There is no such pin in the database.

Figure 5-39: Pin Coverage Display for a UUT

Converting from UFI to GFI   5.6.3.

To convert from UFI to GFI:

- Enter the UUT node list.

- Compile the UUT, selecting TROUBLESHOOT UUT for GFI.

- Since GFI may require node responses for both the probe and the I/O module, and since UFI may not have required both, additional response information may be required. The compiler will identify nodes where this additional information is needed. Use the response LEARN command to gather this information, then compile the UUT again.

USING THE GFI DATABASE WITH TL/1 FUNCTIONS   5.7.

TL/1 contains several commands that interact with the GFI database. These commands include *dbquery, gfi, count, level, sig, storepatt,* and *writepin.*

- The *dbquery* command allows a TL/1 program to retrieve information from the UUT database.

- The *gfi* commands allow TL/1 to interact with the resident GFI software.

- The *count, level,* and *sig* commands allow a TL/1 program to retrieve data for a pin.

- The *storepatt* and *writepin* commands allow a TL/1 program to overdrive a node with a sequence of patterns sent through the I/O module.

The *count, level, sig, storepatt,* and *writepin* commands all have a refpin option that allows information to be requested using reference designator pin names, such as "U1-b4" or "conn1-aa3". The option also supports numeric pin names, such as "U1-12".

If the refpin option is used, the UUT directory must contain a compiled GFI database (named UFIDATA or GFIDATA). The database contains information that allows the system to determine which physical pin corresponds to the pin name.

If you are using the resident GFI software, the database that you created to perform GFI contains all the information that the refpin option requires.

If you are not using GFI or UFI, you can create a minimal database by using the editor. This minimal database requires a reference designator list and part descriptions, but does not require a node list, stimulus programs, or response files. To create a minimal database:

1. Use the editor to create a reference designator list (REFLIST) for the UUT. The reference designator list should contain an entry for each reference designator that will be used in the refpin option. Fill in the REF and PART columns.

    The refpin option ignores the TESTING DEVICE column.

2. For each part that was named in the reference designator list, use the editor to create a part description in the part library. In the information window, fill in the NO. PINS and PKG fields. In the edit window, fill in the PIN NAME column if the part has pin names. If the part uses numeric pin names (such as "1", "2", "3", ...), leave this column blank.

    The refpin option ignores the pin type and related input pin information.

3. Use the editor to compile the UUT, selecting
    LEARN RESPONSES for UFI. The resulting
    database will contain all the information needed by
    the refpin option, although it will be inadequate to
    perform GFI.

Refer to previous paragraphs in this section for more detailed
information on how to edit the reference designator list and part
descriptions, and on how to compile the GFI database for a
UUT.

THE GFI USER INTERFACE   5.8.

At the operator's interface, GFI provides a summary that shows
an overall picture of what GFI has found and a suggestion list
that indicates the location at which backtracing should resume.
To invoke GFI at the operator's interface:

1. Press the GFI key on the operator's keypad and use
    the left arrow key to position the cursor at the left-
    most field.

2. Use one of the following softkeys to specify a
    command. Figure 5-40 shows example displays for
    complete commands. For details, refer to the
    Technical User's Manual.

    RUN: Executes GFI. Backtracing starts from the
    specified location.

    SUGGEST: Displays the GFI suggestion list that is
    generated from previous GFI activity and shows
    points at which backtracing can resume.

```
RUN GFI UUT DEMO REF U27 PIN 1

RUN    CLEAR   SUGGEST  SUMMARY   SETUP
```

Example 1: GFI RUN Command

```
                HINT U3-2
                HINT U16-3
                HINT U25-5
```

Example 2: Results of the GFI SUGGEST Command (suggestion list)

```
     REF    BAD INS    BAD OUTS    UNKNOWN
     U27      0           1           20
     U3       1           0           5
```

Example 3: Results of the GFI SUMMARY Command

Figure 5-40: GFI User-Interface Example Commands

SUMMARY: Lists the number of bad inputs, bad outputs, and untested pins of each component that has been tested. Figure 5-35 shows an example GFI SUMMARY display.

CLEAR: Erases the GFI summary and suggestion list. Also can be done by executing a TL/1 *gfi clear* command or by powering down the system.

SETUP: Enables or disables the automatic startup of GFI. Also can be done by executing a TL/1 *gfi autostart* command.

3.   If you pressed the RUN softkey, the display will instruct you to clip over or probe a component and press the device's Ready button when done.

4.   Follow the displayed instructions of step 3. Figure 5-41 shows examples of GFI recommendations (resulting from step 3), which could take one of three forms:

GFI accuses the probed component.

GFI recommends where next to probe. Use the left and right arrow keys to move the cursor to each pin of the displayed IC. A status message is shown for each pin. Use the up and down arrow keys to scroll through long messages.

GFI has no recommendations.

An operator who does not want to probe at the location recommended by GFI can use the keypad GFI RUN command to specify another location, based on knowledge of the UUT or information from the suggestion list.

Hints in the suggestion list can be generated by any programs that previously performed functional tests on the UUT. These hints are generated by the TL/1 *gfi hint* command.

```
┌─────────────────────────────────────────────┐
│  U27 is BAD or OUTPUT  ₂₈...¡....¡...↑.¹₅     │
│  U27-17 is LOADED      ▓▓▓▓▓▓U27▓▓▓▓▓▓        │
│                        ▓" " ¡ " " ¡" " "¹₄   │
└─────────────────────────────────────────────┘
```

Example 1: GFI Accuses Probed Component

```
┌─────────────────────────────────────────────┐
│  CLIP U27              ₂₀...▓...¡....¹₁       │
│                        ▓▓▓▓▓U27▓▓▓▓▓          │
│  BAD - DETAILS ↓       1" " "¡" " "¹₀         │
└─────────────────────────────────────────────┘
```

Example 2: GFI Makes Probing Recommendation

```
┌─────────────────────────────────────────────┐
│  NO RECOMMENDATION     ₂₀...¡...¡...¹₁        │
│                        ▓▓▓▓▓U27▓▓▓▓▓          │
│  GOOD AS OUTPUT        1▓" " ¡ " " "¹₀        │
└─────────────────────────────────────────────┘
```

Example 3: GFI Makes No Recommendation

Figure 5-41: GFI User-Interface Example Recommendations

Section 6
Terminal Emulator

The terminal emulator lets you use the programmer's interface as a 24-line by 80-column display terminal, to be connected to a remote computer through one of the serial ports. With this feature, you can transfer files between the 9100A and other computers (including other 9100A/9105A machines). Such transfers are useful for obtaining UUT information from CAD systems, for example.

ENTERING AND EXITING THE TERMINAL EMULATOR   6.1.

Before starting the terminal emulator, you must configure the serial port that you will use as a communication channel. The SETUP MENU key on the operator's keypad lets you set the baud rate, parity, number of data bits and stop bits, protocol for data flow control, and newline character. See the "Keypad Reference" section of the *Technical User's Manual* for more information about these port settings. The terminal emulator automatically operates in full duplex mode.

You should use flow control protocol to prevent the loss of characters at high baud rates. (The 9100A sounds a beep when it detects character loss.) The flow control protocol must match the setting of the computer at the other end of the communication

channel. The 9100A can implement software (XON/XOFF) protocol, hardware (CTS/RTS) protocol, or both.

You invoke the terminal emulator from the editor by pressing the TERM softkey and selecting the name (/PORT1 or /PORT2) of the serial port to use. To return to the editor, press the Quit key.

TERMINAL EMULATOR DISPLAY 6.2.

When you invoke the terminal emulator, the monitor is cleared. To turn on the information window, which is shown in Figure 6-1, press the Info key. You set the terminal mode and tab stops with this information window.

While the information window is on, you can press the Help key to display help information. If the information window is off, the Help key has no editor function, and pressing the Help key causes a special character code to be sent to the device that is connected to the terminal emulator.

When you turn the information window off (by pressing the Info key), the keyboard and display send and receive characters as a terminal.

The 9100A collects input characters in a buffer. If the buffer becomes full, the 9100A automatically sends the signal required by the flow control protocol to suspend input.

To exit the terminal emulator, press the Quit key. Pressing Shift and Quit simultaneously will exit the editor directly to the operator's keypad. Then the next time the editor is invoked, it will return to the editor screen that invoked the terminal emulator the previous time.

```
/PORT1 (TERM)
INSERT MODE: OFF        AUTO NEW LINE: OFF      LINE TERMINATOR: CR
AUTO WRAP: ON           TEXT CURSOR: ON         WAIT FOR TERMINATOR: OFF
       T       T       T       T       T       T       T       T       T
12345678901234567890123456789012345678901234567890123456789012345678901234567890




















F1    F2    F3        F4    F5    F6    F7        F8    F9    F10
RESTORE SAVE          RECEIVE SEND  ABORT
```
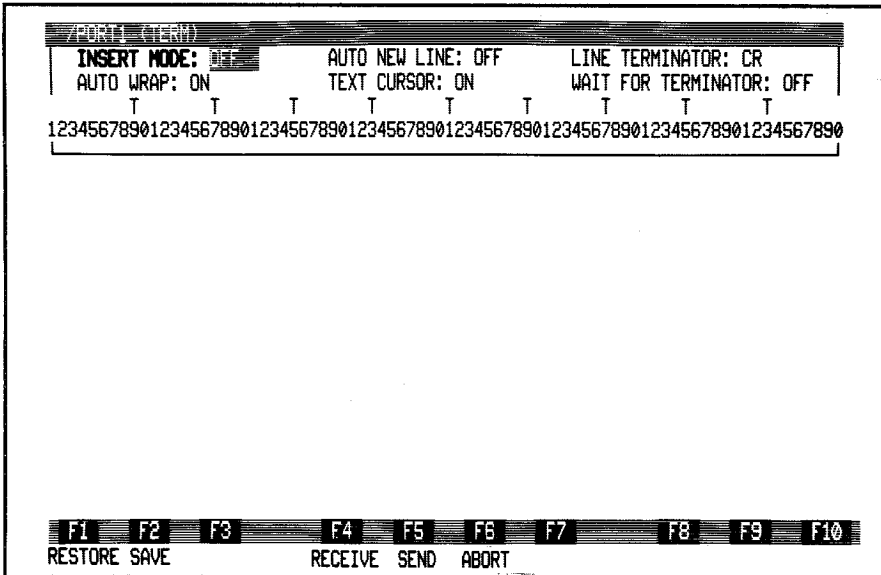
Figure 6-1: Terminal Emulator Screen Example

The following fields in the information window set the terminal modes. In all cases except for LINE TERMINATOR and WAIT FOR TERMINATOR, you press the Field Select key to set the field to either ON or OFF:

- INSERT MODE -If ON, an incoming character is inserted at the cursor location and the characters to the right are move one column to the right. If OFF, an incoming character replaces the character at the cursor location and moves the cursor one column to the right. The default setting is OFF.

- AUTO WRAP -If ON, when the cursor is at column 80, an incoming character moves the cursor to column 1 of the next line. If OFF, when the cursor is at column 80, it remains at column 80. The default setting is ON.

- AUTO NEW LINE - If ON, an incoming newline character moves the cursor to the beginning of the next line and the Return key outputs a carriage return and a linefeed. If OFF, an incoming newline character moves the cursor to the next line in the same column and the Return key outputs a carriage return only. The default setting is OFF.

- TEXT CURSOR - If ON, a blinking cursor appears in the display. If OFF, no cursor appears. The default setting is ON.

- LINE TERMINATOR - When using the send operation, this field determines whether output is terminated with a carriage return, a linefeed, or both. Press the Field Select key to set this field to CR, LF, or CR/LF. The default setting is CR.

- WAIT FOR TERMINATOR - This field contains one of four values: OFF, CR, LF, or CR/LF. When the setting is other than OFF, the emulator waits for a carriage return, linefeed, or carriage return then linefeed before sending the next line of output. The default setting is OFF.

- A tab setting is indicated by a "T" in the line above the column numbers line. Default tab stops are at every eighth column. To set or clear a tab setting, position the cursor at the desired column in the line above the column numbers and press the Field Select key. (When you move the cursor into the tab setting line, the cursor always moves to column 1.)

The INSERT MODE, AUTO WRAP, and TEXT CURSOR modes can be changed by the remote system through incoming escape sequences. These sequences are listed in the "Control Codes for Monitor and Operator's Display" appendix of the *TL/1 Reference Manual*.

TERMINAL EMULATOR OUTPUT   6.3.

The terminal keyboard sends all standard, seven-bit ASCII (ANSI X3.41-1974) key codes. To send the key codes from hexadecimal 00 to 20, type the control sequences shown in Figure 6-2. The softkeys, cursor control keys, and editor keypad keys (except for Quit and Info) send the ANSI-compatible escape sequences listed in Figure 6-3. The Quit and Info keys perform their regular functions.

Control Key Sequence	ASCII HEX	DEC	CHR
CTRL-@	00	0	NUL
CTRL-A	01	1	SOH
CTRL-B	02	2	STX
CTRL-C	03	3	ETX
CTRL-D	04	4	EOT
CTRL-E	05	5	ENQ
CTRL-F	06	6	ACK
CTRL-G	07	7	BEL
CTRL-H or Back Space	08	8	BS
CTRL-I or Tab	09	9	HT
CTRL-J	0A	10	LF
CTRL-K	0B	11	VT
CTRL-L	0C	12	FF
CTRL-M or Return	0D	13	CR
CTRL-N	0E	14	SO
CTRL-O	0F	15	SI
CTRL-P	10	16	DLE
CTRL-Q	11	17	DC1
CTRL-R	12	18	DC2
CTRL-S	13	19	DC3
CTRL-T	14	20	DC4
CTRL-U	15	21	NAK
CTRL-V	16	22	SYN
CTRL-W	17	23	ETB
CTRL-X	18	24	CAN
CTRL-Y	19	25	EM
CTRL-Z	1A	26	SUB
CTRL-[	1B	27	ESC
CTRL-\	1C	28	FS
CTRL-]	1D	29	GS
CTRL-^	1E	30	RS
CTRL-_	1F	31	US

Figure 6-2: Keyboard Control Sequences

Key Sequence	Escape Sequence
Up Arrow	ESC [A
Down Arrow	ESC [B
Right Arrow	ESC [C
Left Arrow	ESC [D
F1	ESC [1 0 ~
Shift-F1	ESC [1 1 ~
F2	ESC [1 2 ~
Shift-F2	ESC [1 3 ~
F3	ESC [1 4 ~
Shift-F3	ESC [1 5 ~
F4	ESC [1 6 ~
Shift-F4	ESC [1 7 ~
F5	ESC [1 8 ~
Shift-F5	ESC [1 9 ~
F6	ESC [2 0 ~
Shift-F6	ESC [2 1 ~
F7	ESC [2 2 ~
Shift-F7	ESC [2 3 ~
F8	ESC [2 4 ~
Shift-F8	ESC [2 5 ~
F9	ESC [2 6 ~
Shift-F9	ESC [2 7 ~
F10	ESC [2 8 ~
Shift-F10	ESC [2 9 ~
Edit	ESC [3 0 ~
Shift-Edit	ESC [3 1 ~
Msgs	ESC [3 2 ~
Shift-Msgs	ESC [3 3 ~
Help	ESC [3 4 ~
Shift-Help	ESC [3 5 ~
Begin File	ESC [3 6 ~
Shift-Begin File	ESC [3 7 ~
End File	ESC [3 8 ~
Shift-End File	ESC [3 9 ~
Scroll Forward	ESC [4 0 ~
Shift-Scroll Forward	ESC [4 1 ~
Scroll Backward	ESC [4 2 ~
Shift-Scroll Backward	ESC [4 3 ~
Begin Line	ESC [4 4 ~
Shift-Begin Line	ESC [4 5 ~
End Line	ESC [4 6 ~
Shift-End Line	ESC [4 7 ~
Field Select	ESC [4 8 ~
Shift-Field Select	ESC [4 9 ~

Figure 6-3: Keyboard Escape Sequences

The terminal accepts all standard, seven-bit ASCII key codes and recognizes a subset of the ANSI 3.64 terminal-control sequences. All of the key codes from hexadecimal 00 to 20 are ignored except for ESC (escape), CR (carriage return), LF (linefeed), BS (backspace) and HT (horizontal tab). These codes are interpreted as follows:

- CR (0D) moves the cursor to return to the beginning of the current line.

- LF (0A) moves the cursor to the next line in the current column if AUTO NEW LINE mode is disabled. LF moves the cursor to the beginning of the next line and scrolls the display when necessary if AUTO NEW LINE mode is enabled.

- BS (08) moves the cursor one column to the left on the current line.

- HT (09) moves the cursor forward to the next tab stop.

ANSI 3.64 terminal-control sequences are recognized by the terminal emulator. These include terminal-control features such as enabling or disabling the cursor, moving the cursor, enabling or disabling AUTO WRAP mode, enabling or disabling AUTO NEW LINE mode, changing display attributes (bolding, underscoring, blinking, and reverse video), and setting or clearing tabs. Also, included are terminal-control sequences for inserting or deleting a line, inserting or deleting characters, and block erasing functions. A complete listing of all available terminal-control features and their associated terminal-control sequences is provided in the "Control Codes for Monitor and Operator's Display" appendix of the *TL/1 Reference Manual*. The actions associated with these control sequences are defined in the ANSI standard.

FLOW CONTROL 6.5.

To suspend data input to the terminal, either press the Scroll
Lock key or type a CTRL-S (press the CTRL-Z key and the
letter "S" at the same time). To allow input again, either press
the Scroll Lock key a second time or type a CTRL-Q. The
9100A automatically sends the appropriate signal for the flow
control protocol you have specified. For example, in a
CTS/RTS protocol, typing CTRL-S causes the 9100A to assert
the CTS line; in an XON/XOFF protocol, typing CTRL-S
causes the 9100A to send a CTRL-S to the other computer.

TERMINAL COMMANDS
(SOFTKEY DEFINITIONS) 6.6.

The softkey commands described below are available only to the
terminal emulator:

- RESTORE: This command sets the terminal modes and tab
  settings to the default values stored on the system disk. (A
  restore command is performed automatically when you
  invoke the terminal emulator.)

- SAVE: This command saves the current terminal modes
  and tab settings on disk. These settings will be used as
  future default settings when the RESTORE softkey is
  pressed.

- SEND: This command prompts you for the full pathname
  of a text, program, or nodelist file. You can then select the
  format of the file using the Field Select key. If the file
  exists, it is output to the serial port using the current line
  terminator and wait-for-terminator settings. To stop the
  send operation, press the ABORT softkey or the Quit key
  on the programmer's keyboard.

- RECEIVE: This command prompts you for the full
  pathname of a text, program, or nodelist file. You can
  then select the format of the file using the Field Select key.

If the file exists, its contents are deleted and all input characters are stored in the specific file name. An incoming carriage return, line feed, or carriage return and line feed is stored as a single end-of-line character. To stop the receive operation and close the file, press the ABORT key.

- ABORT: This key stops a send operation or a receive operation in progress.

TRANSFERRING FILES TO AND FROM THE 9100A   6.7.

Through the terminal emulator, programs, text files, and nodelist files can be transferred between a 9100A and a remote computer. Uploading files from the 9100A allows you to modify programs and other files (part, response, REFLIST, NODELIST, etc.) on another computer. This frees the 9100A for debugging and executing programs.

Since some 9100A files (for example, parts and responses) are field oriented, modification on another computer should be done by first converting and uploading an existing file. This uploaded file can be used as a template for creating new files on another computer. The entries in field oriented files must be in the correct columns for the conversion from a text file to another type of file to be successful.

The following is the general transfer procedure. If you are transferring a text file, program, or node list, skip steps 1 and 6:

1. Convert the desired file to a text file.
2. Upload the text, program, or nodelist file to another computer.
3. Copy the file on the computer to a working file.
4. Make changes to the working file.
5. Download the working file to the 9100A.
6. Convert the downloaded text file to the proper type.

Converting Files for Uploading
from the 9100A   6.7.1.

Before you upload a file that is not a text file, program, or node
file, it must be converted to a text file.

Text files can be stored in a UUT or USERDISK directory and
copied to the part or program library directory using the full path
name specification. For example, a text file version of a part in
the PARTLIB named "7400" can be copied to the USERDISK
directory using the 9100A COPY (softkey F4) function as
shown in the following:

```
FROM NAME   /HDR/PARTLIB/7400   TYPE PART
TO NAME     /HDR/7400           TYPE TEXT
```

These procedures can be reversed for copying (converting) a text
file to a part in the part library.

To convert a file, perform the following steps:

1. From the editor enter the COPY (softkey F4)
    command.

2. At the FROM NAME prompt, enter the file name to
    be converted.

3. At the TYPE prompt, use the Field Select key to
    select the type (i.e. REF, PART, etc.).

4. At the TO NAME prompt, enter the new name of the file.

5. At the TYPE prompt, select TEXT using the Field Select key.

6. Press the Return key, and the file will be converted to a text file that may now be uploaded to a remote system.

General Upload Procedure   6.7.2.

Uploading from the 9100A requires the following steps:

1. Connect the 9100A RS-232 Port 2 and host device (see Figure 6-4) or modem (see Figure 6-5). Although both 9100A RS-232 connectors are wired as DTE, use Port 2 (the earth-referenced port) for connections to other computers.

2. Set up Port 2's baud rate, parity, data bits, stop bits, and other parameters using the SETUP MENU key for PORT2 on the operator's keypad. These parameters must match the setup of the remote computer. Use software handshake control (XON/XOFF ENABLED; CLEAR TO SEND DISABLED) for host computers and modems that support this. If your host does not support software handshaking, you must use CLEAR TO SEND ENABLED hardware handshake control (see Figure 6-6). The 9100A monitors CTS (pin 5) to see if the host device is asking the 9100A to hold sending of data. Typically this signal is DTR (pin 20) from the host.

3. Use the EDIT key on the operator's keypad to start the editor.

4. Select the appropriate UUT directory where the file is to exist after the downloading is complete (edit the appropriate UUT directory).
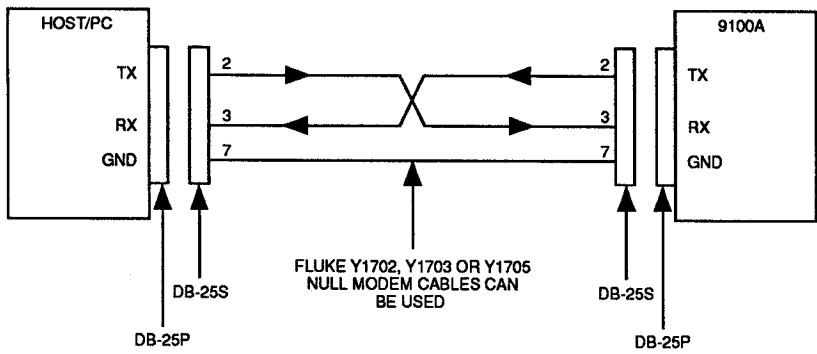
Figure 6-4: Host to 9100A Connections - XON/XOFF Control

MODEM

TX

RX

GND

2

3

7

9100A

TX

RX

GND

DB-25S

DB-25P

FLUKE Y1707 OR Y1708
INTERFACE CABLES
CAN BE USED

DB-25S

DB-25P
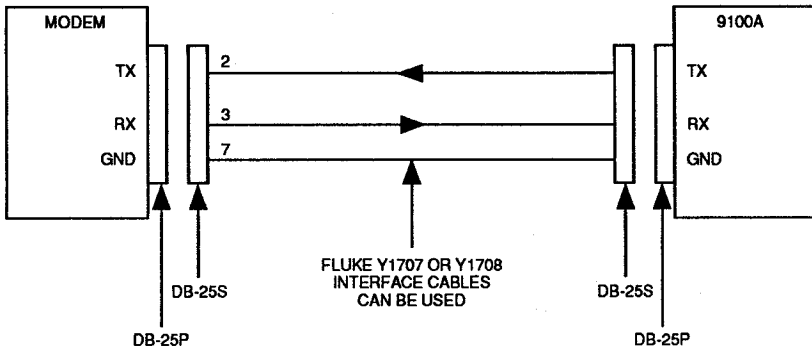
Figure 6-5: Modem to 9100A Connections - XON/XOFF Control
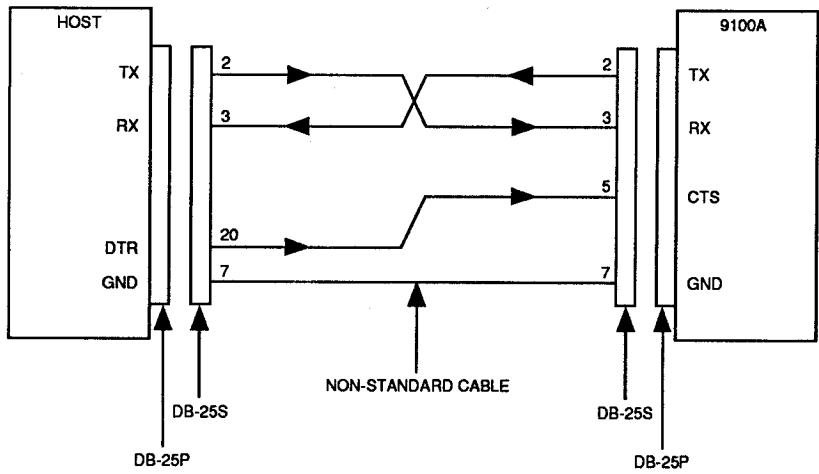
Figure 6-6: Host to 9100A Upload Connections -
Clear to Send Control

5. Use the TERM softkey to enter the terminal emulator; then use the Field Select key to select /PORT2. The terminal emulator starts with the screen that last selected TERM. (The screen is blank the first time TERM is selected after the 9100A is powered on.) Keys pressed on the keyboard are sent to the remote computer and characters received are shown on the monitor.

6. Using the terminal emulator keyboard, instruct the remote computer to receive the file to be uploaded. This is called "log" for some emulators.

7. Press the Info key on the keyboard. Change the LINE TERMINATOR in the top line of the Information Window as required by the remote computer (use right arrow and Field Select key). Then press the SEND (softkey F5) to select uploading. The SEND softkey prompts for the name of the file to be sent and then the type of file format to be sent. The file type is TEXT, press the Return key to start the transfer.

8. The file is transferred to the remote computer. The characters are not displayed unless the remote computer echoes them.

9. When the file has been transferred, the message "Send Completed" is displayed.

10. Press the Info key to get back to the terminal emulator mode. Send the file terminal character (eg. CTRL-Z) to close the file.

11. Press the Quit key to return to the UUT directory.

Uploading from the 9100A to a PC   6.7.3.

To upload a file from the 9100A to a PC, a terminal emulation
software package must be running on the PC. The 9100A is
used in the software handshake control mode (XON/XOFF
ENABLED). The PC and the 9100A are connected as shown in
Figure 6-4.

Set up the PC communication port (com1) to match the 9100A
default RS-232 parameters (9600 baud, no parity, 8 bits, and 1
stop bit) by initiating the following DOS "mode" command:

mode com1:96,n,8,1,p

The "p" flag sets the retry on timeout error mode. The flag
prevents the system from aborting the communication if the
9100A stalls the transfer to prevent overrunning its buffer.

Refer to the "General Upload Procedure" for the step-by-step
uploading procedure. The DOS "copy" command does not
work for uploading. This command does not perform the
needed hardware handshake to hold off the 9100A from sending
data to prevent the PC's buffer from overrunning.

Downloading Files to the 9100A   6.7.4.

The main benefit of downloading files is that a program or data
file can be written on another system, then transferred to the
9100A. Creating program or data files on another system frees
the 9100A to be used only for debugging and executing
programs. Text, program, and node files can be downloaded
directly to the 9100A. To download any other file type, it must
be downloaded as a text file and then converted to the
appropriate file type.

The general steps required to download files are as follows:

•   Transfer the file to the 9100A as a text, program, or node
         file.

- If required, use the COPY softkey to convert the text file to the desired file type. Refer to "Converting Files that have been Downloaded to the 9100A" for information on the conversion.

General Download Procedure   6.7.5.

Downloading to the 9100A requires the following steps:

1. Connect the 9100A RS-232 Port 2 to the host device (see Figure 6-4) or modem (see Figure 6-5). Although both 9100A RS-232 connectors are wired as DTE (data terminal equipment), use Port 2 (the earth-referenced port) for connections to other computers.

2. Set up Port 2's baud rate, parity, data bits, stop bits, and other parameters using the SETUP MENU key for PORT2 on the operator's keypad. These parameters must match the remote computer set up. Use software handshake control (XON/XOFF ENABLED, CLEAR TO SEND DISABLED) for host computers and modems that support this. If your host does not support software handshaking, you must use CLEAR TO SEND ENABLED hardware handshake control. The 9100A asserts DTR (pin 20) to hold up the host when needed. Connect this signal to the signal(s) of the host that will hold sending of the data (typically DSR, pin 6) shown in Figure 6-7.

3. Use the EDIT key on the operator's keypad to start the editor.

4. Select the appropriate UUT directory where the file is to exist after the downloading is complete (edit the appropriate UUT directory).
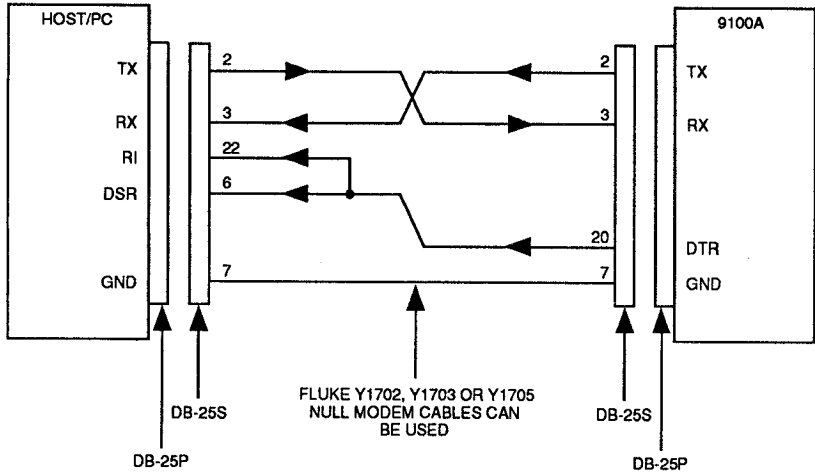
Figure 6-7: Host to 9100A Download Connections -
Clear to Send Control

5. Use the TERM softkey to enter the terminal emulator, and then use the Field Select key to select /PORT2. The terminal emulator starts with the screen that last selected TERM. (It will be blank the first time TERM is selected after the 9100A has been powered on.) Keys pressed on the keyboard are sent to the remote computer and characters received are shown on the monitor.

6. Press the Info key on the keyboard and then the RECEIVE softkey. Enter the name of the file that the download data should be stored in and press the Return key.

7. Use the Field Select key to select the desired file format (text, program, or node) and press the Return key.

   If the desired file format is not a text, program, or node, receive the file as text and convert it to the desired file type. Refer to "Converting Files that have been Downloaded to the 9100A" for information on the conversion.

8. The message "Receiving" appears at the bottom of the CRT. Press the Return key to remove the Info Window and initiate the actual download.

   The data scrolls by as the file is being received. At this point every character being sent from the remote computer to the 9100A is being stored in the specified file on the 9100A.

9. Using the terminal emulator keyboard, instruct the remote computer either to display the file to be downloaded or to otherwise send the file to its serial port. This command will be stored as the first line of the downloaded file since the receive operation has been started.

10. After the file has been transferred, press the Info key and use the ABORT softkey to stop storing characters into the file.

11. Press the Quit key to return to the UUT directory.

Downloading Files from a PC to the 9100A   6.7.6.

The following three methods can be used to transfer files from a PC to the 9100A:

- Using a terminal emulator with software handshaking. Insure flow control is setup for XON/XOFF.
- Using the DOS "print" command and software handshaking.
- Using the DOS "copy" command and hardware handshaking.

USING A TERMINAL EMULATOR WITH SOFTWARE HANDSHAKING

Set up the PC communication port to match the 9100A default RS-232 parameters (9600 baud, no parity, 8 bits, and 1 stop bit) by initiating the following DOS "mode" command:

mode com1:96,n,8,1,p

The "p" flag sets the retry on the timeout error mode. The flag prevents the system from aborting the communication if the 9100A stalls the transfer to prevent overrunning its buffer.

If you use a terminal emulator software package, the procedure is exactly as outlined in the "General Download Procedure" using software handshake control (XON/XOFF ENABLED). Most PC terminal emulator software packages have a "send file" capability. Be sure to setup the file transfer to be an "ASCII" transfer.

SENDING FILES TO THE 9100A USING THE
DOS "PRINT" COMMAND

Sending files to the 9100A using the DOS "print" command requires the following steps:

1. Set up the PC communication port to match the 9100A default RS-232 parameters (9600 baud, no parity, 8 bits, and 1 stop bit) by initiating the following DOS "mode" command:

   mode com1:96,n,8,1,p

   The "p" flag sets the retry on the timeout error mode. The flag prevents the system from aborting the communication if the 9100A stalls the transfer to prevent overrunning its buffer.

2. Connect the 9100A RS-232 Port 2 and the PC's communication port (com1) as shown in Figure 6-4.

3. Use the SETUP PORT2 XON/XOFF ENABLE and SETUP PORT2 CLEAR TO SEND DISABLE commands to enable software handshake control, and disable hardware handshake control of the 9100A.

4. Follow steps 3 through 7 in the "General Download Procedure". From the PC, enter the following command to send the file to the 9100A:

   print filename

   where filename is the name of the file that is to be sent to the 9100A. The first time the DOS "print" command is used, it asks:

   Name of list device [PRN:]

   Type: com1

   then press the ENTER key.

5. After the file has been transferred, press the Info key and use the ABORT softkey to stop storing characters into the file.

6. Press the Quit key to return to the UUT directory.

SENDING FILES TO THE 9100A USING THE DOS "COPY" COMMAND

Sending files to the 9100A using the DOS "copy" command requires the following steps:

1. Set up the PC communication port to match the 9100A default RS-232 parameters (9600 baud, no parity, 8 bits, and 1 stop bit) by initiating the following DOS "mode" command:

   mode com1:96,n,8,1,p

   The "p" flag sets the retry on timeout error mode. The flag prevents the system from aborting the communication if the 9100A stalls the transfer to prevent overrunning its buffer.

2. Connect the 9100A RS-232 Port 2 and the PC's communication port (com1) as shown in Figure 6-6. The DTR line (pin 20) from the 9100A must connect to the PC's DSR (pin 20) and RI (pin 22) lines to allow the 9100A to hold the transfer when needed.

3. Use the SETUP PORT2 XON/XOFF ENABLE and SETUP PORT2 CLEAR TO SEND ENABLE commands to enable software handshake control, and enable hardware handshake control of the 9100A.

4. Follow steps 3 through 7 in the "General Download Procedure". From the PC, enter the following command to send the file to the 9100A:

    copy filename com1:

    Where filename is the name of the file that is to be sent to the 9100A.

5. After the file has been transferred, press the Info key and use the ABORT softkey to stop storing characters into the file.

6. Press the Quit key to return to the UUT directory.

Converting Files Downloaded to the 9100A   6.7.7.

Command information in a downloaded text file should be removed before the text file is converted to a file type that can be edited on the 9100A. Removing the command information involves editing the text file and removing lines at the beginning or end of the file that were not part of the downloaded file. If a terminal emulator "send file" sequence, or a DOS "copy" or "print" command from a PC is used, command information does not have to be removed, because no command information has been received.

To convert the text file to another type of file, perform the following steps:

1. Edit the UUT that contains the text file you wish to convert to another file type.

2. Press the COPY (softkey F4). At the FROM NAME prompt, enter the name of the text file to be converted. At the TYPE prompt, press the Field Select key to select the file type TEXT.

3. At the TO NAME prompt, enter the name of the file after conversion. At the TYPE prompt, press the Field Select key to select the file type (most likely PROGRAM).

4.   You can now use the 9100A's editor to edit the new
      file. Any conversion error is flagged within the
      newly created file. Any errors may be corrected
      offline using a remote system, then repeating the
      download/conversion process. Errors can also be
      corrected directly on the 9100A, then
      converted/uploaded (see the following paragraphs for
      further information) to the remote system if you want
      to have the corrected version on the remote system.

USING THE 9100A BULLETIN BOARD   6.8.

9100A users who have purchased a Software Support
Agreement have access to the Electronic Bulletin Board. Access
to the Electronic Bulletin Board allows 9100A users to
send/receive mail from other users and upload/download files
(programs, parts, etc.) that have been posted to the Electronic
Bulletin Board. Information about training and new 9100A
products are also posted.

The following paragraphs describe how to log in, download
files from, and send files to the Electronic Bulletin Board.

The Electronic Bulletin Board operates at 1200 baud, no parity,
8 data bits, and 1 stop bit.

Logging into the Bulletin Board from the
9100A Terminal Emulator   6.8.1.

To log into the Electronic Bulletin Board use the following steps:

1.   Connect the 9100A RS-232 Port 2 to a modem as
      shown in Figure 6-5.

2.   Set up the 9100A's RS-232 Port 2 to 1200 baud, 8
      bits, no parity, 1 stop bit, and XON/XOFF enabled
      using the SETUP MENU key for PORT2 on the
      operator's keypad.

3. Move to the UUT directory that you are transferring files to or from by editing the UUT.

4. Start the terminal emulator by selecting the TERM (F5) softkey.

   A modem is required to dial the access number of the Electronic Bulletin Board. Type the following dial number command (Hayes compatible) to the modem, then press the Return key:

   ATD18008259100

   This dials 1-800-825-9100 (the telephone number of the Electronic Bulletin Board). Outside the U.S., the number is (206) 356-5957. The modem indicates when the modem link is established.

5. After the Electronic Bulletin Board connects, you are prompted for your first name, last name, and password. Enter this information and wait for the Electronic Bulletin Board main menu. At this point you have a series of options. The following sections describe only the (U) Upload and (D) Download options. All of the other options are menu driven and are self explanatory.

Downloading Files from the Bulletin Board to the 9100A   6.8.2.

To download a file to the 9100A from the Electronic Bulletin Board use the following steps:

1. Enter the D (Download) menu or the M (Mail System) menu.
2. If you are in the Download Menu, select the directory from which you wish to download. Now select the file number you wish to download.
3. If you are in Mail Menu, enter L (List) to obtain a list of messages available.

4. Enter the number of the message (file) you wish to download.
5. Enter Y to set up the download. In either mode (Download or Mail) you are now at the system prompt that asks for an Enter (Return) to begin the download.
6. Press the Info key on the 9100A Programmer's Keyboard. This brings up the soft key selections. Select the RECEIVE (softkey F4).
7. Enter the file name that will be created and press the Return key.
8. Use the Field Select key to select the file type (TEXT, PROGRAM, or NODE) and press the Return key.
9. The message "Receiving" appears at the bottom of the CRT. Press the Return key to remove the Info window and initiate the actual download. The data will scroll by as the file is being received.
10. When the transfer is complete, select ABORT (softkey F6) to close the file on the 9100A.
11. To sign off the Electronic Bulletin Board, initiate the hang up sequence.
12. Exit the terminal emulator by entering QUIT.

The received file will require some editing. Inspect the beginning and ending of the file and delete any bulletin board commands that have been added to the file.

Save this corrected text version of the file.

Refer to "Converting Files that have been Downloaded to the 9100A" to convert the text file types to a desired file type. Program, text, and node file types can be received directly without conversion, but all other file types must be received as text files, and then converted.

Uploading Files to the Bulletin Board
from the 9100A   6.8.3.

Text, program, and node files can be uploaded to the Electronic
Bulletin Board. To upload any other file type, it must first be
converted to a text file. Refer to "Converting Files that have
been Downloaded to the 9100A" for information on the
conversion.

To upload a file to the Electronic Bulletin, use the following
steps:

1. Log on to the Electronic Bulletin Board as shown in
    "Downloading Files from the Bulletin Board to the
    9100A."

2. Select the U (Upload) menu. Press U then the
    Return key.

3. Select U (Upload) to upload to the system operator,
    or select M (Mailbox) to upload to the system
    mailbox.

4. Enter a name for the file to be uploaded. The system
    is now ready to receive the file.

5. On the 9100A Programmer's Keyboard, press the
    Info key to bring up the Information Window and the
    soft key options. Change the Line Terminator in the
    Information Window to CR/LF (carriage return/line
    feed) by using the arrow key to move to the field and
    the Field Select key to change the field.

6. Press the SEND (softkey F5). Enter the name of the
    text file to be uploaded to the Electronic Bulletin
    Board and press the Return key.

7. Use the Field Select key to select the file format
    (TEXT, PROGRAM, or NODE) and press the
    Return key to initiate the transfer of the file. The data
    is not displayed as it is sent because the Electronic
    Bulletin Board does not echo them. When the
    transfer is complete, the 9100A reports "Send
    Completed".

8. Press the Info key to remove the Information Window.

9. Type a CTRL-Z to close the file that has been sent to the Electronic Bulletin Board.

If the file was uploaded to the system operator, the transfer is complete. If the file was uploaded to the mail system, continue with the following steps:

1. The system now prompts for a subject. Enter a suitable subject.

2. The system now prompts for the name of the person the file is to be sent to. Enter the first and last name on the same line. It must match exactly the name of user.

3. Sign off the Electronic Bulletin Board by initiating a hang up sequence.

4. Exit the terminal emulator by entering QUIT.

Section 7
CAD Translator

INTRODUCTION 7.1.

The CAD Translator (also referred to as CADTrans) is a software package that converts CAD output files into a 9100A/9105A node list and a reference designator list which are readable by Guided Fault Isolation (GFI), the 9100A editor, and other 9100A/9105A applications.

Section 7 begins with an overview of the CAD Translator. The output file downloading procedure is described using a step-by-step process that includes illustrations of the 9100A monitor screen. The optional files prompted by CADTrans are explained, and examples of name translation (aliasing) are included. The section ends with an explanation of the regular expression grammar used to match part and reference name patterns to transform CAD output file format to legal 9100A NODELIST and REFLIST format.

Figure 7-1 shows an overview of the CAD Translator process and the files which are associated with it. The CAD Translator translates CAD information from a CAD output file format into a 9100A/9105A-usable format. After the necessary CAD files have been downloaded to the 9100A/9105A (see paragraph 7.3), the CAD Translator is invoked to translate CAD system-specific output files to 9100A/9105A Reference Designator List (REFLIST) and Node List (NODELIST) formats. Because of the display limitations on the 9100A/9105A front panel, part names can be only 10 characters long and reference designators are limited to 6 characters. There are also limitations on the characters that are allowed. For example, the characters "-" and "\" are not allowed in a reference designator or a part name. The CAD Translator allows you to specify rules that help meet 9100A/9105A requirements for REFLIST and NODELIST files, and automatically truncate long reference designators and part names to their maximum length.

In addition to providing help when translating the CAD system output files to a format acceptable by the 9100A, the CAD Translator also helps speed programming by providing other features. The most useful of the features is part name aliasing. The part library contains information on device types and pinouts. The library is keyed by the part name as described in the REFLIST file on the 9100A. Because the device pinouts are the same for functionally identical parts using different technologies (i.e., "7400", "74LS00", "74C00", etc.), the 9100A/9105A can use a single entry in the library to cover all of the devices of a type (in this case a quad NAND gate). CADTrans assists its user by allowing aliasing of part names so all of the similar parts can be coerced to a single name. In this example, "7400" would be a likely library name choice.

Aliasing is a mechanism that allows for specification of a pattern as a search string (see paragraph 7.5) and another pattern as a replace string. Search patterns and replace patterns always come in pairs: one specifies the pattern to be searched for and the other is the type of replacement that is to done on that string. For example, the search pattern may be something like, "All parts starting with '74' and ending in numbers should be tagged." The replace pattern corresponding to the search pattern
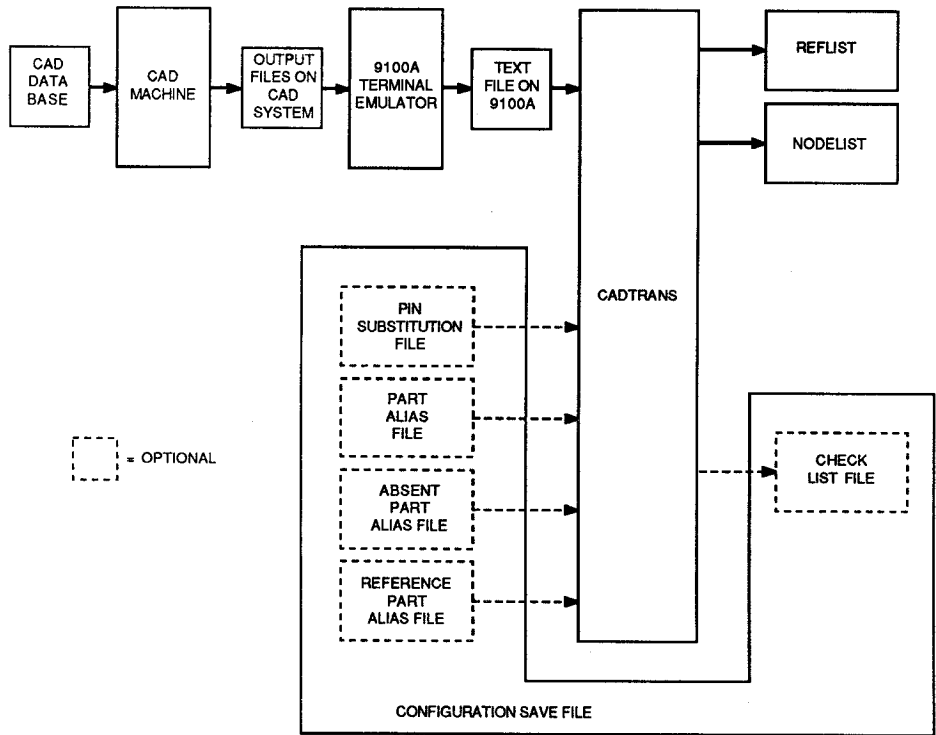
Figure 7-1: CADTrans Process

may be, "Throw away everything between the '74' and the ending numbers and write the result to the output file." The results of running the string "74ALS00" through this aliasing pattern would be "7400." Likewise, since we generalized the rule to handle all 74-series parts, the string "74LS273" would result in "74273", etc. Aliasing may be used to change reference designators to legal 9100A names, to change part names to legal 9100A names, and to assign part names to parts without names in the CAD source file based upon the reference designator. (Information regarding the details of aliasing can be found in paragraph 7.5.).

TRANSFERRING A CAD OUTPUT FILE
TO THE 9100A   7.3.

CADTrans expects the downloaded file to be available to the 9100A filesystem as a file of type TEXT. Refer to paragraph 6.7. for the general process of transferring a file from a remote host.

USING THE CAD TRANSLATOR   7.4.

CADTrans is executed by pressing the F9 key while editing a UUT directory. To execute CADTrans successfully, the UUT being edited must not contain a REFLIST or NODELIST, and a downloaded CAD file must be available. After the F9 key is pressed, you are prompted for input file names. The majority of the file names are optional. If you type the name of an input file that does not exist, or you type an output file with an invalid path name, CADTrans asks you to re-enter the file name. After entering the correct information, CADTrans automatically creates a REFLIST and a NODELIST that reside in that UUT directory. The following paragraphs summarize the input files that are prompted for and their use.

Required Inputs

The following inputs are required in order to use the CAD Translator:

- System Type
- Source File name

System Type - System Type is a prompt that requires the name of one of the supported CAD systems. No file name is required. Press the HELP key to obtain a summary of currently supported CAD systems. The name of the CAD system must be entered. (It must match the spellings used in the HELP window.)

Source File Name - The Source File Name is the downloaded text file from the original CAD system. CADTransrequires an unedited version, direct from the CAD system. This is the only required file, the following files are optional. As with all of the other input files, if a simple file name is entered, CADTrans will assume it is a file of type TEXT in the current UUT. If a file path is entered, CADTrans will look for a file of type TEXT at the location specified by the path.

Optional Files

The following files are optional for use with the CAD Translator:

- Configuration File
- Pin Substitution File
- Part Alias File

- Absent Part Name Alias File
- Reference Alias File
- Output Check List File
- Name of Configuration Save File

Configuration File

The configuration file is an optional file that allows you to execute a CADTransset-up configuration. It is a text file containing a number of keywords followed by file names, denoting which files CADTrans should use during its execution. The following example shows how a configuration file might appear when viewed with the editor:

```
SYSTEM¹ SCICARDS²
INPUT¹ main_brd³
PARTALIAS¹ parts⁴
PINSUB¹ ABSENTPARTS¹
REFALIAS¹
CHECKLIST¹ check⁴
```

[1] denotes keyword
[2] denotes name of CAD system
[3] denotes name of CAD output file
[4] denotes file name

If the name of the configuration file is entered at the configuration file prompt, CADTrans uses the filenames after the keywords as if you had entered them in response to a CADTrans prompt. The configuration file may be edited using the 9100A editor. The file names contained in the configuration file can be changed, but the keyword names must remain the same. If a keyword appears with no file name (as in Figure 7-2 for PINSUB, ABSENTPARTS, and REFALIAS), CADTrans

assumes that you are not using the option associated with the missing file name and continues its execution. If a keyword does not appear at all in the configuration file, you are prompted for the missing keyword during CADTrans start-up. Not all of the keywords shown in the Configuration File example need to appear in the configuration file, nor do they need to appear in any particular order.

Pin Substitution File

Most users will not need the Pin Substitution File. It is an optional file that is entered only if pin names are to be changed to legal 9100A numbers, or if you wish to change the pinout on a UUT part to match the definition in your part library. To create a pin substitution file, begin by editing a new text file. For each part for which you wish to change the pinouts, type the part name when prompted, followed by a space and a list of the pins. The order of the list determines the pin numbers assigned in the node list. The following example is an entry used to change the pin names on a 24 pin connector.

> 24-PIN   1A 2A 3A 4A 5A 6A 7A 8A 9A 10A 11A 12A \
>   1B 2B 3B 4B 5B 6B 7B 8B 9B 10B 11B 12B

If pin 1B appeared in the CAD output file, it would be assigned to pin number 13 in the 9100A node list, (1B is in the 13th position of the pin list). The "\" character is used to denote the continuation of a pin list to the next line. If you wish to include comments with the pin configuration, begin the comment line with a "!" character.

Part Alias File

The Part Alias file is an optional file that is used to change part names from the CAD system's naming conventions to legal part names which appear in the 9100A part library. Legal part names are 1 to 10 characters in length and may contain any of the following characters:

- A through Z (uppercase)
- a through z (lowercase)
- 0 through 9
- "_"
- "."

The part alias file uses regular expression grammar to match patterns in part names and reference designators and transform them into a different pattern recognizable by the 9100A.

Absent Part Name Alias File

The Absent Part Name Alias file is an optional file that fills in missing or absent part names in the CAD source file and uses the same format as the part alias file. The file looks at the reference name of parts with blank part names. A search string matches the reference name and the part name is filled in with the replacement string. This file is useful for capacitors, resistors, and other parts that are given reference designators like "C12" but no part description name.

Reference Alias File

The Reference Alias file is an optional file that modifies reference designator names to legal 9100A format. This file has the same format as the part alias file. For more information on alias file format see paragraph 7.5. Reference names are from 1 to 6 characters long, and may contain the same characters listed in paragraph 7.6. The characters must begin with a letter or digit, and are case-insensitive.

Output Check List File

If a text file name is entered, the Output Check Listfile is written in text format in the current UUT directory. The file allows you to check aliasing by writing out all part aliases, absent part aliases, and reference aliases that occur during the translation. Also the file allows you to quickly check for inaccurate or ambiguous aliases. Warning and error messages are written to this file and to the screen.

NOTE

To avoid errors when using GFI you should examine the output check list file when using CADTrans.

Name Of Configuration Save File

The Name Of Configuration Save file is an optional output file that contains all the file name parameters that are currently being used by CADTrans. A prompt "Name Of Config Save File" appears only if a configuration file name has not been previously entered at the beginning of CADTrans execution. If a file name is entered at the prompt, a configuration file is written by CADTrans in text format to the current UUT directory. The next time CADTrans is executed, you are only required to enter the configuration file name.

The three types of alias files (Part, Absent Part, and Reference) follow the same format of alternating search and replace lines. Comments are legal, and may be placed anywhere in the file if preceded by a "!" character. The following is an example of the alias file format:

```
! This is a comment
SEARCH <regular expression>
REPLACE <regular expression>
```

The keywords SEARCH and REPLACE must be all upper case, and a space must separate the SEARCH and REPLACE from the regular expression. The order of regular expressions determines the priority of the alias rules. The first rule that matches is used for the alias. The most specific aliasing rules should be listed first, continuing to the most general alias rules that affect the greatest number of parts listed at the end of the alias file. Alias files are created and edited using the 9100A editor, or downloaded using term. Alias files are of type TEXT, and there may be up to 50 alias rules associated with each alias file.

The following Part Alias file examples use rules (Figure 7-2) that change CAD output file part names to 9100A part names. You can use these alias rules as shown, modify them for your application, or create new ones of your own.

```
! Use this rule for converting LSxxx parts to
! 74xxx

SEARCH ^LS\([0-9]+\)
REPLACE 74\1

! Use this rule for converting ALSxxx parts to
! 74xxx

SEARCH ^ALS\([0-9]+\)
REPLACE 74\1

! Use this rule for converting SNxxx to 74xxx

SEARCH ^SN\([0-9]+\)
REPLACE 74\1

! Use this rule for converting HCxxx and
! HCTxxx to 74xxx

SEARCH ^HC[^0-9]*\([0-9]+\)
REPLACE 74\1

! Use this rule for converting parts ending in
! K to resistor (ex. 10K, 5K)

SEARCH K$
REPLACE RESISTOR

! Use these rules convert any 1, 2, or 3-digit
! number parts to RESISTOR
```

Figure 7-2: Part Alias File Examples

```
SEARCH ^[0-9]$
REPLACE RESISTOR

SEARCH ^[0-9][0-9]$
REPLACE RESISTOR

SEARCH ^[0-9][0-9][0-9]$
REPLACE RESISTOR

! Use these rules for converting names with uF
! or pF to CAP

SEARCH uF
REPLACE CAP

SEARCH pF
REPLACE CAP
```

Figure 7-2: Part Alias File Examples (*continued*)

The Absent Part Alias file allows you to use SEARCH strings to locate the reference names and add part names that are missing from the CAD output file. You can use the following examples as a guide for writing rules to fill in missing part names in your alias file.

```
! Use this rule to convert reference names
! beginning with C and followed by numbers
! into CAP (example: C1, C10, and C113 to CAP.
! Missing part names only).

SEARCH C[0-9]+
REPLACE CAP

! Use this rule for reference names beginning
! in R to resistor.

SEARCH R[0-9]+
REPLACE RESISTOR
```

The Reference Alias file allows you to change CAD output file reference names to legal 9100A reference names. When making alias rules for the reference designators, make sure that each reference name is unique. Avoid making rules that are ambiguous. Also, determine if the proper aliases have been made in the output check file. You can use the following examples as a guide for reference aliases:

```
! Use this rules to change illegal "-"
! characters to legal "_" characters.

SEARCH \([^\-]*\)-\(.*\)
REPLACE \1_\2

! Use this rule to change illegal "/"
characters! to legal "." characters.

SEARCH \([^\/]*\)/\(.*\)
REPLACE \1.\2
```

REGULAR EXPRESSIONS 7.6.

Regular expressions are used in alias files to match patterns in part names and reference designators and transform them into a different pattern recognizable by the 9100A. The regular expression analyzer implemented in CADTrans can describe almost any set of characters possible. If you are just beginning to use regular expressions, they are used at a simplified level. The following regular expressions progress from the simplest expressions to more advanced expressions.

The simplest form of regular expression is a direct, one to one correspondence between the SEARCH expression and the part. For example, the alias entry:

```
SEARCH ALS00
REPLACE 7400
```

The search statement finds every part with the identification ALS00 and replace it with 7400. This accomplishes the required replacement, but a more general regular expression would cover every ALS part and not just ALS00:

```
SEARCH ALS\([0-9]+\)
REPLACE 74\1
```

Notice how this alias rule finds any part with a name containing ALS and replaces it with 74 along with the part number found after the ALS. The grouping characters "\(\)" identify the portion of the string used in the REPLACE statement; the [0-9] identifies that portion as one or more numeric characters; and the \1 appearing in the REPLACE string directs the new part name to be filled with the contents of the grouping characters.

```
SEARCH \([^\-]*\)-\(.*\)
REPLACE \1_\2
```

In the search and replace strings above, the SEARCH string first reads all characters that are not a "-" character ("not" is denoted by the "^" character preceding the "\" character).

NOTE

*A "\" character is placed before the "-"
character as a literal and not a set range of
characters delimiter.*

When the search string finds a "-" character, it is read in and
ignored, because it is not contained in a \(\) grouping. Trailing
characters are read into a second grouping so they may be
restored in the replace string. The replace string recalls what
was read into the first grouping with a \1, followed by a "_"
character to replace the "-" character. \2 follows the "_"
character and recalls the last grouping read by the search string.

The types of characters used in a regular expression (including a
description of each) are listed in Figure 7-3. With the
description is an example to show how the character is used to
successfully write rules to convert CAD output file format to
9100A file format.

Character	*Description*
| char | Matches itself, unless it is a special character (metacharacter): ".", "\\", "[", "]", "*", "+", "^", "$".<br><br>Example: Convert 74S00 to 7400<br><br>```
SEARCH 74S00
REPLACE 7400
``` |
| \\ | Matches the character following it, except when followed by a left or right parentheses ( ), a digit (1 to 9), or a left or right angle bracket < >. A "\\" character can be used to literalize a character, such as itself for searching.

Example: \\\\ searches for \\ |
| * | Any regular expression listed in this example group followed by the closure character "*", matches zero of more occurrences of that form. |
| + | Same as "*" above, except it matches one or more expression listed in this example group. Used in alias files to specify the search and replace rules.<br><br>Example: Convert 74S00 to 7400<br><br>```
SEARCH 74S0+
REPLACE 7400
! Replaces one or more
! occurrences of 0 with 7400
``` |

(continued on the next page)

Figure 7-3: Regular Expression Characters

[set]	Matches one of the characters in the set. If the first character in the set is "^", it matches a character not in the set. A shorthand [E-S] is used to specify a set of characters E up to S, inclusive. The special characters "]" and "-" have no special meaning if they appear as the first characters in the set.
	Example: Convert 74S00, 74LS00, 74C00 to 7400
| | ```
SEARCH 74[SLC]+00
REPLACE 7400
``` |
[a-z]	Matches any lowercase alpha.
[^]-]	Matches any character except "]" and "-".
[^A-Z]	Matches any character except uppercase alpha.
[a-zA-Z]	Matches any alpha.
\(\)	A regular expression listed in this example shown as: \(form\), matches what the expression "form" matches. The enclosure creates a set of tags, used for replacement. Replacement enclosures are numbered starting from 1.

Figure 7-3: Regular Expression Characters (*continued*)

\#

A "\" followed by a digit 1 to 9 matches whatever a previously enclosed \(\) search regular expression matched.

Example: Convert 74Sxxx, 74LSxxx, 74Cxxx to 74xxx

```
SEARCH 74[SLC]+\([0-9]+\)
REPLACE 74\1
```

^$

A regular expression starting with a "^"character and/or ending with a "$" character, restricts the pattern matching to the beginning of the line, or the end of line. The "^" and "$" characters are treated as ordinary characters in any other location of the pattern.

Example: Convert LS00 to 74LS00 without confusing an existing 74LS00. To obtain an incorrect result of 7474LS00.

```
SEARCH ^\(LS[0-9]+\)
REPLACE 74\1
```

\<\>

Matches where a word is delimited by whitespace.

Example: In the sentence, the six brown mice are in a row. Row appears twice, but you want to match on the word row itself, use \<row\>.

Figure 7-3: Regular Expression Characters (*continued*)

To use CADTrans effectively, first use CADTrans on the CAD output file with no file modifications. Then, if any errors occur, add a part alias file or a pin substitution file that contains rules to correct these errors. The translation becomes an iterative process consisting of translating the output file, identifying the errors, creating rules to correct the errors, and repeating the process until all errors are corrected. The reference designator list and node list are the result of executing CADTrans. The node list is checked first for any errors during the translation process. Correct the errors using the optional aliasing files. Once the node list requires no further modification, the reference designator list may need some changes to alleviate possible errors with part name syntax. If you notice part name errors that occur frequently, (for example: ALS00 instead of 7400), the following example solves this error:

1. Edit the part alias file.

2. Type in:

```
SEARCH ^[ALS]+\([0-9]+\)
REPLACE 74\1
```

The search string looks for one or more occurrences of ALS and one or more occurrences of a number 0 through 9. The replace string converts the CAD output file format into a format recognizable by the 9100A. If a part name error occurs only once or twice in the entire reference designator list, it is easier to change the error manually, than to create a new rule.

NOTE

When making manual changes to reference designator names in either the node list or reference designator list, make sure the same change is made in both the node list and the reference designator list.

Futurenet*, Scicards®, and Cadnetix* are the CAD systems supported by CADTrans. The output files of these systems are downloaded to the 9100A using the terminal emulator (the TERM softkey in the USERDISK directory screen or a UUT directory screen).

NOTE

The output from the CAD systems must not be altered. Any changes in the file output format could cause CADTrans to fail.

Each CAD system paragraph contains an example of an output file. Since CAD systems produce many output files, compare the CAD output file with those in the following sections to make sure you are using the correct file from your CAD system.

---

* The following are trademarks of their respective companies: Cadnetix of Cadnetix Corporation, Futurenet of Futurenet Corporation.

Scicards is a registered trademark of Scientific Calculations Incorporated.

Futurenet

CADTrans supports the NETLIST format created by Futurenet.
Refer to your Futurenet manual for specific instructions on
creating NETLIST output files. The following is an example of
Futurenet output file format:

```
NETLIST,2
(DRAWING,\DASH3\BOARDEX1.DWG,1-1
DATA,0,SHT-2
DATA,0,RAM 8K x 16 @ 00000
DATA,0,ADDRESS DECODER
)
(SYM,1-1,82
DATA,2,C13
DATA,26,1
DATA,24,4
)
(SYM,1-1,17
DATA,2,U12
DATA,3,ALS00
DATA,103,1
)
(SIG,,RESET,1,,
PIN,1-1,25,C13,23,1
PIN,1-1,41,U12,23,2
)
(SIG,,+5,1,,
PIN,1-1,25,C13,23,2
PIN,1-1,33,U12,23,1
)
```

Scicards

CADTrans supports output files from the Scicards system when the "LIST PINS FULL ALL" option is used. The x-y coordinate positions are not necessary, and are ignored. The following data is an example of Scicards output:

```
SN7414      U1       1    41    0.3125    5.1125
SN7414      U1       2    28    0.3125    5.0125
SN7414      U1       3    18    0.1325    4.9125
SN7414      U1       4     5    0.3125    4.8125
SN7414      U1       5    55    0.3125    4.7125
SN7414      U1       6    53    0.3125    4.6125
SN7414      U1       7    12    0.3125    4.5125
SN7414      U1       8    13    0.6125    4.5125
SN7414      U1       9    22    0.6125    4.6125
SN7414      U1      10     1    0.6125    4.7125
SN7414      U1      11    49    0.6125    4.8125
SN7414      U1      12    57    0.6125    4.9125
```

Cadnetix

The Cadnetix compilation process produces a standard parts list and a net list. The following shows a sample Cadnetix output file:

```
PARTS LIST

TESTPAT,1      TESTPAT,1      TP101
C1206V,0.01UF             X    C10 C11 C12 C13 C14 C15
R1206V,200K              X    R20 R21 R22 R23 R24 R10
SO20V,IC,ALS244          X    U1 U2 U18
SO16V,IC,ALS668          X     U13 U22
EOS
NET LIST
NODE      1 $ ***050157-1-3
                U18 1 U13 10
NODENAME NC $
                C10 1 C13 1 C15 1 R20 1 R23 1 U1 8 $
                U2 3 U18 8
NODE      2 $ ***050158-1-2
                R10 2 C11 2 U18 6
EOS
```

Section 8
Glossary

If you cannot find a term in the glossary, search the index for a reference to that term.

Active Edge
A signal transition used to initiate action.

Address Decoding
The conversion of address bits into a signal that activates a component or components.

Address Mapping
The correspondence between addresses and components in the UUT.

Aliasing
A condition where a component address responds to more than one combination of address bus bits.

Assert
To cause a signal to change to its logical "true" state.

Asynchronous
Not synchronized to the microprocessor or not synchronous to any clock signal.

Automated Test

An automated activity that verifies the correct operation of a circuit by comparing its output to the expected output.

Automated Troubleshooting

An automated process of locating a fault on a UUT.

Backtracing

A procedure for locating the source of a fault on a UUT by checking logic along a logical path from bad outputs to bad inputs until the point where no bad inputs are found.

BASIC

An acronym for Beginner's All-Purpose Symbolic Instruction Code.

Bit Logical

Considering each bit of a value, rather than the value as a whole, to perform a logical operation.

Block

A group of program lines delimited by a beginning statement and an ending statement.

Buffer

1. In software, a storage area for holding characters until a device is ready to accept them. 2. In hardware, a component that drives an output identical to its input. A hardware buffer provides electrical separation between two or more other components.

Built-in Test

Functional tests built into the 9100A/9105A that test the bus, ROM, and RAM.

Bus

A group of functionally similar signals.

CAD

An acronym for Computer-Aided Design. CAD systems let the user create, manipulate, and store designs on a computer.

Case-Sensitive
Capable of distinguishing between upper-case and lower-case characters.

Channel
A means for communication of data from one location to another.

Comment
Text in a program that is not executed. A comment in a TL/1 program or a node list must begin with an exclamation point (!).

Component
A passive or active part on a UUT.

Conditional Branching
The execution of particular statements based on the value of a logical expression.

Continuation Character
In the editor, a character that indicates that the next line is a continuation of the same statement, not a new statement.

Control Line
A signal that comes out of a microprocessor and is used to control the UUT.

Control Sequence
A combination of the CTRL key and another key. When these keys are pressed simultaneously, a single key code is produced. Control sequences are noted in this manual as CTRL-S, CTRL-Q, etc.

CRC Signature
CRC is an acronym for Cyclic Redundancy Check. A CRC signature is a compression of a long data stream into a 16-bit number.

Cursor
A symbol on a display (usually a box or an underscore) that indicates where a typed character will appear.

Cursor Control
Mechanisms that control the location and movement of the cursor.

Data Bus
A set of signal paths on which parallel data is transferred between two or more devices.

Data Type
In TL/1, the available data types are numeric and string. Both numeric and string types can be used in arrays.

Declaration
A statement that sets scope, data type, or default value of a variable.

Default Value
The value given a variable if no other value is specified.

Device
1. Refers to the probe, an I/O module, a reference designator, or the pod. 2. Also used with I/O operations to specify a port or a disk drive.

DIP
An acronym for Dual In-line Package. A DIP has an equal number of pins on each of its long sides. See also SIP.

Directory
A collection of related sets of data (files, for example) on a disk.

Drivability
Testing whether lines can be driven to the appropriate active high or active low level.

Dynamic Coupling
Data in one memory location is affected by combinations of data in other memory locations.

Edge
The transition from one voltage level to a different voltage level.

Exerciser
See Fault Condition Exerciser.

Expression
A combination of symbols and names that can be evaluated (according to TL/1 syntax rules) to yield a value.

External Synchronization
Synchronizing a node response measurement using signals external to the pod.

Fault
A defect in a UUT that causes circuitry to operate in a manner that is inconsistent with its design.

Fault Condition
A recognition by the 9100A/9105A that a fault exists on the UUT.

Fault Condition Exerciser
A group of statements that attempts to repetitively reproduce the conditions that generate a fault condition. (Sometimes called just an "exerciser.")

Fault Condition Handler
A group of statements that is executed when a particular fault condition occurs. (Sometimes called just a "handler.")

Fault Condition Raising
The generation of a fault condition either from detecting a fault on a UUT or from using a TL/1 *fault* statement.

Feedback Loop
A circuit in which one or more outputs is routed to the circuit's input.

Fill-in Field
An area of the monitor or the operator's display, usually shorter than a single line, into which characters can be entered.

Forcing Line
Input to the microprocessor that forces it to a particular known state.

Format Picture
A character string for formatted input or output that represents the format for a single value.

Format String
One or more format pictures which represent the format for a series of values. See also Format Picture.

Functional Test
An activity that verifies the correct operation of a circuit by comparing its output to the expected output.

GFI
See Guided Fault Isolation.

GFI Summary
A record of the components that have been tested by GFI.

Global Variable
A variable whose name and value are valid inside and outside of the invocation of the block in which the variable is declared.

Guided Fault Isolation
An algorithm that uses backtracing to troubleshoot a UUT.

Handler
See Fault Condition Handler.

Hexadecimal
Pertaining to the base 16 numbering system. (Often abbreviated as "hex.")

I/O
An abbreviation for Input/Output. The transfer of data to and from devices other than the local memory of the microprocessor system.

I/O Module
An option for the 9100A/9105A that allows simultaneous stimulus or response for multiple points on a UUT.

Implicit Declaration
A declaration assigned to a variable if no explicit declaration is given. See also Declaration.

Invocation
The execution of a program, function, handler, or exerciser block. Each invocation maintains its own set of local variables.

Keyword
The name of a value used in keyword notation of a TL/1 command.

Keyword Notation
The specification of arguments by name and value, in an arbitrary order.

Label
A string that identifies a line.

Level History
A character string that represents a record of the logic levels measured at a point over a period of time. "1", "X", and "0" represent high, invalid, and low states, respectively.

Library
A directory that contains a collection of only a particular type of file. The 9100A/9105A uses four libraries: a part library, a program library, a pod library, and a help library.

Local Variable
A variable whose name and value are valid only for the invocation of the block in which the variable is declared.

Mask
A value where each logic "1" represents a bit that is to be acted on.

Monitor

A 24-line, 80-column display that connects to the rear panel of the 9100A/9105A.

Node

A set of points that are all electrically interconnected.

Node List

A file containing a description of the interconnection of all pins on a UUT.

Non-Printing Characters

ASCII codes that do not represent letters, numbers, or punctuation.

One's Complement

The result of changing every bit of a binary number to its complement value.

Operand

A value or expression that receives the action of an operator. See Operator.

Operator

1. A symbol that acts on one or more values or expressions to produce another value. 2. A person who uses the 9100A/9105A for testing or troubleshooting.

Operator's Display

Three-line display on the mainframe of the 9100A/9105A.

Operator's Interface

The operator's display and the operator's keypad.

Operator's Keypad

The set of keys on the front panel of the mainframe of the 9100A/9105A.

Overdriver

A circuit in the probe or an I/O module that forces a voltage level on the probe or a pin of the I/O module.

Part Description
A file that describes a component on a UUT.

Part Library
A library of part descriptions.

Pod Library
A library of pod descriptions, each of which contains a pod database and pod-related TL/1 programs.

Pod Synchronization
Synchronizing a node response measurement using signals generated by the pod to indicate the sampling time.

Positional Notation
The specification of command arguments without using keywords.

Priority Pin
A pin that the GFI program will test first if a particular node is bad.

Probe
A device that can stimulate and measure any single point on the UUT.

Program Library
A library of programs that can be called by any program in the userdisk.

Programmer's Interface
The monitor and the programmer's keyboard.

Programmer's Keyboard
The keyboard that connects to the side panel of the 9100A.

Raise
See Fault Condition Raising.

Reference Designator
A one to ten character string naming a component on the UUT.

Related Input Pin
An input pin on a part that affects an output pin on that same
part.

Response File
A file containing data generated by executing a specific stimulus
program to a UUT and measuring the responses for the
execution of the stimulus program.

RUN UUT Test
A feature that allows the normal operation of a UUT using its
own program.

Scope
The definition of a variable as being valid only within an
invocation block (local scope) or as being valid both within an
invocation block and outside it (global scope).

Selectable Field
An area of the editor's display, usually shorter than a single line,
whose contents can be selected from a limited number of choices
(by pressing the Field Select key).

Signature
See CRC Signature.

SIP
An acronym for Single In-line Package. See also DIP.

Softkey
A key that has its function determined by software.

Statement
In a program, a group of words and/or symbols that cause the
9100A/9105A to perform some action.

Stimulus Program
A program that exercises a circuit while responses of circuit
nodes are gathered to see if the circuit produces the expected
response.

String
A group of characters enclosed in double-quote characters (")
and manipulated as a single entity.

Subscript
A number that selects one dimension of an array.

Synchronous
Activated by transitions of a clock signal.

Termination Status
An indication of whether a program or function ended with
"passes" or "fails" as a result.

Timeout
A condition in which an expected event has not occurred within
the expected time period.

Toggle
To change to the complementary logic state.

Transition Count
A record of the number of times the logic level at a point changes
to the high state within a period of time.

Troubleshooting
A process of locating the area of a UUT that is causing a fault.

Userdisk
1. A diskette containing test programs and information about a
particular UUT. 2. The current disk drive that is used as a
source for UUT programs and data.

UUT
Unit Under Test. A physical item, i.e., a board or a system to
be tested.

UUT Directory
A set of files that contain information about a particular UUT.

Wildcard
A symbol that represents any sequence of characters. The
9100A/9105A uses the asterisk character (*) for this purpose.

Window
An area of the monitor reserved for certain information to be
displayed.

Index

Writing
    a TL/1 program, 3-9
    stimulus programs, 3-137, 3-139, 5-56

XON/XOFF, 6-2, 6-9

YANK, 2-41