Logic Engine User Manual

Caleb Hess, Steven D. Johnson, Robert W. Wehrmeister, Ingo Cyliax,

Revised 2001

1 Introduction 4
2 Installation 6
2.1 Introduction. 6
2.2 Before Installation. 6
2.3 Basic Installation. 7
2.4 Advanced Installation 7
2.5 Starting Up the Logic Engine Software 8
2.6 Installing Only the LE PI Library 9
2.7 Installing the LE Board. 9
3 Using the Logic Engine Board 12
3.1 Introduction. 12
3.2 Tie Points 12
3.3 Clock 12
3.4 Switches and Buttons 12
3.5 LEDs 14
3.7 Serial Port 14
3.8 Placing Sockets in the Prototype Area 14
4 LE Panel 19
4.1 Introduction. 19
4.2 How the LE Panel Tool Works. 19
4.3 The LE Panel Tool upon Startup 20
4.4 Status Field and Modes of Operation 21
4.5 Label and I/O Fields 22
4.6 Using Input Fields. 24
4.7 Switch Fields. 24
4.8 File I/O 25
4.9 Symbol Files 25
4.10 Key Bindings and Menu Selections 25
5 TERM 28
5.1 Introduction. 28
5.2 Menus and Key Bindings 28
6 Common User Interface 30
6.1 Introduction. 30
6.2 DESQview Interface 30
6.3 Menus 31
6.4 Dialog Boxes. 32
6.5 File I/O 32
6.6 Tools Menu 34
6.7 Help System 35
6.8 Communication with the LE Board 35
4 ED PLD 37
4.1 Introduction. 37
4.2 Cypress PLD C 20G10 37
4.3 EDPLD Fuse Map Editor 40

4.4 File I/O 41
4.5 EDPLD PLD Programmer 45
4.6 Testing a 20G10 45
4.7 Key Bindings 47
5 LE Assembler 50
5.1 Introduction 50
5.2 Editor 50
5.3 LEASM Debugger 53
5.4 A Design Example 54
5.5 Developing the Control Program 57
5.6 The Micro Assembly Language 61
I Logic Engine Programmer's Interface 79
A Introduction 79
B How the LEPI Works 79
C Low Level Interface Routines 80
D High Level Interface Routines 83
E Declaration File Syntax 84
F Nomenclature 87
G Linking the Library 89

1. Introduction

The Logic Engine consists of the Logic Engine Board and a set of software tools providing an interface to the board. This manual describes the use of the software tools. A description of the Logic Engine Board can be found in the Logic Engine Board Technical Reference Manual.

The Logic Engine Tool Set consists of four software tools, a library of routines which allow for the development of custom tools and an environment in which to run them. We have found this set of tools to be extremely useful in our senior level digital design course. Some of these tools were designed with the course explicitly in mind, while others were designed as general purpose tools. Below is a brief description of each tool:

LE Panel: The LE Panel tool is used to display information from the LE board on the PC screen and to send information from the PC keyboard to the LE board. It is in effect a virtual display which serves the same purpose as the lights, switches, and push buttons on the LE board, but which offers a more powerful interface. Signals wired to the lights, switches and push buttons of the LE board can be grouped and displayed in a number of formats. Signals wired from the switches and push buttons can be controlled from the PC.

LE Asmb: The LE Asmb Tool is a microcode development system. It consists of a text editor, microcode assembler, downloader, and debugger, all in a unified environment. Features include:

- a source level debugger with single stepping and breakpoints.
- an emacs-like editor.
- viewing of object code.
- a rich micro-assembly language.

ED PLD: The ED PLD Tool is a PLD fuse map editor for use with the PLD burner on the Logic Engine Board. ED PLD also has the capability to read, burn, verify, and test PLDs. ED PLD supports only the PLD20G10 at this time.

LE Term: LE Term is a simple terminal emulator for use with designs that require serial I/O.

LE PI: The LE PI is a library of routines for use with the Microsoft C[a] complier. With this library, users can develop specialized tools for use with their designs.

a.Microsoft C® is a trademark of Microsoft Corporation.

All the tools, with the exception of the LE PI, were written to run in the DESQview[1] environment.

1. DESQview is a trademark of Quarterdeck Office Systems.

They will not run unless DESQview is installed and running. The DESQview environment offers three major features utilized by the Logic Engine tool set: the ability to switch between applications, a standard way of dealing with windows and fields, and the ability to multitask.

2. Installation

2.1. Introduction

There are two ways in which the Logic Engine software can be installed. The basic method, in which only the Logic Engine Tool set is available from within DESQview or the advanced method, in which the Logic Engine Tool set is available along with other applications that have been set up for use with DESQview. If you are not currently using DESQview, it is recommended that the basic method be used. Later, when the full power of DESQview is desired, the Logic Engine software can be reinstalled using the advanced method. If you are currently using DESQview or wish to use all the power of DESQview immediately, use the advanced method of installation. If you wish to use only the LE PI tool to develop your own tools, you will not need to install DESQview or the Logic Engine software at all. See section 2.6 for the procedure to install only the LE PI tool.

2.2. Before Installation.

In order for the LE software tool set to run properly, your system should meet the following requirements:

IBM Personal Computer or 100% compatibles

(386 class machine recommended)

640K Memory minimum

(1M to 4M recommended)

1 Floppy drive, 1 Hard drive with 1M available

Monochrome or Color monitor

(VGA Color monitor recommended)

Mouse

(Optional but highly recommended)

PC-DOS 2.0-3.3 or MS-DOS 2.0-3.3

DESQview version 2.01 or greater

All DESQview requirements

(see the DESQview manual)

DESQview must be installed before any of the software tools can be used (with the exception of the LE PI tool.) To install DESQview, see the DESQview manual.

2.3. Basic Installation

Place the diskette labeled **INSTALL** in floppy drive **A** (other floppy drives can be used, simply use the appropriate drive letter instead of **A**)
Type **A:** and press ↵
Type **TYPE README** and press ↵for any last minute instructions or modifications to the manual
Type **INSTALL <dv> <le>** and press ↵,where:

- **<dv>** is the location where DESQview has been installed
- **<le>** is the location where the LE software tools are to be installed
- e.g. **INSTALL C:\DV C:\LE** ↵

If an error occurs during the installation, an appropriate error message will be displayed. After an error, installation can be continued by correcting the problem and restarting the installation.

During installation, messages will be displayed indicating the progress of the installation. When installation is complete, a message will be displayed indicating so. At this point the Logic Engine software is ready to run as described in section 2.5.

2.4. Advanced Installation

The advanced installation procedure will install the following programs into the DESQview Open Window menu with the key stokes listed below:

Program Name	*Key Stroke used to Open Program*
LE Panel	PA
LE ASMB	LE
ED PLD	ED
TERM	TR

If any of these key strokes interfere with a program that is already installed in DESQview, the advanced installation script (**ADVANCE.BAT**) can be edited to make the appropriate changes.

Performing the advanced installation procedure will create several new files in the DESQview directory and modify the file **DESQVIEW.DVO**. The old version of the file **DESQVIEW.DVO** will be stored in the file **DESQVIEW.OLD**.

The advanced installation procedure is as follows:
•Place the diskette labeled **INSTALL** in floppy drive **A** (other floppy drives can be used, simply use the appropriate drive letter instead of **A**)
•Follow the procedure for the basic installation (if this has not already been done)
•Type **ADVANCE <dv> <le>** and press ↵,where:

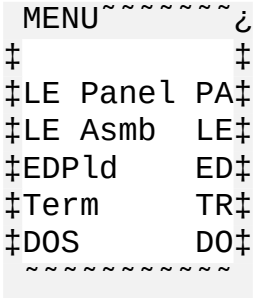
<dv> is the location where DESQview has been installed
<le> is the location where the LE software tools have been installed
e.g. **ADVANCE C:\DV C:\LE** ↵

During the advanced installation, messages will be displayed indicating the progress of the installation. If there is a conflict between one of the programs being installed and another program already installed, an error message will be displayed indicating that the program was not installed and installation will proceed with the next program. Once the advanced installation is complete, the programs will be available from the DESQview Open Window menu the next time DESQview is started.

2.5. Starting Up the Logic Engine Software

After installing the Logic Engine Software, you should do the following:
Add **<le>/bin** to the **PATH** environmental variable in autoexec.bat
Add **set LE=<le>** to autoexec.bat,where: **<le>** is the location of the installed Logic Engine Software
Reboot

If the Logic Engine software was installed using only the basic method, you will be using the Logic Engine Menu to start up the Logic Engine tools. To start the Logic Engine Menu:
Type **LE** ↵This will start up DESQview and bring up a menu as illustrated in Fig. 1. DESQview should not be running before the Logic Engine Menu is started. The menu has five selections, one for each of the Logic Engine tools and one for DOS. To start up one of these programs, select it in one of three ways.



Type the two letters on the right side of the selection.
eg. Type **PA** to select the **LE Panel Tool**.
Use the cursor keys to highlight the selection then press the Space Bar or Enter Key.
Use the mouse to highlight the selection then press the left mouse button.

If the Logic Engine menu is ever obscured by other windows, it can be raised by moving the mouse to a spot on the screen which is not occupied by a window and pressing the left mouse but-

ton.

If the Logic Engine software was installed using the advanced method, any of the Logic Engine Tools can be started from the DESQview Open Window menu. See the DESQview manual for details about starting programs from the Open Window menu.

2.6. Installing Only the LE PI Library

If you desire to use only the LE PI library, it is possible to avoid installing DESQview and the Logic Engine software. To be able to link your own programs to the LE PI library you need two files off of the **INSTALL** diskette

\bin\lelib.lib

\include\lelib.h

These can be copied to a convenient location on your hard disk. See Chapter 8 for details about using these files.

2.7. Installing the LE Board

In order to operate the LE Board in a stand-alone fashion (no host PC), the power supply must be connected to the LE Board (see below) and plugged in and turned on. In order to operate the LE Board with the host PC, the parallel connector must also be connected to the host (see below.) Fig. 2 illustrates these connections.

There are three connectors located on the right side of the LE Board (board oriented with switches face up and forward.) These connectors are, starting closest to the right most button and moving

away:

Parallel Connector:

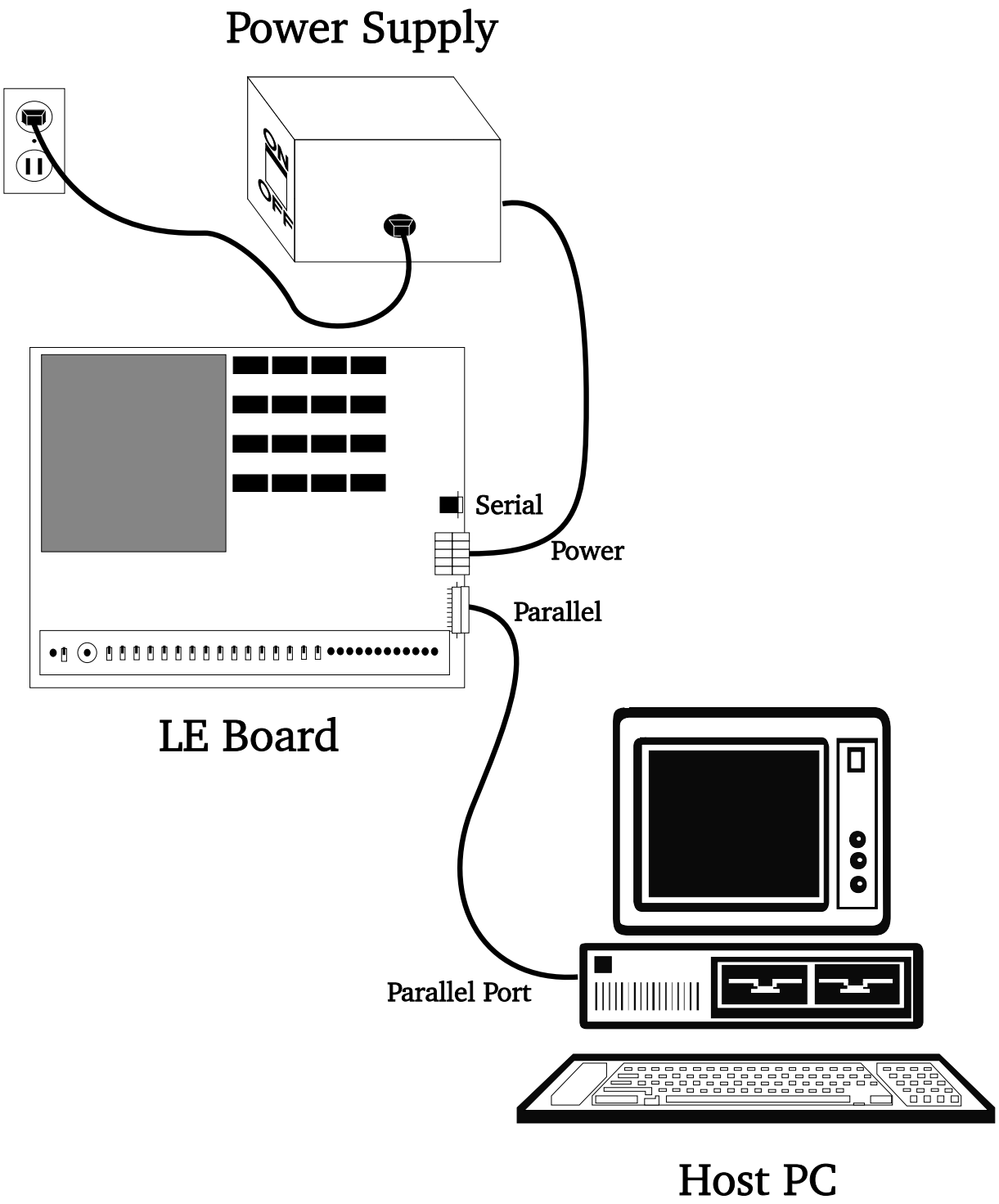
This is a male DB25 (25 pin) connector. It is connected to the host PC's parallel port (usually a female DB25 connector) via a male DB25/female DB25 parallel cable. This port is used for communication between the host PC and the LE Board. If any of the LE software is used to communicate with the LE Board, this connection must be made.

Power Connector:

This is a 5 conductor, unisex, color coded connector. It is connected to the power supply with an identical connector supplied with the power supply. This connection provides +5V, ground, -12V, +12V and 'power good' signals to the LE Board. The connection is made so that like colors of each connector match up. This connection must be made in order to use the LE Board.

Serial Connector:

This is a female DB9 connector. It is connected to the host PC's serial port (usually a male DB25 connector, sometimes a male DB9 connector) via a female DB9/male DB25(DB9) serial cable. This undedicated port is used to provide a serial port between the LE Board and the host PC. It is strictly for the designers use and is not used by any of the LE software.



3. Using the Logic Engine Board

3.1. Introduction

The Logic Engine Board has several features which aid in the testing and prototyping of chips and systems. The main features are: a general purpose prototyping area, a system clock, 32 switch and button inputs, 128 LED's for display of outputs, a micro-sequencer with 40 command bits, and a serial port. The following sections describe how to use these features.

3.2. Tie Points

Fig. 1 illustrates the Logic Engine board with all the user tie points highlighted and labeled. Figs. 2-4 are more detailed drawings of the board. The tie points are wire-wrap pins that can be used by simply wiring from the user design in the prototype area to the tie point. Table 1 below, lists and describes all the tie points on the board.

3.3. Clock

The clock supplied on the board is a variable rate clock with three selectable modes of operation. Each mode can be selected using the three position toggle switch located in the lower left corner of the board. The three positions are down: fast clock rate, up: slow clock rate, and middle: manual clock. The modes are also indicated by the lights above the switch: green: fast clock rate, yellow: slow clock rate and red: manual clock. The exact range of rates of the fast and slow mode can be selected by wiring from tie points C1 (fast) and C2 (slow) to one of the clock divisor tie points. This should already be set to some default configuration. In these two modes, the frequency can be adjusted with the clock pot. To increase the frequency, turn the pot clockwise, to decrease, turn it counter clockwise. When the clock is in manual mode, it is controlled by the push button in the lower left corner of the board. When the button is depressed, the clock is high, when released, the clock is low.

There are six tie points (U0 - U5) on the board for access to the user clock. They are all generated from the same signal run through separate buffers.

3.4. Switches and Buttons

There are 16 switch tie points (S0 - S15) and 16 button tie points (B0 - B15) on the Logic Engine board. All of the switches and 12 of the buttons can be controlled in manual mode from the switches and buttons across the front of the board. The 16 switches on the board starting on the

Table 1: Tie Points

Function	Tie Points	Input Output	Illustrated in Figure:	Description
Clock	D0-D23	Output	Fig. 2	Clock Divisor:
	C1-C2	Input	Fig. 2	Clock Selector:
	U0-U5	Output	Fig. 2	User Clock:
Switches	S0-S15	Output	Fig. 2	Switch Outputs:
Buttons	B0-B15	Output	Fig. 3	Button Outputs:
LED's	L0-L127	Input	Fig. 4	LED Inputs:
Micro-sequencer	P0-P39	Output	Fig. 4	Pipeline Outputs:
	PE.L	Input	Fig. 4	Pipeline Enable:
	MAP0-MAP11	Input	Fig. 4	Jump Map Inputs:
	JMAP.L	Output	Fig. 4	Jump Map Enable:
	CC.L	Input	Fig. 4	Condition Code:
Serial Port	DTR	Input	Fig. 4	
	TD	Input	Fig. 4	
	RTS	Input	Fig. 4	
	RD	Output	Fig. 4	
	CD	Output	Fig. 4	
	DSR	Output	Fig. 4	
	CTS	Output	Fig. 4	

right, are connected to tie points S0 - S15. The 12 buttons on the board starting on the right, are connected to tie points B0 - B5 and B8 - B13. In manual mode, when a switch is positioned toward the front of the board, the value on the correspoding tie point is low (0V). When positioned away from the front of the board, the value is high (5V). In manual mode, when a button is in the up position, the value on the correspoding tie point is low (0V). When in the down position, the value is high (5V).

When in host mode, all 32 of the switch and button tie points can be controlled from the host computer. In this case the switches and buttons on the board are disconnected from the tie points and have no effect on these signals. Refer to the proper chapter for information on how each software tool can control the switches and buttons.

3.5. LEDs

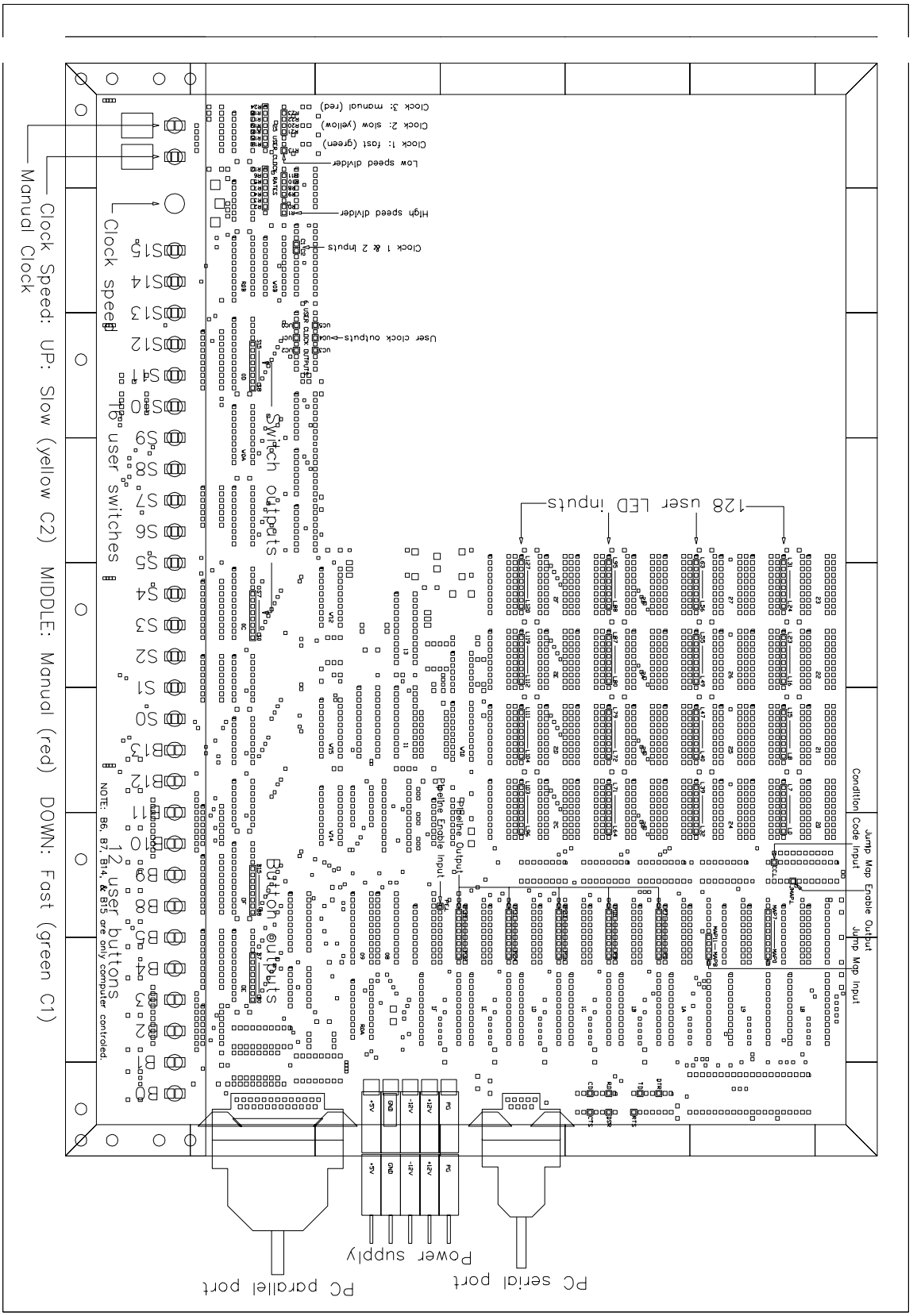
There are 128 LED's that can be used to display signals. They are used by wire wrapping the signal to be displayed to one of the LED tie points (L0 - L128). The values of the LED's and hence the value of any signal wired to an LED can be read by software when in host mode. Refer to the proper chapter for information on how each software tool can control the switches and buttons.

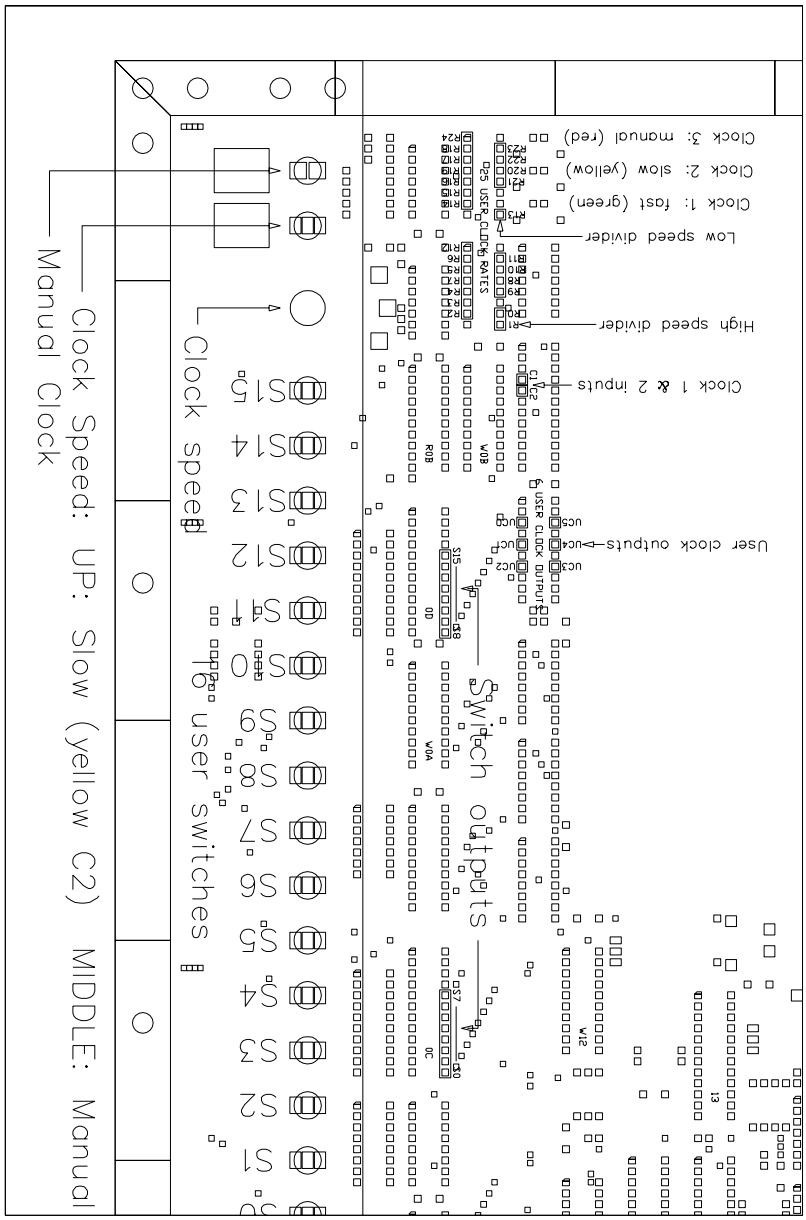
F. Microsequencer

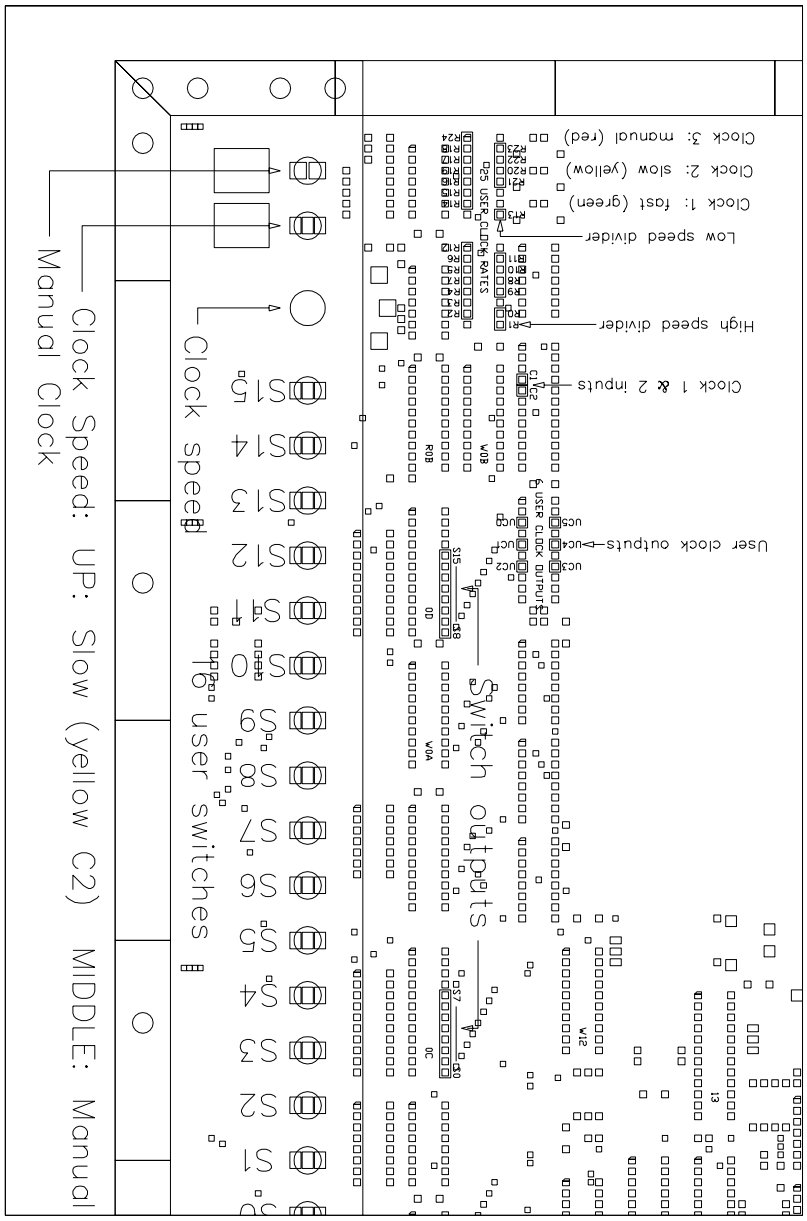
3.7. Serial Port

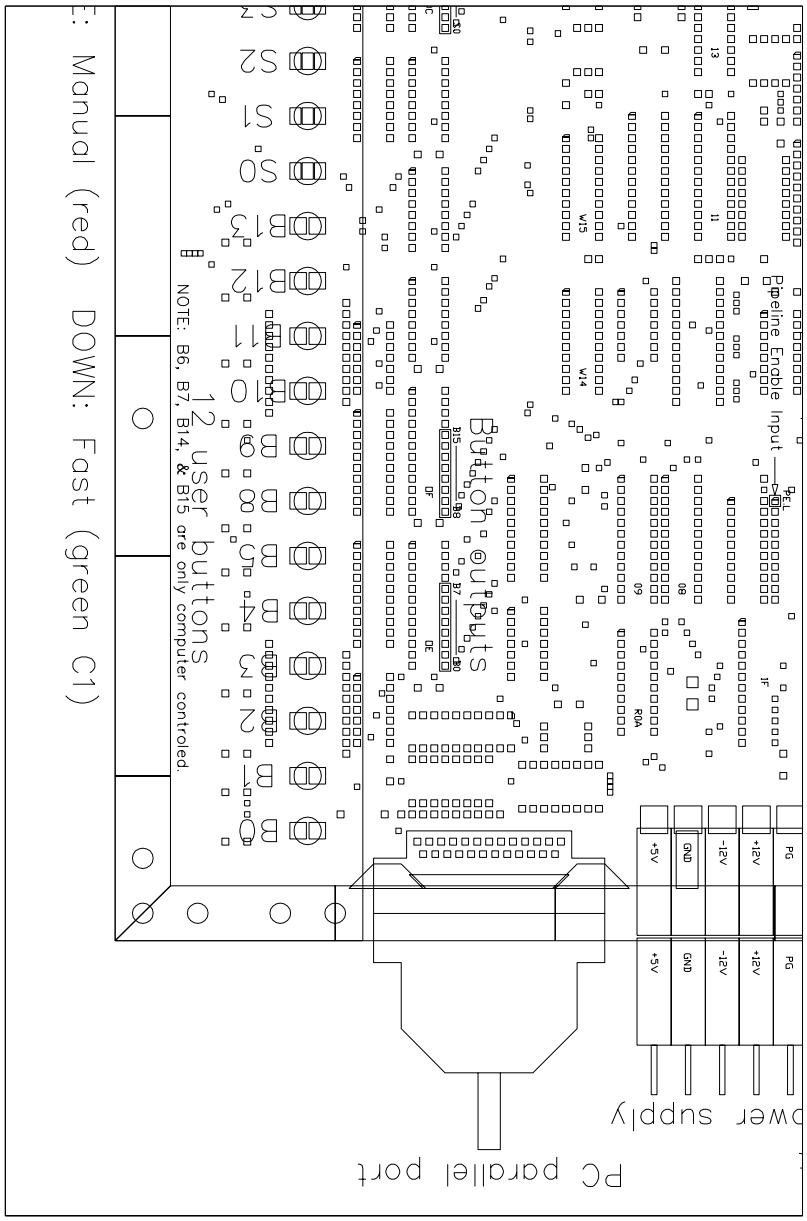
3.8. Placing Sockets in the Prototype Area

The prototype area has a power grid on each side. The top side has a grid of 5V. And the bottom side has a grid of 0V. To place a socket in the prototype area, insert the socket, and make a solder bridge from the Vcc pin to the grid on the top side and make a solder bridge from the GND pin to the grid on the bottom side. Alternatively, you can use stake pins. Solder these to the top and bottom grids and wire wrap from these to the socket.









4. LE Panel

4.1. Introduction.

The LE Panel tool is used to display information from the LE board on the PC screen and to send information from the PC keyboard to the LE board. It is in effect a virtual display which serves the same purpose as the lights, switches, and push buttons on the LE board, but which offers a more powerful interface. Signals wired to the lights, switches and push buttons of the LE board can be grouped and displayed in a number of formats. Signals wired from the switches and push buttons can be controlled from the PC. This capability allows the user to interact with the LE board entirely from the PC host. Displays can be constructed that are best suited to the current needs of the design. And since the display is easily reconfigured, the user can change the display as the need arises. Each configuration can be saved to disk to be retrieved at a later time.

4.2. How the LE Panel Tool Works

The LE Panel tool has two major tasks. The first is to take user input from the keyboard and react to it. The second is to read information from the LE board and display it on the screen. This is accomplished by having one process for each task. The **user input process** takes all input from the user, manages all the menus and windows and sends information to the LE board. The **panel update process** reads information from the LE board and displays it in the proper format in the panel window.

The **panel update process** can operate in two modes: continuous update or user update. In continuous update mode the panel update process is continuously running and updating the display. This gives the panel the feel of the light panel on the LE board since the signal values are displayed almost instantaneously. The rate at which the display can be updated depends on the speed and load of the host processor. In user update mode the user requests all updates. Each request causes the panel update process to perform one update of the display and then become dormant. Although this mode may not be very useful while debugging, it is handy to be able to turn off the panel update process while doing other tasks on the computer.

The LE Panel tool can also control the operation of the clock on the LE board. When in run mode the clock is controlled from the LE board. Alternatively, the clock can be pulsed from the panel. This will turn the clock off if it was in run mode and issue a single pulse of the clock. When the clock is in run mode and running at a high rate, the **panel update process** may not be able to keep up with the changing signals and the values displayed may be invalid.

The main part of the LE panel display consists of 24 label field, I/O field pairs. The label field of each pair is used to associate a name with a group of signals and the I/O field is used to display the group of signals in one of several formats. The I/O field can be configured to be an input field, in which case it is used to display the value of a group of switches or buttons from the LE board, or

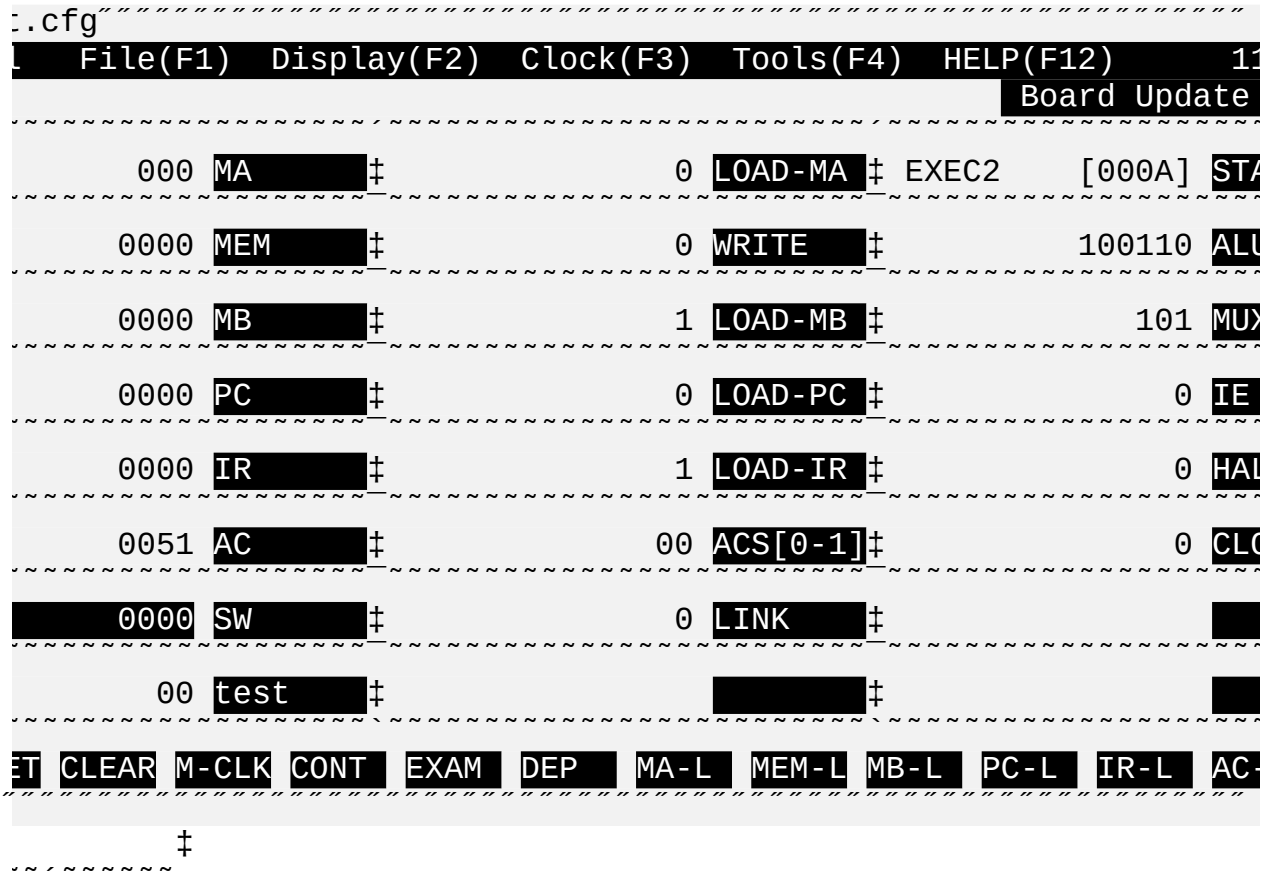
an output field, in which case it is used to display the value of a group of lights from the LE board. Input fields can also be configured to accept input from the user, thus overriding the physical switches and buttons on the LE board. The user input is then sent to the LE board in place of the switches and buttons.

From the perspective of the host, the switches and buttons behave identically and will both be referred to as switches. Associated with each switch is a register which can be written by the PC host. The output of each switch is connected to the output of its corresponding register. Normally, the output of the register is disabled and the output of the switch is enabled. The PC host can take control of the switches by disabling the output of the switches and enabling the output of the registers. In addition to the 28 registers associated with the switches, there are 4 hidden registers which are accessible from the PC host which do not have an associated switch or button.

In addition to the 24 label, I/O fields, the panel has 12 toggle input fields. Each can be mapped to one switch or button signal, creating a field that behaves like a toggle switch. Whenever one of these fields is selected it toggles the value of the signal to which it is mapped.

All communication between the LE board and the PC host is done over the parallel port. The LE board contains numerous registers to store data or to control the behavior of the board. All of these registers can be written or read. The LE Panel tool assumes that it is the only process that is writing to these registers, so when it puts the LE board in a certain mode of operation, it expects it to remain in that mode until otherwise instructed. If another process is also writing to the LE board, the LE Panel tool can get confused about the state of the LE board. It is always best to run only one application that accesses the LE board at a time.

4.3. The LE Panel Tool upon Startu



I/O Fields

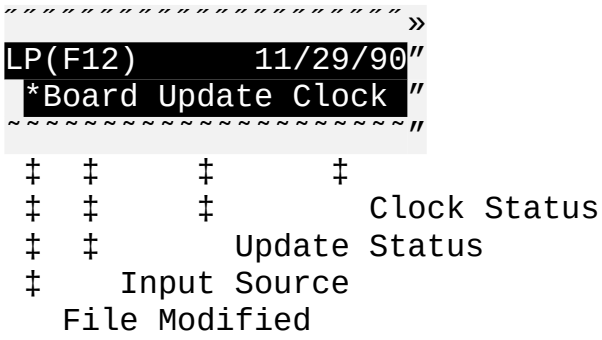
When the LE Panel tool is opened, a window similar to Fig. 1 should appear. In the upper left corner is the number assigned by DESQview to this window. To the right of the number is the title of the window. The LE Panel tool uses the window title to display the current file name. Whenever the current file name changes, for instance when a new configuration is loaded, the title of the window will change. The configuration shown in Fig. 1 is simply an example. Upon startup, LE Panel loads its initial configuration from the file "init.cfg".

Upon startup, the LE Panel tool first changes the current directory to the path indicated by the **HOME** environmental variable. If the variable is not set or the path does not exist, the current directory will be the directory from which the LE Panel tool was started (initially set to "\le\config"). It then tries to load the file "init.cfg" from the current directory. If this file does not exist a bell will sound. If after reading the mode of operation from "init.cfg" it is necessary to communicate with the LE board, it is first determined if the board is responding to requests. If so, the LE Panel is configured to that mode of operation. If not, the mode of operation is set to the default. For more information about configuration files and the various modes of operation, see sections 4.4 and 4.8.

The LE Panel window consists of five kinds of fields: Menu, Status, Label, I/O, and Switch fields.

The purpose of each will be described in the following sections.

4.4. Status Field and Modes of Operation



The status field is located in the upper right of the LE Panel window. The status field consists of four flags as illustrated in Fig. 2. The **File Modified** flag indicates whether the configuration of the panel has been modified since the last save or load. If the file has been modified an asterisk (*) will be displayed for this flag. The remaining three flags indicate the current mode of operation. The **Input Source** flag indicates whether input to the LE board comes from the board or the LE Panel. If input comes from the LE Board, **Board** is displayed, otherwise **Panel** is displayed for this flag. The **Update Status** flag indicates the status of the panel update process. When the panel update process is running, **Update** is displayed for this flag. The **Clock Status** flag indicates the status of the clock on the LE board. If the clock is running, **Clock** is displayed for this flag.

There are five commands that change the mode of operation: **Update Once**, **Update Continuous**, **Pulse Clock**, **Run Clock**, and **Input from Panel/Board**. In addition, when a new configuration file is loaded with the **Load File** command, the mode of operation may change. For details on these commands see section 4.10.

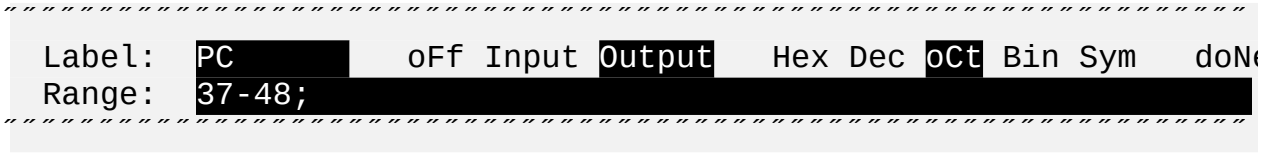
To change the mode of operation, the LE Panel tool must communicate with the LE board. Before doing this it first determines if the board is responding. If so, the command is executed. If not, a message to that effect is displayed and the mode of operation is changed to the default (input from board, **panel update process** stopped, clock stopped). To remove the displayed message, hit the ESC key.

4.5. Label and I/O Fields

The label and I/O fields are used to give a name and a view to a group of signals. An I/O field can be configured to be an input or output field. An input field can be configured to display any of up to 16 of the buttons or switches of the LE board. These can be displayed in either hexadecimal,

decimal, octal, or binary. An output field can be configured to display any of up to 16 of the lights of the LE board. These can be displayed in either hexadecimal, decimal, octal, binary, or symbolically. An I/O field that has been configured as an input field is highlighted while output fields are not.

When a label field is selected, a field definition dialog box as shown in Fig. 3 will be opened. The dialog box indicates the current configuration of this label, I/O field pair. From here the label field and its associated I/O field can be reconfigured. The field definition dialog box consists of five sections: the label, I/O, display, and signal definition sections along with the Done field. Since this menu has a text input field, the TAB key is used to move between fields. Also, the ALT key can be used along with the capitalized letter of the select fields as a quick way to select a field. For instance, ALT-H will select the Hex field even when the cursor is not over it.



The label definition field is used to assign a name of up to 8 characters to this group of signals. This name will be displayed in the label field. The only limitation is that a comma (`,') cannot appear in the label.

The I/O definition section consists of 3 fields: off, input, and output. One of these can be selected to configure the I/O field as either inactive, an input, or an output field. The field that is currently selected is highlighted.

The display definition section consists of 5 fields: hex, dec, oct, bin, and sym. These are used to configure the contents of the I/O field to be displayed as either hexadecimal, decimal, octal, binary, or symbolically. The field that is currently selected is highlighted. If the field is configured as an input field, the sym field configures the I/O field to be displayed as binary.

The signal definition section is used to define which signals are to be grouped and displayed in the I/O field. Signals are specified by their number (0-31 for input fields and 0-127 for output fields); a group of signals is specified by a list of signal numbers or signal ranges separated by commas and terminated with a semi-colon. A signal range is specified by two signal numbers separated by a dash (`-'); it defines a list of signals beginning with the first signal number in the pair and ending with the second signal number of the pair. These can be in either increasing or decreasing order. For instance, the signal groups: "12-5;", "12,11,10-7,6,5;", and "12,11,10,9,8,7,6,5;" are equivalent. Signal groups can consist of non-sequential signal numbers and can contain replicated signal numbers. For instance the signal group: "1,13,9,5-7,6,1;" is legal. Signal groups can contain up to 16 signals. Chapter 9. describes how the lights, buttons, and switches are numbered on the LE board.

The Done field is used to accept the current configuration. If there are no errors in the configuration the field definition menu is closed. If there are errors, a bell will sound and the field definition menu will remain open. At any time the ESC key can be used to abort the process.

A signal group is interpreted as a binary number with the most significant bit being the leftmost

signal in the group and the least significant bit being the rightmost. Whenever an update is done the value of all the signal groups are read from the LE board, converted to their appropriate format and displayed in the I/O field. The I/O field is not changed after reconfiguration until an update is completed.

The value displayed in an I/O field is right justified in the field. In the case of hexadecimal, decimal, and octal formats, the letters `H', `D', and `O' are displayed in the leftmost position of the field respectively, to indicate the format of the display. In the case of binary format, no indicator is displayed. In the case of a symbolic display, a symbol of up to 8 characters is displayed, followed by a four-digit hexadecimal number enclosed in brackets, representing the value of the signal group. If no symbol is mapped to the current value of the signal group, only the hexadecimal value is displayed.

Upon closing a field definition menu configured as an output with symbolic display, the user is prompted for the symbol map file name. Section 4.9 discusses how to create mapping files for symbolic display.

4.6. Using Input Fields

An I/O field that has been configured as an input field operates in two different modes. When input is being taken from the LE board (**board** appears in the status field), the input field will display a value from the switches and buttons on the board. If the input field is selected a bell will sound and nothing will happen. When input is being taken from the panel (**panel** appears in the status field), the switches and buttons are disabled and the switch register on the LE board is enabled. The contents of this register is displayed in the input field. Selecting the input field in this mode allows the user to modify the contents of the switch register. When selected, a field will appear directly below the input field with the current value being displayed. A new value can be entered into this field and registered upon hitting the enter key. The new value is entered in the format that the field is configured to display. Note that the input field is not changed until an update is completed.

4.7. Switch Fields

There are 12 switch fields located along the bottom of the LE Panel window. Each of these is used to give a name to a single switch or button signal on the LE board. The Switch fields operate in two modes. When input is being taken from the LE board, the switch fields operate in field-definition mode. If a switch field is selected in this mode, a field definition menu is displayed as illustrated in Fig. 4. This menu has three fields: the label definition field, the switch definition field, and the done field. The label definition field is used to give a name of up to 5 characters to this signal. The name will be displayed in the switch field. The switch definition field is used to define the switch that this field will control. Switches are specified by their number (0-31). The done field is used to accept the current field definition.

When input is being taken from the panel, the switches and buttons on the board are disabled, the switch register on the LE board is enabled and the switch fields operate in toggle mode. If a switch field is selected in this mode, the value of the switch will be toggled on the LE board. The current value of the switch does not get displayed in the switch field. If it is desired to display the switch values, a label, I/O field can be used.

4.8. File I/O

Any configuration of the panel can be saved to a file for later retrieval. When a configuration is saved, the current configuration of all the label, I/O and switch fields is written to a file along with the state of the status field. When a file is retrieved, the configuration is restored to the state it was in when saved. It may not be possible to restore the state of the status field if the LE board is not responding. In this case a message will be displayed and the mode of operation will be: input source - board, panel update process - stopped, clock - stopped.

There are three commands that pertain to the saving and retrieval of configuration files: Load File, Save File, and Save File As. See section 4.10. for details about these and other commands. In addition, the Exit command will prompt the user whether to save a file that has been modified; if an output field has been reconfigured to display its value symbolically, the user will be prompted for the name of the symbol file to be loaded.

4.9. Symbol Files

Symbol files are used to define a symbol map to map signal group values to symbolic names. When an output field is configured to display its value symbolically, the symbol map is searched for the current value of the signal group. If found, the symbol corresponding to that value is displayed in the output field. Symbol files are simple text files that can be created using any text editor. The first line of the file must contain the number (in decimal) of entries that follow. The subsequent lines contain one entry per line. Each entry consists of a value (in hexadecimal) followed by its corresponding symbol of up to 8 characters.

4.10. Key Bindings and Menu Selections

Below is a table of all the commands and menu selections for the LE Panel tool. The first column is the key stroke for the command. The key strokes preceded by an `A-' refer to the key combined

with the ALT key. The key strokes preceded by a `C-' refer to the key combined with the CTRL key. The second column contains the title of the menu that contains this command as a selection. The third column contains a description of the command. This table can be viewed within the LE Panel tool by selecting the help menu item or by hitting the F12 function key.

Table 2:

Key Stroke	*Menu*	*Description*
F1	-	Activate File Menu.
F2	-	Activate Display Menu.
F3	-	Activate Clock Menu.
F4	-	Activate Tools Menu.
F5	File	Load File: The user is prompted for a configuration file to be loaded. This file will become the new current file. If the current configuration has been modified since the last save, the user is prompted to save or abort the changes before loading the new file. The modification flag in the status field is cleared after the new file is loaded.
F6	File	Save File: The current configuration is saved into the current file and the modification flag in the status field is cleared.
A-F6	File	Save File as: The current configuration is saved into a named file. The user is prompted for a file name. If the file exists, it is overwritten, otherwise it is created. The saved file becomes the current file. The modification flag in the status field is cleared.
F7	File	Clear Configuration: All fields of the current configuration are cleared and the modification flag is set, the input source flag is set to **board**, the **panel update process** is stopped and the clock is stopped.
F8	Display	Update Once: Request a single update from the **panel update process**. This will cause the panel update process to be stopped if it is currently in continuous update mode. If the LE board is not responding, a message to that effect will be displayed and no update will occur.
A-F8	Display	Update Continuous: Put the **panel update process** into continuous update mode. If it is already in continuous update mode, nothing happens. If the LE board is not responding, a message to that effect will be displayed and the **panel update process** will be stopped.
C-F8	Display	Input from Panel/Board: Toggle the source of input between the board and the panel. This also affects the behavior of the switch fields. The status field will reflect the current source of input. If the LE board is not responding, a message to that effect will be displayed and the source of input will be set to board.

Table 2:

F9	Clock	Pulse Clock: Issue one pulse of the LE board clock. This will stop the clock if it is currently in run mode. If the LE board is not responding, a message to that effect will be displayed and no pulse will occur.
A-F9	Clock	Run Clock: Put the LE board clock in run mode. The clock will run at the rate set by the controls on the LE board. If the LE board is not responding, a message to that effect will be displayed and the clock will not be put in run mode.
F10	Tools	LE Assembler: Start up an LE Assembler process in a new window. If one is already running, make it the topmost window.
F11	Tools	Terminal: Start up a terminal emulator process in a new window.
F12	-	Help: Display this table in the help window.
ESC	File	Exit: Exit the LE Panel tool. If the configuration has been modified since the last save, the user is prompted to save or abort these changes.

5. TERM

5.1. Introduction

The TERM tool was developed in order to have a simple terminal emulator that conformed to the Common User Interface. It is intended to be used with Logic Engine designs that require terminal I/O. It does not however, have any direct connection with the Logic Engine.

Any other terminal emulator would work equally well if not better.

In order to use the TERM tool, a serial cable must be connected from the 9 pin serial port of the Logic Engine board to the serial port of the PC host. See Section 2.07 for details about this cable and how to install it.

5.2. Menus and Key Bindings

Key Stroke	Menu	Description
F1	-	Activate File Menu.
F2	-	Activate Baud Menu.
F3	-	Activate Port Menu.
F4	-	Activate Tools Menu.
F5	File	Script File: Transmit named file.
F6	File	Log File: Log all recieved characters to the named file.
ESC	File	Exit: Exit the TERM tool.
A-1	Baud	1200: Set the baud rate of the current port to 1200.
A-2	Baud	2400: Set the baud rate of the current port to 2400.
A-3	Baud	4800: Set the baud rate of the current port to 4800.
A-4	Baud	9600: Set the baud rate of the current port to 9600.
A-5	Baud	19200: Set the baud rate of the current port to 19200.
A-A	Port	Com 1: Set the current port to be COM1.

A-B	Port	Com 2: Set the current port to be COM2.
F10	Tools	LE Panel: Start up an LE Panel process in a new window.
A-F10	Tools	LE Asmb: Start up an LE Asmb process in a new window.
F11	Tools	EDPLD: Start up an EDPLD process in a new window.
A-F11	Tools	TERM: Start up an EDPLD process in a new window.
C-F11	Tools	DOS: Start up a DOS process in a new window.
F12	HELP	Help: Display help information on the TERM tool.

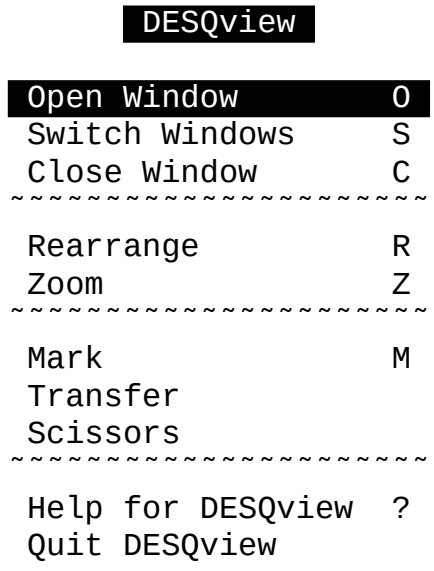
6. Common User Interface

6.1. Introduction

Several of the tools (LE Panel, LE ASMB, ED PLD, and TERM) were written to run in the DESQview[1] environment. They will not run unless DESQview is installed and running. The DESQview environment offers three major features utilized by these tools: the ability to switch between applications, a standard way of dealing with windows and fields, and the ability to multi-task.

These tools also share a common philosophy towards the user interface. Wherever possible, a common interface was used to accomplish similar tasks. This is most notable in the menus, file I/O, dialog boxes, and the DESQview interface.

6.2. DESQview Interface



After the tools have been installed, they can be started from the DESQview menu. This is assuming the advanced installation procedure was used. The DESQview menu is usually displayed in the upper right hand corner of the screen but at times the menu will be hidden. If so, simply hit the DESQview key (ALT-key) to redisplay the menu. The DESQview menu is illustrated in Fig. 1.

1. DESQview is a trademark of Quarterdeck Office Systems.

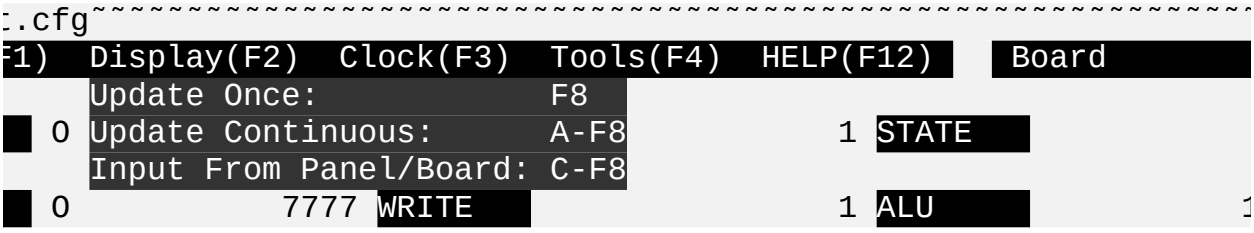
To open a tool, first select the **Open Window** item from the DESQview menu. This can be done by either moving the cursor over the item with the cursor keys or mouse and hitting the space bar, Enter key or left mouse button, or by simply typing the letter **O**. This will open up the **Open Window** menu. From here, a tool can be selected either by moving the cursor over the entry and hitting the space bar or the enter key or by typing the letters associated with the tool. If the entry is not visible on this menu, the **Page Down** key can be used to list more entries. Once selected, a window will be opened for that tool.

DESQview also provides an interface for managing the windows. These functions include: moving, resizing, scrolling, closing and switching windows. See the DESQview manual for details about these and other functions provided by the DESQview interface.

In various situations there will be a selection of fields presented from which the user must choose, or a field in which the user must enter text. In general, the cursor keys or the mouse can be used to move between the fields and the space bar, Enter key or left mouse button can be used to select the field the cursor is over. The exception is in the case of a window that has a field in which text is to be entered. In this case the TAB key moves the cursor to the next field and the Shifted TAB key moves to the previous field. The space bar can be used to select the field that the cursor is over. Text is entered into fields by moving the cursor over the field and entering the text. The text can be edited using the left and right cursor keys, the end, home, insert, delete, and backspace keys. As a general rule the ESC key will cause the current activity to be aborted.

6.3. Menus

Each of the tools that conform to the Common User Interface (LE Panel, LE ASMB, ED PLD, and TERM) have a common menu interface. Across the top of the window of each of these tools, is a menu bar consisting of six fields. The first is the name of the application and has a special function described later. The next five are menu fields, the first labeled **File**, the fourth and the fifth labeled **Tools** and **HELP** respectively. The second and third menu fields are labeled differently for each tool. Each of the menu fields is associated with a function key shown parenthetically in each field. The date of the last modification of the tool is shown on the far right of the menu bar.



The menus can be activated either by selecting them as described above or by pressing the associated function key. For example, the Tools menu can be activated by pressing the F4 key. When a menu is activated, a small pull-down menu containing a list of possible selections is displayed below the menu title. The Display menu for the LE Panel tool is shown in Fig. 2. Displayed to the

right of each selection is an associated key stroke. The key strokes preceded by an `A-' refer to the key combined with the ALT key. Those preceded by a `C-' refer to the key combined with the CTRL key. This key stroke is used as a quick way to select an item without first selecting a menu. They are not available when a menu is active. The actions that are performed when each menu item is selected are described in detail in the respective chapter for that tool. The ESC key can be used to exit the menu without selecting an item. To activate another menu while one is currently active, the mouse, the left and right cursor keys, or the function keys for the menus can be used.

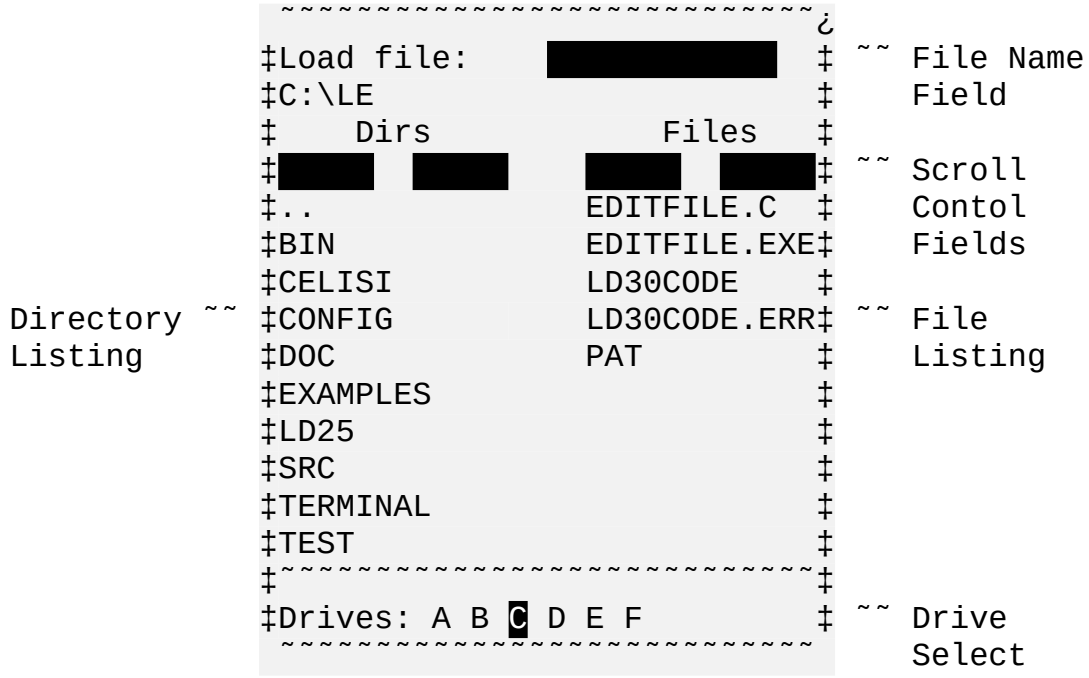
The first field of the menu bar which contains the name of the tool, is the icon field. When selected, the window for the tool is reduced to the size of this field and positioned at the far right of the screen. When selected again, the window is expanded to full size and repositioned. This is useful when you want to hide the window for a time without exiting the tool.

6.4. Dialog Boxes

```
 Label:  PC       oFf Input Output   Hex Dec oCt Bin Sym   doNe
 Range:  37-48;
```

At various times the user will be prompted for input with a dialog box. This is simply a small window that appears on top of the current window and has a selection of fields to choose from and/or a field in which to enter text. The fields can be selected as described above. Fig. 3 shows a field definition dialog box from the LE Panel tool.

6.5. File I/O



When a requested operation requires a file name, the user is prompted with the browser dialog box, as illustrated in Fig. 4. From here the user can type in a file name, select a file from a list, change directories, or change drives. The browser dialog box consists of five major sections: the file name field, the scroll control fields, the directory listing, the file listing, and the drive select fields. The file name field is located at the top right of the browser menu and is used to enter a file name or file name pattern (contains one or more of '*' or '?'). When the enter key is entered into this field, the file name is accepted. If it is not a pattern, the browser menu is closed, and the appropriate action is taken on the named file. If it is a file name pattern, the file listing is updated with only those files which match the pattern.

The directory and file listing sections are located in the middle of the browser dialog box - the directory listing on the left and the file listing on the right. These contain a listing of up to ten of the subdirectories and files of the current directory. If there are more than ten subdirectories or files, the scroll control fields for each listing can be used. These are located just above each listing. Each has two controls, one for scrolling up (on the left) and one for scrolling down (on the right). When one of these fields is selected, the corresponding listing will scroll up or down.

Selecting one of the items in the directory listing section will change the current directory. A new directory and file listing will be displayed. The path of the current directory will be displayed just below the file name field. To move up one level in the directory tree, the sub-directory name ".." can be selected.

Selecting one of the items in the file listing section will accept that file name. The browser dialog box is closed and the appropriate action is taken on the selected file name.

The drive select fields are located at the bottom of the browser dialog box, labeled with the drive letters with the current drive highlighted. Selecting one of the drives will change the current drive, update the directory and file listings, and update the path of the current directory.

The ESC key can be used to abort out of the browser with a null file name being accepted and the appropriate action being performed for a null file (usually nothing). However, a change of the current directory or drive will take effect no matter how the browser is exited. Subsequent file I/O will be relative to the new current directory and drive.

There are several shortcuts available for the browser that allow the user to avoid having to move the cursor around a lot with the cursor keys or the mouse. They are listed below:

Home	Move the cursor to the File Name field
Page Up	Scroll File Listing up
Page Down	Scroll File Listing down
CTL-Page Up	Scroll Directory Listing up
CTL-Page Down	Scroll Directory Listing down
ALT-<n>	Selects the <n>th item in the File Listing, where <n> is one of the numeric keys (1 representing the first item and 0 representing the tenth item).
ALT-<c>	Changes to the directory indicated by the letter <c>. Where <c> is one of the keys on the top row of the keyboard starting with 'q' (representing the first item) and ending with 'p' (representing the tenth item).
ALT-<c>	Changes to the drive indicated by the letter <c>. Where <c> is one of the keys on the bottom row of the keyboard starting with 'z' (representing drive A) and ending with 'm' (representing drive G).

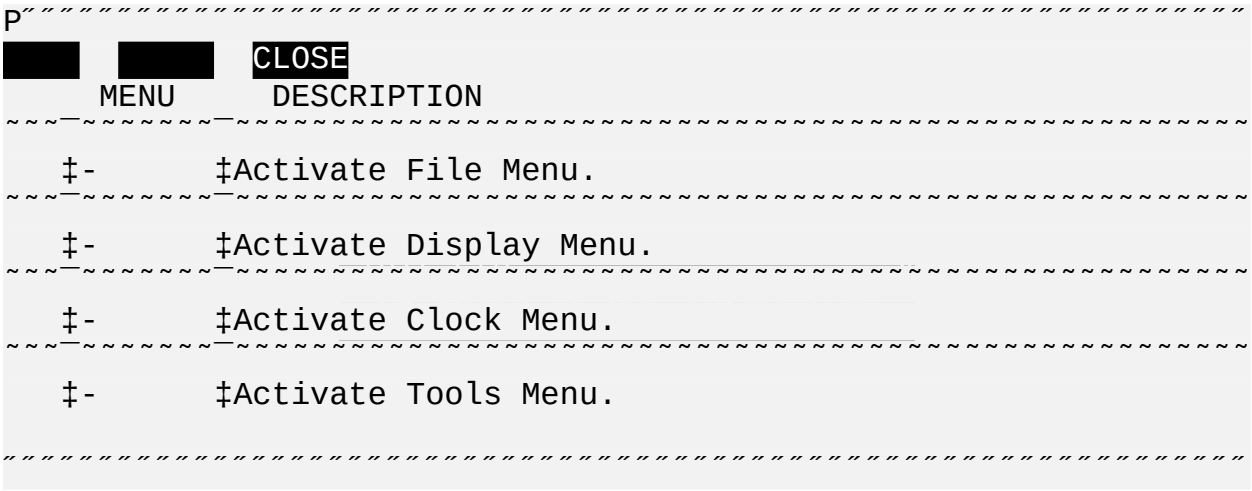
6.6. Tools Menu

Each of the tools that conform to the Common User Interface (LE Panel, LE ASMB, ED PLD, and TERM) have an identical tools menu. This menu consists of five selections used to start up one of the tools or a DOS window. The key strokes associated with each selection are identical

within each tool. They are:

F10	LE Panel
ALT-F10	LE ASMB
F11	ED PLD
ALT-F11	TERM
CTL-F11	DOS

6.7. Help System



Each of the tools that conform to the Common User Interface (LE Panel, LE ASMB, ED PLD, and TERM) use a common help system. Fig. 5 shows the help dialog box for the LE Panel tool. The help dialog box consists of three fields across the top and a display area below. The display area contains a portion of the help information. The first two fields are used to scroll through the help information and the third is used to close the help dialog box.

6.8. Communication with the LE Board

Most of the tools need to communicate with the LE Board at various times. In order to do this, the LE Board must be installed and turned on. Before any communication is attempted, the tool will first determine if the LE Board is active, if so the communication will take place, if not the communication is aborted and a dialog box as shown in Fig. 6, is displayed indicating that the LE Board is not responding. This dialog

box can be closed by pressing the ESC key or the left mouse button. This will prevent most communication attempts when the board is not installed. However, if the board is turned off or disconnected while the tool is running, the tool can get confused and may attempt to communicate with a non-functioning board. In this case it is possible for the tool to become hung.

Another problem that can arise while communcating with the LE Board is due to multitasking. Since more than one tool can be running at the same time, it is possible for more than one tool to be commmunicating with the board. The system allows only one tool to communicate with the board at any one time. The problem can occur when the communications of several tools are interweaved, such as:

1) tool A reads the state of the board

2) tool B changes the state of the board

3) tool A acts based on the state it read (which is now invalid)

The system does little to prevent this situation, so it is best to avoid it. It does attempt to prevent it if tool A and tool B in the example above are the same tool. In the case of the LE ASMB and ED PLD tools, if subsequent instances of the tool are opened, the subseqent ones are not allowed to communicate with the board. This is indicated by a bell when the window is opened. In the case of the LE Panel tool, subseqent instances are not allowed. If a second instance is opened it is immediately closed and the first instance is raised to be the top window.

4. ED PLD

4.1. Introduction

The ED PLD Tool is a PLD fuse map editor with the capability to read, burn, verify, and test PLDs using the PLD burner on the Logic Engine Board. ED PLD supports only the Cypress PLD C 20G10 at this time.

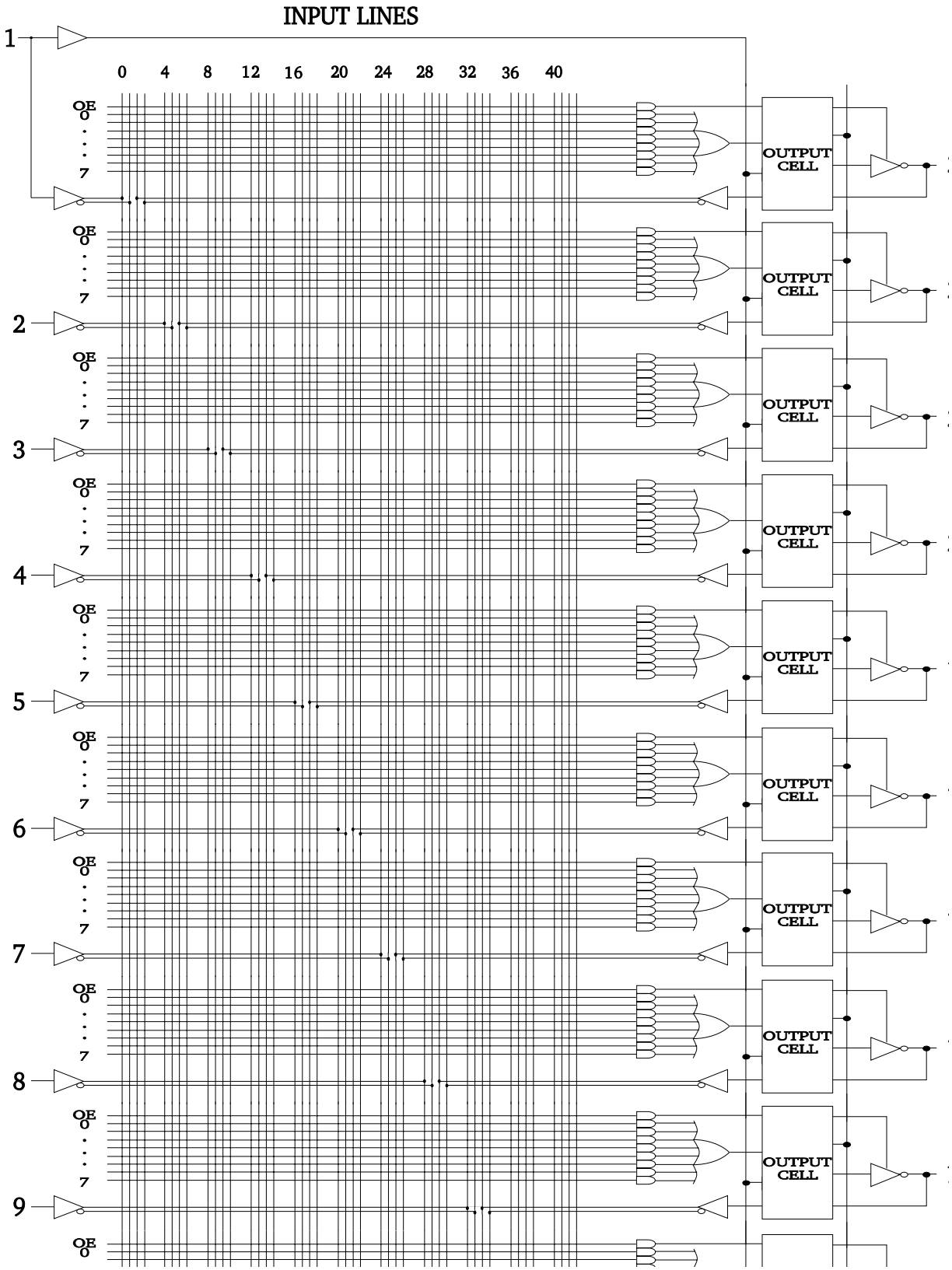
4.2. Cypress PLD C 20G10

The 20G10 is a 24 pin PLD with 12 input pins and 10 I/O pins. Each of the 10 output cells can be configured as registered or combinational outputs, true high or true low outputs, and product term or pin 13 output enable signals. The registered outputs are clocked by input pin 1. The fuse matrix consists of 44 signals (12 input, 10 output feedbacks, and their compliments). Each output cell has 9 product terms, 1 output enable term and 8 terms feeding an OR-gate. The functional logic diagram of the PLD C 20G10 is shown in Fig. 1 and the eight possible configurations of the output cells are shown in Table 1 and Figs. 2-3. For more information on the PLD C 20G10, con-

sult the data sheet.

Table 3:

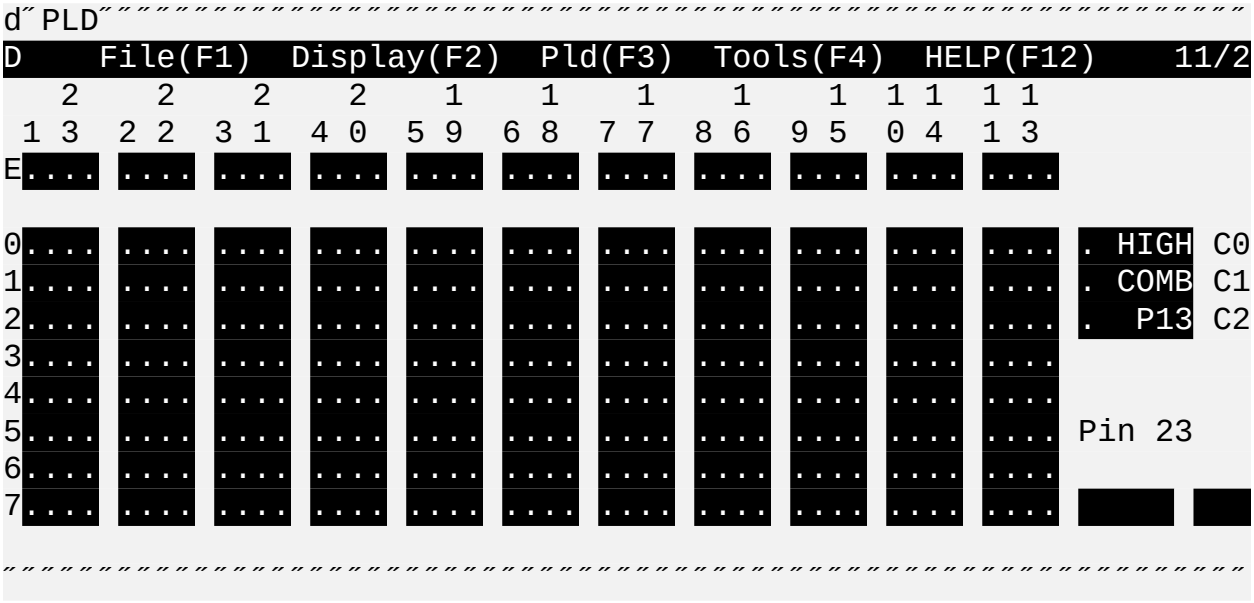
Figure	*C2*	*C1*	*C0*	*Configuration*
2A	0	0	0	Product Term OE Registered Active LOW
2B	0	0	1	Product Term OE Registered Active HIGH
3A	0	1	0	Product Term OE Combinational Active LOW
3B	0	1	1	Product Term OE Combinational Active HIGH
2C	1	0	0	Pin 13 OE Registered Active LOW
2D	1	0	1	Pin 13 OE Registered Active HIGH
3C	1	1	0	Pin 13 OE Combinational Active LOW



4.3. EDPLD Fuse Map Editor

As a fuse map editor, the EDPLD tool provides an easy and integrated way to generate a file in the proper format for the PLD programmer. The files generated by the editor are simple ASCII files that conform to a specific format (see section 6.6). These files can of course be generated and edited by any text editor. The EDPLD editor will however, ensure that the file is generated in the proper format.

When the ED PLD tools is started, a window such as that shown in Fig. 4 is opened. This window represents a view of one output cell of a 20G10 and its fuse matrix. The window consist of four sections: a menu bar across the top, fuse matrix fields through the middle, output configuration fields on the right, and two scroll fields on the bottom right.



The fuse matrix fields are arranged in 9 rows. One for the output enable signal (labeled OE) and eight for the product terms (labeled 0-7). Each row contains 22 fuse matrix fields organized in pairs of two. Each of the fuse matrix fields represents one signal of the 12 input and 10 feedback signals. Each column is labeled with the pin number of the signal it represents. The fuse matrix fields consist of two characters representing the two polarities of the signal. Each character can be either a '.' indicating the absence of a fuse or a 'x' indicating the presence of a fuse. For the input signals the true high signal comes first followed by the true low signal. For the output feedback signals the order of the signals depends on the configuration of the output cell. Refer to Figs 1-3 to determine the proper orientation.

The fuse matrix fields can be in one of four configurations. When selected, the configuration of the field will advance to the next configuration in the sequence ('..' '.x' 'x.' 'xx' '..').

The output configuration fields can each be in one of two states. When selected, the fields will toggle states. Besides a ',' or a 'x' being displayed in each field, a mnemonic is displayed in each field indicating the current configuration.

The two scroll fields are used to move the view to the next or previous output cell. The pin number of the current output cell is displayed above the scroll fields. The **Page Up** and **Page Down** keys can also be used for this function.

4.4. File I/O

```
    1 23 2 22 3 21 4 20 5 19 6 18 7 17 8 16 9 15 1014 1113  PIN 23
0 OE x... .... .... .... .... .... .... .... .... .... ....
0  0 .... x... .... x... .x.. .x.. .x.. .x.. .xx. .x.. .x.x .  H
0  1 .... x... .... x... .x.. .x.. .x.. .x.. .x.. .xx. .x.x .  C
0  2 .... .x.. x... .... .... .... .... .... .... .... .... x  P
0  3 .... .x.. x... .... .... .... .... .... .... .... ....
```

The ED PLD software has several functions used to save and retrieve fuse maps. **Save File** -saves the current fuse map into the current file. If there is no current file, the user is prompted for a file name. **Save File As** - saves the current fuse map into the named file and **Load File** -loads the named file into the editor. Initially the current file is undefined. Each time a file is named, using any of these functions, that file becomes the current file and will be displayed as the title of the ED PLD window.

The files created by the ED PLD editor, called fuse map files, have a specific format. These files are text files and can be edited using other text editors, but they must conform to the fuse map format described below.

Each fuse map file consists of 10 configuration blocks which describe the product terms and architecture of each output cell. Fig. 5 is an example of one configuration block and Fig. 6 is an example of a complete fuse map file.

Each configuration block consists of exactly 10 lines. The first line is a comment that typically contains the pin numbers corresponding to each column of the product terms and the pin number of the output cell. The second line contains the product term for the output enable signal. The following 8 lines contain the product terms which feed the OR-gate of the output cell. Each of the product term lines must contain in the first column, the number of the configuration block (0-9). The product term for the output enable signal must contain the letters 'OE' in the third and fourth column. The remaining 8 product terms must contain in the fourth column, the number of the product term (0-7).

Starting in column 5, each product term line contains a fuse pattern consisting of 44 fuse bits. These are typically arranged in 11 groups of 4 bits separated by a space, however, spaces can occur throughout the line and are ignored. In addition, product terms 0, 1, and 2 contain an extra fuse bit beyond the 44 fuse bits which configure the architecture of the output cell. Each fuse bit is either a `.' or a `x'. A `.' indicates the absence of a fuse and a `x' indicates the presence of a fuse. Any characters beyond the fuse bits are considered comments and are ignored.

The product terms are arranged as a sequence of pairs of fuse bits, one pair for each input signal and output feedback signal. One fuse bit of each pair is for the true high and the other is for the true low version of the signal. For the input signals the true high signal comes first followed by the true low signal. For the output feedback signals the order of the signals depends on the configuration of the output cell. Refer to Figs 1-3 to determine the proper orientation.

The three fuse bits beyond product terms 0, 1, and 2 (referred to as C0, C1, and C2) configure the architecture of the output cell. Each fuse bit controls one aspect of the configuration as described below:

C0	.	: True High (H)
	x	: True Low (L)
C1	.	: Combinational Output (C)
	x	: Registered Output (R)
C2	.	: Output Enable source is pin 13 (C)
	x	: Output Enable source is product term (P)

```
    1 23 2 22 3 21 4 20 5 19 6 18 7 17 8 16 9 15 1014 1113  PIN 23
0 OE x... .... .... .... .... .... .... .... .... .... .... ....
0  0 .... x... .... x... .x.. .x.. .x.. .x.. .x.. .xx. .x.. .x.x .  H
0  1 .... x... .... x... .x.. .x.. .x.. .x.. .x.. .xx. .x.x .  C
0  2 .... .x.. x... .... .... .... .... .... .... .... .... .... x  P
0  3 .... .x.. x... .... .... .... .... .... .... .... .... ....
0  4 .... .x.. x... .... .... .... .... .... .... .... .... ....
0  5 .... .x.. x... .... .... .... .... .... .... .... .... ....
0  6 .... .x.. x... .... .... .... .... .... .... .... .... ....
0  7 .... .x.. x... .... .... .... .... .... .... .... .... ....
    1 23 2 22 3 21 4 20 5 19 6 18 7 17 8 16 9 15 1014 1113  PIN 22
1 OE x... .... .... .... .... .... .... .... .... .... .... ....
1  0 .... x... .... x... .x.. .x.. .x.. .x.. .xx. .x.. .x.x .  H
1  1 .... x... .... x... .x.. .x.. .x.. .x.. .x.. .xx. .x.x .  C
1  2 .... .x.. x... .... .... .... .... .... .... .... .... .... x  P
1  3 .... .x.. x... .... .... .... .... .... .... .... .... ....
1  4 .... .x.. x... .... .... .... .... .... .... .... .... ....
1  5 .... .x.. x... .... .... .... .... .... .... .... .... ....
1  6 .... .x.. x... .... .... .... .... .... .... .... .... ....
1  7 .... .x.. x... .... .... .... .... .... .... .... .... ....
    1 23 2 22 3 21 4 20 5 19 6 18 7 17 8 16 9 15 1014 1113  PIN 21
2 OE x... .... .... .... .... .... .... .... .... .... .... ....
2  0 .... x... .... x... .x.. .x.. .x.. .x.. .x.. .x.. .xx. .  H
2  1 .... .x.. .x.. .... .... .... .... .... .... .... .... .  C
2  2 .... .x.. .x.. .... .... .... .... .... .... .... .... x  P
2  3 .... .x.. .x.. .... .... .... .... .... .... .... .... ....
2  4 .... .x.. .x.. .... .... .... .... .... .... .... .... ....
2  5 .... .x.. .x.. .... .... .... .... .... .... .... .... ....
2  6 .... .x.. .x.. .... .... .... .... .... .... .... .... ....
2  7 .... .x.. .x.. .... .... .... .... .... .... .... .... ....
    1 23 2 22 3 21 4 20 5 19 6 18 7 17 8 16 9 15 1014 1113  PIN 20
3 OE x... .... .... .... .... .... .... .... .... .... .... ....
3  0 .... x... .... x... .x.. .x.. .... .x.. ..x. .... ...x .  H
3  1 .... x... .... x... .x.. .x.. .... .x.. .... ..x. ...x .  C
3  2 .... x... .... x... .x.. .x.. .... .x.. .... .... x... x  P
3  3 .... x... .... x... .x.. .x.. .... .x.. .... x... .... ....
3  4 .... x... .... x... .x.. .x.. .... .x.. x... .... .... ....
3  5 .... x... .... x... .x.. .x.. x... .... .... .... .... ....
3  6 .... x... .... x... .x.. .x.. x... .... .... .... .... ....
3  7 .... x... .... x... .x.. .x.. x... .... .... .... .... ....
    1 23 2 22 3 21 4 20 5 19 6 18 7 17 8 16 9 15 1014 1113  PIN 19
4 OE x... .... .... .... .... .... .... .... .... .... .... ....
4  0 .... .... .... x... .x.. .... .x.. .... .xx. ...x ...x .  H
4  1 .... .... .... x... .x.. .... .x.. .... .x.. .... x... .  C
4  2 .... .... .... x... .x.. .... .x.. .... .x.. x... .... x  P
4  3 .... .... .... x... .x.. .... .x.. x... .... .... .... ....
4  4 .... .... .... x... .x.. x... .... .... .... .... .... ....
4  5 .... .x.. .... .... .... .... .... .... .... .... .... ....
4  6 .... .x.. .... .... .... .... .... .... .... .... .... ....
4  7 .... .x.. .... .... .... .... .... .... .... .... .... ....
    1 23 2 22 3 21 4 20 5 19 6 18 7 17 8 16 9 15 1014 1113  PIN 18
5 OE x... .... .... .... .... .... .... .... .... .... .... ....
5  0 .... x... .... x... .... .x.. .x.. .... .... .x.. x... .  H
5  1 .... x... .... x... .... .x.. .x.. .... x... .... .... .  C
5  2 .... x... .... x... .... .x.. .x.. x... .... .... .... x  P
5  3 .... x... .... x... x... .... .... .... .... .... .... ....
5  4 .... x... .... x... x... .... .... .... .... .... .... ....
5  5 .... x... .... x... x... .... .... .... .... .... .... ....
5  6 .... x... .... x... x... .... .... .... .... .... .... ....
5  7 .... x... .... x... x... .... .... .... .... .... .... ....
    1 23 2 22 3 21 4 20 5 19 6 18 7 17 8 16 9 15 1014 1113  PIN 17
6 OE .... .... .... .... .... .... .... .... .... .... .... ....
6  0 ..xx .... .... .... .... .... .... .... .... .... .... .  H
6  1 ..xx .... .... .... .... .... .... .... .... .... .... .  C
6  2 ..xx .... .... .... .... .... .... .... .... .... .... x  P
6  3 ..xx .... .... .... .... .... .... .... .... .... .... ....
6  4 ..xx .... .... .... .... .... .... .... .... .... .... ....
6  5 ..xx .... .... .... .... .... .... .... .... .... .... ....
6  6 ..xx .... .... .... .... .... .... .... .... .... .... ....
6  7 ..xx .... .... .... .... .... .... .... .... .... .... ....
    1 23 2 22 3 21 4 20 5 19 6 18 7 17 8 16 9 15 1014 1113  PIN 16
7 OE .... .... .... .... .... .... .... .... .... .... .... ....
7  0 .... .... .... .... .x.. .x.. .x.. .x.. .x.x .x.x .x.x x  L
7  1 .... .... .... .... .x.. .x.. .x.. .x.. .x.x .x.x .x.x .  C
7  2 .... .... .... .... .x.. .x.. .x.. .x.. .x.x .x.x .x.x x  P
7  3 .... .... .... .... .x.. .x.. .x.. .x.. .x.x .x.x .x.x
7  4 .... .... .... .... .x.. .x.. .x.. .x.. .x.x .x.x .x.x
7  5 .... .... .... .... .x.. .x.. .x.. .x.. .x.x .x.x .x.x
7  6 .... .... .... .... .x.. .x.. .x.. .x.. .x.x .x.x .x.x
7  7 .... .... .... .... .x.. .x.. .x.. .x.. .x.x .x.x .x.x
    1 23 2 22 3 21 4 20 5 19 6 18 7 17 8 16 9 15 1014 1113  PIN 15
8 OE ..xx xxxx xxxx xxxx xxxx xxxx xxxx xxxx xxxx xxxx xxxx xxxx
8  0 .... .... .... .... .... .... .... .... .... .... .... .  H
8  1 .... .... .... .... .... .... .... .... .... .... .... .  C
8  2 .... .... .... .... .... .... .... .... .... .... .... x  P
8  3 .... .... .... .... .... .... .... .... .... .... .... ....
8  4 .... .... .... .... .... .... .... .... .... .... .... ....
8  5 .... .... .... .... .... .... .... .... .... .... .... ....
8  6 .... .... .... .... .... .... .... .... .... .... .... ....
8  7 .... .... .... .... .... .... .... .... .... .... .... ....
    1 23 2 22 3 21 4 20 5 19 6 18 7 17 8 16 9 15 1014 1113  PIN 14
9 OE ..xx xxxx xxxx xxxx xxxx xxxx xxxx xxxx xxxx xxxx xxxx xxxx
```

Beyond each of these three fuse bits there is typically a single character comment indicating the configuration of the output cell. These characters are given above in parenthesis.

Although the fuse map file format is rigid, creating and editing a fuse map file is quite easy. To create a new file, simply enter the **pld** program and select option 2 (Save buffer) and specify a file-name. This will save the initial empty buffer into the named file. To edit an existing fuse map file, use your favorite text editor in overstrike mode to alter the fuse bits.

4.5. EDPLD PLD Programmer

There are several functions available in the ED PLD editor for testing, programming, and verifying a 20G10 in the programmer socket on the Logic Engine Board. These functions are all available under the **Pld** menu and are described below in section 6.7. Each of these functions will prompt the user to insert a 20G10 into the programmer socket. The programmer socket is the zero insertion force socket located near the center of the Logic Engine board below the bank of leds. The 20G10 should be oriented in the socket so that pin 1 is on the left (the same orientation as the other chips on the board.) The lever on the socket should be in the up position when inserting or removing a chip and in the down position to lock a chip into the socket. It is important that the 20G10 not be inserted until the prompt appears and at not time should a 20G10 be inserted or removed when one of the leds surrounding the programmer socket is on. After the 20G10 is inserted, hit any key to proceed with the operation. Some of the functions will display dialog boxes indicating the sucess or failure of the operation. Hit any key to remove one of these messages. The **Test Device** function is a little more involved and is described in the next section.

4.6. Testing a 20G10

The EDPLD tool has the ability to apply a set of test vectors to a 20G10 in the programmer socket and read the outputs of the 20G10. When the **Test Device** function is selected, the user will first be prompted to supply the file name of a test vector file. The format of a test vector file is described below. After supplying the file name, the user is prompted to insert the 20G10 into the programming socket. After the 20G10 is inserted, hit any key to begin the testing of the device. After the device has been tested, a window will be displayed showing the results of the test. This window can be scrolled using the cursor keys. The window can only display up to fifty lines of information. If the information is longer than fifty lines, it will be truncated and a message will be displayed at the bottom of the window so indicating. This window can be closed by hitting the ESC key. After the window is closed, a dialog box will be displayed indicating the number of errors that occured during testing.

The output of the testing can also be logged to a file. This can be done by use of the **Log File** function under the file menu. This function will prompt the user for the name of the log file. All subsequent output from testing will be sent to this file and no window will be displayed containing this information. Also, the information being sent to the log file will not be truncated.

Each test vector consists of a line of 24 characters, one for each pin of the device, in order from pin 1 to pin 24. Any number of test vectors can occur in a file, one per line. A blank line or a line starting with a `#' or a ` ' (space) is considered a comment and ignored. The valid characters for each type of pin are given below:

IAfter a file has been created containing the desired test vectors, they can be applied to the device

Input Pins: 1-11,13		
	1	: Apply 5V
	0	: Apply 0V
	C	: Clock (0V-5V-0V)
	K	: Clock (5V-0V-5V)
	N	: This pin is not tested
Input/Output Pins: 14-23		
	1	: Apply 5V (assumes pin is configured as an input)
	0	: Apply 0V (assumes pin is configured as an input)
	C	: Clock (0V-5V-0V) (assumes pin is configured as an input)
	K	: Clock (5V-0V-5V) (assumes pin is configured as an input)
	H	: Expected result is 5V (assumes pin is configured as an output)
	L	: Expected result is 0V (assumes pin is configured as an output)
	Z	: Expected result is High Z
	I	: Expected result is opposite of High Z (i.e when the pin is pulled high, its value is low and when it is pulled low its value is high). It is not exepected that this will be used in a test vector, but it does show up in the output of the test. This usually occurs when the pin is not being tested and it depends on other pins that are also not being tested.
	N	: This pin is not tested
Power Pins: 12,24		
	N	: This pin is not tested

under test as described above. After testing each vector will be displayed. If there is an error in the format of the test vector, an error message will be displayed and the test vector will be skipped. If there is an error in the expected results, the actual results will be displayed along with an error message indicating the location of the error. After all the test vectors have been applied,

a message will be displayed indicating the total number of errors which occurred. Below is an example set of test vectors followed by an example output from running the set of test vectors:

```
Test Vectors:
     10000010000N0NNNNNNNNNHN
     10000001000N0NNNNNNNNNHN
     10000000100N0NNNNNNNNNHN
     10000000000N0NNNNpNNNNLN
     01111111111N1NNNNHHNNNZN
 Output:
     VECTOR:  10000010000N0NNNNNNNNNHN : Passed
     VECTOR:  10000001000N0NNNNNNNNNHN : Passed
     VECTOR:  10000000100N0NNNNNNNNNHN : Passed
     ERROR: Illegal character in vector: p
     VECTOR:  10000000000N0NNNNpNNNNLN : Skipped

     VECTOR:  01111111111N1NNNNHHNNNZN : Failed
     RESULTS: 01111111111N1ZZZZZLLLZZN
     ERRORS:                   ^^
     Test Completed with 3 Errors
```

4.7. Key Bindings

Table 4:

Key Stroke	Menu	Description
F1	-	Activate File menu.
F2	-	Activate Display menu.
F3	-	Activate Pld menu.
F4	-	Activate Tools menu.
F12	-	Activate Help menu.
F5	File	Load File: Load a fuse map file into the buffer. The user is prompted for the file name. This file becomes the current file.
F6	File	Save File: Save the current buffer into the current file. If there is no current file the user is prompted for the file name.
A-F6	File	Save File As: Save the current buffer into the named file.
F7	File	Log File: Name the file for logging output from the application of test vectors.
ESC	File	Exit:
Page Down	Display	Next Block: Move view to the next output cell.
Page Up	Display	Previous Block: Move view to the previous output cell.
DEL	Display	Clear Block: Set all fuses for this output cell to '.' (absence of a fuse).
A-B	Pld	Blank Test: Test for a blank device in the PLD Burner socket. The user is prompted to insert the device in the socket at the proper time.
A-P	Pld	Program Device: Program the device in the PLD Programmer socket with the contents of the buffer. The user is prompted to insert the device at the proper time. If the device is not blank, the user is prompted whether to continue.
A-R	Pld	Read Device: Read the contents of the device in the PLD Programmer socket into the buffer. The user is prompted to insert the device in the socket at the proper time.
A-V	Pld	Verify Device: Read the contents of the device in the PLD Programmer socket and compare it to the contents of the buffer. The number of differences between the two is displayed. The user is prompted to insert the device in the socket at the proper time.

Table 4:

A-T	Pld	Test Device: Apply a set of test vectors to the device in the PLD Programmer socket. The user is prompted to insert the device in the socket and for the name of the file containing the test vectors.

5. LE Assembler

5.1. Introduction

LEASM is a microcode developmenttool which consists of a text editor, microcode assembler, downloader, and debugger, all in a unified environment. Features include:

- a source level debugger with single stepping and breakpoints.
- an emacs-like editor.
- viewing of object code.
- a rich micro-assembly language.

5.2. Editor

Built into the LEASM Tool is a simple text editor with emacs-like key bindings. It is not a full featured editor and as such is not intended to be used for large editing task. It is intended to be used as a quick and easy way to make changes to source code during debugging sessions. Table 1 lists the key bindings available in the editor

There are several functions built into the editor which allow for the assembly of microcode programs. These are listed in Table 2. Section 5.6 describes the syntax and sematics of the microassembly language.

Once a microassembly program has been loaded into the editor or created within the editor, it can be assembled using the Assemble/Load function. This will assemble the program starting from the top of the file, no matter where the cursor is currently located. If any errors occur during assembly, the cursor will be moved to the beginning of the offending line and an error message will be displayed in the status region (see below). The Next-Error function will move the cursor to the beginning of the line of the next error and display an error message in the status region. If no more errors exist, the cursor will be moved to the top of the file.

If there are no errors, the assembled code is downloaded to the Logic Engine Board and the cursor will be moved to the starting line of the microcode (location 0). This line will also be highlighted. If the board is not present, a message will be displayed indicating that the code was not downloaded and the cursor will be moved to the top of the file.

The status region is used to display various information about the status of the file. There are three contexts in which different information is displayed. As mentioned above,

cursor is on has an assembly error, an error message is displayed. If the code has been successfully assembled and the cursor is on a line which contains a program statement, then the object code for that line is displayed. In all other cases, the status region contains three fields which dis-

play the status of the Saved flag, Assembled flag, and the number of errors. The Saved flag indicates whether the file has been saved to disk since the last change made to the file in the editor. If it has been saved, an asterisk (*) will be displayed, otherwise nothing will be displayed in this field. The Assembled flag indicates whether the file has been successfully assembled since the last change made to the file in the editor. If it has been assembled, an asterisk (*) will be displayed, otherwise nothing will be displayed in this field. The third field contains the number of errors that were detected during the last assembly.

The display of the status region is controlled by the Display Status function. Initially the status region is not displayed. The Display Status function will toggle the display of the status region. If it is currently being displayed, it will turn it off. If it is currently not being displayed it will turn it on. Also, if any errors occur during assembly or if the Next Error function is used, the status region will be displayed.

There are three functions which allow for the naming of various output files associated with assembly. These are: Listing File, Symbol Table File, and Error File. If any of the files are named, the appropriate information will be stored in the files after each assembly. Each subsequent assembly will overwrite contents of these files as long as name has not been changed. To disable the storing of the information, give the file a null name by hitting ESC when the browser menu is displayed.

key	*meaning*
^C-A	beginning-of-line
^C-E	end-of-line
^C-N	down-line
^C-P	up-line
^C-F	forward-character
^C-B	backward-character
^C-L	center-window
^C-V	next-page
^A-V	previous-page
^C-Z	scroll-up
^A-<	goto-beginning
^A->	goto-end
^CN8	beginning-of-window
^A-F	forward-word
^A-B	backward-word
^C-X g	goto-line
N.	delete-character
Backspace	backward-delete-character
^C-K	kill-line
^A-H	backward-kill-word

key	*meaning*
^CX ^CS	save-file
^CX i	insert-file
F-12	help
^CX ^CC	exit
^A-F8	display 2910
F5	next-error
F6	single step
^A-F7	run with breakpoint
^C-F7	clear 2910
F11	EDPld
^C-F11	DOS
N7	beginning-of-line
N1	end-of-line
N2	down-line
N8	up-line
N6	forward-character
N4	backward-character
N5	center-window
N3	next-page
N9	previous-page

key	*meaning*
^A-Z	scroll-down
^C-N7	goto-beginning
^C-N1	goto-end
^C-N2	end-of-window
^C-N6	forward-word
^C-N4	backward-word
^C-D	delete-character
^C-H	backward-delete-character
^C-O	open-line
^A-D	kill-word
^C-X^C-V	visit-file
^CX ^CW	write-file
^A-?	display bindings
N0	overwrite-mode
F8	display status
^A-F5	assemble/load
^A-F6	run
^C-F6	idle
F7	toggle breakpoint
F10	LEPanel
^A-F11	Term

Function	*Key*	*Description*
Assemble/Load	A-F5	Assemble the current file and download the resulting object code if the LE Board is present
Next-Error	F5	Advance the cursor to the next assembly error and display an error message in the status region
Display Status	F8	Toggle the display of the status region.
Listing FIle	F9	Name the file for saving of listings of subsequent assemblie
Symbol Table File	A-F9	Name the file for saving of the symbol table of subsequent assemblies
Error File	C-F9	Name the file for saving of error messages of subsequent assemblies

5.3. LEASM Debugger

There are several functions built into the editor which allow for the debugging of microcode programs. These are listed in Table 3.

Function	*Key*	*Description*
Run	^A-F6	Put the 2910 into run mode. It will run at the rate of the LE Board clock.
Single Step	F6	Execute one instruction of the microcode.
Idle	^C-F6	Take the 2910 out of run mode.
Run Breakpoint	^A-F7	Put the 2910 into run mode and wait for a breakpoint.
Toggle Breakpoint	F7	Set/Clear a breakpoint for the current instruction.
Display 2910	^A-F8	Toggle the display of the 2910 window.
Clear 2910	^C-F7	Reset the 2910.

Once a microassembly program has be successfully assembled and downloaded, it can be run

from the LEASM environment. In the most simple case, the program can be started, stopped, and restarted. To start the program running, use the Run function. This will start the program from the current location (highlighted) and run at the rate of the LE board clock. When the program is running, there is no feedback to the software. To stop the program from running, use the Idle function. This will stop the execution of the 2910 and update the current location to the instruction where the program was stopped. The Clear 2910 function will reset the contents of the 2910 and thus reset the microprogram to be started at location 0.

The Single Step functions causes one microinstruction to be executed. After execution, the cursor will be relocated to the next instruction to be executed and this instruction will be highlighted.

The Toggle Breakpoint function allows the user to set or clear a breakpoint on any microinstruction. In order to set a breakpoint, the microcode must have been successfully assembled. If the code has been assembled, but the LE board is not responding, a bell will be sounded when the Toggle Breakpoint function is used, indicating that the breakpoint was not downloaded to the LE Board. Breakpoints are used in conjunction with the Run/Breakpoint function which is used to put the LE Board into **run with breakpoint** mode. The Run/Breakpoint function is similar to the Run function with the exceptions that none of the other LEASM functions are available while in **run with breakpoint** mode. The board will stay in this mode until an instruction with a breakpoint set is to be executed or the cancel field is selected in the dialog box that is displayed upon entering **run with breakpoint** mode. When the board leaves this mode, the dialog box will be closed, the cursor will be moved to the current microinstruction, and the current microinstruction will be highlighted.

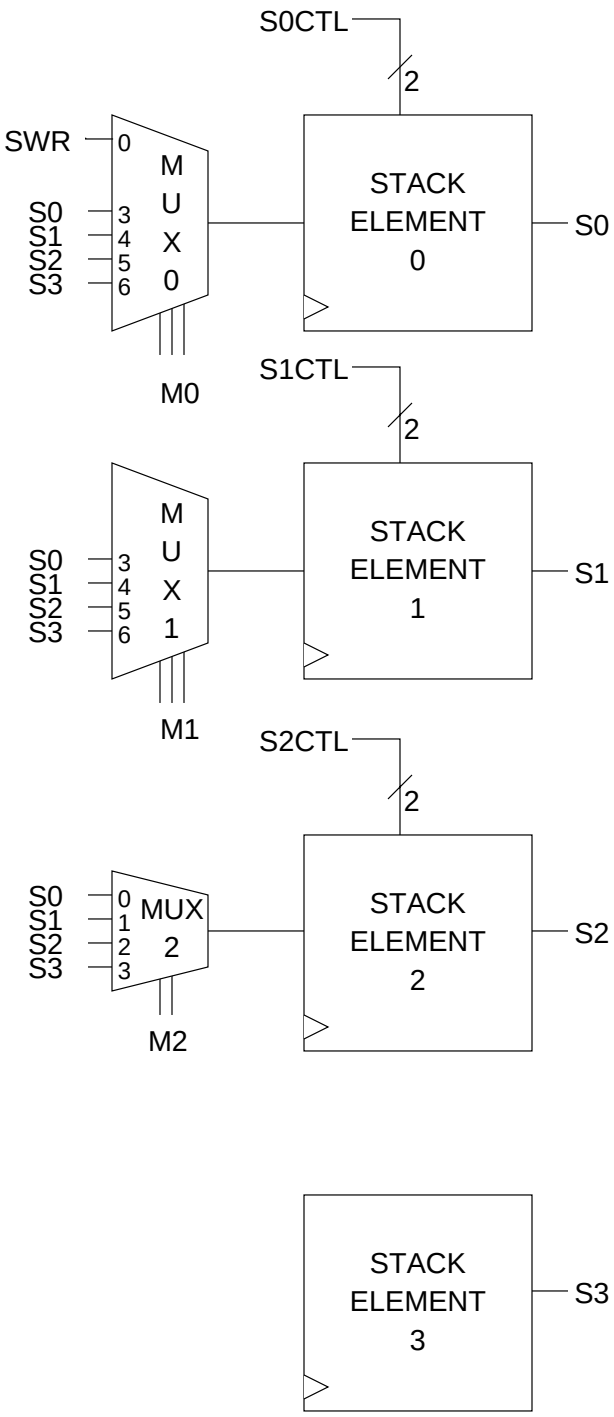
The Display 2910 function allows the user to observe the contents of the 2910 while debugging. This function will toggle the display of the 2910 window. When displayed, this window contains the values of the **I** and **D** inputs to the 2910 and the contents of the **PC** register, **R** register and the **Stack** of the 2910. The values in this window are updated whenever the execution of the 2910 is stopped as after an Idle, Single Step or Clear function. The window will not continuously update while the 2910 is running.

5.4. A Design Example

Modern control microprograms usually have complex responsibilities, and the code is often quite complicated. We developed the LEASMB Microprogram Assembler with the hardware designer's needs in mind. We strove for simplicity and for a small set of powerful functions that support structured design. Before describing the details of the LEASMB language, we will present a design illustration to give the flavor of the language and its use.

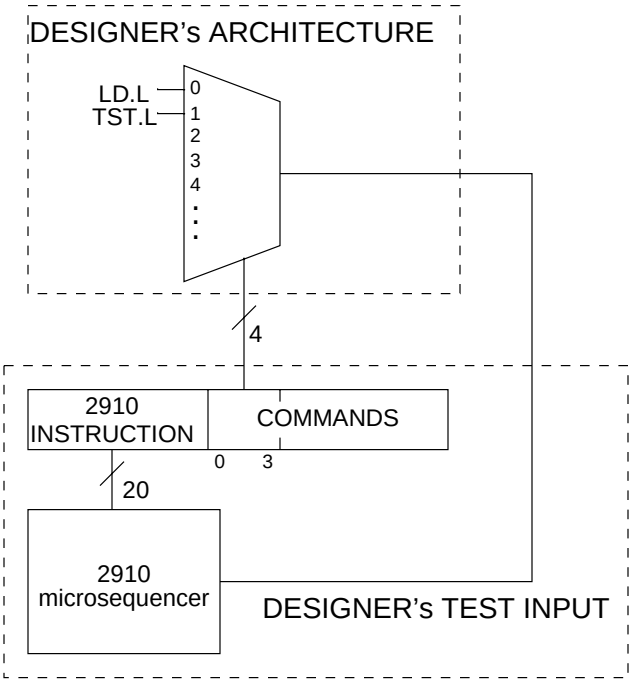
Hardware designers separate a design problem into an architecture (the registers, data paths, etc) and a control algorithm (for our purposes, a microprogram). The control program receives status information from the architecture, and delivers command information to the architecture. The command information is contained in the microinstruction, along with information governing the sequence of microinstructions. The microprogram assembler must give the designer a flexible and powerful language in which to express the command and sequencing information. The 2910

microprogram sequencer is at the core of Logic Engine control, and in this manual we will assume that you are familiar with this chip. Even so, since we present this example to familiarize you with the style of the LEASMB assembler, not the 2910, you should have little difficulty even if you are not yet familiar with the 2910.



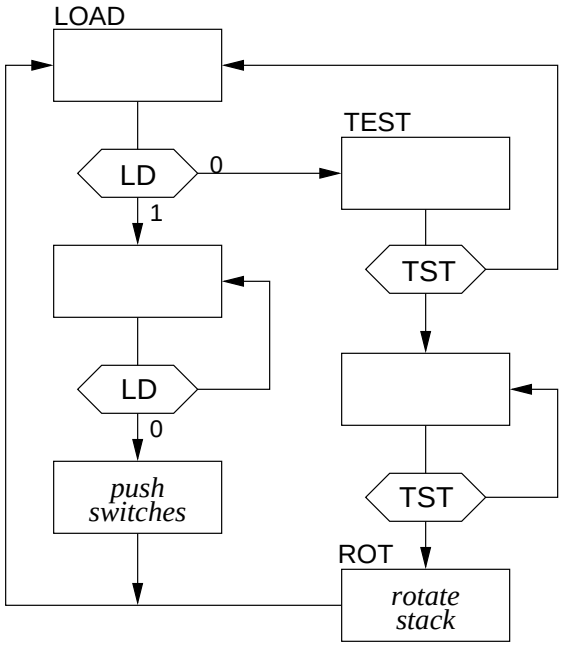
To eliminate unwanted detail, yet provide a real design example, our model uses hardware extracted from a larger design -- a high-speed stack-oriented computer. Our microprogram illustrates the control of a small set of tasks required in the debugging of this hardware. Fig. 1 shows the architecture for our illustration. We focus on the top three elements of the main stack. The larger task involves various movements of the data among these stack elements, but our illustration will deal with two operations: (a) a load operation that pushes new external data onto the stack for debugging purposes, and (b) a cyclic rotation of the top three stack elements.

Each stack element (which may contain as many bits as necessary) receives its input from a multiplexer (actually, a set of multiplexers, one for each bit, controlled identically). Each desired source for a given stack element becomes an input to that element's multiplexer. For testing purposes, in addition to the stack elements, inputs include the switch register on the display panel. We shall call the select signals for the multiplexers M0, M1, and M2, and we shall refer to the entire collection of multiplexer select signals as MUXCTL. In the full design, each stack element requires two control inputs; we call these pairs of bits S0CTL, S1CTL, and S2CTL, and we call the collection of stack element controls REGCTL.



To direct the actions of the 2910, the designer will usually need to make available several status signals from the architecture. The 2910 sequencer accepts a single signal as a test input. 2910 instructions may interrogate this signal and branch based on its value. The designer of a microprogrammed system faces the problem of extracting the desired signal from the architecture and presenting it to the sequencer at the proper time. Of the several mechanisms for selecting one signal from many, perhaps the easiest and simplest is to construct a multiplexer in the architecture. Since in each microinstruction we know which signal, if any, is required for testing, we may use some of the command bits in the microinstruction to serve as the select code for the multiplexer. The full design from which this example is taken requires about a dozen test inputs, so a 16-input multiplexer with a four-bit select code is appropriate. Fig. 2 shows the structure of the test input selection apparatus, centered around the test multiplexer INMUX. You will notice that we have decided to allocate the first four microinstruction command bits (bits 0-3) to control the test multiplexer. This is an arbitrary choice.

5.5. Developing the Control Program



The diagram to the left specifies the controller design for our example. If the LD button is pushed (and then released), the switches are pushed onto the stack. Otherwise, if the TST button is pushed, a *rotate* operation (permuting the top three stack items) is performed. The ASM has been designed with a microcode implementation in mind: each state has a single test and there are no conditional outputs.

An LEASMB microprogram assembly has two parts. In the **declaration phase**, we specify symbolic names for all variables and quantities of interest, and we describe the structure of the microinstruction. The **program phase** begins with the directive PROG and contains the microcode itself, in symbolic form. A microinstruction has a sequencer part, which drives the 2910 and determines the address of the next microinstruction, and usually a command part, in which the programmer specifies values for command signals that control the architecture.

The diagram to the left shows Fig. 3 shows a small portion of microcode created to load data from the Logic Engine display panel switch register and test the stack rotate operation. This code includes all the necessary declarations and microinstructions to support our example, and uses a variety of notations to illustrate features of the LEASMB microprogram assembler.

The SIZE directive specifies the number of command bits in the microinstruction (23 in our example). Command bit fields are defined with the COM directive. In Fig. 3, the definition of MUXCTL provides the following information: MUXCTL is a field of 11 bits, which we choose to refer to in our program with indices running from 10 (for the leftmost bit) to 0. In the command bit area of the microinstruction, MUXCTL occupies command bits 4 through 14. Thus MUXCTL(10) occupies command bit 4 of the microinstruction. We have chosen to describe the control in terms of logic (true or false), rather than in terms of voltages. In the definition of MUXCTL, we specify that, for each bit, truth is to be implemented as a high voltage (T=$7FF, hexadecimal 7FF). Further, we declare that, whenever any bit or bits of MUXCTL are not specifically referenced in a particular microinstruction, the default values for the bits are true logic levels (D=%TTTTTTTTTTT).

In many instances, we wish to deal with the set of command signals that controls a particular multiplexer in Fig 1. For our convenience, with the next three lines of the program we define three variables M0, M1, and M2. M0 is declared to be a field of three bits, numbered 2 to 0, which is equivalent (EQU) to bits 9 to 7 of MUXCTL. M1 and M2 are declared similarly to be equivalent to bits 6 to 4 and bits 3 to 2 of MUXCTL. With these definitions, we may refer to the field MUXCTL as a whole, or to subfields M0, M1, and M2, or to any bit in any of the fields.

In similar fashion, we declare the attributes of the collection of stack control signals REGCTL, and a supporting symbol LOAD3.

In examining Fig. 1, you will see that, in order to select stack element S0 as the output of multiplexer 1, we must supply the code 3 (binary %011) into the M1 select inputs. For convenience, we define a symbol M1S0 that will invoke (INV) the value 3 on the field M1. If in a microinstruction we wish to pass element S0 through multiplexer 1, we may simply write M1S0, thus assigning the value 3 (%011) to the field M1 in the microinstruction. In the microcode in Fig. 3, the instruction at location 002 illustrates this usage.

The definition of the symbol ROTATE shows how we may easily develop complex invocations. The use of ROTATE in the instruction at location 005 invokes the previously-defined invocations M0S2, M1S0, and M2S1, and invokes the value LOAD3 in the command bits defined for REGCTL. The advantage of such symbolic notations is that, when producing the microcode, we do not need to be concerned about the detailed location and values of the signals; the assembler will fill in the proper bits for us.

The declaration of the structures for the test input multiplexer appears at the end of the declaration phase in Fig. 3. INMUX, a field of 4 bits, defines the position in the command bits of the multiplexer controls. LD.L and TST.L describe values to be invoked upon the INMUX field, as discussed below.

The illustrative microprogram in Fig. 3 performs two debugging operations: loading the contents of the display panel switch register into stack element S0 (and pushing the stack), and performing a rotation of the top three elements of the stack. Two pushbuttons on the display panel, LD and TST, control the actions. When LD is pressed and released, the load and push operation will occur; pressing and releasing TST will enable the rotate operation. For each button, it is necessary to assure that the microcode performing the loads and rotates will be executed only once per button push. To accomplish this, the code at locations 000 and 001 performs a "single pulser" function for the LD button [Winkel and Prosser, The Art of Digital Design, Prentice-Hall, Inc., 1980, Chapter 6]. The code at locations 003 and 004 performs a similar function for the TST button. The "*" in the microinstruction operand fields means "value of the assembler's location counter".

In the declaration phase, we have specified that a use of TST.L, such as at locations 003 and 004, should invoke the value 1 for command field INMUX; LEASMB will take care of generating the proper values for the test multiplexer select lines. Using information supplied in the declarations and in a given microinstruction, LEASMB will determine the logic level entering the 2910 sequencer's test signal input that should cause a jump, and will translate this logic level into the proper voltage. For instance, the microinstruction at location 003 implies that a jump should

occur if TST.L is false. The definition of TST.L states that the input test signal selected by TST.L has truth represented as a low voltage (T=%L). Thus LEASMB will conclude that, in order to jump at location 003, the incoming test signal must be a high voltage level.
With this informal description of the language, you should be able to follow the test program. We hope that you will appreciate how LEASMB can help define useful structures and suppress unwanted detail. In the next sections, the LEASMB language is presented in full.

```
            ID   LEASMB_demo

* SAMPLE DECLARATIONS AND SAMPLE MICROCODE

            SIZE 23; number of command bits
            MODE LOGIC

* COMMAND FIELD DECLARATIONS

MUXCTL(10:0) COM  (4:14),T=$7FF,D=%TTTTTTTTTTT
M0(2:0)      EQU  MUXCTL(9:7); mux 0 select signals
M1(2:0)      EQU  MUXCTL(6:4); mux 1 select signals
M2(1:0)      EQU  MUXCTL(3:2); mux 2 select signals
M0S2         INV  M0=5; select reg S2 through mux 0
M0SWR        INV  M0=0; select switch reg through mux 0
M1S0         INV  M1=3; select reg S0 through mux 1
M2S1         INV  M2=1; select reg S1 through mux 2
REGCTL       COM  (15:22), T=%HHHHHHHH,D=%FFFFFFFF
LOAD3        EQU  %11111100; load S0,S1,S2
ROTATE INV   M0S2,M1S0,M2S1,REGCTL=LOAD3; rotate stack

* INPUT TEST MUX CONFIGURATION (2 input signals defined)

INMUX(3:0)   COM  (0:3),T=%HHHH,D=0
LD.L INV     INMUX=0,T=%L
TST.L INV    INMUX=1,T=%L

PROG

LOC XDDDI CCCC CC
000 ORG  0
000 BEGIN EQU  *
000 10033 0FFE 00  LOAD JUMP TEST IF LD.L=%F
001 50013 0FFE 00  JUMP *    IF LD.L=%T
                   JUMP BEGIN;M0SWR,M1S0,M2S1,
002 30003 086F F8             REGCTL=LOAD3; push switches onto stack
003 10003 1FFE 00  TEST JUMP LOAD IF TST.L=%F
004 50043 1FFE 00  JUMP *    IF TST.L=%T
005 30003 0D6F F8  ROT JUMP BEGIN;ROTATE; rotate top 3 stack elements

END

0 ERROR(S) DETECTED
```

5.6. The Micro Assembly Language

In the language description that follows, **`bold courier`** letters are used for concrete syntax.
NAMES

Symbolic names in LEASMB may contain upper case or lower case letters, digits, "_", or "**.**". The first character must be a letter. Names may be any length. Upper case letters are distinct from lower case letters. The following symbols are reserved, and must not be used as symbolic names:

IF **T** **F** **H** **L**

SUBSCRIPTS

A subscript is one of the following forms:

<*subscript*> ::= **(** <*index*>$_1$ **)** | **(** <*index*>$_1$ **:** **<***index*>$_2$ **)**

where <*index*>$_1$ and <*index*>$_2$ are expressions with numeric values. Subscripts are used with command variables and as command bit range specifiers.
NAME TYPES

Depending on how it is defined, a name may represent a constant, a control poing, a command field, or an invocatoin.l

A **constant name** (defined with the EQU statement) may be of type numeric constant, voltage constant, or logic constant, governed by the type of the defining expression. Depending on the context, a numeric constant may be treated as a number, a voltage value, a logic value, or a program label.

A **program label**, which names a location in the microcode, is usually defined by its appearance in the label field of a microinstruction statement. It may also be defined using the EQU and ORG statements.

A **command variable** describes a field of contiguous command bits in the microinstruction. It is usually defined with the COM statement, but may also be defined with the EQU statement. It may appear as a subscripted or unsubscripted symbolic name:

<*command variable*> ::= <*name*> | <*name*> <*subscrip*t>

The unsubscripted form implies a range of command bits, as determined by the context. The subscripted form with a single index refers to exactly one command bit; the subscripted form with two indices refers to a range of command bits. Examples of command variable names are:

```
X.REX(3:7)
PROD
Headload.L(2)
ALUCTL(5:0)
```

An **invocation variable** describes a particular pattern to appear in specified command bits in the microinstruction. It is defined with the INV statement or with the EQU statement, and is a simple unsubscripted name.

CONSTANTS

Constants may be numeric (expressed in decimal, hexadecimal, or binary), voltage, or logic. There is also one special program label constant.

A **decimal numeric constant** appears as a number between 0 and 65523. Example: `2544`.

A **hexadecimal numeric constant** appears as a number between 0 and FFFF, preceded by a "$" character. Example: `$4FA`.

A **binary numeric constant** appears as a string of 1's and 0's, preceded by "%". The maximum usable length of a binary constant is 16 bits. Example: `%111010`.

A **voltage constant** appears as a string of up to 16 "H" (for high voltage) or "L" (for low voltage) characters, preceded by "%". Within LEASMB, "H" is maintained as a 1, and "L" as a 0. Example: `%HHLHLLL`.

A **logic constant** appears as a string of up to 16 "T" (for true) or "F" (for false) characters, preceded by "%". Within LEASMB, "T" is maintained as a 1, and "F" as a 0. Example: `%FTFFFFTTT`.

An asterisk "*" in the operand field of a microinstruction means "current value of the LEASMB microinstruction location counter", and is of type **program label**.

EXPRESSIONS

Expressions may involve the arithmetic operators "**+**" (add) and "**-**" (subtract), and the logical operator "**/**" (one's complement). Any numeric, voltage, or logic constant may be preceded by "/" to generate the logical complement of the constant. Expressions are evaluated in strict left-to-right order, with no operator precedence. Grouping parentheses are not permitted in LEASMB. LEASMB maintains constants as 16-bit unsigned quantities. LEASMB does not distinguish between negative quantities and large positive quantities (greater than 32767). To avoid unwanted results, the programmer should arrange for all expressions to evaluate to positive quantities.

LEASMB evaluates expressions involving arithmetic "+" and "-" as shown in Table 1.

From Table 1, we note the following:

(f) No operations are allowed on command variables or invocation variables.

(g) Voltage and logic types may not be mixed in an expression.

(h) A numeric type, if combined in an expression with an operand of another type, will take that operand's type.

(i) Arithmetic is done in two's complement notation; "-" does not perform the logical (one's) complement function, which is performed by "/".

1st operand	*Operator*	*2nd operand*	*result*
-	+ or -	numeric	numeric*
-	+ or -	voltage	voltage
-	+ or -	logic	logic
numeric	+ or -	numeric	numeric*
numeric	+ or -	voltage	voltage
numeric	+ or -	logic	logic
numeric	+	program label	program*
numeric	-	program label	(illegal)
voltage	+ or -	numeric	voltage
voltage	+ or -	voltage	voltage
voltage	+ or -	logic	(illegal)
voltage	+ or -	program label	(illegal)
logic	+ or -	numeric	logic
logic	+ or -	voltage	(illegal)
logic	+ or -	logic	logic
logic	+ or -	program label	(illegal)
program label	+ or -	numeric	program*
program label	+ or -	voltage	(illegal)
program label	+ or -	logic	(illegal)
program label	+	program label	(illegal)
program label	-	program label	numeric

**designates the most useful forms of expressions.The other forms, although legal in LEASMB, have limited utility.*

LEASMB PROGRAM STRUCTURE

Structure of the source program file

A program is contained in a DOS text file, usually prepared using a text editor. A program consists of a sequence of statements. The statements are partitioned into two phases: In the **declaration phase**, the programmer describes the structure of the microinstruction and declares symbolic names to describe components of the structure. In the **program phase**, which, if present, must follow the declaration phase, the programmer writes the microinstructions that will form the object code destined for the Logic Engine writable control store.

Structure of source program statements

Each statement must begin on a new line. Statements consist of three major fields: label, operation, and operand. Major fields are separated by one or more blanks. If a statement has a label field, it must begin in column 1; otherwise, column 1 must be blank. Within numeric or symbolic elements, no blanks should appear. Elements must be separated by suitable punctuation, which may be one of the characters , ; = or blank. Elements may be preceded or followed by any number of blanks. With these provisos, the statement format is free-field. Although not required by LEASMB, you will usually wish to line up the operation fields, to improve the appearance of your listing.

Comments

Comment lines, beginning with an asterisk "*" in column 1, may appear preceding or following any statement. They appear on the output listing, but are otherwise ignored by LEASMB. Comments may also be included at the end of any statement. Such comments must be separated from the statement's major fields by a semicolon.

LEASMB ASSEMBLY DIRECTIVES

Declaration segment	
ID	Specifies the program name
SIZE	Specifies the number of command bits
MODE	Specifies logic or voltage expression mode
COM	Defines a command bit field
INV	Defines a command assertion pattern
EQU	Equates a symbol to a value
Program segment	
PROG	Marks start of program phase

ORG	Specifies load location of object code
EQU	Equates a symbol to a value
Either segment	
END	End of source program
TITLE	Header title for listing

DESCRIPTION OF LEASMB STATEMENTS

Each LEASMB assembly directive defines a statement of the same name, with the assembly directive's mnemonic appearing in the operation code field of the statement.

ID statement:

The operand field is a symbolic name representing the name of the program. One ID statement may appear anywhere in the declaration phase. The program name in the ID statement appears in the object file. If the program has no ID statement, then the object file will contain a name of blanks.

SIZE statement:

The operand field is an expression that evaluates to an integer between 0 and 39. This integer is the number of **command bits** in the microinstructions to be created by LEASMB. The SIZE statement, if present, must appear prior to any COM, INV, or EQU statement. If no SIZE statement appears, LEASMB assumes that the number of command bits is 0.

MODE statement:

The operand field consists of the word "LOGIC" or the word "VOLTAGE". In several assembly contexts, numeric values may represent either logic or voltage values. This statement declares the mode for numeric values whose mode is not otherwise specified by the context. One MODE statement may appear anywhere in the declaration phase, and applies to the entire program. In the absence of a MODE statement, LEASMB uses a mode of LOGIC.

COM statement:

The COM statement allows the programmer to define the nature of a contiguous field of command bits. The form is:

[<*command variable*>] ::= **COM** <*command bit range*> [,<*voltage values*>] [,<*default values*>]

A label field is optional, and if present represents a simple or subscripted <*command variable*>. The operand field begins with <*command bit range*>, a subscript specifying a command bit or a

range of command bits. The values of the <*command bit range*> indices must be from 0 to 39, and if the subscript indicates a range, then the second index cannot be smaller than the first index. The range size of the <command bit field> must be no greater than 16. For this discussion, call the size of the range N (N is 1 to 16). If the <command variable> in the label field is unsubscripted, then it is assumed to have a range of size N, with index values of 0 through N-1. If the <command variable> is subscripted, then the declared size of the subscript range must be N, agreeing with the range size of <command bit range>. The initial (leftmost) index of <command variable> corresponds to the initial (leftmost) index of <command bit range>. Subsequent index values of <command variable> may increase or decrease, depending on whether the second index is greater or less than the first index. All indices must be positive. Any symbols encountered in the operand field must have been previously defined.

Consider two illustrations:

```
VAR1          COM (4:9)
VAR2(32:29)   COM (12:15)
```

VAR1 refers to command bits 4 through 9 (a range size of 6 bits). VAR1 has implied indices of 0 through 5, with VAR1(0) corresponding to command bit 4. VAR2 refers to command bits 12 through 15 (4 bits), and has an explicitly stated range of 32 through 29 (decreasing), with VAR2(32) corresponding to command bit 12, VAR2(31) to command bit 13, etc. The range sizes must match.

Following the command bit range may appear up to two optional fields, in either order, each preceded by a comma. The fields specify (a) the default values for each command bit and (b) the voltage values for assertion (truth) for each command bit.

<voltage values> for asserting each bit are specified by "**T=**" followed by an expression that evaluates to a numeric or voltage value. A numeric value is taken to be a voltage specification, regardless of the MODE declaration. An alternative notation, for specifying the negation (false) values for each bit, uses "**F=**" instead of "**T=**". If the <voltage values> field is absent, LEASMB assumes that assertion of each bit requires a high voltage.

<default values> are specified using the notation "**D=**" followed by an expression that evaluates to a numeric, voltage, or logic constant. If the expression is numeric, then the value takes on the type specified by the MODE statement. If <default values> is absent, then LEASMB assumes that the default for each bit is a low voltage for VOLTAGE mode, and is the voltage for a false logic level for LOGIC mode. In a microinstruction, LEASMB will use the default values for any command bits that are not specifically referenced.

The programmer should take care not to overlap <*command bit range*>s in COM declarations, since such overlap can cause ambiguities as to the correct default and voltage values. (The programmer can create alternative names for command bit fields and subfields with the EQU statement.)

LEASMB views the value of the <default values> and <voltage values> expressions as binary numbers, and assigns the least significant bit to the rightmost bit of <command bit range>. Extra significant bits on the left of those required to fill the field are discarded, but will cause a warning

message.

Here are some examples of command statements:

```
VAR3(6:0)       COM (0:6),D=%1011100,T=%1110000
INPUTMUX(3:0)   COM (11:14),D=%TTFF
REGLOAD         COM (15:16),D=2,T=%LL
HALTFF.SET      COM (10)
```

The effect of these definitions is summarized in the following table:

	indices	*Commad bits*	*Mode VOLTAGE*		*Mode LOGIC*	
			truth	*default*	*truth*	*default*
VAR3	6-0	0-6	HHHLLLL	HLHHHLL	HHHLLLL	HLHLLHH
INPUTMUX	3-0	11-14	HHHH	HHLL	HHHH	HHLL
REGLOAD	0-1	15-16	LL	HL	LL	LH
HALTFF.SET	0	10	H	L	H	L

We recommend that you study the table carefully, so that you fully appreciate the power and use of the notations at your disposal.

In using command variables in other statements, you may use a subscripted or unsubscripted form. An unsubscripted usage implies the entire defined range of the variable. In a subscripted usage, the specified range must be completely within the range defined for the command variable.

INV Statement

The INV statement is used to define an invocation variable which, when used in a microinstruction, will cause a specified value to be imposed on specified command bits. This is an exceedingly powerful notation, and is the key to programming in a structured, disciplined manner. The INV statement has this form:

<*invocation variable*> **INV** <*specification list*> [**,** <*input truth value*>]

Any symbols appearing in the operand field must be previously defined. The <*invocation variable*> is a simple, unsubscripted name. The <*specification list*> consists of one or more <*specification*>s, separated by commas. Each <*specification*> is one of the following forms:

<*specification*> *::=* <*command bit range*>
<*command variable*>
<*command bit range*> **=** <*code*>
<*command variable*> **=** <*code*>
<*invocation variable*>

The form <*command bit range*> implies that, when the invocation variable is used, the specified

command bits will be **asserted**.

<command variable> may be subscripted or unsubscripted, and the form implies that the specified command bits will be **asserted**. An unsubscripted *<command variable>* implies the entire defined range for that variable.

*<command bit range>***=***<code>* implies that the specified command bits will receive the voltages specified by *<code>*. *<code>* is an expression that evaluates to a numeric value (with logic or voltage mode implied by the MODE declaration), a logic value, or a voltage value.

*<command variable>***=***<code>* implies that the specified command bits will receive the voltages specified by *<code>*. *<code>* is an expression that evaluates to a numeric value (with logic or voltage mode implied by the MODE declaration), a logic value, or a voltage value.

<invocation variable> implies that the invocation specified in the definition of that variable will be performed.

Consider the following declarations:

```
VAR4  COM (5:9),T=%HLHHL
VAR5  COM (10:13),T=%LLHH
      COM (0:3),T=%HLHL
INV1  INV (6:8)
INV2  INV VAR4=%HHHLL
INV3  INV VAR4(0:4)
INV4  INV (0:3)=%1100
INV5  INV VAR5(1:3)=%TFT
INV6  INV INV1,VAR5=%0110,INV4
```

The effect of the invocation statements is presented in the following table:

Invocation variable	Command bits affected	VOLTAGE mode command bits	LOGIC mode command bits
INV1	6 to 8	LHH	LHH
INV2	5 to 9	HHHLL	HHHLL
INV3	5 to 9	HLHHL	HLHHL
INV4	0 to 3	HHLL	HLLH
INV5	11 to 13	LLH	LLH
INV6	6 to 8	LHH	LHH
	10 to 13	LHHL	HLHL

	0 to 3	HHLL	HLLH

The programmer should take care not to refer to overlapping command bit ranges in an INV statement, as LEASMB may produce unpredicted results.

<input truth value> in invocation statement:

The 2910 can accept one input signal for testing to help determine which 2910 instruction option -- pass or fail -- is executed. The Logic Engine requires that this signal appear on the Designer's Condition Code input. Although invocations have quite general application in microcode development, frequently the designer will wish to use an invocation to select the particular signal to appear on Designer's Condition Code. The invoked code selects that particular signal from among the designer's collection of possible test input signals. The mechanism by which the selection occurs is the designer's responsibility; the selection apparatus may use a multiplexer, three-state bus, or other method, controlled from the Logic Engine microcode with an invocation variable. Each potential input test signal has its own code.

With <input truth value>, LEASMB allows the programmer to specify the logic convention for the particular invoked signal. <input truth value> has the form:

<*input truth value*> := **T=**<*voltage expression*> | **F=**<*voltage expression*>

where <*voltage expression*> evaluates to a numeric value (with implied VOLTAGE mode) or a voltage value. If <*input truth value*> is absent, LEASMB assumes a **T=%H** convention.

Note that the <*input truth value*> has no direct connection with the <*code*>; <*input truth value*> describes the logic convention for a signal that will presumably appear on designer's condition code when the particular code is invoked in the specified command bits.

Examples of <*input truth value*> usage:

The first JUMP microinstruction requires a jump to location ABC if TESTSIG.a is false at the

```
TESTSIG.a  INV  (15:18)=%LLHL,T=%L
TESTSIG.b  INV  (15:18)=%LLHH,T=%H
           ...
           JUMP ABC IF TESTSIG.a=%F
           JUMP XYZ IF TESTSIG.b=%T
```

time of execution of the microinstruction. The definition of TESTSIG.a declares that the incoming test signal has T=%L, so false is a high voltage. LEASMB will create proper object code to **invoke** TESTSIG.a (with voltages %LLHL on command bits 15 to 18), and will recognize that a high voltage on Designer's Condition Code is required for the jump to occur.

In similar manner, the second JUMP microinstruction requires truth on TESTSIG.b for the jump to occur. LEASMB will invoke TESTSIG.b (with voltages %LLHH on command bits 15 to 18), and will arrange for a high voltage on Designer's Condition Code to cause a jump.

These powerful LEASMB structures can free the programmer from many tedious details of bit positions, codes, and voltage values, allowing the designer to concentrate fully on the higher-level aspects of the control program.

EQU Statement

The EQU statement allows the programmer to define values for names, and to equate variable names. For assigning values to names, the form of the statement is:

<*name*> **EQU** <*expression*>

Any symbols appearing in the expression must be previously defined. The <*name*> must be a simple, unsubscripted name. The <*expression*> must evaluate to a numeric, logic, voltage, or program label value, which will become the value and type of the specified <*name*>. Thus this form of EQU can define <*constant name*>s and <*program label*>s.

For equating variable names, the form of the EQU statement is:

<*variable*> **EQU** <*specifier*>

Any symbols appearing in the operand field must be previously defined. <*variable*> may be a simple or subscripted name. The forms allowed for <*specifier*> and the type of the resulting <*variable*> are:

<*specifier*> :=	<*command bit range*>	produces a <*command variable*>
	<*command variable*>	produces a <*command variable*>
	<*invocation variable*>	produces an <*invocation variable*>

For a command variable definition, a subscript range on the defined variable may be explicitly provided, in which case the range size must match the range size of the specifier. If a range is not specified, then the defined command variable assumes the same range size as the specifier, with indices starting at 0. Invocation variables may not be subscripted.

Within the **definition phase** of a source file, EQU may be used to define numeric, logic, or voltage constants, and to create command and invocation variables.

Within the **program phase** of a source file, EQU may be used to define numeric constants or program labels.

The following statements contain illustrations of valid EQU statements:

```
VAR10      COM(10:20)
INV10      INV         VAR10=$1D
TURNON     EQU         %HLLHL
ACCLR      EQU         $10
SUB1(3:0)  EQU         VAR10(0:3);   SUB1 is a 4-bit subfield of VAR10
SAME10     EQU         INV10
           ...
           CONT
```

```
ABC         EQU         *+2;          ABC is location of CRTN instruction + 2

            CRTN
```

PROG statement:

The PROG statement must be the first statement in the program phase of the source file. It marks the end of the declaration phase and the beginning of the program phase. It has no label field or operand field.

ORG statement:

The ORG statement sets the LEASMB microinstruction location counter, and thus defines the writable control store address of the next microinstruction in the program. The form of the ORG statement is:

[<*program label*>] **ORG** <*expression*>

The required operand field is an expression of type numeric constant or program label. LEASMB truncates the value of the <*expression*> to 12 bits, and issues a warning message if significant bits are lost. A label field is optional; if present, the symbolic name will be given the value of the operand field expression, with type program label.

If ORG sets the location counter backwards, the programmer may (intentionally or unintentionally) overlay previous microinstructions. The last usage of an address will prevail. LEASMB does not issue an error or warning message.

END statement:

The END statement is the last statement in the program. It marks the end of the program phase, if any, or the end of the declaration phase if there is no program phase. It has no label field or operand field.

TITLE statement:

The TITLE statement allows the programmer to specify a title that will appear at the head of each new page after the occurrence of the TITLE statement. The title begins with the first non-blank character in the operand field, and may be up to 74 characters in length. The TITLE statement must be contained on one source line. A blank or void title will disable the titling feature. TITLE statements may appear anywhere in the source program; they do not themselves appear in the listing, and do not cause a page eject. TITLE statements have effect only if the assembly page option is operating.

THE MICROINSTRUCTION STATEMENT

Each microinstruction statement in the source program results in the production of one object code microinstruction. LEASMB assigns microinstructions to consecutive locations in writable control store, starting with 0, unless directed otherwise with the ORG assembly directive. The

microinstruction statement has the form:

[<*program label*>] <*sequencer field*> [**;**<*command list*>]

If a comment appears at the end of a microinstruction statement, then a command list must be present, even if it is null. Thus, if a comment is present, two semicolons appear, one marking the start of the (possibly null) command list, and the other marking the start of the comment.

The sequencer field

<*sequencer field*> allows the programmer to direct the activities of the 2910, the device whose task is to produce the address of the next microinstruction. The 2910 instruction set consists of sixteen basic operations, each with two options, "pass" and "fail", governed by the status of 2910 inputs CCEN.L and CC.L. The LEASMB <*sequencer field*> provides the ability to exert complete control of the 2910. As you study this section, you may wish to refer to the sample program that appears in section XXX.

The structure of <*sequencer field*> is:

<*sequencer field*> ::= <*operation*> [<*D-field*>] [**IF** <*test condition*>]

<operation> is one of the standard 2910 I-field mnemonics, or one of a set of alternative mnemonics defined within LEASMB. Allowable forms are:

<*operation*> ::= <*I-field mnemonic*>
<*I-field mnemonic*>**,PASS**
<*I-field mnemonic*>**,FAIL**

where the latter two forms will force the 2910 to execute its pass or fail option, respectively. Table 2 shows the 2910 I-field mnemonics recognized by LEASMB.

Table 2. 2910 Instructions.

I-field value	Mnemonic	Function
Standard 2910 mnemonics		
$0	**JZ**	Jump to location 0
$1	**CJS**	Conditional jump to subroutine at PL address
$2	**JMAP**	Jump to map address
$3	**CJP**	Conditional jump to PL address
$4	**PUSH**	Push with conditional load of counter
$5	**JSRP**	Jump to subroutine at R address or at PL address
$6	**CJV**	Conditional jump to vector address
$7	**JRP**	Conditional jump to R address or PL address
$8	**RFCT**	Repeat loop if counter is non-zero
$9	**RPCT**	Jump to PL address if counter is non-zero
$A	**CRTN**	Conditional return from subroutine
$B	**CJPP**	Conditional jump to PL address and pop
$C	**LDCT**	Load counter and continue
$D	**LOOP**	Test end of loop
$E	**CONT**	Continue
$F	**TWB**	Three-way branch
Alternate forms supported be LEASMB		
$3	**JUMP**	Equivalent to CJP

The programmer may specify the contents of the 2910 D-field. The optional <*D-field*> is an expression of type numeric constant or program label. The expression is truncated to 12 bits; if significant bits are dropped, LEASMB issues a warning message. If <*D-field*> is absent, LEASMB inserts the value 0.

The optional **IF** <*test condition*> field allows the programmer to control the choice of pass or fail option within the 2910. Refer to the earlier discussion of the INV Statement.

<*test condition*> has one of these forms:

<*test condition*> ::= <*invocation variable*>
<*invocation variable*>**=**<*test input value*>

where <test input value> is an expression that evaluates to a one-bit numeric (1 or 0), voltage (H or L), or logic (T or F) constant. (Expression values larger than one bit are truncated, with a warning message.)

The purpose of <*invocation variable*> is to issue the proper command bit values to select a desired test input signal from the designer's hardware.

<*test input value*> allows the programmer to specify the value of the test input signal that will cause the 2910 to execute its pass option. If the **=**<*test input value*> phrase is absent, then LEASMB assumes the value T.

As an illustration, consider the following definitions and microinstruction statements:

```
ACZERO   INV   (4:7)=%1011,T=%H
COMPARE  INV   (4:7)=%0010,T=%L
         ...
         CJP   XYZ IF ACZERO = %F
         CRTN  IF COMPARE
```

The use of ACZERO in the CJP instruction invokes the value %1011 (= %TFTT = %HLHH) onto command bits 4 through 7. The designer should arrange that this pattern in command bits 4 through 7 will cause a signal ACZERO to appear at the Designer's Condition Code input. Further, the declaration states that when this occurs, the incoming test signal uses a high voltage for truth. In the CJP instruction, the jump is to occur if ACZERO is false (a low voltage). LEASMB will arrange that a low voltage delivered at the Designer's Condition Code input will cause the jump to occur.

In the second example, the use of COMPARE causes the value %0010 (= %FFTF = %LLHL) to appear on command bits 4 through 7, which we infer is going to cause a signal COMPARE to appear at the Designer's Condition Code input. The 2910 will execute a Return (the pass option for CRTN) if this incoming signal is true. Since the definition of COMPARE states that truth is a low voltage, LEASMB will arrange that the pass option will occur when a low voltage appears at the Designer's Condition Code input.

The **IF** <*test condition*> phrase will have no effect if used in a microinstruction statement in which the programmer has used the explicit **PASS** or **FAIL** form of <*operation*>. The simple form <*I-field mnemonic*> without the **IF** <*test condition*> phrase will cause the 2910 to execute its pass option. The following table summarizes the programmer's control of the 2910's pass and fail options:

Form	*Result*
<*I-field mnemonic*>	Pass
<*I-field mnemonic*>**,PASS**	Pass
<*I-field mnemonic*>**,FAIL**	Fail
<*I-field mnemonic*>...**IF** <*test condition*>	Governed by Designer's Condition Code input

The Microinstruction Command List

In the command list, the programmer specifies signal values for command bits. The purpose of the entire microprogram is to deliver an orderly sequence of command bit values to the designer's architecture. <command list> consists of a list of <command specification>s, each separated by a comma, where each <command specification> has one of these forms:

<*command specification*> ::= <*command bit range*>
<*command variable*>
<*command bit range*>**=**<*field value*>
<*command variable*>**=**<*field value*>
<*invocation variable*>

Note that these structures are equivalent to those permitted in defining an invocation variable.

The form <*command bit range*> implies that the specified command bits will be **asserted**.

<*command variable*> may be subscripted or unsubscripted, and the form implies that the specified command bits will be **asserted**.

<*command bit range*>**=**<*field value*> implies that the specified command bits will receive the voltages specified by <*field value*>. <*field value*> is an expression that evaluates to a numeric value (with logic or voltage mode implied by the MODE declaration), a logic value, or a voltage value.

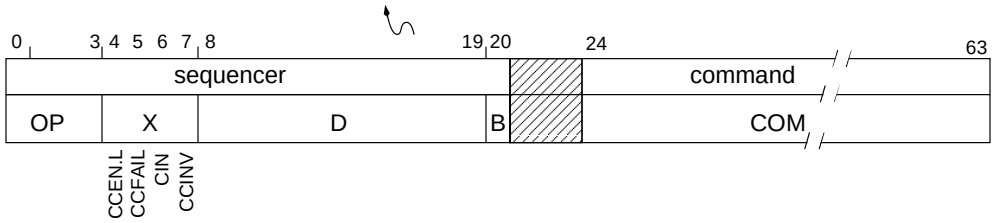
<*command variable*>**=**<*field value*>implies that the specified command bits will receive the voltages specified by <*field value*>. <*field value*> is an expression that evaluates to a numeric value (with logic or voltage mode implied by the MODE declaration), a logic value, or a voltage value.

<*invocation variable*> implies that the invocation specified in the definition of that variable will be performed.

In the object code for a microinstruction, all command bits not specifically addressed in a command specification will receive their default voltage values.
STRUCTURE OF A LOGIC ENGINE OBJECT CODE MICROINSTRUCTION

An LEASMB microinstruction consists of a fixed-format sequencer field (20 bits) and a command

bit field of size specified by the designer.



The structure of the **sequencer field** is as follows (bits are numbered from left to right, beginning with bit 0):

Field		*range*	*Description*
I (4 bits):		0-3	2910 instruction code field (I-field) (high-active
X (4 bits)	CCEN.L:	4	2910 Condition Code Enable (low-active)
	CIN:	5	2910 Carry In (high-active)
	CCFAIL:	6	Condition Code Fail (high-active)
	CCINV:	7	Condition Code Invert (high-active)
D (12 bits):		8-19	2910 address field (D-field) (high-active)
B (1 bit):		20	Breakpoint (high active)

The structure of the **command bit field** is:

µ*Instruction bit*	*Command bit*
24	*0*
25	1
⋮	⋮
63	39 (maximum)

The D-field is specified by the programmer as an expression of type program label or numeric constant.

The I-field is generated by LEASMB from the mnemonic in the operation code field of the micro-instruction.

The B-field is generated by LEASMB when a breakpoint is requested for that microinstruction. If breakpoints are enabled, the clock will be stopped before the execution of a microinstruction that

has this bit set.

The X-field

The bits in the X-field allow the programmer to exert, through the prescribed LEASMB syntax, detailed control over certain of the 2910 microprogram sequencer features. In normal programming, LEASMB will set the X-field bits automatically, and the programmer will not deal with them. The use of the X-field is explained below. The discussion assumes you are familiar with the operations of the 2910.

For each of its instructions, the 2910 sequencer provides two sub-operations, PASS and FAIL. The 2910 performs the PASS sub-operation if its CCEN.L input is high (false) or if its CC.L input is low (true). The 2910 performs the FAIL sub-operation under other conditions, namely, if its CCEN.L input is low (true) and its CC.L input is high (false). A designer may wish to control the choice of sub-operation by forcing a PASS, forcing a FAIL, or allowing his Designer's Condition Code input to determine the choice. The Logic Engine microinstruction provides the means to select each of these alternatives easily.

The designer may force the 2910 to select the PASS sub-operation by setting the CCEN.L bit to 1 (false) in the microinstruction. (LEASMB performs this bit setting automatically if you specify the PASS option in the sequencer field of a symbolic microinstruction, or if you fail to specify a conditional clause.)

The designer may force the 2910 to select the FAIL sub-operation by setting the CCEN.L bit to 0 (true) and the CCFAIL bit to 1 (true) in the microinstruction. (LEASMB performs this bit setting automatically if you specify the FAIL option in the sequencer field of a symbolic microinstruction.)

The designer may allow his Designer's Condition Code signal to control the choice of PASS or FAIL sub-operation by setting the CCEN.L bit to 0 (true) and the CCFAIL bit to 0 (false) in the microinstruction. (LEASMB performs this bit setting automatically if you specify the conditional "IF" clause in a symbolic microinstruction.)

Here is a summary of the bit values required to influence the choice of sub-operation:

	CCEN.L	CCFAIL
Force a PASS	1 (high, false)	---
Force a FAIL	0 (low, true)	1 (high, true)
Act on incoming Designer's Condition Code signal	0 (low, true)	0 (low, false)

The signal appearing on the Designer's Condition Code input may be low-active (asserted by a low voltage) or high-active (asserted by a high voltage). The Logic Engine microinstruction contains the CCINV bit to allow the programmer to specify which voltage polarity is in effect. To specify that Designer's Condition Code is low-active, the designer may set CCINV to 0 (false). To specify that Designer's Condition Code is high-active, the designer may set CCINV to 1 (true). (LEASMB sets CCINV automatically, based on the declarations for the signal referenced in the

"IF" clause of a symbolic microinstruction.)

In the Logic Engine, the logic equation governing the signal entering the 2910's Condition Code (CC.L) pin is

2910.CONDITION.CODE = (DESIGNER.CONDITION.CODE **xor** CCINV) **and** $\overline{\text{CCFAIL}}$

The CIN bit of the microinstruction connects directly to the 2910 CIN pin. Normally, this bit will be a 1 (high, true), implying normal sequencing if a branch does not occur. Normal sequencing is inhibited if CIN is 0 (low, false). The programmer may override the normal value of CIN by placing a '**/**' preceding the 2910 operation mnemonic..

2910 Signal	*Disposition*
D (12 bits)	Input. When PL.L is asserted, the Logic Engine pipeline register provides these bits. When PL.L is negated (i.e., when MAP.L or VECT.L is asserted), the designer's circuit must supply these bits.
I (4 bits)	Input received from Logic Engine pipeline register.
CC.L	Input received from Logic Engine. Derived from Designer's Condition Code, CCINV, and CCFAIL.
CCEN.L	Input received from Logic Engine pipeline register.
CI	Input received from Logic Engine pipeline register. Programmer may specify the value through LEASMB.
RLD.L	Input available at 2910 chip on Logic Engine board. Not used by Logic Engine.
OE.L	Input controlled exclusively by Logic Engine.
CP	Input supplied exclusively by Logic Engine. Designer must supply the System Clock from which CP is derived.
Y (12 bits)	Output available on 60-pin flat-cable connector. Used by Logic Engine.
FULL.L	Output available at 2910 chip on Logic Engine board. Used by Logic Engine.
PL.L	Output available at 2910 chip on Logic Engine board. Used by Logic Engine.
MAP.L	Output available on the 2910 PLD. Not used by Logic Engine.
VECT.L	Output available on the 2910 PLD. Not used by Logic Engine.
Note: Directions are relative to the 2910.	

I. Logic Engine Programmer's Interface

A. Introduction

The Logic Engine Programmer's Interface (LEPI) is a library of Microsoft C routines that allow the user to monitor and control the LE board from the PC host. At the lowest level, the LEPI allows the user to read from the lights and write to the switches, buttons and pipeline of the LE board. It can also control the clock on the LE board. At a higher level, the LEPI provides a powerful way to name and invoke values onto groups of signals. Some of the many uses of the LEPI include, test vector generation, providing control signals to systems in lieu of microcode, the migration of software simulations to hardware implementations and high level diagnostics.

B. How the LEPI Works

The LE board contains 128 lights, 16 toggle switches, 12 buttons and 40 bits of pipeline. The PC host has the ability to read from the lights and write to the switches, buttons and pipeline. With this ability any signal that is wired to a light can be monitored by the PC host and any input signal that is wired to a switch, a button or the pipeline can be controlled by the PC host. The LE board also contains a clock which can be stopped, started and pulsed under the control of the PC host.

From the perspective of the host, the switches and buttons behave identically and will both be refered to as switches in this document. Associated with each switch is a register which can be written by the PC host. The output of each switch is connected to the output of its corresponding register. Normally, the output of the register is disabled and the output of the switch is enabled. The PC host can take control of the switches by disabling the output of the switches and enabling the output of the registers. In addition to the 28 registers associated with the switches, there are 4 hidden registers which are accessible from the PC host which do not have an associated switch or button.

The pipeline is part of the microcontroller which is on the LE board and is used to control input signals to a design. The PC host can however, write directly to the pipeline to provide these signals.

The PC host has the ability to stop a running clock, pulse a stopped clock, or start a stopped clock by writing to the clocks control register. When the clock is stopped the clock controls on the LE board are disabled. When the clock is running, its frequency is controlled from the LE board.

With these resources, the LE board has 128 parallel output signals which can be monitored and 72 parallel input signals which can be controlled. Section 8.6 describes how to access the switches, buttons and pipeline on the LE board.

C. Low Level Interface Routines

The low level interface includes routines to read the lights, write the switches and pipeline and stop, start or pulse the clock. #include "/le/include/lelib.h"

```
#include "/le/include/lelib.h"
```

In order to use these routines, the file **/<le>/include/lelib.h** has to be included into the source file and the file **/<le>/bin/lelib.lib** has to be linked to the object files. See section 8.7 for instructions on linking this library.

```
int initboard(void);
```

Configures the LE board so that the PC host can control the operation of the clock, switches, buttons and pipeline. It also determines on which port the board is attached and does a minimal test to determine if the board is functioning. If the board is present and functioning **initboard** returns 0 otherwise it returns a positive integer. If no board is found, it could mean that the board is not powered on, the interface cable is not connected properly or the LE board is not functioning properly. This function should be called before any of the LEPI functions are used.

```
void restoreboard(void);
```

Restores the LE board to a state in which the clock, switches and buttons are all enabled. This function should be called before exiting the user program.

```
void readlights(lights);
```

```
unsigned char *lights;
```

Reads the current value of all the lights. The values are stored in the user supplied array **lights**. Each entry in the array holds eight bits representing the value of the lights. The first entry (**lights[0]**) contains lights 0-7, the second, lights 8-15 and so on up to the sixteenth entry which contains lights 120-127. In each entry the lowest numbered light is the least significant bit.

```
int readlight(n);
```

```
int n;
```

Reads the **n**th 8-bit group of lights. **Readlight(n)** returns the value of the lights **n***8 through **n***8+7. The lowest numbered light is the least significant bit of the value returned. The value of **n** can range from 0 to 15. If **n** is outside this range, -1 is returned indicating an error.

```
void readswitches(switches);
```

```
unsigned char *switches;
```

Reads the current value of all the switches and buttons. The values are stored in the user supplied array **switches**. Each entry in the array holds eight bits representing the value of the switches. The first entry (**switches[0]**) contains switches 0-7, the second, switches 8-15 and so on up to the forth entry which contains switched 24-31. In each entry the lowest numbered switch is the least significant bit.

```
int readswitch(n);
int n;
```

Reads the **n**th 8-bit group of switches. **Readswitch(n)** returns the value of the switches **n***8 through **n***8+7. The lowest numbered switch is the least significant bit of the value returned. The value of **n** can range from 0 to 3. If **n** is outside this range, -1 is returned indicating an error.

```
void writeswitches(switches);
unsigned char *switches;
```

Writes the values in array **switches** to all the switches and buttons. Each entry in the array holds eight bits representing the value of the switches. The first entry (**switches[0]**) contains switches 0-7, the second, switches 8-15 and so on up to the forth entry which contains switches 24-31. In each entry the lowest numbered switch is the least significant bit.

```
int writeswitch(n,val);
int n;
unsigned char val;
```

Writes the value **val** to the **n**th 8-bit group of switches. **Writeswitch(n,val)** writes the value **val** to the switches **n***8 through **n***8+7. The least significant bit of **val** is written to the lowest numbered switch. The value of **n** can range from 0 to 3. If **n** is outside this range, -1 is returned indicating an error.

```
void writepipeline(pipe);
unsigned char *pipe;
```

Writes the values in array **pipe** to the pipeline. Each entry in the array holds eight bits. The first entry (**pipe[0]**) contains bits 0-7 of the pipeline, the second, bits 8-15 and so on up to the fifth entry which contains bits 32-39. In each entry the lowest numbered pipeline bit is the least significant bit of the value.

```
int writepipe(n,val);
int n;
unsigned char val;
```

Writes the value **val** to the **n**th 8-bit group of pipeline bits. **Writepipe(n,val)** writes the value **val** to the pipeline bits **n***8 through **n***8+7. The least significant bit of **val** is written to the lowest numbered pipeline bit. The value of **n** can range from 0 to 4. If **n** is outside this range, -1 is returned indicating an error.

```
void pulseclock(ticks);
int ticks;
```

Issues **ticks** number of clock pulses on the user clock. The clock is left in the stopped state after this function is called, so a value of 0 for **ticks** will simply stop the clock.

```
void startclock(void);
```

Puts the clock in the running state. If the clock was already in the running state, nothing happens.

Fig. 1. is a small example of a C program that uses some of the low level routines. The routine tests one NAND gate (74ls00). It assumes that the inputs (pins 1 and 2) have been wired to switches 0 and 1 and the output (pin 3) has been wired to light 0. This example uses the routines: **initboard**, **restoreboard**, **writeswitch**, and **readlight**.

```
#include <stdio.h>
#include "\le\include\lelib.h"

void
main()
  { while(initboard()!=0)
     { /* initialize the board*/
       fprintf(stderr,"LE Board not present or not functioning\n");
       fprintf(stderr,"<Hit Enter key to continue>");
       getchar();
      }
    dotest();/* perform the test*/
    restoreboard();/*restore the board*/
   }

void
dotest()
  {
    static int expected[4] = {1,1,1,0}; /*Truth Table for 74ls00*/
    int result;                         /*result of running test*/
    int error;                          /*number of errors so far*/
    unsigned char i;                    /*input value and loop index*/
    error = 0;
    for(i=0;i<4;i++)
     {
       writeswitch(0,i);/*Present inputs to gate*/
       result = readlight(0);/*Read output of gate*/
       if(result != expected[i])
         { fprintf(stderr,"Error: Inputs: %d Result: %d Expected: %d\n",
                   i,result,!(i==3));
           error++;
          }
       }
    fprintf(stderr,"The test completed with %d error(s)\n",error);
   }
```

Sample program using low-level LEPI routines

D. High Level Interface Routines

The high level interface includes routines to define field declarations and invocations, invoke values onto the defined fields and read values from the defined fields. Table II. is a list of all the high level routines available, including the signature and a description of each. The field declarations and invocations are defined in a separate file. The syntax for this file is nearly identical to the declaration section of the LE Micro Assembly Language described in Chapter 5 of this manual with a few additions This file must be read by the user program before any of the high level routines can be used.

```
int initboard(void);
```

Besides initiating the board as described above in the low level routines, this routine initiates the pipeline, switches and buttons to their default values as defined in the declaration file. It should therefore be called after calling **declare**.

```
int declare(filename);

char *filename;
```

Reads the named declaration file. This will construct an internal symbol table that is used by the **initboard**, **command**, **mask** and **readval** routines. The syntax of the declaration file is described in section XX. This routine should be called before the **initboard** routine. The value returned is the number of errors that occurred while parsing the declaration file.

```
int command(comlist[,arg]...);

char *comlist;
```

The **command** routine is used to assert values onto the defined field. **Comlist** is a string with the same syntax as the command list portion of a microinstruction statement of the LE Micro Assembly Language as described in chapter 5 of this manual. In addition, the **comlist** can contain format specifications as in the **printf** routine (See the Microsoft C Run-time Library Reference). The **command** routine asserts only the signals as defined in **comlist**. All other signals take on their default value. A negative value will be returned if a error occurred while parsing **comlist** otherwise 0 will be returned.int mask(comlist[,arg]...);

```
char *comlist;
```

The **mask** routine is identical to the **command** routine with the exception that signals which are not asserted as defined in **comlist**, remain at their current value.

```
int readval(field);

char *field;
```

The **readval** routine returns the value of the named field as defined in the declaration file. The **field** argument is a string containing the name of the field to be read. A negative result indicates that an error has occurred in parsing **field**.

Fig. 2. is a small example of a C program that uses some of the high level routines and Fig. 3. is

the accompanying declaration file. The routine tests an ALU (74ls181). It assumes that the inputs and outputs have been wired as defined in the declaration file. This example uses the routines: **declare**, **initboard**, **restoreboard**, **command** and **readval**. The testing algorithm used is to present the device with all possible inputs, checking the outputs for each against a software model of the device. The code for the software model (LS181 in fig. 2) is not shown in the figure. Notice that in the code of Fig. 2, there is no reference to light or switch numbers. All signals are referenced by name.

```
A   SW(0:3)
B   SW(4:7)
S   SW(8:11)
M   SW(12)
C0  SW(13), T=%L
F   LT(8:12), T=%LHHHH
```

E. Declaration File Syntax

The declaration file is used to give logically names to a group of signals and to describe the use of these signals. The syntax is similar to the declaration portion of the Micro Assembly Language described in Chapter 5 of this manual with a few addtions to support the switches, buttons, and lights. The description of the syntax given here is only a subset of the complete syntax. For a complete description of the sytax, see Chapter 5.

The three basic directives used in the declaration file are **COM**, **LT**, and **SW**. All of which have the same syntax

```
#include <stdio.h>
#include “lelib.h”

void main()
  {
    if(declare(“ls181.dec”)!=0)
      {
        fprintf(stderr,”Error: Declaration File\n”);
        exit(-1);
       }
    while(initboard()!=0)
      {
        fprintf(stderr,”LE Board not present or not functioning\n”);
        fprintf(stderr,”<Hit Enter key to continue> “);
        getchar();
       }
    dotest();
    restoreboard();
   }

void dotest()
  {
    int a,b,s,m,c0,f,expected,errors;

    errors = 0;
    for(s=0;s<16;s++){
        fprintf(stderr,”Testing function %x for all possible inputs\n”,s);
        for(b=0;b<16;b++){
            for(a=0;a<16;a++){
                for(m=0;m<2;m++){
                    for(c0=0;c0<2;c0++){
                        command(“A=%d,B=%d,S=%d,M=%d,C0=%d”,a,b,s,m,c0);
                        f = readval(“F”);
                        expected = LS181f(a,b,s,m,c0);

                        if(f != expected){
                            fprintf(stderr,
                                “Error: A=%x,B=%x,S=%x,M=%x,C0=%x->F=%x:Expected:%x\n”
                                ,a,b,s,m,c0,f,expected);
                                errors++;
                                printf(“test>”);if(getchar()==’q’) exit(-1);
                        }
                    }
                }
            }
        }
    }
    fprintf(stderr,”The test completed with %d error(s)\n”,errors);
}
```

Sample program using high-level LEPI routines

<*name*> <*dir*> <*range*> [**,**<*truth values*>] [,<default values>
where]

<*name*>	::=		A string of any length containing, upper or lower case letters, numerals, "_", or ".". The first character must be a letter. This defines the name of the field.
<*dir*>	::=	**COM**, **LT**, or **SW**	The directive defines the type of field.
<*range*>	::=	**(***n,m***)**	n and m are integers and n is less than m. The range defines the signals in the field.
	::=	**(***n***)**	
<*truth values*>	::=	**T=**<*numeric value*>	The truth value defines the interpretation of the voltage of each signal in the field. If absent, each signal is assumed to be true high.
	::=	**T=**<*voltage value*>	
<*default values*>	::=	**D=**<*numeric value*>	The default value defines the value each signal gets when it is not asserted. Used only for **COM** and **SW** directives. If absent, the default for each signal will be a low voltage for **VOLTAGE** mode and a false value for **LOGIC** mode. See the MODE directive below.
	::=	**D=**<*voltage value*>	
	::=	**D=**<*logic value*>	
<*numeric value*>	::=	[**0-9**]*	decimal value
	::=	**$**[**0-9a-fA-F**]*	hexadecimal value
	::=	**%**[**01**]*	binary value
<*voltage value*>	::=	**%**[**HL**]*	high and low voltage values
<*logic values*>	::=	**%**[**TF**]*	true and false logic values

The **COM** and **SW** directives both define input fields to the board. The name defined by these directives can be used in **command** or **mask** statements. The **LT** directives define output fields from the board. The name defined by these directives can be used in **readval** statements.

F. Nomenclature

The naming and numbering of the switches, lights, and pipeline for the software is slightly different from that of the hardware and can be a point of confusion. The following tables describe how they relate to each other. For exact location of these points on the Logic Engine board, see the Logic Engine Board User Manual.:

Table 5: Switches

	Type	*Switch*	*Switch*	*Switch*	*Switch*	*Hidden*	*Push-Button*	*Push-Button*	*Hidden*	*Push-Button*	*Push-Button*
numbering	*Software*	31-28	27-24	23-20	19-16	15-14	13-12	11-8	7-6	5-4	3-0
	Hardware	S15-S12	S11-S8	S7-S4	S3-S0	B15-B14	B13-B12	B11-B8	B7-B6	B5-B4	B3-B0
	IC/pin	XX/20-17	XX/16-13	XX/20-17	XX/16-13	XX/20-19	XX/18-17	XX/16-13	XX/20-19	XX/18-17	XX/16-13

Table 6: Lights 127-64

	Type	*Light*	*Light*	*Light*	*Light*	*Light*	*Light*	*Light*	*Light*
numbering	*Software*	127-120	119-112	111-104	103-96	95-88	87-80	79-72	71-64
	Hardware	L127-L120	L119-L112	L111-L104	L103-L96	L95-L88	L87-L80	L79-L72	L71-L64
	IC/pin	XX/2-9	XX/2-9	XX/2-9	XX/2-9	XX/2-9	XX/2-9	XX/2-9	XX/2-9

Table 7: Lights 0-63

	Type	*Light*	*Light*	*Light*	*Light*	*Light*	*Light*	*Light*	*Light*
numbering	*Software*	63-56	55-48	47-40	39-32	31-24	23-16	15-8	7-0
	Hardware	L63-L56	L55-L48	L47-L40	L39-L32	L31-L24	L23-L16	L15-L8	L7-L0
	IC/pin	XX/2-9	XX/2-9	XX/2-9	XX/2-9	XX/2-9	XX/2-9	XX/2-9	XX/2-9

Table 8: Pipeline

	Type	*Pipeline*	*Pipeline*	*Pipeline*	*Pipeline*	*Pipeline*
numbering	*Software*	39-32	31-24	23-16	15-8	7-0
	Hardware	P39-P32	P31-P24	P23-P16	P15-P8	P7-P0
	IC/pin	XX/19-12	XX/19-12	XX/19-12	XX/19-12	XX/19-12

G. Linking the Library

In order use the LE PI library, the source files must be compiled using the large memory model and then linked with the library as illustrated in the example below. For more details about compiling and linking see the Microsoft C Compiler User's Guide and Reference Manual.

Compile:

```
cl /c /AL file1.c
cl /c /AL file2.c
```

Link:

```
link file1.obj file2.obj, prog.exe, prog.map, lelib.lib ;
```