WORKSHOP USER'S GUIDE

for the Lisa™

Beta Draft
April 1983

CONTENTS

APPENDICES

A. ERROR MESSAGES

Chapter 1
INTRODUCTION

INTRODUCTION

1.1 The Workshop Manager.

The Workshop allows you to develop and run programs on the Lisa. It provides tools necessary to write, debug, and run programs in Pascal, BASIC, and COBOL. This manual explains how to use the Workshop and all of its tools.

Access to all Workshop functions is provided by command lines. The main command line, WORKSHOP allows you to edit programs, run utilities or user programs, and use the various languages available on the system. It also provides access to two subsystems; the File Manager, and the System Manager.

The File Manager allows you to copy, delete, rename, and list disk files. It includes a backup function, and functions for manipulating volumes. These functions are listed in the FILE- MGR command line, which is similar to the main command line. (See Chapter 2.)

The System Manager provides for system configuration and defaults and process managment. Its commands are listed in the SYS-MGR command line. (See Chapter 3.)

All command lines are displayed at the top of the Lisa screen. If there are more commands than will fit on one line, a "?" is at the end of the line. Pressing "?" will display the remaining commands. To access any command, press the first character of the command name. To redisplay the first command line, press RETURN.

Most commands will ask for additional information. Type in the information using the Lisa keyboard. Some questions have a default value, displayed in square brackets ([default]). To accept the default value, press RETURN. If you don't want the default value, type in the value you want.

The Lisa system can display one of two screens, called the main screen and the alternate screen. The Workshop system normally displays on the main screen. The alternate screen is used by the system debugger. You can change to the other screen display by pressing the right hand OPTION and ENTER keys. The System Manager contains the Console command, which can be used to specify where the Workshop should display.

The Workshop can be used to write programs in Pascal, COBOL, and BASIC. To use these languages, refer to the appropriate language manuals. In addition to this manual, you will need:

For Pascal Programming:

- Pascal Reference Manual for the Lisa
- MC68000 16 Bit Microprocessor User's Manual (for assembly language programming)
- Operating System Reference Manual for the Lisa (for information on system calls)

For BASIC Programming:

- BASIC User's Guide for the Lisa

For COBOL Programming:

- COBOL User's Guide for the Lisa
- COBOL Reference Manual for the Lisa

If you have only a BASIC or COBOL system, you will not have all the software described in this manual. The portions of this manual that will be most useful to BASIC and COBOL programmers are:

- The Introduction, which describes how to use the Workshop.
- The File Manager, which describes files and how to manipulate them.
- The System Manager, which describes setting up the system configuration parameters.
- The Editor, which describes how to create and modify text files that are used as source files.

You may also use some of the utilities if they are included in your software.

1.2 Starting the Workshop

The Workshop can be booted from a diskette or a Profile. It will most commonly be used with a Profile.

To start the system, boot from a disk that contains the Workshop software. If your disk contains only the Workshop environment, the Workshop command line will appear at the top of the screen. If you have more than one environment (for example, the Workshop and the desktop) you can use the Environments window to start up the environment you want, and switch between them.

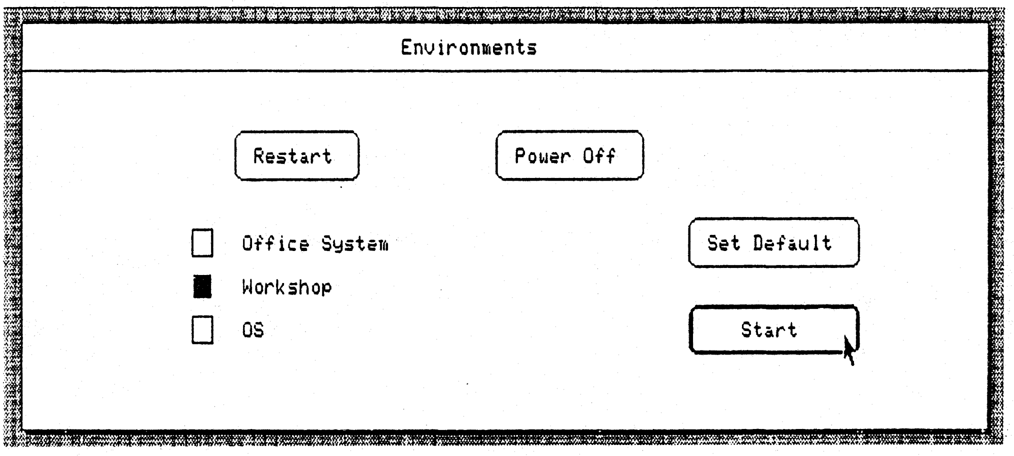
The Environments Window allows you to select the environment you want to start. You can also set a default environment that will be started automatically when you boot the system. To access the environments window while booting the system, press any key while the Lisa is starting up. The environments window will be displayed.

The Environments window is shown in Figure 1-1. It displays five buttons:

Power Off	Turn off the Lisa
Restart	Reboot or reset the Lisa
Start	Start the selected environment
Set Default	Set the default to the selected environment
No Default	The Environments window will always be displayed on startup.

To select an environment, move the pointer to the checkbox of that environment and click the mouse button. Then move the pointer to the start button and click. The selected environment will start. -

To access the Environments window from the Workshop, and select another environment, use the Quit command from the Workshop command line, or press the on-off button. To access the Environments window from the Desktop, press the on-off button while holding down the (apple) key.



□ Office System
■ Workshop
□ OS

Figure 1-1. The Environments Window

1.3 The Workshop User Interface.

When the workshop environment is selected, the system will come up with the Workshop command line at the top of the screen. This command line lists all the actions you can currently request of the system. The Workshop line displayed contains only some of the commands available. The rest of the commands can be displayed by pressing "?", the last symbol on the line. The original command line can be redisplayed by pressing RETURN. A command is executed by pressing the first letter of the command name.

There are two other subsystems that have separate command lines; the

File-Manager, and the System-Manager. Their command lines can be accessed from the Workshop command line, and are used the same way.

You can terminate the operation of most commands by pressing (apple) period. You can turn off the Lisa by pressing the on-off button at any time. The system will shut down in an orderly manner. A diskette can be inserted at any time. It will automatically be mounted and accessible. Diskettes are ejected by pressing the diskette button.

The main, or Workshop, command line is as follows:

```
WORKSHOP: FILE-MGR, SYSTEM-MGR, Edit, Run, Pascal, Basic, Cobol, Quit, ?
```

The additional portion, displayed by pressing "?", is:

```
Assemble, Debug, Link, MakeBackground, Generate
```

All the main command line commands are described below.

FILE-MGR (F)
This command puts you into the File Manager subsystem, which is used to manipulate the files and volumes on the system. For more information on the file manager, see Chapter 2 in this manual.

SYSTEM-MGR (S)
This command puts you into the System Manager subsystem. This subsystem provides various configuration and utility functions. See Chapter 3 in this manual for more information.

Edit (E)
The Edit command puts you into the text editor, which is used to create and modify text files. The Editor is used to create source files for BASIC, COBOL, and Pascal. It is also used for assembly language programming and to create exec files. The Editor is described in Chapter 4 in this manual.

Run (R)
The Run command causes a compiled and linked program to execute. This command is used for user-written Pascal programs, utility programs, and any other software that runs under the Workshop. The Run command asks you for the file to run. This file must be an executable object file or an exec file. (An exec file name must be preceded by a "<" .) If you do not give it a complete pathname, the Run command will search through up to three default volumes for the file. These defaults can be set by the File-Manager's Prefix command. See the Prefix command in Chapter 2 for more information.

The Run command will also accept an "exec file" as input. An exec file is a scenario of commands for the Workshop system to carry out. An exec file name must be preceded by a "<" to be processed correctly. For more information on exec files, see Chapter 9 in this manual.

Pascal (P)
This command starts the Pascal compiler. The compiler asks for the input file, which must be a text file; the listing file; and the output file, which will contain the object file. The Pascal compiler is described in Chapter 5. Further information on the Pascal language can be found in the Pascal

Reference Manual for the Lisa.

The compilation is done in two steps. The first step, done by the Pascal command, produces an intermediate code file. After this, you must use the Generate command, (press G) to generate an object file from the intermediate code file.

Basic (B)
This command puts you into the BASIC interpreter. More information on BASIC programming can be found in the BASIC User's Guide for the Lisa.

Cobol (C)
This command puts you into the COBOL language system. More information on COBOL programming can be found in the COBOL User's Guide for the Lisa and the COBOL Reference Manual for the Lisa .

Quit (Q)
The Quit command ends the Workshop environment. You can access the Environments window to start another environment.

Assemble (A)
The Assemble command starts the assembler. Further information on the assembler can be found in this manual in Chapter 6. Additional information on the assembly language can be found in the MC 68000 Microprocessor User's Manual .

Debug (D)
The Debug command causes your program to run with a breakpoint inserted at the first instruction in the program, so you can use the debugger on the program. More information on the Debugger can be found in Chapter 8 of this manual.

Link (L)
The Link command executes the Linker. The Linker is used to prepare compiled Pascal programs and assembled routines for execution, and to link together separately compiled pieces of a program. The Linker is described in Chapter 7.

MakeBackground (M)
The MakeBackground command allows you to start up a background process, then continue using the Workshop for other functions. It is assumed that the backgrond process will not try to display on the console.

Generate (G)
The Generate command converts intermediate code files produced by the Pascal compiler into object code. It is used with the Pascal compiler and is described in Chapter 5.

1.4 File system organization and naming

Files are stored on volumes, that are mounted on devices. A volume has a name and a directory of files that it contains. A file is specified by giving the name of the volume and the name of the file:

```
-volumename-filename
```

The Workshop maintains a working directory; you can access files in it without specifying a volume name. The working directory can be changed by using the File Manager's Prefix command. Files on the working directory can be specified by just the file name, with no leading "-":

```
filename
```

Further information on the file system can be found in Chapter 2 of this manual and in the Operating System Reference Manual for the Lisa.

1.5 Utility Programs.

There are various utility programs provided with the Workshop. These are used for functions not as commonly used as the commands.

The utilities are described in Chapter 10.

You must Run utilities. Select the Run command from the main command line by pressing R when the main command line is displayed. The system will ask you for the name of the file to run. Type in the name of the utility you want to run.

1.6 How do I Write and Run a Pascal Program?

To write and run a Pascal program, proceed as follows:

1. Use the Editor to create a text file with the Pascal source program. See Chapter 4 in this manual for more information on editing the file. See the Pascal Reference Manual for the LIsa for information on the language.

2. Compile the program using the Pascal command (press P while the Workshop command line is displayed) from the main command line. The output from the compiler is an intermediate file.

3. The output from the Pascal command is an I-code file. Use the Generate command to convert the I-code file into an object file. To use the Generator, press G when the Workshop command line is displayed. See Chapter 5 for more information on compiling Pascal programs.

4. Link the program using the Link command. In order to be executable, the program must be linked with the Pascal support routines contained in IOSPASLIB. For other applications you may also use other libraries and units, or assembly language routines. More information on the Linker can be found in Chapter 7.

5. The linker produces an executable object file. Press R to run the program.

Information on making system calls from Pascal can be found in the Operating System Reference Manual for the Lisa.

1.7 How do I Write and Run an Assembly Language Program?

Assembly language programs must be called as procedures of functions from a Pascal main program. To write an assembly language routine, proceed as follows:

1. Use the Editor to create an assembly language source program. See

Chapter 6 of this manual for information on assembly language. Chapter 4 describes the Editor.

2. Press A to execute the Assembler. The Assembler accepts the text file you created and produces an object file.
3. Declare the routines you wrote in assembly language as EXTERNAL in the main Pascal program that calls them.
4. Use the Pascal and Generate commands to create an object file from the Pascal program. See Section 1.6 for more information.
5. Use the Link command to link the Pascal object file, the assembly object file, IOSPASLIB, and any other needed units or libraries.
6. Use the Run command to run the resulting object file.

1.8 How do I use the BASIC Interpreter?

To use the BASIC interpreter, proceed as follows:

1. Use the Basic command by pressing B when the main command line is displayed. You will enter the BASIC interpreter.
2. Enter the BASIC language statements and commands necesary to write and execute your program. The BASIC interpreter can execute statements immediatly or save them to run later. You can return to the main command line by using the BASIC command BYE.

You may also use the Editor to prepare or modify the BASIC source program, then use the BASIC interpreter to run it. See Chapter 4 in this manual for more information on the Editor.

See the BASIC User's Guide for the Lisa for more information on the language.

1.9 How do I Write a COBOL Program?

To write a COBOL program, proceed as follows:

1. Create a text file containing the source program by using the Editor. See Chapter 4 in this manual for more information on the editor.
2. Press C to enter the COBOL language system. More information on COBOL programming can be found in the COBOL User's Guide for the Lisa and the COBOL Reference Manual for the Lisa .

1.10 The Operating System.

The Workshop runs under the Operating System of the Lisa computer. You can use some operating system routines from a Pascal program to perform special system functions for you. These system calls are defined in the intrinsic unit SYSCALL. More information on the syscall interface and routines can be found in the Lisa Operating System documentation.

Chapter 2

THE FILE MANAGER

THE FILEMANAGER

2.1 The File Manager

The File Manager is a subsystem of the Workshop that provides file and device manipulation facilities. It handles most of the tasks of transferring information from one place to another. Using the file manager, you can do such things as make copies of files, list directories, rename or delete files, find out what volumes are on line, initialize new disks or diskettes, print files, and so on. See the Operating System Reference Manual for the Lisa for more information on the file system and supported devices.

A file specifier can be an OS pathname (representing a file on a disk or diskette), an OS volume name (for example, -MYDISK), the name of a physical device (for example -RS232A), or the name of a logical device (for exampel -PRINTER). File specifiers may contain wildcards (see section 2.5) allowing them to specify a collection of files.

2.2 Using the File Manager

To use the File Manager, press F in response to the Workshop command prompt. The File Manager begins executing, and displays the File Manager prompt line.

The File Manager prompt line is:

FILE-MGR: Backup, Copy, Delete, List, Prefix, Rename, Transfer, Quit, ?

To display the additional commands, press "?". The line of additional commands is:

Equal, FileAttributes, Initialize, Mount, Names, Online, Scavenge, Unmount

To redisplay the original command line, press RETURN.

To execute any command, press the first character of that command when the File Manager command line is displayed. Most commands will ask for file names, or other input parameters. If there is a default value for a parameter, it is displayed in square brackets ([default]). To accept the default, just press RETURN. If you do not want the default, type in the response you want.

2.3 The File Manager Commands

The File Manager commands are listed in the File Manager prompt line. They are: Backup, Copy, Delete, List, Prefix, Rename, Transfer, Quit, Equal, FileAttributes, Initialize, Mount, Names, Online, Scavenge, and Unmount.

Some of these operations can be performed either on a single file, or on a list of files specified by wild card characters.

Each of these operations is described below. Information on wild card characters can be found in section 2.5 below.

2.3.1 Backup (B)

This command executes a simple backup utility, similar to Copy. It asks for source and destination file specifiers, which will most likely contain wild cards, (see Section 2.5) and compares the source files to the destination files. Whenever the contents of the two files are not equal, the file is copied. If a source file is missing from the destination, it is copied.

2.3.2 Copy (C)

The Copy command copies files. It asks for a source file specifier and a destination file specifier. You may use wild cards if you want to copy more than one file. The source file(s) are not changed by this command.

The default is not to verify copy operations. You can change this default with the Validate command in the System Manager. If you change the default, the source file will be compared to the destination file after the copy operation to insure that they are the same. The Validate command is described in Chapter 3.

You can copy files to the -PRINTER or the -CONSOLE logical devices. Text files (ending in ".text") will be displayed as a text file. All other files will be sent byte by byte .

2.3.3 Delete (D)

The Delete command is used to delete a file or a number of files specified by a wild card expression. It asks you to specify the files to be deleted.

2.3.4 List (L)

The List command lists information about the files matching the given file specification. If all you need is the names of the files, use the Names command described below.

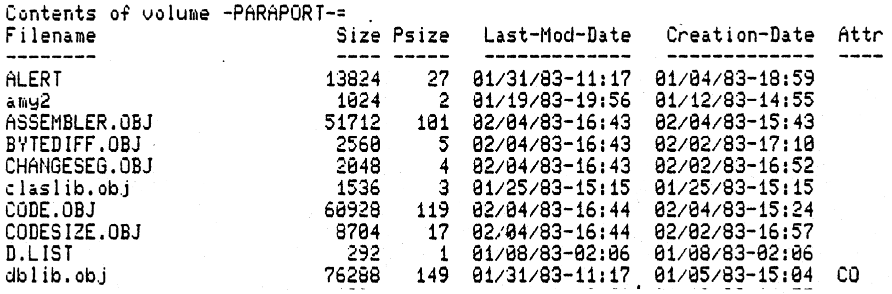
- If the file specifier is a file name (for example -MYDISK-example.text) that file is listed.
- If the files specifier is a volume name (for example -MYDISK), information about all files on the volume is listed.
- If the file specifier includes a wildcard character (for example, -MYDISK-=.text) information about all matching files is listed.

The list command displays the following information:

Filename	The name of the file.
Size	The logical file length in bytes.
Psize	The physical length of the file in blocks.
Last-Mod-Date	Date and time the file was last changed.
Creation-Date	Date and time the file was created.
Attr	File attributes, a combination of the following:

C	File was closed by the OS
L	File is locked (cannot be deleted)
O	File was left open when the system crashed
P	File is Protected
S	File has been Scavenged.

An example of the list display is shown in figure 2-1.



```
Contents of volume -PARAPORT-=
Filename                         Size Psize  Last-Mod-Date   Creation-Date  Attr
--------                         ---- -----  -------------   -------------  ----
ALERT                           13824    27  01/31/83-11:17  01/04/83-18:59
amy2                             1024     2  01/19/83-19:56  01/12/83-14:55
ASSEMBLER.OBJ                   51712   101  02/04/83-16:43  02/04/83-15:43
BYTEDIFF.OBJ                     2560     5  02/04/83-16:43  02/02/83-17:10
CHANGESEG.OBJ                    2048     4  02/04/83-16:43  02/02/83-16:52
claslib.obj                      1536     3  01/25/83-15:15  01/25/83-15:15
CODE.OBJ                        60928   119  02/04/83-16:44  02/04/83-15:24
CODESIZE.OBJ                     8704    17  02/04/83-16:44  02/02/83-16:57
D.LIST                            292     1  01/08/83-02:06  01/08/83-02:06
dblib.obj                       76288   149  01/31/83-11:17  01/05/83-15:04  CO
```

Figure 2-1. The List Display

2.3.5 Prefix (P)

This command allows you to set up default volume names to search when you specify a file name without a volume name. You can set a sequence of up to three volume names that will be searched in order when you try to run a program until the file is found. The first prefix is the name of the working directory. It will be searched anytime you specify a filename without a volume name. Boot defaults for prefixes can be set using this command. The second and third prefixes will be searched when you try to Run a program without specifying the volume it is on.

This command asks you for the three prefixes. If you want to accept the default, (if any), press RETURN. If you want to set a prefix, type in the volume name. If you want to have no prefix, press CLEAR as the prefix for that level.

2.3.6 Rename (R)

The Rename command allows you to change the name of a file. It asks for the filename to change and the name to change it to. You can also use the Rename command to change the name of a volume. The Rename command can change the name of a number of files by using wild cards. See Sections 2.5 and 2.9 for more information.

2.3.7 Transfer (T)

The Transfer command asks for an input file specification and a destination file specification. It copies the input file(s) to the destination and then, if the copy was successful, deletes the input file(s). If you Transfer to the -console or the -printer, the input file will not be deleted.

2.3.8 Quit (Q)

This command exits from the File Manager subsystem to the Workshop command line.

2.3.9 Equal (E)

The Equal command compares the contents of two files to determine whether they are exactly the same. It asks for the names of the files to compare, then compares them byte by byte and tells you if they are equal or unequal.

2.3.10 FileAttributes (F)

This command is used to set file attributes. You can set the safety attribute, which makes the file so you cannot accidentally delete it. In order to delete a file with the sasfety attribute set, use the FileAttributes command to unset the attribute on the file. You can also make a file into a protected master.

Use the FileAttributes command by pressing F in response to the File Manager command prompt. It displays a command line:

```
FileAttributes: ClearAttributes, Safety, Protect, Quit.
```

These commands are accessed by pressing the first character of the command. They perform the following functions:

ClearAttributes (C)
The clear attributes command clears the C, O, and S attributes on the specified volume. These attrubutes are set by the system, and have the following meanings:

C	File was closed by the Operating System
O	File was left open when the system crashed.
S	File has been scavenged.

The clear attributes command should be used before scavenging a volume so that you can tell if any files were changed. See the Scavenge command in Section 2.3.15 below for more information.

Safety (S)
The Safety command allows you to set or remove the safety attribute on any file. When the safety attribute is set, the file cannot be deleted. To delete a file with safety on, use this command to remove the attribute, then delete the file.

Protect (P)
The Protect command is used to make a file into a protected master. This is a form of copy protection for object files. Once a file is made into a protected master, this protection cannot be removed. A protected master has the following characteristics:

- It can be run on any Lisa machine
- It can be copied on any one Lisa machine.
  - Copies made will run only on the machine that made the copies.
  - After the file is copied the first time, further copies of the master can be made only on the same machine.

NOTE

Once a file is made into a protected master, there is no way to unprotect it. Be sure you understand the characteristics of a protected master before you create one.

This protection scheme is for executable object files. Note that protecting a file does not prevent you from deleting it.

Quit (Q)
The quit command exits you from the file attributes subsystem to the File Manager.

2.3.11 Initialize (I)

The Initialize command is used to set up an OS device. It is used to format and initialize the file system on a diskette or ProFile. It asks you for the device name to initialize, the number of blocks to initialize, the volume name, and password. If you want the entire device to be initialized, enter RETURN (accepting the default) for the number of blocks. If the device is a diskette, it is formatted (ProFiles are factory formatted). Boot tracks are automatically written to any device that is initialized. An initialized device is automatically mounted.

The initialize command will warn you if you attempt to initialize a disk that already contains a volume. A volume is initialized to allow a certain maximum number of files. You can make this number larger or smaller (if you know you will have a large number of small files, for example) when initializing it.

2.3.12 Mount (M)

This command is used to make an OS device accessible. It requests a device name. It should be used whenever you connect a new device, such as a Profile. The Unmount command, described below, is used to remove a device. All configured devices are mounted at boot time. The configuration can be changed with the Preferences tool, which is described in Section 3.3

2.3.13 Names (N)

The names commandis a faster version of the List command. It gives you a list of file names only. It asks for a file specifier, and displays the names of all files matching the given file specifier.

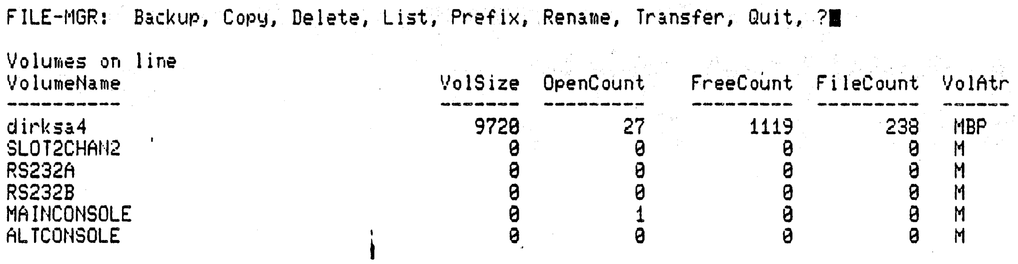
2.3.14 Online (O)

The Online command produces a list of all the devices that are currently mounted and available. It tells you the devices mounted, the names of the volumes contained on them, the number of files on each volume, the size of the volume, and the amount of free space on it. The online display gives the following information:

VolumeName	The name of the volume.
VolSize	The number of blocks on the volume.
OpenCount	The number of files open.
FreeCount	The number of blocks still available.

FileCount The number of files stored on the volume.
VolAt The attributes of the volume:
B the boot volume.
P the prefix volume.
M volume is currently mounted.

The Online display is shown in Figure 2-2.



```
FILE-MGR:  Backup, Copy, Delete, List, Prefix, Rename, Transfer, Quit, ?█

Volumes on line
VolumeName                           VolSize  OpenCount     FreeCount  FileCount  VolAtr
-----------                          -------  ---------     ---------  ---------  ------
dirksa4                                 9720         27          1119        238   MBP
SLOT2CHAN2                                 0          0             0          0   M
RS232A                                     0          0             0          0   M
RS232B                                     0          0             0          0   M
MAINCONSOLE                                0          1             0          0   M
ALTCONSOLE                                 0          0             0          0   M
```

Figure 2-2. The Online Display

2.3.15 Scavenge (S)

This command runs the OS Scavenger which restores damaged files. Files can be damaged any time the system terminates abnormally. The Scavenger searches through a disk and restores its directories, files, and allocation tables to a consistent state.

A disk must be unmounted before it can be scavenged. Use the unmount command to unmount the disk, scavenge it, then mount it again to continue using it. The boot volume cannot be unmounted; therefore it cannot be scavenged. If the ProFile is normally your boot volume and you need to scavenge it, it is necessary to boot from a diskette and run the Scavenger from it.

If a file is changed in any way by the Scavenger, the file attributes will be set to S, for scavenged. This attribute is displayed by the List command. The changes made to the file may or may not affect the data in the file, depending on what state the file was in when it was scavenged. Check any file with the Scavenged attribute before relying on its contents. After the file has been checked, the Scavenged attribute can be removed with the FileAttributes command.

NOTE

The file system can get into an inconsistent state because the directories and allocation tables are kept in memory and only written out to disk periodically. If there is an abnormal termination, such as a power failure, the changes to the state of the file system since these tables were written to disk will be lost. Information can also be lost if you disconnect a ProFile from the Lisa without first unmounting it. If the disk is used after such an event, more data can be lost if the system allocates the same blocks to more than one file.

The Scavenger will always return the disk to a consistent state, but it is possible to lose data when the system crashed. This damage can become even worse if the disk is used while in an inconsistent state.

All Scavenged files should be checked before you depend on their contents.

2.3.16 Unmount (U)

This command makes a device inaccessible. It asks for a device name. Always unmount a device before disconnecting it.

2.4 Disk Storage Organization and Naming

Each disk contains a volume. The volume name is the name of the disk. Volumes are created with the Initialize command, which sets up the disk and puts an empty directory on it. As files are entered on the disk, their names are entered in the directory. A complete path name consists of a volume name followed by the file name in the following format:

```
-volname-filename
```

A working directory is maintained by the Workshop allowing you to access files on it without using the volume name. This working directory defaults to the boot device. The working directory can be changed by the Prefix command. The working directory is the first prefix specified in the Prefix command. Files on the working directory are specified by just the file name, with no leading "-":

```
filename
```

A volume must be mounted before it can be accessed. Volumes are mounted with the Mount command in the File Manager. To mount a volume, you specify the device on which it resides. Device names that can be used for disks are as follows:

-UPPER	The upper diskette. Drive 1.
-LOWER	The lower diskette. Drive 2.
-PARAPORT	ProFile attached to the parallel port.
-SLOT2CHAN2	ProFile attached to the N-port card in slot 2, channel 2, etc.

There are also two serial devices, -RS232A and -RS232B . These provide access to external RS232 devices.

There are three logical devices that can be used for input and output. These devices are:

-CONSOLE Used for output to the screen and input from the keyboard. The actual device which is used as the console can be changed by the Console command in the System Manager. See Section 3.2.

-PRINTER Used to output to the printer. The physical port that the printer is connected to is set by the Preferences tool, described in Section 3.3.3.

-KEYBOARD Used as a non-echoing input device from the keyboard. This is the keyboard on the console device.

Certain types of files in the system have standard file extensions. These extensions make it easier to keep track of the different types of files. These file extesions are:

.TEXT This indicates a text file in the format created by the Editor.

.OBJ This indicates an object code file. Object files are created by the code generater, the Assembler, and the Linker. Object files created by the Linker are executable.

.I This indicates an intermediate (I-CODE)file produced by the Pascal compiler. The Generate command will convert an intermediate file into an object code file.

.LIB This indicates a library file.

.SHELL This indicates a shell file that can be started by the environments window.

2.5 Using Wild Card Characters

Wild card characters allow you to specify a set of files to operate on. The command is performed on all files whose pathname matches the set specified. Wild card characters are "=", "?", and "$". These characters are used as follows:

string1=string2

The "=" character stands for any sequence of characters that can be ignored. The surrounding strings (string1 and string2) must be matched exactly, ignoring case. Either or both strings can be null. Here are some examples of using the "=" wild card character as a source file name:

ds=.text	all files beginning with ds and ending in .text.
=.obj	all files ending with .obj.
=	all files.

When "=" is used in a destination file name, it is replaced with the characters that were matched by a wild card in the source file. This allows you to do operations like change the name of a list of files as they are copied. Here are examples of using "=" as a destination file name:

ds=.text	to	bu/ds=.text	Change all files starting with ds and ending with .text so they are prefixed with bu/
=.obj	to	x/=.obj	Put x/ in front of the file name.

string1?string2

The "?" character is the same as the "=", except that the system asks you to confirm each file name before performing the operation. The "?" wild card can be used only as a source string.

When you use a "?" in a source specifier, you are presented with a list of files that match it. You can move backwards and forwards through the list by using the up and down arrows on the numeric keypad. Press "Y" beside every file that you want to be processed. When you have selected all the files you want, press RETURN. The operation will then be performed on the files you selected.

string1$string2

The "$" character is used only as a destination file name. It is replaced by the entire source ffle name. For example, if you have the source files matching ds=.text:

```
dsfmgr.text
dssmgr.text
```

If the destination expression is bk$, the output files will be:

```
bkdsfmgr.text
bkdssmgr.text
```

Contrast this with the output expression bk=, which results in:

```
bkfmgr.text
bksmgr.text
```

2.6 How do I Copy a File?

You can either Copy a file and leave the original file intact, or you can Transfer the file, which will copy the file, then delete the original file. To copy a file, proceed as follows:

1. If you are not in the File Manager subsystem, enter it by typing F in response to the Workshop command prompt.
2. Press C to start the Copy command. (Press T, for transfer, if you want the original file to be deleted after the copy operation.)
3. Enter the pathname of the file you want copied. Press RETURN.
4. Enter the pathname you want the file to be copied to. Press RETURN.

The file will be copied or transferred as you specified.

If you want to copy a number of files with similar names, or all the files on a volume, you can use wild card characters. See section 2.5 for more information on using wild cards. Wild cards can also be used to rename all the copies of the selected files.

You can use a shorthand method of entering the file names by entering both the source and destination file names, separated by a comma (,) in response to the request for the source file.

See Figure 2-3 for examples of copy and transfer operations.

```
Copy from what existing file(s)? myprog
Copy to what new file? -backup-$
```

(This copies the file myprog on the working directory to the volume -backup with the same name, myprog.)

```
Copy from what existing file(s)? ds=
Copy to what new file? -backup-$
```

(This copies all files beginning with ds on the working directory to the volume backup with the same file name.)

```
Transfer from what existing file(s)? -osback-osg=
Transfer to what new file? -oswork-$
```

(This copies all files beginning with osg on the volume -osback to the volume -oswork using the same file name. When the files have been copied successfully, the original files are deleted.)

```
Transfer from what existing file(s)? -osback-osg=,-oswork-$
```

(This is the shorthand version of the above transfer operation.)

```
Copy from what existing file(s)? ds=,-backup-backds=
```

(This copies all files beginning with ds in the working directory to the volume -backup with back inserted as the beginning of each file name.)

Figure 2-3. Copy and Transfer operations

2.7 How do I Delete a File?

To delete a file, proceed as follows:

1. If you are not in the File Manager subsystem, enter it by typing F in response to the Workshop command prompt.
2. Select the Delete command by pressing D.
3. Enter the pathname of the file you want to delete.
4. The system asks you to confirm that you want to delete the file. Reply Y to delete the file or N to keep it.

If you want to delete more than one file, you can use wild cards. See the section "Using Wild Card Characters" in this chapter for more information.

2.8 How do I Create and Use a Volume?

A volume can be created on either a diskette or a ProFile disk. Each disk can contain one volume. Creating a volume on a disk gives it a name and sets up a directory for files.

1. If you are not in the File Manager subsystem, enter it by typing F in response to the Workshop command prompt.
2. Press I to invoke the Initialize command. This command asks for:

- The device name (upper or lower for a diskette, slot2chan2 for a ProFile, etc.)
- The number of pages to initialize. The default is to initialize the whole device.
- The volume name.
- The volume password (optional).
- The maximum number of files on the device. The default is a good value unless you are using a large number of very small files or a few very large files.

The volume is initialized, with an empty directory. (If the device is a diskette it is first formatted.) The system will warn you if you are initializing a device that has an existing volume on it, and give you a chance to change your mind before destroying the existing volume.

After initialization, the device is automatically mounted so it can be used.

2.9 How do I Change the Name of a File or Volume?

The Rename command allows you to change the name of any file.

1. If you are not in the File Manager subsystem, enter it by typing F in response to the Workshop command prompt.
2. Execute the Rename command by pressing R.
3. Enter the pathname of the file or volume you want to rename.
4. Enter the new name.

The name of the file or volume is changed.

You can use the Rename command to change the name of a group of files by using wild card expressions.

2.10 How do I List Existing Files?

You can use either the List command, or the Names command to list existing files. The Names command executes much faster than the List command, but it gives you only the file names.

1. If you are not in the File Manager subsystem, enter it by typing F in response to the Workshop command prompt.
2. Execute the List command by pressing L, or the Names command by

pressing N.

3. If you want to list an entire volume, enter the pathname of the volume or device. If you want to list only a certain set of files, enter a wild card expression or pathname describing the files to be listed.

The listing produced by the list command is explained in Section 2.3.4.

For more information on wild card characters, see Section 2.5 in this chapter.

Chapter 3
THE SYSTEM MANAGER

THE SYSTEM MANAGER

3.1 The System Manager.

The System Manager allows you to set system defaults and configuration. It allows you to:

- Set the Lisa system characteristics such as screen contrast, speaker volume, and time lags for repeating keys.
- Set the configuration of external devices such as disks and printers.
- Set the default start up device.
- Set processes to be resident or non resident, to allow you to performance tune your Workshop system.
- Set what device is to be the console.
- Redirect output from the console to a file or external device.
- Monitor all currently existing processes, and remove processes.

3.2 The System Manager Functions.

By pressing S in the main comand line, you can enter the System Manager subsystem. The System Manager command line works the same as the main Workshop command line. Pressing "?" shows you the additional line of commands.

The System Manager command line is:

SYSTEM-MGR: ManageProcess, OutputRedirect, Preferences, Time, Quit, ?

Press "?" to see the additional commands:

Console, FilesPrivate, Validate

Each System Manager command is described below.

ManageProcess (M)
This command puts you into a process management subsystem, which allows you to select which processes should be resident for performance reasons. It also allows you to display the status of all currently existing processes, and remove processes. This subsystem is described in section 3.4 below.

OutputRedirect (O)
The OutputRedirect command allows you to send a copy of all output that is displayed on the console to another device (such as the -printer) or to a file on a disk. The command asks you for the pathname to send the copy to. In order to return to displaying only on the console, use the command again and redirect the output to the -console device (the default).

Preferences (P)
The Preferences tool is used to set up the configuration of the Lisa system and the Workshop. It is described in section 3.3 below.

Time (T)

The Time command allows you to set the date and time. The date and time will be maintained automatically by the Lisa system.

Quit (Q)
The Quit command exits from the System Manager back to the main Workshop command line.

Console (C)
This command allows you to change where the Workshop console is displayed. It may be displayed on the main screen (the default) or on the alternate screen (where LisaBug displays), or on an external terminal connected to the RS232A or B port.

FilesPrivate (F)
The FilesPrivate command selects whether or not the private system files should be displayed by the List command. The default is to not display the private files. Private files are any files with a name beginning with "{". These file names are used by the system for files you should not normally need access to.

Validate (V)
The validate command is used to set up defaults for verifying operations. Currently the only default of this type tells if the system will verify file copies or not. The system verifies a copy by comparing the original file with the copy to be sure they are the same. The boot default is to never verify. You should have no reason to verify unless you something is wrong with your disk.

3.3 The Preferences Tool

The Preferences tool is started by pressing P in response to the System Manager command line. After you are finished with it, you can exit back to the System Manager by selecting Quit from the Tools menu.

The Preferences tool allows you to set up your Workshop system the way you want it. It contains four sections:

- Convenience settings that allow you to set up the screen contrast, the speaker volume, and repeat delays.
- Device connections that tell the Lisa system what external devices are connected.
- Startup that tells the Lisa what device to use as a startup device.
- Workshop defaults that set up things the Workshop needs to know.

These default settings are stored in parameter memory, a small area of memory that is preserved as long as the Lisa is plugged into a working outlet and for up to 10 hours when the Lisa is unplugged. If your Lisa is without power for longer than this, the preference settings will be restored from information on the startup disk.

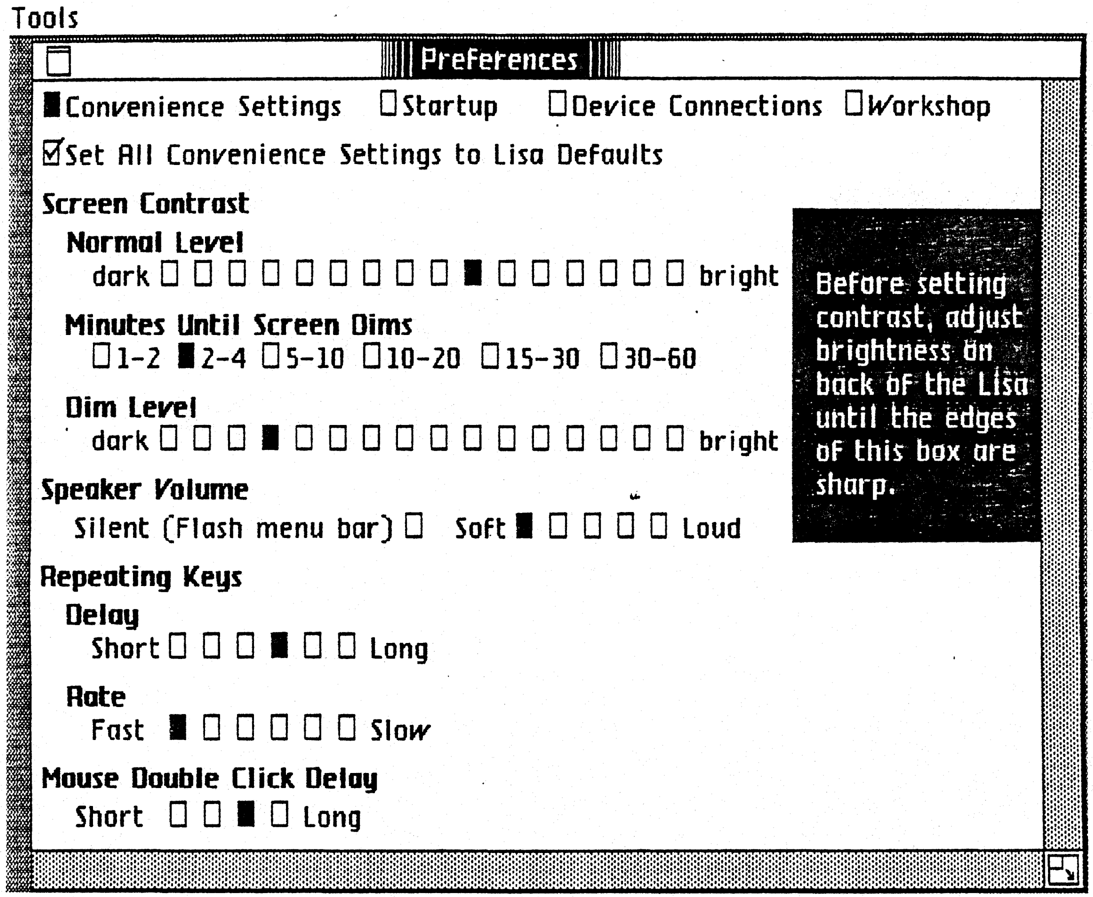
Any changes made with the Preferences tool change Parameter Memory immediately, but some of them, such as device connections and startup options have no effect until the system is booted again.

The preferences tool displays a window containing a number of buttons and checkboxes. You set the values you want by using the mouse to move the pointer to the desired options and clicking.

These four areas are described briefly below. More information on the first three areas can be found in the Lisa Owners Guide Section D. Select the area you want to view or change by moving the pointer with the mouse to the checkbox in front of the section name and clicking.

3.3.1 Convenience Settings.

The Convenience Settings portion of the Preferences tool allows you to customize the input and output characteristics of the Lisa. These characteristics are divided into three sections: Screen Contrast, Speaker Volume, and Rates. The Convenience Settings display is shown in Figure 3-1.



■Convenience Settings □Startup □Device Connections □Workshop

☑Set All Convenience Settings to Lisa Defaults

Screen Contrast

Normal Level

dark □ □ □ □ □ □ □ □ □ ■ □ □ □ □ □ □ bright

Minutes Until Screen Dims

□1-2 ■2-4 □5-10 □10-20 □15-30 □30-60

Dim Level

dark □ □ □ ■ □ □ □ □ □ □ □ □ □ □ □ □ bright

Before setting contrast, adjust brightness on back of the Lisa until the edges of this box are sharp.

Speaker Volume

Silent (Flash menu bar) □ Soft ■ □ □ □ □ Loud

Repeating Keys

Delay

Short □ □ □ ■ □ □ Long

Rate

Fast ■ □ □ □ □ □ Slow

Mouse Double Click Delay

Short □ □ ■ □ Long

Figure 3-1. Convenience Settings.

Screen Contrast

The contrast portion contains three sections. The first allows you to select the normal screen contrast level. Check in a contrast box until the contrast level is comfortable. Checking a box immediately changes the contrast.

The Lisa screen automatically dims if no activity is taking place on the screen to protect the screen from damage. The delay time before this dimming takes place is set with the Fade Delay section.

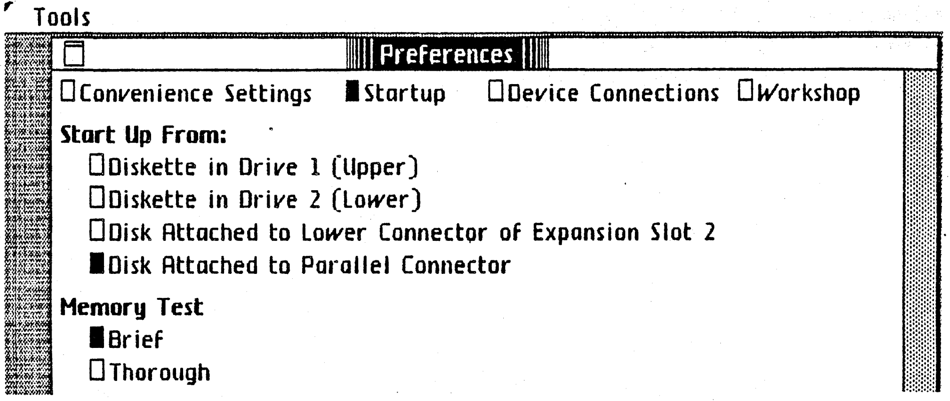
The third section allows you to set the dim contrast level. Checking a box in the Dim Level section makes the screen dim to that level until you move the mouse.

Speaker Volume
The speaker volume section allows you to set how loud the Lisa's audible alerts will be. Checking a box causes two beeps at the level you selected.

Rates
There are three rates that can be set, two for the keyboard and one for the mouse. The first is the initial keyboard repeat delay. This is the length of time a key must be depressed before it begins repeating. The second is the subsequent repeat delay. This is how quickly a key repeats after it has started repeating. The third rate is the mouse double click delay. This sets the maximum amount of time between two clicks that will be considered a double click. These three values should be set for your most comfortable use.

3.3.2 Start Up.

The Start Up display allows you to specify the boot device, and the type of memory test to be performed on startup. The Start Up display is shown in Figure 3-2.



□Convenience Settings ■Startup □Device Connections □Workshop

Start Up From:
- □Diskette in Drive 1 (Upper)
- □Diskette in Drive 2 (Lower)
- □Disk Attached to Lower Connector of Expansion Slot 2
- ■Disk Attached to Parallel Connector

Memory Test
- ■Brief
- □Thorough

Figure 3-2. The Start Up Display.

The Start Up display lets you select the Lisa system boot device. You are given a list of all possible boot devices. Select the one you want.

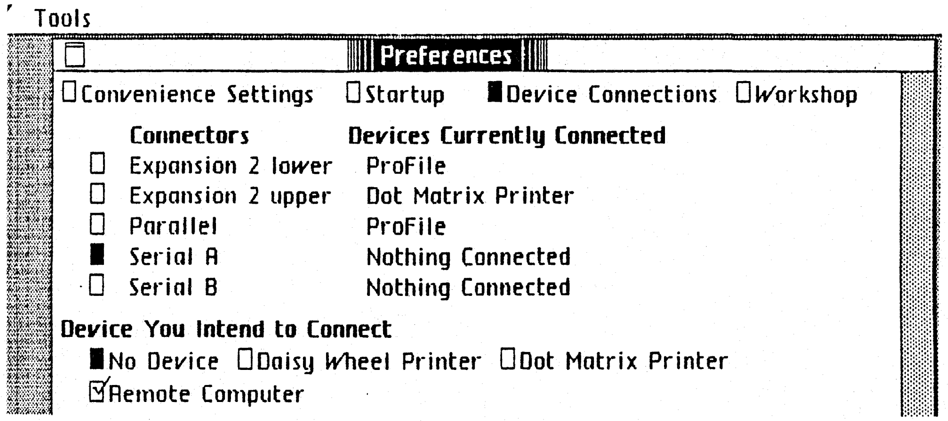
The Start Up display also allows you to select a long or short memory test. The brief test takes about 30 seconds, the long test takes about a minute.

Changes made to the Start Up display are put into Parameter Memory immediately, but have no effect until the system is booted again.

3.3.3 Device Connections.

The Device Connections display allows you to specify what devices are connected to the Lisa. When it is selected, it displays all ports that currently exist, along with the devices that are currently connected. To add, delete, or change the device connected to a port, select the port. All devices that may be connected to that port are displayed; you may also choose to have no device connected. When you select the device to connect, any additional configuration options for that type of device are displayed.

Any changes made to the device connections are made immediately to Parameter Memory, but they do not take effect until the next time the Lisa is booted. A typical device connections display is shown in Figure 3-3.



□Convenience Settings □Startup ■Device Connections □Workshop

	Connectors	Devices Currently Connected
□	Expansion 2 lower	ProFile
□	Expansion 2 upper	Dot Matrix Printer
□	Parallel	ProFile
■	Serial A	Nothing Connected
□	Serial B	Nothing Connected

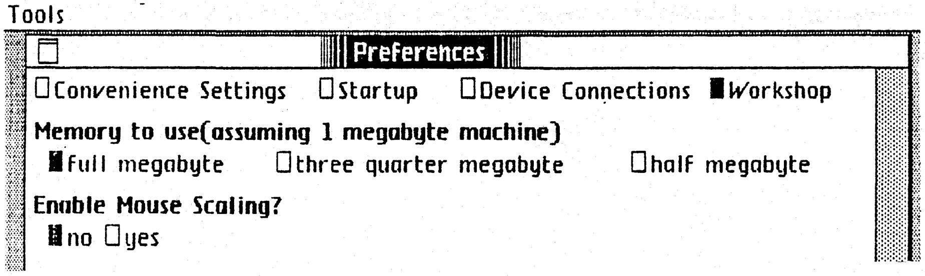
Device You Intend to Connect

■No Device □Daisy Wheel Printer □Dot Matrix Printer
☑Remote Computer

Figure 3-3. A Device Connections Display.

3.3.4 Workshop

The Workshop display allows you to set parameters of the Workshop system. The Workshop display is shown in Figure 3-4.



□Convenience Settings □Startup □Device Connections ■Workshop

Memory to use(assuming 1 megabyte machine)

■full megabyte □three quarter megabyte □half megabyte

Enable Mouse Scaling?

■no □yes

Figure 3-4. The Workshop Display.

3.3.5 The Tools Menu

The tools menu provides you with functions to access Parameter Memory. There are three functions provided: Set PM to defaults; Quit; and Print PM. Set PM to defaults sets parameter memory to the standard Lisa defaults. Quit exits you from the Preferences tool, and puts a copy of the current settings of parameter memory on the disk. Print PM displays all the values in parameter memory on the console.

3.4 Process Management

The Process Management subsystem is started by pressing M in response to the System Manager command line. This subsystem displays the following command line:

ManageProcess: AddResident, DeleteResident, KillProcess, ProcessStatus, Quit ?

This subsystem is used to control which processes will be resident. Making a process resident means that after it has run to completion, it will be suspended and retained in memory rather than terminated and removed from memory. This allows it to restart faster, because it does not have to be reloaded from disk. For example, if you are often using the Pascal compiler and the Editor, you can improve the performance of your Workshop system for these applications by making the compiler and the Editor resident. This will allow much more rapid shifting between the two.

See the Operating System Reference Manual for the Lisa for more information on processes

AddResident (A)
The AddResident command adds a process to the list of processes that are resident. You supply the file name of the object file that you want to be made resident the next time it is executed.

DeleteResident (D)
The DeleteResident removes a process from the list of resident processes.

KillProcess (K)
This command terminates a currently existing process.

ProcessStatus (P)
The ProcessStatus command gives you information about all currently existing processes. It provides the following information:

Pathname	The name of the object file in the process.
ProcessID	The unique identifier assigned to the process.
State	The current state of the process: Active, Suspended, or Waiting.
Resident	Tells you if this is a resident process.

Quit
Exit from the process **management** subsystem back to the System Manager command line.

Chapter 4
THE EDITOR

THE EDITOR

4.1 The Editor

The Editor is used to create and modify text files. These files may be used for many purposes including input to the language processors and as exec files.

If the file you are editing is too big to fit on the screen, a portion of the file is displayed. This "window" into the file can be moved to display any part of the file you want. An example of the Editor display window is shown in Figure 4-1.

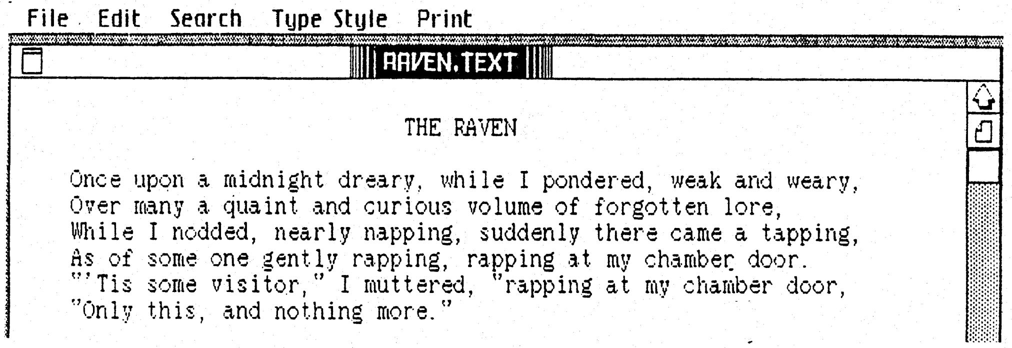
The basic editing operations are inserting characters, cutting a portion of the text, and pasting text into a new location. Items that are cut go into a special window called The Clipboard. Text on the Clipboard can be pasted into any place in the file, or into another file.

All editing action takes place at the insertion point. The insertion point is marked by a blinking vertical line where the next character will be placed. Any characters typed, or pasted from the Clipboard will be inserted at this point. This is true even if the insertion point is not currently displayed in the window. The window will automatically be scrolled to show the insertion point.

NOTE

> The editor is memory based. This means that there is a practical limit on the size of the file that can be edited. If a file is too big to edit, it should be split into more than one file of manageable size. The Filediv and Filejoin utilities can be used for this. They are described in Chapter 10.

The mouse is used to scroll the text in the window, move the insertion point, and select text to be cut or copied. Other operations, provided in five menus, are selected using the mouse.



THE RAVEN

Once upon a midnight dreary, while I pondered, weak and weary,
Over many a quaint and curious volume of forgotten lore,
While I nodded, nearly napping, suddenly there came a tapping,
As of some one gently rapping, rapping at my chamber door.
"'Tis some visitor," I muttered, "rapping at my chamber door,
"Only this, and nothing more."

Figure 4-1. The Editor Display Window

4.2 Using the Editor

Start the Editor by pressing E in response to the Workshop command prompt. The Editor will prompt you for a document name. If you want to edit an existing file, enter its name. If you want to create a new file, select Tear Off Stationery from the filing menu. The Editor will prompt you for the stationery name. Press RETURN for the default, which is blank paper. For more information on stationery, see below.

The file that you are working on is called the active document. You may have several documents open and accessible at any one time, but only the active document may be edited. The active window is indicated by a darkened title bar.

4.2.1 Editing Operations

The basic editing operations are Cut, Paste, and Copy. To cut or copy text, you must first select the text to be cut or copied. Select text by moving the mouse while holding down the button. See section 4.3 below for complete information on selecting text. Text that is selected and cut is removed from the active document and placed in a special window called The Clipboard. Text that is copied is placed on The Clipboard and also left in place in the active document.

The contents of The Clipboard may be inserted at any point in the active document by moving the insertion point to where you want the text inserted and selecting Paste from the edit menu.

4.2.2 The Menus

Operations are provided in five menus: File, Edit, Search, Type Style, and Print. The File menu is used to access things outside the Editor, such as documents and stationery. The Edit menu contains the editing operations. Search provides for finding strings in the active document. The Type Style menu selects the font for document display. The Print menu controls printing. Each of these menus is described in more detail below.

You select an operation from a menu by moving the arrow pointer to the menu name on the menu bar and holding down the button. The menu is displayed. Select the menu item by moving the mouse up or down until the right item appears in reverse video. Releasing the button starts the operation.

4.2.3 Creating and Using Stationery

Stationery for a special purpose (such as a letterhead) can be created with the Editor. Stationery is just a regular document containing the desired text. To use any stationery other than the default blank paper, select Tear Off Stationery from the File menu, and type the name of the document containing the stationery when it asks you for the stationery name.

To create stationery, make a document containing the standard text you want on the stationery. Save this document on the disk. To use this stationery, select Tear Off Stationery from the Edit menu, and give it the file name of the stationery you created.

4.2.4 Editing Multiple Files

More than one file may be open at one time, but only one document is the active document. To read in a document when you already have an active document, select Open from the File menu. It will ask you for the document name. The new document will be read in to a window on the screen and will become the active document. To make another document the active document, use the mouse to move the pointer into a portion of that document and click.

This capability may be used to copy text from one file to another by using the following sequence of operations:

- Open the document containing the text you want to copy.
- Select the text you want to copy and select Copy from the Edit menu. This places a copy of the text onto the Clipboard. You can use Cut if you want the text to be removed from its original file.
- Open the document you want the text to be copied to. It becomes the active document.
- Move the insertion point to the place you want the text to be inserted.
- Select Paste, which will copy the text from the Clipboard to the active document.

Further information on each of these operations may be found below.

4.3 Selecting Text

The basic editing functions are Cut, Copy, and Paste. Before you can Cut or Copy text, you must select the text to be cut or copied. Before you Paste, move the insertion point to where you want the text to be placed. You select text and move the insertion point by using the mouse to move the pointer on the screen.

When there is an active document, the pointer will have one of two shapes:

Text pointer in a document

Arrow pointer for menus and scroll bars

Use the mouse to move the pointer on the screen. The shape of the pointer will change when you move in and out of the document display window.

Within the display window, the text pointer is used to move the insertion point and to select text.

In selecting text, you may select characters, words, or lines. You may also select any number of characters, words, or lines. Selected text is displayed in reverse video.

4.3.1 How do I Move the Insertion Point?

The insertion point is indicated by a blinking vertical line where the next character will be inserted. All insertion, whether from typing or pasting, takes place at this point in the file, even if it is not visible in the window.

To move the insertion point, move the text pointer to where you want it to be and click. Note that the insertion point is also moved when you select text.

4.3.2 How do I Select Characters?

To select characters, move the text pointer to the beginning of the characters you want selected, press and hold the button while moving to the last character you want selected.

An alternate way of selecting characters, which is especially useful when selecting a large block of text, is as follows. Move the pointer to the beginning of the text you want selected and click. Then move the pointer to the end of the text you want selected and shift click (hold down the shift key on the keyboard and click the mouse button). You may use the scrolling controls to display the end of the text you want selected if it is too big to fit in the window.

4.3.3 How do I Select Words and Lines?

To select a word, move the text pointer into the word and click twice. To select a line, move the pointer into the line and click three times.

To select multiple words or lines, click the required number of times, and hold. Move the pointer to the last word or line you want selected and release.

An alternate method, especially useful when you want to select more text than will fit in one display window, is as follows. Click the required number of times to select the first word or line. Scroll the window if necessary to display the last item you want selected. Move the pointer to the last item you want selected, shift click, and the entire block of text will be selected.

4.3.4 How do I Adjust the Amount of Text Selected?

To change the amount of text selected, move the pointer to the position that you want the selection to extend to and shift click. This can be used to either expand or contract the selection.

4.4 Scrolling and Moving the Display

When a document is longer than will fit into the display window, only part of the document is displayed at one time. You can change what part is displayed by "scrolling" through the display. The vertical bar on the right side of the active window is the scroll bar. An example of a text window showing the scroll bar is in Figure 4-1.

The display window can be changed in size and moved on the screen. This allows you to have multiple files displayed on the screen. These operations are done using the title bar and size control box.

4.4.1 Scrolling the Display

There are three ways of moving the display window through the document. The first is by using the elevator. The elevator is the white rectangle in the scroll bar. Its position in the "elevator shaft" (the grey portion of the bar) indicates the relative position of the currently displayd text window in the document. If the elevator is near the top, you are near the beginning of the document. If it is near the middle, the text displayed on the screen is near the middle of the document, and so on. To change the position of the text window, you can use the mouse to move the arrow pointer into the elevator, click and hold the button down while you move the elevator to the position in the document you want to display. When you release the button, the display will be updated to the new position.

The second way of moving the window makes use of the view buttons. The view buttons are the boxes at each end of the elevator shaft. If you move the arrow pointer to a view button and click, the display will move one text window toward the beginning or end of the document, depending on which button you clicked.

The third way of moving the window uses the scroll arrows, which are just above and below the view buttons. If you move the arrow pointer to the bottom scroll arrow and click, the display window will move one line toward the end of the document. If you hold the button down, the window will continue to move a line at a time until you release it. The upper scroll arrow works the same way, except it moves the window towards the beginning of the document.

4.4.2 Moving the Display

You can move the display window on the screen and change its size. This lets you display multiple files on the screen. You can make any visible window be the active window by moving the pointer into it and clicking.

To move a window, move the pointer to the title bar, press the mouse button and hold it while you move the window. When you release the button, the window will be redisplayed at the new location.

To change the size or shape of the active window, move the pointer to the size control box, press the button, and move the pointer until the window is the right size and shape. Release the button and the resized window will be displayed. The size control box is the box in the lower right hand corner of the window. Only the active window can be resized.

4.5 The File Functions

The file menu provides functions for communicating with the outside world. Functions are provided for reading in and writing out documents, and for exiting the Editor. The Filing menu is shown in Figure 4-2. Each function is explained below.

filingmenu

Figure 4-2. The Filing Menu

Save & Put Away
This writes out the active document and closes it.

Save a Copy in ...
This writes out a copy of the active document to another file name. You are prompted for the name of the file to write to.

Save & Continue
This saves all changes made so far by writing out the document to disk, without closing the document.

Revert to Previous Version
This returns the document to the way it was before you started editing it, or when you last saved it. This is done by reading in the file from the disk.

Open ...
This tells the Editor to get a new document. It prompts you for the document name, then reads it in and makes it the active document. The Editor will supply the .TEXT extension on the file name.

Duplicate ...
This allows you to read in a copy of an existing document to edit into a new file. It is read in with the default name "untitled"

Tear Off Stationery ...
This gets a new piece of stationery and makes it the active document. See section 4.2.3 above for more information. The stationery is given the default name "untitled".

Exit Editor
This first asks you if you want to put away any modified documents. If you answer yes, they are written out to disk. Then it exits the Editor.

4.6 The Edit Functions

The Edit menu provides the editing functions and tab setting. It is shown in Figure 4-3.

The three basic edit functions are Cut, Paste, and Copy. These make use of the special window called The Clipboard. The Clipboard can hold one piece of text. Text is put into The Clipboard by selecting it in the active document, and either cutting it or copying it. Text is copied from the Clipboard and inserted at the insertion point with the paste operation.

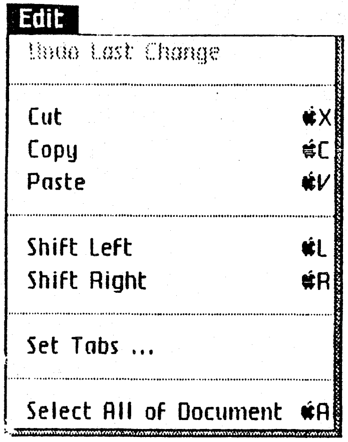


Figure 4-3. The Edit Menu

For example, to move a block of text from one place in a document to another, follow these steps:

1. Select the block of text to be moved.
2. Select Cut from the Edit menu. The text is removed from the active document and placed on the Clipboard.
3. Move the insertion point to where you want the text to be.
4. Select Paste from the Edit menu. The text on The Clipboard is inserted at the insertion point.

The edit menu also allows you to adjust selected text left or right by inserting or deleting spaces. It also allows you to set tabs.

Some edit functions may also be done by holding down (apple) and pressing another key. The key that corresponds to each function is shown in the edit menu. See figure 4-3.

Undo Last Change
This command puts the document back to the way it was before the previous operation if possible. The system will tell you if the last operation cannot be undone.

Cut
Cut places a copy of the currently selected text into The Clipboard and removes the text from the active document. You may also Cut by pressing (apple) X.

Copy
Copy places a copy of the currently selected text onto The Clipboard, but does not remove it from the active document. You can also Copy by pressing (apple) C.

Paste
Paste inserts a copy of the text on The Clipboard at the insertion point in the active document. You can also Paste by pressing (apple) V.

Shift Left
Shift Left moves selected text left by deleting a single space from the left of each line. It will not delete any characters other than spaces. It is most often used to adjust the left margin of a block of text. You can shift left by pressing (apple) L.

Shift Right
Shift Right is similar to Shift Left, except that it moves the selected text to the right by inserting spaces at the beginning of each line. This can also be done by pressing (apple) R.

Set Tabs ...
Set Tabs allows you to set the spacing of the tab stops.

Select All of Document

This command selects the entire document. You can select the entire document by pressing (apple) A.

4.7 The Search Functions.

The Search menu gives you the ability to search for a text string in the active document. The basic operation is Find, which locates the next occurrence of the string and selects it. Find & Paste All will replace each occurrence of the string with the contents of The Clipboard. Several options are provided to specify how the match is to be found. The Search menu is shown in Figure 4-4.

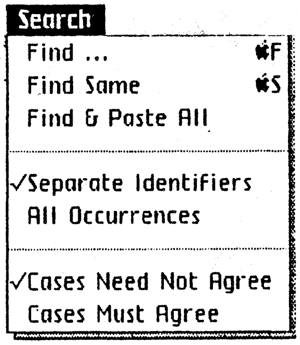


Figure 4-4. The Search Menu

All searches start at the insertion point, and go to the end of the file.

There are three search operations in the Search menu, as follows:

Find ...
Find prompts you for the string to search for, then finds the next occurrence of the string. If a match is found, it will be selected and displayed. The Find command can also be executed by pressing (apple) F.

Find Same
Find Same repeats a previously specified Find, and selects the next occurence of the string. You may do a Find Same by pressing (apple) S.

Find & Paste All
This finds all occurrences of the specified string from the current insertion point to the end of the file, and replaces each of them with the contents of the Clipboard.

The other four items in the search menu tell how a match is to be found. There are two areas to describe: searching for tokens or characters, and whether or not case must be matched. The options currently in effect have a check mark in front of them. To change the option, use the mouse to select the one you want.

The first set of options tells whether to search for tokens or to search literally:

Separate Identifiers
When Separate Identifiers is selected, the search operation will look for a "token" or word to match the search string. Only the first 8 characters are significant in a this type of search.

All Occurrences
When All Occurrences is selected, the search operation will match any string containing the same characters, even if it is only part of a word.

The next options indicate if case is significant in finding a match:

Cases Need Not Agree
When this item is selected, any string with the same characters will be a match, regardless of whether they are in upper or lower case.

Cases Must Agree
When this item is selected, the string must exactly match the search string, including case, to be selected.

4.8 The Type Style Functions

The Type Style menu allows you to change the display font. The Type Style menu is shown in Figure 4-5. A check appears in front of the font that the file is currently displayed in. You may change the font by selecting another font from the menu.

The font selected will affect how many characters may be displayed on a line, and whether or not the display is proportionally spaced. When a file is printed, it will be printed in the same type style it is displayed in.

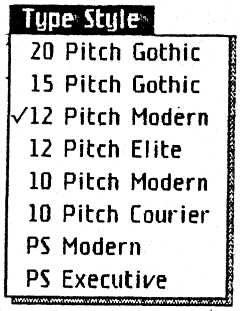


Figure 4-5. The Type Style Menu

4.9 The Print Functions

The Print menu provides functions for printing a document. You can print all or part of a document, choose what form of footers are to be printed, specify if Pascal keywords are to be emphasized, and tell what type of printer is being used. The Print menu is shown in Figure 4-6.

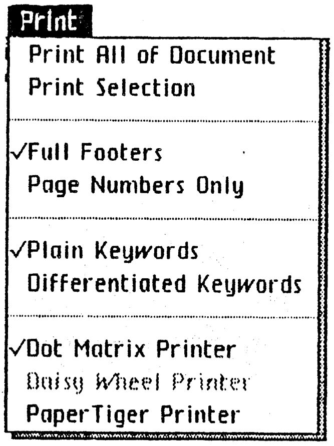


Figure 4-6. The Print Menu

The Print functions are as follows:

Print All of Document
This command prints the entire document.

Print Selection
This command prints only the currently selected portion of the document.

Both of the print commands will wait if the printer is not ready.

The remaining options in the Print menu chose how the print is to be performed. They are organized into 3 sets of 2 options. The currently selected option in each set is indicated by a check mark. You can select any combination of options you want.

The first options control what type of footers will be printed at the bottom of the page.

Full Footers
When Full Footers is selected, Each page printed will have a footer consisting of the file name, the page number, and the date.

Page Number Only
Selecting Page Number Only results in only a page number on the bottom of each printed page.

The next options are used for printing Pascal programs.

Plain Keywords
Selecting Plain Keywords makes Pascal keywords print with normal text.

Differentiated Keywords
Selecting Emphasized Keywords makes the printed output emphasize all Pascal keywords by underlining them.

The next options select the type of printer to print on. Select the type of printer you have attached to your Lisa:

Dot Matrix Printer

Daisy Wheel Printer

Chapter 5

THE PASCAL COMPILER

THE PASCAL COMPILER

5.1 The Pascal Compiler

The compiler translates Pascal source statements into object code. This translation is done in two steps. The first step (parsing) converts the program into semantically equivalent tree structures called I-code. The second step translates the resulting I-code into machine language.

A complete definition of Lisa Pascal is found in the Pascal Reference Manual for the Lisa.

The Pascal run-time support routines are in the library IOSPASLIB. After generating the object code, it is necessary to link the program with IOSPASLIB before you can run it. For information on how to link the program, see chapter 7 in this manual.

5.2 Using the Pascal Compiler

The compiler expects a text file containing a Pascal source program as input. You can create this text file using the Editor.

When you have prepared a source program, use the Compiler to translate it into object code. Start the compiler by pressing P in response to the workshop command prompt. The compiler first asks for the

```
Input file [.text] -
```

Type the name of the file that contains the source program. You do not need to add the .TEXT extension. The compiler then asks you for the

```
List file -
```

Type the name of the file that you want the listing to go to, or press RETURN if you don't want a listing. You can display the listing on the console by using the -console pathname. The compiler next asks you where to store the I-code form of the program:

```
I-code file [<input name>][.I] -
```

If you want the I-code to be stored in a file with the same name as the source file, but with a .I extension instead of the .TEXT, just press RETURN. If you want another name, type the name and press return.

After the last input, the compiler translates the program into I-code and stores it in the I-code file. If there were any errors, they will be displayed on the console.

5.2.1 Using the Code Generator

To translate the I-code into object code, press G in response to the shell command prompt. The code generator first asks you for the

```
Input file [.I] -
```

Type the name of the I-code file. You do not need to add the .I extension. The generator then asks you for the

```
        Output File [<input name>][.OBJ] -
```

To accept the default name, press RETURN. If you want a different name for the output file, type the name and press RETURN. The .OBJ extension will be added to the name for you.

The output file from the code generator is object code, but it is not executable because it does not contain the Pascal run-time support routines. The run-time support routines are contained in IOSPASLIB. These routines must be added to the object file by using the Linker. See chapter 7 in this manual for more information on the Linker.

5.2.2 Compiling with a Different Intrinsic Library

The Compiler and the code Generator both access INTRINSIC.LIB, the library of intrinsic units. It contains information about the intrinsic units used by the program. If you want the program to be compiled with a different intrinsic library, you can enter "?" to the request for an input file in both the Compiler and the Generator. They will ask you for the name of the intrinsic library you want to use. After entering the name of the intrinsic library, the compilation proceeds in the usual way.

5.3 The Pascal Compiler Commands

Compiler commands allow control of code generation, input file control, listing control, and conditional compilation. The commands all start with a $, and are placed as comments in the source program where you want the command to take effect. A complete list of the compiler commands is found in the Pascal Reference Manual for the Lisa .

5.4 Further Information

For further information on the Pascal language, refer to the Pascal Reference Manual for the Lisa. A Pascal program can call assembly language routines. More information on assembly Language is in chapter 6 of this manual.

The Debugger, described in Chapter 8, can be used for run time debugging of Pascal programs. More information on the run time environment of a Pascal program is found in Chapter 6.

The Operating System provides a number of routines that can be called from a Pascal program to perform various system functions. These routines are in the SYSCALL unit, which is described in the Operating System Reference Manual for the Lisa.

Chapter 6

THE ASSEMBLER

THE ASSEMBLER

6.1 The Assembler

The assembler is a program that translatesassembly language source code into object code. The assembler accepts a text file containing the source code as input, and produces an object file as output.

The object file produced must be linked with a Pascal main program before it can be executed.

Assembly language routines are used to implement low level or time critical functions. This chapter describes how to use the assembler, and the syntax of assembly language programs. Information on the machine instructions available on the 68000 processor is found in the Motorola manual.

6.2 Using the Assembler

To assemble a program, press A from the Workshop command line. Then specify the input file (the file that contains your source program) and two output files: the object file (the file that contains the machine-language code produced by the assembler) and an optional listing file.

The input file must be a text file containing assembly language source statements. You can make this file with the editor. The output file produced is an object file (.OBJ), that must be linked with a Pascal main program to be run.

6.2.1 Assembler Options

When you start the assembler, the option settingsare displayed. You may change the options by responding to the input file prompt with "?". There are two assembler options:

P	Pretty Listing.
S	Printinformation about available space.

Each option may be set to + or -:

+	On
-	Off

When Pretty Listing is on, the forward jump addressesare filled in with the correct values.

After setting any options desired, press return, and the assembler asks you for the name of the input file. The assembler then asks you for the name of the listing, and the output files.

6.2.2 The Input File

The input file is a text file containing assembler language source statements. A file created using the Editor will be in text file format.

When the assembler asks you for the name of the input file, type "?" if you want to change assembler options at this time; otherwise type the pathname of your source file.

6.2.3 The Object File

The object file produced by the assembler contains a machine language versionof your source program. The name of an object file ends with .OBJ. An object file is not executable; it must be linked with a Pascal program thatcalls it. See Section 6.6 forfurtherinformation.

The output file will be an object file which must be linked with a Pascal main program before itcan be executed.

6.2.4 The Listing File

The listing file produced by the assembler contains a list of source statements and their machine-language equivalent. If Pretty Listing is off, all addresses for forward referencing branches will be displayed as asterisks("****). If Pretty Listing is on, the actual value will be filled in.

Source statement errors are flagged in the listing. Refer to the Appendix for a listof assembler errormessages.

An example of an assembler listingfile isshown in Figure6-1.

assemblerlisting

```
TAIL      MC --------|
PAGE -    5  ASMSTR     FILE: EX/ASM/STR.TEXT


0000|                              .proc      AsmStr
0000|  205F                        move.l     (a7)+, a0        ; return address
0002|  225F                        move.l     (a7)+, a1        ; address of string passed from Pascal pgm
0004|  2F0A                        move.l     a2, -(a7)        ; save scratch reg a2
0006|
0006|  45FA 0016                   lea        size, a2
000A|  4280                        clr.l      d0
000C|  1012                        move.b     (a2), d0         ; get size of string
000E|
000E|  12DA                        move.b     (a2)+, (a1)+     ; copy size of string (first byte of string)
0010|  5340                   copy subq       #1, d0           ; done copying string?
0012|  6500 0006                   blo        done             ; yes, return to pascal
0016|  12DA                        move.b     (a2)+, (a1)+     ; copy one char of the string
0018|  60F6                        bra        copy
001A|
001A|  245F                   done move.l     (a7)+, a2        ; restore scratch reg
001C|  4ED0                        jmp        (a0)             ; return to pascal
001E|
001E|  26                     size      .byte     38
001F|  74 68 69 73 20 73 74   myStr     .ascii    'this string is from the LISA assembler'
0026|  72 69 6E 67 20 69 73
002D|  20 66 72 6F 6D 20 74
0034|  68 65 20 4C 49 53 41
003B|  20 61 73 73 65 6D 62
0042|  6C 65 72
0045|  00                               .align    2            ; just to be sure next instruction is on word
0046|                                                          ; boundary (even address)
0046|
0046|                              ;.end
0046|
```

Figure 6-1. Assembler Listing

If you specify a device name such as -PRINTER or -CONSOLE for the listing file, the listing will be printed on that device. If you specify a disk file, the listing will be created as a text file; you may then print it by using the Copy command in the File Manager command line.

6.3 Assembler Opcodes

The 68000 opcodes are described in the Motorola MC68000 Microprocessor User's Manual. The assembler has two variant mnemonics for branches (BHS for BCC and BLO for BCS). The variant names are more indicative of how the instruction is being used after unsigned comparisons. The default radix is decimal.

The size of an operation (byte, word, or long) is specified by appending either .B, .W, or .L to the instruction. The default operation size is word. To cause a short forward branch, append a .S to the instruction. The default branch size is Long.

6.3.1 Optimization

It should be noted that the Assembler accepts generic instructions and assembles the correct form. The instruction ADD, for example, is assembled into ADD, ADDA, ADDQ, or ADDI, depending on the context.

```
                ADD       D3,A5
becomes         ADDA      D3,A5.
```

MOVE, CMP, and SUB are handled in a similar manner.

6.4 Assembler Syntax

This section describes the form in which the assembler expects an assembly language program. We describe the structure of an assembly language program in section 6.4.1. We then describe the form of constants, identifiers, labels, expressions, and how to specify addressing modes.

6.4.1 Structure of an Assembly language Program

An assembly language program contains one or more procedures or functions. The structure of an assembly language source file lookes like Figure 6-2. First it contains an (optional) section of non code generating operations. This is usually where any constants or macros are defined. Next it conains one or more procedures (.PROC) or functions (.FUNC). These each contain a sequence of code generating operations and directives. A procedure or function is ended when the assembler encounters the next .PROC or .FUNC. A .END directive is the last statement in the program. Any text beyond the .END is ignored.

```
non code generating operations

.PROC (or .FUNC)
code generating operations and any directives needed

.PROC
...

etc.

.END
```

Figure 6-2. Structure of an Assembly Language Program

The non code generating directives are:

.EQU	.MACRO	.IF	.LIST	.MACROLIST
	.ENDM	.ELSE	.NOLIST	.NOMACROLIST
.REF		.ENDC	.PAGE	.PATCHLIST
.DEF			.TITLE	.NOPATCHLIST

6.4.2 Constants

Constants in the Assembler can be either numeric or string constants.

6.4.2.1 Numeric Constants

Numeric constants in the assembler can be expressed in decimal, hexadecimal, octal, or binary. The default radix is decimal . The other three bases are expressed as follows:

Hexadecimal
Hex numbers can be expressed in two ways:

1. Preceed the number with a "$". Examples of this are:

   $FF13
   $127

2. Follow the number with an "H". Using this form, the number must start with a digit (0-9). Examples:

   0FF13H
   195H

Octal
Octal numbers are followed by the character "O". Note that this is the letter O, not the character zero (0). Examples:

   77O
   104O

Binary

Binary numbers are followed by the character "B". Examples:

```
1011B
111000B
```

6.4.2.2 String Constants

String constants are delimited by matching pairs of single or double quotes. Examples of string constants are:

```
"this is a string constant"
'using single quotes as delimiters lets you include "double" quotes'
```

6.4.3 Identifiers

Only the first eight characters of identifier names are meaningful to the assembler. The first character must be alphabetic; the rest must be alphanumeric, period, underbar, or percent sign.

Examples of identifiers are:

```
LOOP
EXIT_PRC
NUM
```

6.4.4 Labels and Local Labels

Labels begin in column one. They can be followed by a colon, if you like.

Local labels can be used to avoid using up the storage space required by regular labels. The local label stack can handle 21 labels at a time. It is cleared every time a regular label is encountered. Local labels in this assembler start with the character @. A local label is an @ followed by a string of decimal digits (0-9). Examples of local labels are:

```
@123
@2
@79
```

6.4.5 Expressions and operators

All quantities are 32 bits in size unless constrained by the instruction. Expressions are evaluated from left to right with no operator precedence.

Angle brackets can be used to control expressionevaluation. The following operatorsare available:

+	unary or binary addition
−	unary minus or subtraction
~	ones complement (unary operator)
^	exclusive or
*	multiplication
/	division(DIV)
\	MOD
|	logical OR
&	logical AND
=	equal (used only by .IF)
<>	not equal (used only by .IF)

There is no operator precedence in expressions. For example, in the expression 2 + 9 * 4, the addition is performed first. To make the multiplication be performed first, the expression can be rewritten with brackets to show precedence: 2 + <9 * 4>, or the operands can be reordered as: 9 * 4 + 2.

6.4.6 Addressing Modes

The following is a summary of the addressing mode syntax for the 68000. Refer to the Motorola 68000 manual for information on the addressing modes supported by the 68000. Table 6-1 gives a summary of the addressingmodes including their syntax.

Table 6-1. Summary of Addressing Modes

Mode	Register	Syntax	Meaning	ExtraWords
0	0..7	Di	Data direct	0
1	0..7	Ai	Addressdirect	0
2	0..7	(Ai)	Indirect	0
3	0..7	(Ai)+	Postincrement	0
4	0..7	-(Ai)	Predecrement	0
5	0..7	e(Ai)	Indexed	1
6	0..7	e(Ai,Ri)	Offsetindexed	1
7	0	e	Absolute short address	1
7	1	e	Absolute long address	2
7	2	e	PC Relative	1
7	3	e(Ri)	PC Relative indexed	1
7	4	#e	Immediate	1 or 2

Notes:

1) The indexed and PC relative indexed modes are determined by the opcode.

2) The absolute address and PC relative address modes are determined by the type of the label (absolute or relative).

3) The absolute short and long address modes are determined by the size of the operand. Long mode is used only for long constants.

4) The number of extra words for immediate mode is determined by the opcode size modifier (.W or .L).

6.4.7 Miscellaneous Syntax

Comments

A semicolon begins a comment in an assembly language program. All characters on a line after a semicolon are ignored. This is an example of comments:

```
;   This is a comment on a line by itself
CLR.L  D0                ;comment after a statement
```

Current Program Location

The current program location is indicated in assembly language by the symbol "*". Examples of its use are:

```
JMP  *                   ;  Loop infinitly
JMP  *-4                 ;  Jump back 4 bytes
```

Move Multiple (MOVEM)

To specify which registers are affected by Move Multiple (MOVEM), specify ranges of registers with "-", and specify separate registers with "/". For example, to push registers D0 through D2, D4, and A0 through A4 onto the top of the stack:

```
MOVEM.L  D0-D2/D4/A0-A4,-(A7)
```

6.5 Assembler Directives.

The Assembler directives (pseudo-ops) are:

.PROC	<identifier>[,Expr]	begin procedure with Expr args
.FUNC	<identifier>[,Expr]	begin function with Expr args
.DEF	<identifier-list>	make identifiers externally available
.REF	<identifier-list>	declare external identifiers
.SEG	'<name>'	put following code in segment 'name'
.END		end of entire assembly
.ASCII	'<character-string>'	place ASCII string in code
.BYTE	<value-list>	allocate a byte in code for each value
.BLOCK	<length>[,value]	allocate length bytes of value
.WORD	<value-list>	allocate a word for each value
.LONG	<value-list>	allocate a long word for each value
.ALIGN	<Expr>	allign next code on multiple of Expr
.ORG	<value>	place next byte at <value>
.RORG	<value>	same as .ORG
.EQU	<value>	set label equal to <value>

.MACRO	<identifier>	begin macro definition
.ENDM		end macro definition
.IF	<expr>	begin conditional assembly
.ELSE		optional alternate to .IF block
.ENDC		end conditional assembly
.LIST		turn on assembly listing
.NOLIST		turn off assembly listing
.PAGE		issue a page feed in listing
.TITLE	'<title>'	title of each page in listing
.MACROLIST		turn on macro expansion listing
.NOMACROLIST		turn off expansion listing
.PATCHLIST		turn on patchlist
.NOPATCHLIST		turn off patchlist
.INCLUDE	<filename>	insert <filename> into assembly

6.5.2 Space Allocation Directives.

The space allocation directives are .ASCII, .BYTE, .WORD, .LONG, and .BLOCK.

.ASCII 'string'
converts 'string' into the equivalent ASCII byte constants and places the bytes in the code stream. The string delimiters must be matching single or double quotes. To insert a single quote into the code use double quotes as delimiters. Similarly for double quotes:

```
.ASCII   "AB'CD"        ; string containing a single quote
.ASCII   'AB"CD'        ; string containing a double quote
```

.BYTE <values>
allocates a byte of space in the code stream for each of the values given. Each value must be between -128 and 255.

.BLOCK <length>[,value]
allocates <length> bytes, each filled with the value given. If no value is given, a block of zeros is allocated.

.WORD <values>
allocates a word of space in the code stream for each of the values listed. The values must be between -32768 and 65535.

Forexample,

```
TEMP  .WORD    0,65535,-2,17
```

createsthe assembled output:

```
    0000
    FFFF
    FFFE
    0011
```

.LONG <values>
allocates two wordsof space foreach value in the list. Forexample,

```
STUFF .LONG    0,65535,-2,17
```

createsthe output:

```
    00000000
    0000FFFF
    FFFFFFFE
    00000011
```

<label> .EQU <value>
assigns<value> to<label>. <value> can be an expressioncontaining other labels.

.ORG <value>
puts the next byte of code at <value> relative to the beginning of the assembly file. Bytes of zero are inserted from the current location to <value>.

.RORG
is similar to .ORG. It indicates that the code is relocatable. Because the loader does not support absolute loading, .ORG and .RORG accomplish the same function. **All addressingmust be PC relative.**

RORG (without the leading period) is the same as .RORG. Similarly, END = .END, EQU = .EQU, PAGE = .PAGE, LIST = .LIST, NOL = .NOLIST.

6.5.3 Macro Directives.

A macro consistsof a macro name, optional arguments, and a macro body. When the assembler encounters the macro name, it substitutesthe macro body for the macro name in the assembly text. Wherever %n occurs in the macro body (where n is a single decimal digit), the text of the n-th parameter is substituted. If parameters are omitted, a null stringis used in the macro expansion. A macro can invoke other macros up to five levels deep. In the assembly listing, the listing of the expanded macro code is controlled by the options .MACROLIST and .NOMACROLIST. These options are described in Section 6.5.5.

```
        .MACRO   <identifier>
        .
        .
        .ENDM
```

defines the macro named <identifier>. The following is an example of a macro:

```
        .MACRO   Help
        MOVE     %1,D0
        ADD      D0,%2
        .ENDM
```

If 'Help' is called in an assembly with the parameters 'Alpha' and 'Beta', the listingcreated would be:

```
        Help    Alpha,Beta
#       MOVE    Alpha,D0
#       ADD     D0,Beta
```

6.5.4 Conditional Assembly Directives.

The conditional assembly directives .IF, .ELSE, and .ENDC are used to include or exclude sections of code at assembly time based on the value of some expression.

.IF <expression>
identifies the beginning of a conditional block. <expression> is considered to be false if it evaluates to zero. Any non-zero value is considered true. The expression can also involve a test for equality (using <> or =). Strings and arithmetic expressions can be compared. If <expression> is false, the Assembler ignores code until a .ELSE or .ENDC is found. The code between the optional .ELSE and .ENDC is assembled if <expression> is false. Otherwise it is ignored.Conditionals can be nested. The macros HEAD and TAIL given in section 6.6.1 provide examples of the use of conditionals. The general form is:

```
        .IF      <expr>
        .                        ;assembled if <expr> is true
        .
        [.ELSE]                  ;optional
        .                        ;assembled if <expr> is false
        .
        .ENDC
```

6.5.5 External Reference Directives.

Separate routines can share data structures and subroutines by linkage between assembly routines using .DEF and .REF. These directives cause the Assembler to generate link information that allows separately assembled assembly routines to be linked together. .DEF and .REF associate labels between assembly routines, not between assembly routines and Pascal.The only way to communicate data between Pascal and assembly routines is by using the stack. This is done by passing them as parameters in the procedure or function call. Information on parameter passing between Pascal and assembly language is found in section 6.6.

.DEF <identifier-list>
identifies labels defined in the current routine as available to other assembly routines through matching .REFs. The .PROC and .FUNC directives also generate code similar to that generated by a .DEF with the same name, so assembly routines can call external .PROCs and .FUNCs with .REFs.

```
      .PROC    Simple,1
      .DEF     Alpha, Beta
      .
      .
      BNE      Beta
      .
      .
Alpha MOVE
      .
      RTS
Beta  MOVE
      .
      RTS
      .END
```

This example defines two labels, Alpha and Beta, which another assembly routine can access with .REF.

.REF <identifier-list>
identifies the labels in <identifier-list> used in the current routine as available from some other assembly routines which defined these identifiers using the .DEF directive.

```
      .PROC    Simple
      .REF     Alpha
      .
      .
      JSR      Alpha
      .
      .END
```

uses the label 'Alpha' declared in the .DEF example.

When a .REF is encountered, the assembler generates a short absolute addressing mode for the instruction (the opcode followed by a word of 0's) and a short external reference with an address pointer to the word of 0's following the opcode. If the referenced label and the reference are in the same segment module, the Linker changes the addressing mode from short absolute to single-word PC relative. If, however, the referenced procedure is in a different segment, the Linker converts the reference to an indexed addressing mode (off A5) and the word of zeros is converted into the proper entry offsetin the jump table. If the referenced procedure is in an intrinsic unit (and therefore in a differentsegment), the IUJSR, IULEA, IUJMP, and IUPEA instructionsare used (seepage ##). The Linker blindly assumes that the word immediately before the word of zeros is an opcode in which the low order 6 bitsare the effective address. Thus, a .REF label cannot be used with any arbitrary instruction. The .REF labels are intended for JSR, JMP, PEA, and LEA instructions.

.SEG

default segment name is "   " (8 blanks). .SEG "segment name" puts the code in segment called "segment name".

6.5.6 Listing Control Directives.

The directives that control the Assembler's listing file output are .LIST, .NOLIST, .PAGE, .TITLE, .MACROLIST, .NOMACROLIST, .PATCHLIST, and .NOPATCHLIST. If you do not specify a name for the listing file in response to the Assembler'sprompt:

```
Listingfile (<cr> for none) -
```

the listingdirectivesare ignored.

The default for the assembleris for .LIST, .MACROLIST, and .PATCHLIST to be in effect when the assembler starts. .TITLE defaultsto blank.

.LIST and .NOLIST
can be used to select portions of the source to be listed. The listinggoes to the specified output file when .LIST is encountered. .NOLIST turns off the listing. .LIST and .NOLIST can occur any number of times during an assembly.

.PAGE
insertsa page feed into the listingfile.

.TITLE '<title>'
specifies a title for the listing page. <title> can contain up to 80 characters, and can be enclosed in either single or double quotes.

```
.TITLE  'Interpreter'
```

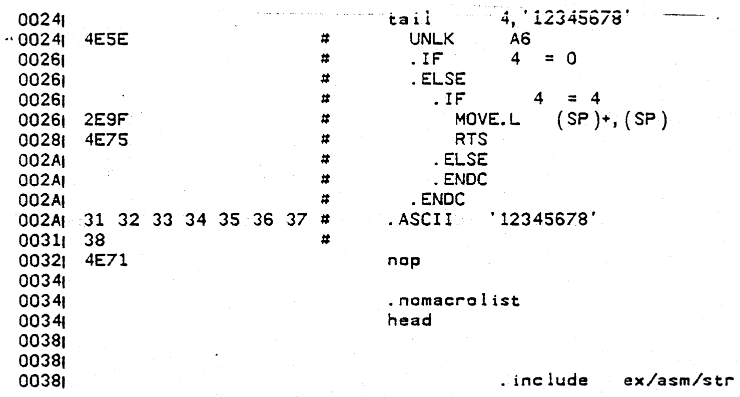
places the word, Interpreter, atthe head of each page of the listing.

.PATCHLIST
patches the forward referenced labels in the listing. It must be on if you want prettylisting.

.NOPATCHLIST
turnsoff patching of forwardreferences.

.MACROLIST
turnson listingof the expanded code from a macro.

.NOMACROLIST
turns off listing of macro expansion. See Figure 6-3 for examples of the macro listingoptions.



```
0024|                               tail      4,'12345678'
0024|  4E5E                     #     UNLK      A6
0026|                           #     .IF       4  = 0
0026|                           #     .ELSE
0026|                           #       .IF       4  = 4
0026|  2E9F                     #         MOVE.L    (SP)+,(SP)
0028|  4E75                     #         RTS
002A|                           #       .ELSE
002A|                           #       .ENDC
002A|                           #     .ENDC
002A|  31 32 33 34 35 36 37 #   .ASCII    '12345678'
0031|  38                       #
0032|  4E71                         nop
0034|
0034|                               .nomacrolist
0034|                               head
0038|
0038|
0038|                                          .include   ex/asm/str
```

Figure 6-3. Macro Listing Options

6.5.7 File Directives.

The pseudo-op

.INCLUDE <filename>
causes the contents of <filename> to be assembled at the point of the .INCLUDE. <filename> need not specify the .TEXT suffix.An included file cannot itselfcontain a .INCLUDE statement.

6.6 Communication with Pascal.

Pascal programs can call assembly language procedures. The Pascal program declares the assembly language procedure or function to be EXTERNAL. If the assembly routine does not return a value,use .PROC. If .FUNC is used, space for the returned value is inserted on the stack just before the function parameters, if any. The amount of space inserted depends on the type of the function. A LongInt or Real function resulttakes two words, a Boolean resulttakes one word with the resultin the high order byte, and other types take one word. In the following example, we link a bit-twiddling assembly language routine into a Pascal program. The Pascal hostfile is:

```
PROGRAM BITTEST;
VAR I,J: INTEGER;
FUNCTION Iand( i, j : INTEGER ) : INTEGER;
  EXTERNAL;        (* external =Assembly language *)

BEGIN
  i := 255;
  j := 33;
  WRITELN (I,J,' AND= ',Iand (I, J));
END.
```

The Assembler file is:

```
.FUNC    IAND,2          ; two arguments
MOVE.L   (A7)+,A0        ; return address
MOVE.W   (A7)+,D0        ; J
MOVE.W   (A7)+,D1        ; I
AND.W    D1,D0           ; I AND J
MOVE.W   D0,(A7)         ; put function result on stack
JMP      (A0)
.END
```

In the example given above we have made little attempt to make the assembly language procedure mimic the structure of a procedure generated by the Pascal Compiler. A complete description of this structure requires some preliminary discourse.

6.6.1 The Run Time Stack

Automatic stack expansion code makes procedure entries a little complicated. To ensure that the stack segment is large enough before the procedure is entered, the compiler emits code to 'touch' the lowest point that will be needed by the procedure. If we 'touch' an illegal location (outside the current stack bounds), the MMU hardware signals a bus error that causes the 68000 to generate a hardware exception and passcontrol to an exception handler. This code, provided by the operating system, must be able to restore the state of the world at the time of the exception, and then allocate enough extra memory to the stack that the original instruction can be re-executed without problem. To be able to back up, the instruction that caused the exception must not change the registers, so a TST.W instruction with indirect addressing is used.

In the normal case, the procedure's LINK instruction should be preceded by a TST.W e(A7) which attempts to reach the stack location that can accomodate the static and dynamic stack requirements of the procedure. If the static and dynamic stack requirements of your assembly language procedure are less than 256 bytes, you can assume that the compiler's fudge factor will protect the assembly language procedure, so the TST.W can be omitted. If the requirements are greater than 32K bytes, e(A7) may not be sufficient because only 16 bits of addressability are available (the 68000 does call a 16-bit processor). In this case, the compiler currently emits code something like:

```
MOVE.L   A7,A0
SUB.L    #Size,A0        ;#size=dynamic + static needed
TST.W    (A0)
```

If the compiler option D+ is in effect (the default), the first eight bytes of the data area following the final RTS or JMP (A0) contain the procedure name. LisaBug gets the procedure name from this block, making debugging much more pleasant. The following example is provided to show how an assembly language programmer can provide LisaBug with all the information it needs to perform fully symbolic low level debugging.

```
;
; ASSEMBLY LANGUAGE EXAMPLE

 DEBUGF .EQU 1                 ; true => allow debugging with
proc names

;   HEAD -- This MACRO can be used to signal the
;   beginning of an assembly language procedure.  HEAD
;   should be used when you do not want to build a stack
;   frame based on A6, but do want debugging information.
;
;   No arguments

      .MACRO   HEAD
       .IF      DEBUGF
            LINK    A6,#0           ; fancy NOP used by debugger
       .ENDC
      .ENDM

;   TAIL -- This MACRO can be used as a generalized exit
;   sequence.  There are two cases.  First, if you build
;   a stack frame, TAIL can be used to undo the stack
;   frame, delete the parameters (if any) and return.
;   Second, if you do not want to build a stack frame
;   based on A6, this MACRO can be used to signal the
;   end of an assembly language procedure.  In either
;   case if DEBUGF is true, the Procedure_name
;   is dropped by the MACRO as an 8-character name.
;
;   Two arguments:
;          1) Number of bytes of parameters to delete
;          2) Procedure_Name as string exactly 8 characters
;
;
      .MACRO   TAIL
         UNLK     A6
         .IF      %1 = 0
            RTS                     ; 0 bytes of parameters
         .ELSE
            .IF      %1 = 4
               MOVE.L   (A7)+,(A7)  ; 4 bytes of parameters
               RTS
            .ELSE
               MOVE.L   (A7)+,A0    ; put return addr into A0
               ADD.W    #%1,A7      ; remove params from stack
               JMP      (A0)        ; return to caller
            .ENDC
         .ENDC
```

```
          .IF      DEBUGF
             .ASCII   %2
          .ENDC
       .ENDM
;
;   The following example demonstrates the use of the
;   TAIL macro for the purpose of debugging.  The example
;   assumes that you want to build a stack frame based
;   on A6.  In a real assembly language procedure the
;   zeroes below would be replaced by the local size and
;   parameter size.

       .PROC    SIMPLE,0
       LINK     A6,#0             ; zero bytes of locals
       NOP                        ; body of procedure
       TAIL     0,'SIMPLE  '      ; zero bytes of parameters
       .END
```

These macros are sufficient for the programmer writing small assembly language routinesto be called from Pascal.

Upon entry to the assembly routine,the stack is as shownin Figure 6-4.

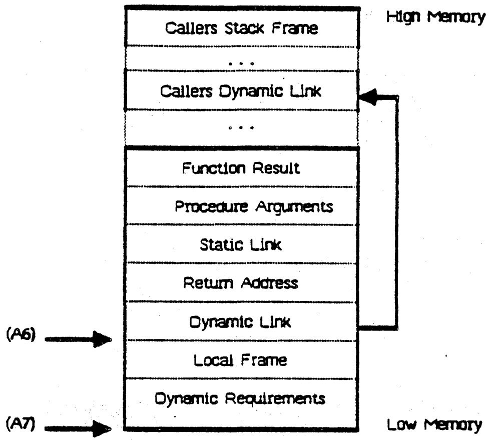


Figure 6-4. The Pascal Run Time Stack

The function result is present only if the Pascal declaration is for a function. It is either one or two words. If the result fits in a single byte (a boolean, for example), the most significant half (the lower addressed half) gets the result value.

Parameters are present only if parameters are passed from Pascal. They are pushed on the stack in the order of declaration: All reference parameters are represented as 32 bit addresses. Value parameters less than 16 bitsin size alwaysoccupy a fullword. All non-setvalue parameters larger than 4 bytes are passed by reference. It is the procedure's responsibilitytocopy them. All large setvalue parameters are pushed onto the stack by the calling routine.

The static link is present only if the external procedure's level of declaration is not global. The link is a 4 byte pointer to the enclosing static scope.

It is the responsibilityof the assembly language procedure to deallocate the return address, the static link (if any), and the parameters (if any). The SP must point to the function result or to the previous top of the stack upon return. Registers D4 through D7 and A3 through A7 must be preserved. It is recommended that you also preserve D3 and A2.

6.6.2 Register Conventions

The following are the registerconventions used in the Lisa system. It is the responsibilityofthe programmer topreserve these registers.

D0-D2/A0-A1:	Scratch registers (can be clobbered)
D3,A2:	Scratch registers, but should be preserved
D4-D7/A3,A4:	Used for code optimization
A5:	Pointer to user globals (must be preserved)
A6:	Pointer to base of stack (must be preserved)
SP:	Top of stack

RegistersD3 and A2 may be used at some time in the future by the compiler for code optimization, so the assembly language programmer should preserve them also.

6.6.3 Assembly Language Examples

The following examples show how to use certain featuresof the assembly language.

The first example illustrates the use of .REF and .DEF. These two directives allow an assembly language routine to reference another assembly routine.

The Pascal host file is:

```
program WasteTime;
procedure Wait (time : integer);
    external;
begin
    writeln ('Going to waste some time');
    wait (50);
    writeln ('Finished wasting time');
end.
```

The assembly language file is:

```
      .proc    wait
      .ref     cycle          ; need to use a piece of code
                              ; whose entry point is cycle
                              ; defined outside procedure wait
      .ref     more_time      ; another outside procedure
      move.l   (a7)+,a0       ; return address in a0
      move.w   (a7)+,d0       ; need to wait this many cycles
                              ; a parameter for cycle
      jsr      cycle
      jsr      more_time      ; waste more time
      jmp      (a0)           ; return

      ; the subroutine used by wait is defined in the
      ; following code.  this proc could do other things
      ; besides the cycle routine
      .proc    def_cycle
      .def     cycle          ; cycle visible to other procs
      ;
      ;  code can go here
      ;
      nop                     ; example of a line of code
cycle                         ; beginning of the cycle routine
                              ; parameter is in d0
      sub      #1,d0
      bne      cycle
      rts
      ;
      ;  more code can go here
      ;
      .proc    more_time      ; waste more time
      clr      d0             ; use d0 as timer
@1    add      #2,d0
      bne      @1
      rts

      .end
```

The following program illustrates how to pass a Pascal string to an assembly language program, modify the string,and returnit. Pascal strings have theirlength storedas the firstbyte in the string.

```
      .proc    AsmStr
      move.l   (A7)+,A0       ;return address saved in A0
      move.l   (A7)+,A1       ;address of string from Pascal
      move.l   A2,-(A7)       ;save scratch register A2
```

```
        lea       size,A2
        clr.l     D0
        move.b    (A2),D0         ;get size of string

        move.b    (A2)+,(A1)+     ;copy size of string
copy    subq      #1,D0           ;done copying string?
        blo       done            ;yes, return to Pascal
        move.b    (A2)+,(A1)+     ;one char of string
        bra       copy

done    move.l    (A7)+,A2        ;restore scratch register
        jmp       (A0)            ;return to Pascal

size    .byte     38
myStr   .ascii    'this string is from the LISA assembler'
```

Chapter 7
THE LINKER

THE LINKER

7.1 The Linker.

The Linker combines object files. Its input consists of commands and object files. Its output consists of object files, link-map information, and error messages. The output of the Pascal compiler must be linked with IOSPASLIB before it can be executed. Other object files, including intrinsic unit libraries, and object files produced by the Assembler, can also be linked into the output object file.

What the Linker does is as follows. When a program is compiled into an object file, it contains the following sorts of things:

- Object code, similar to machine language, that expresses the algorithm of the program.
- Symbolic (named) addresses of all code whose location was unknown to the compiler. These include externally compiled routines (units and intrinsic units) and the Pascal library support routines (PASLIB).
- Other information to be used by the Linker.

The purpose of the Linker is to connect up all the necessary things (linking them together), and output an object file that can be executed.

The Linker does this by going through the main program, and, each time it finds a symbolic address, it looks up that address in all the units and libraries it was given as input, and converts the symbolic address into a real address that will be correct when the program is loaded to be executed.

If the Linker can't find something that is addressed symbolically, this is an error. An error message will be printed, indicating the missing module. This process of finding the real addresses that correspond to the symbolic addresses is called resolving the external references.

The Linker expects to find the file INTRINSIC.LIB even if you are not using any intrinsic units. INTRINSIC.LIB is a directory of libraries and intrinsic units, and includes information for the use of the Linker. INTRINSIC.LIB defines all the intrinsic units supplied with the Workshop system.

7.1.1 Creating an Executable File.

To create an executable file, the Linker must have the following inputs:

- the object file from a main Pascal program.
- object files for all external procedures referenced by the main program. These may be as Pascal units, assembly language routines, or intrinsic units defined in INTRINSIC.LIB.
- All units used by the units the main program uses.
- IOSPASLIB to provide the standard Pascal procedures and functions.

The Linker combines these files and creates an executable object file. If it is unable to link these files correctly to create a legitimate output file, the Linker will display an error message. If there is an error, the object file produced is not executable.

When linking a main program, all references to external objects must be resolved. Partial links are not allowed.

While it is linking the program, the Linker does a "dead code analysis" and does not include any routines that are not referenced. Unnecessary routines are eliminated from the main program, and from the units and libraries given as inputs to the link.

7.2 Using the Linker.

The Linker is started by pressing "L" in response to the Workshop command prompt. The Linker prompts you for the input files, the listing file and the output file. Options may be entered as a response to the input file prompt. After all file names and options are entered, the link begins. This means that the set of options in effect are the same throughout the link. It is not possible to change options part way through the link. When entering an input file name, it is not necessary to enter the .OBJ extension, the Linker will provide that for all inputs.

The Linker will accept option commands and input file names from a command file. A command file is a text file containing the file names and options, one per line. If there is a blank line in the file, the Linker treats this as the RETURN that signals the end of the input files. You use a command file by typing "<" followed by the name of the text file the commands are in. Create the text file by using the Editor.

The default listing file is the -CONSOLE. You may send the listing to a text file by entering its name in response to the listing file prompt.

After entering the ouput file name, the link begins. If no errors occur during the link and all external references are resolved, the output file is executable. A message is printed at the end of the link to tell you if the output is executable.

7.3 The Linker Options.

Linker options can be entered at any time in response to the prompt for an input file name. The order in which options are entered is unimportant, because they have no effect until the link begins. The last value entered for an option is the value used when the link is performed.

Options are represented by a single character. A "+" in front of the character makes that option take effect. A "-" sets the Linker so that option will not happen. In addition to being set on or off, some options have additional parameters. Numeric parameters can be in either decimal or hexadecimal. Hexadecimal numbers are indicated with a leading "$". The current setting of all options can be displayed by entering a "?" in response to the request for an input file.

The Linker options are as follows:

+A Alphabetical listing of symbols. The default is -A.

+D Debug information. The default is -D.

+H num +H sets the maximum amount of heap space the Operating System can give a program before terminating it. Here, as in the other options, 'num' can be either decimal or hexadecimal.

-H num -H sets the minimum amount of heap space needed by a program.

+I Copy interface information into intrinsic library files. The default is -I.

+L Location ordered listing of symbols. The default is -L. The location is the segment name plus offset.

+M fromName toName

+M maps all occurrences of the segment 'fromName' to the segment 'toName'. This allows you to map several small segments into a single larger segment. You can thereby postpone the segmentation decision until link time by using many segment names in the source code.

NOTE

Because options have an effect only when the link begins, it is not possible to map a segment name to several different names using this option.

+P Production link. The default is -P. +P produces a 'production' .OBJ file. A production object file does not contain information used by the debugger and the Linker, and intrinsic unit files do not contain a jump table. The production object file can be executed, but cannot be handled by the Linker or the debugger.

+S num +S sets the starting dynamic stacksize to 'num'. The default is 10000.

+T num +T sets the maximum allowed location of the top of the stack to 'num'. The default is 128K.

+ W + W tells the Linker to get intrinsic unit information from a file other than INTRINSIC.LIB.

? Prints the options available and their current values.

7.4 How do I Link a Main Program?

A main program consists of a Pascal program linked with all routines necessary for it to run. A main program is the only type of executable object file produced by the Linker. To link a main program you must have the following:

o A compiled pascal PROGRAM object file.

- Object files for all the units the program uses. This includes files for regular units and assembly language routines. Any intrinsic units used must be defined in INTRINSIC.LIB.
- IOSPASLIB.

When you have all the above files, proceed as follows:

1. Execute the Linker by pressing "L" when the Workshop command prompt is displayed. The Linker will display a header and ask you for an input file.
2. Enter any desired options. See section 7.3 in this chapter for more information. Press RETURN after each option entered.
3. Enter the file names for all the object files, pressing RETURN after each one. The file names can be entered in any order. Do not enter the .OBJ extension, the Linker automatically appends it.
4. Press RETURN to indicate the end of the input files.
5. The Linker prompts you for a listing file. Enter the file name desired, or press RETURN to accept the default of displaying the listing on the -CONSOLE.
6. The Linker prompts you for the output file. Enter the name of the executable file you want produced. Do not enter the .OBJ extension, that will be supplied automatically.

The linking process begins when you press RETURN after entering the output file name. If the link is successful, the message "Output is executable" will be displayed. If the link is not successful, error messages will be displayed.

7.5 Regular and Intrinsic Units.

The two types of units are regular units and intrinsic units. Both of them are separatly compiled code modules that may be used by a main program or another unit.

The syntax of a Pascal unit is explained in the Pascal Reference Manual for the Lisa.

A regular unit is combined with a main program by the Linker and included in the resulting object file. An intrinsic unit, on the other hand, is stored separately on the disk, and loaded at run time. Thus only one copy of an intrinsic unit is kept on the disk, no mater how many main programs use routines in it. In addition to being shared on the disk, an intrinsic unit is also shared in memory.

NOTE

In the current implementation, there is no provision for creating intrinsic units. Only intrinsic units supplied by Apple can be used.

7.5.1 How do I use a Regular Unit?

A regular unit is a separately compiled segment of code. It is written in Pascal, compiled, and code generated. See the Pascal Reference Manual for the Lisa for information on how to write a unit. See Chapter 5 in this manual for information on compiling the unit.

After you have created a unit, the routines in it may be accessed from any other program or regular unit you write. The Linker is used to combine a main program with all units it uses. The result is an executable object file containing all the needed routines.

To use regular units with a main program, follow the procedure in section 7.4. As input, you must give the Linker:

- The object file of the main program.
- The object files of all units used by the main program.
- The object files of all units used by other units.
- IOSPASLIB.

The Linker will combine all these object files into an executable object file. It will also do a "dead code analysis" to eliminate any routines that are not used, thus preventing the object file from becoming any larger than is necessary.

When regular units are used by more than one main program, a separate copy of each routine used is stored in each executable object file. This "waste" of disk space and memory can be prevented by using intrinsic units instead.

7.6 The Linker Listing.

A listing is produced each time a program is linked. This listing can be sent to a file, or displayed on the console (the default). The +A option will give you an alphabetical list of the symbols (procedure names) used in the link. The +L option gives you a list of the names in order of their location. The listing is produced in stages, as follows:

1. The input files are read, and a summary of the resources used is printed.
2. The linking process begins. Information about the size of each segment is printed.
3. Errors are reported, and you are told if the output is executable or not.

If you requested optional listings, they will also be printed. An example of a Linker listing with no options requested is shown in Figure 7-1.

linkerlisting

Figure 7-1. A Linker Listing.

7.7 Resolving External Names.

An external name is a symbolic entry point into an object module. All such names are visible at all times--there is no notion of the nesting level of an external name. External names can be either global or local. A local name begins with a $ followed by 1 to 7 digits. No other characters are allowed. A global name is any name which is not a local name.

The scope of a global name is the entire program being linked. Unsatisfied references to global names are allowed. Only one definition of a given global name may occur in a given link. (The one exception to this is that the Linker will accept duplicate names where one instance is in a main program or regular unit, and the other is in an intrinsic library file. In this case, a warning is issued, and the entry in the main program or regular unit is used.)

The scope of the local name is limited to the file in which it resides. When a link is done, global names are passed through to the output file unmodified, but local names are renamed so that no conflicts occur between local names defined in different files. All references to a given local name must occur within the same input file.

7.8 Module Inclusion.

There are two different cases of what modules the Linker includes in the output file. When linking an intrinsic unit, all code modules in the unit are included. When linking a main program with regular units, the Linker does a dead code analysis and does not include any modules that are not used.

7.9 Segmentation.

Segmenting a program makes it possible for portions of the program that are not being used to be swapped out to disk, thus making better use of memory. The way a program is segmented will have important effects on its performance.

Segmentation is controlled by two things:

- The $S Compiler command, that assigns segment names to source code modules.
- The +M Linker option, that allows you to remap compiler segment names into new segment names.

The usual strategy for segmenting a program is to use the $S compiler command to divide the code into many small segments, then to map these segments into a few larger physical segments with the +M Linker option. This will allow you to change the segmentation of the program by just relinking it. The segmentation can then easily be adjusted to produce the best swapping characteristics.

Assembly language routines are by default placed in the blank segment. You can use the .SEG directive to specify another segment, or change the segment with the ChangeSeg utility. See the Chapters 6 and 10 for more information.

7.10 Error Messages.

The Linker produces three different types of error messages, depending on the severity of the error it encountered.

The first, and least severe type of message, is called a warning. A warning message is given when the Linker detects a condition that is potentially dangerous, but not definitly an error. A warning message always begins with:

```
*** Warning
```

If the warning message occurs while entering a command or file name, you may simply reenter the command correctly, and the Linker will proceed as though nothing had happened.

The second type of message is called an error. An error means that the Linker has discovered a condition that makes it impossible to complete the link successfully. The link process is continued, so that any further errors can be discovered. An error message begins:

```
*** Error
```

A fatal error is a condition that makes it impossible for the Linker to continue the link. The link is terminated immediatly, and a message is displayed beginning:

```
*** Fatal Error
```

A complete list of all Linker messages is given in Appendix A.

Chapter 8
THE DEBUGGER

THE DEBUGGER

8.1 The Debugger.

The Debugger allows you to examine and modify memory, set breakpoints, assemble and disassemble instructions, and perform other functions for run-time debugging.

Procedure names are available to the debugger for program units compiled with the D option on. The debugger uses the symbolic names wherever appropriate.

The debugger's symbol table combines the user symbol table and the distributed procedure names. The user symbol table contains symbols the user defines while using the debugger and the predefined symbols for registers. Each entry contains twelve bytes. The first eight bytes are the symbol name, and the last four bytes are the symbol's value. Section 6.4 in this manual contains more information about the run-time environment of programs.

8.2 Using the Debugger.

Type D to the command prompt to invoke the debugger. It asks:

```
Debug what OS file?
```

Enter the name of the object file you want to debug. It will be Run with a breakpoint at the first instruction that will drop you into the debugger immediately. The debugger command prompt is '>'. The default radix is hexadecimal.

Another way of getting into the debugger is by pressing the NMI (non maskable interrupt) key which is the "-" key in the top row of the numeric keypad.

When you get the command prompt, the debugger is ready to accept commands that allow you to:

- Display and set memory locations
- Set and display registers
- Assemble and disassemble instructions
- Set breakpoints, patchpoints, and traces
- Manipulate the memory management hardware
- Set up timing buckets for execution timing
- Perform utility functions including:
  - symbol and base conversion
  - move the debugger window

8.2.1 Examples of Using the Debugger.

This section gives examples of how to use the debugger. An explanation of all debugger commands is given below in Section 8.3. A summary of all debugger commands is given in Section 8.4.

If you type a file name to the prompt from the Debug command, the debugger starts up with the program counter at the start of the program. To see one instruction disassembled (say at 32F96), type

```
>ID 32F96
```

ID stands for Immediate Disassemble. Each subsequent ID command, if given without any address, disassembles the next instruction found. In addition to printing the value of each byte, the debugger prints the ASCII equivalent of that value, if a printable one exists. If none exists, it prints a period.

To disassemble 20 consecutive addresses, type

```
>IL
```

IL (Immediate Disassemble Lines) can also be followed by an address. Subsequent IL commands disassemble successive blocks of 20 consecutive locations in memory.

If the object file being examined was compiled with the D+ compiler option, the procedure names are available in the debugger and can be used in any expressions. For example,

```
>IL Foo 5
```

disassembles the first 5 lines of procedure 'Foo'.

```
>BR Foo+40
```

sets a break point 40 bytes into procedure 'Foo'.

You can also use labels in immediate assemblies:

```
>sy Ken 6000

>A Ken NOP
```

assembles a NOP instruction at the address 'Ken', which in this case is 6000.

```
>A 6000

>Rich: JMP $100

> <RETURN>
```

enters the immediate assembler at 6000, defines the label 'Rich', and assembles a JMP instruction.

8.3 The Debugger Commands.

This section gives the definition of each debugger command. The commands are grouped together according to function.

8.3.1 Definitions.

Constant	A constant in the default base.
$Constant	A hex constant.
&Constant	A decimal constant.

'ASCII String' An ASCII string.

Name A symbol in the symbol table.

Expr An expression. Expressions can contain names, regnames, strings, and constants. Legal operators are + - * /. Expressions are evaluated left to right. * and / take precedence over + and -. (and) can be used to indicate indirection. < and > can be used to nest expressions. In those cases where an odd value is probably a mistake, the debugger warns you that you are trying to use an odd address. If you decide to go ahead, it subtracts one from the address given. If the compiler option D+ is used, procedure names are legal in expressions.

Exprlist A list of expressions separated by blanks.

Register The name for any of the 68000 registers, as follows: D0..D7 are the data registers, A0..A7 are the address registers, the program counter PC, the status registers SR, US, or SS. Note that A7 is SP (the stack pointer).

RegName RD0..RD7, RA0..RA7, PC, US, or SS. A predefined symbol in the symbol table with a value set by the debugger. The value is equal to the value of the register in question. The debugger automatically updates the values of these symbols. The 'R' is appended to distinguish the register names from hexadecimal numbers.

8.3.2 Display and set memory locations.

The following commands are used to display and set memory locations.

SM expr1 exprlist
Set memory with exprlist starting at expr1. SM assumes that each element of exprlist is 32 bits long. To load different length quantities, use SB or SW described below. If the expression given is longer than 32 bits, SM takes just the upper 32. For example, if we ask the debugger to:

```
SM 1000 'ABCDE'
```

it deposits the ASCII equivalent of 'ABCD' starting at 1000.

SB expr1 exprlist
Set memory in bytes with exprlist starting at expr1

SW expr1 exprlist
Set memory in words with exprlist starting at expr1

SL expr1 exprlist
Set memory in long words with exprlist starting at expr1. For example,

```
SL 100 1
```

is equivalent to

```
SM 100 0000 0001
```

DM expr
Display memory. Display 16 bytes of memory starting at expr. DM RA3+10, for example, displays the contents of memory from 10 bytes beyond the

address pointed to by A3. DM (110) displays the contents of the memory location addressed by the contents of location 110.

DM expr1 expr2
Display memory. If expr1 < expr2, then display memory from expr1 to expr2. Otherwise, display memory for expr2 bytes starting at expr1.

DB expr
Display memory as bytes.

DW expr
Display memory as words.

DL expr
Display memory as long words.

FB starting_addr count data
Find Byte. Find the byte or bytes 'data' in memory between 'starting_addr' and 'starting_Addr'+'count'.

FM starting_addr count data
Find Memory.

FW starting_addr count data
Find Word.

FL starting_addr count data
Find Long word.

8.3.3 Set and display registers.

TD
Display the Trace Display at the current PC. An example of the trace display is shown in Figure 8-1. It shows the instruction executing at the time the program was interrupted, the current value of all the registers, and the current domain and process.

tracedisplay

~ figure to come ~

Figure 8-1. The Trace Display.

register
Display the current value of the register. D0, for example, is a command to the debugger to display the current value in the register D0. RD0, on the other hand, is a name automatically placed in the symbol table to give you a handle on the contents of D0 in an expression. Thus, to display the current

value in the D0 data register, type the command D0. To display the instruction pointed to by the A0 address register, type the command ID RA0 (Immediate dissassemble at the address RA0, which is predefined to be the contents of the A0 register)

register expr
Set the register to expr. For example, to set register D3 to zero, type D3 0.

8.3.4 Assemble and disassemble instructions.

These commands are used to display code in assembly language format, and to enter code in the form of assembly language statements.

A expr statement
Assemble one or more assembly language statements (instructions) starting at expr. You can continue assembling instructions into consecutive locations, pressing RETURN after each statement. Type just RETURN to exit the immediate assembler. Note that the immediate assembler cannot assemble any intrinsic unit instructions, but they will be correctly disassembled. Code segments may be write-protected, which will prevent you from assembling instructions into them. This can be overridden with the WP 0 command to disable write protection.

A expr
If you use the form A expr, the debugger prompts you for the statement to be assembled.

ID
Disassemble one line at the next address

ID expr
Disassemble one line at expr

IL
Disassemble 20 lines at the next address

IL expr
Disassemble 20 lines starting at expr

IL expr1 expr2
Disassemble expr2 lines starting at expr1

IX statement
Immediate execution of a single instruction. The users PC is not changed by this operation.

8.3.5 Set breakpoints and traces.

These commands are used to trace program execution.

BR
Display the breakpoints currently set. You can set up to 16 breakpoints with the debugger. Break points are displayed both as addresses and as symbols. An asterisk marks the point of the breakpoint in the disassembly.

BR exprlist
Set each breakpoint in exprlist. Symbols are legal, of course, so we can:

```
BR Ralph+4
```

if Ralph is a known symbol.

Expressions can be of the form:

```
pp:aaaaa
```

where pp is the process number, and aaaaa is the address in that process where you want the breakpoint set. If the process number is 0, the breakpoint is set in system code in domain 0. If no process is given, the current process is assumed. The current process is shown in the TD display described above.

Breakpoints cannot be set on intrinsic unit instructions.

CL
Clear all breakpoints

CL exprlist
Clear each breakpoint in exprlist

G
Start running at the current PC

G expr
Starting running at expr

T
Trace one instruction at the current PC

T expr
Trace one instruction at expr

SC expr
Stack Crawl. Display the user call chain. Expr sets the depth of the display. It can be omitted.

RB
Reboot. This command should not be used while you are in the Workshop. The Lisa is reset.

procedure name
This calls a user procedure or function. It is the users responsibility to save and restore registers and push any necessary parameters. If you want execution to stop upon return, you must set a breakpoint on the current PC. For example:

```
BR PC                      ; set break point on PC.
IX MOVEM.L  D0-A6,-(A7)    ; save registers.
                           ; push params if needed.
FOO                        ; call procedure  FOO.
IX MOVEM.L  (A7)+,D0-A6    ; restore registers.
CL PC                      ; remove break point.
```

A function can be called in a similar manner. Remember to allocate space for the function result before pushing any parameters. Use either CLR.W -(A7) ro CLR.L -(A7).

A procedure that may need to be called is OSQUIT. It exits from the OS. We reccomended that you avoid this whenever possible.

8.3.6 Manipulate the Memory Management Hardware.

These commands change the memory management hardware of the Lisa. More information on the memory managment hardware can be found in the Lisa **hardware manual. CHECK NAME.**

LP expr
Convert logical address to physical address.

DO expr
Set the SEG1/SEG2 bits. These bits determine the hardware domain number. If the Status Register shows that you are in supervisor state, then the effective domain is zero, and the domain number returned by the debugger is the domain that would be active if the SR were changed to user state.

WP 0 or 1
Diable (0) or Enable (1) Write Protection. The default is 1.

MM start [end_or_count]
MM with one or two arguments displays information about the MMU registers. The second argument defaults to 1. If the starting address is greater than the second argument, the second argument is a count of the number of MMU registers to be displayed. If the starting address is less than the second argument, the second argument is the last register displayed.

```
MM 70
```

displays

```
Segment[70] Origin[000] Limit[00] Control[C]
```

These values are the Segment Origin, Limit, and Control bits stored by the hardware for each MMU register. As can be seen from a careful perusal of the hardware documentation, a Control value of C means the segment in question is unused (invalid). If the Control value is valid (7, for example), the debugger also displays the Physical Start and Stop addresses of the segment.

```
MM &100 8
```

displays the MMU register information for the 8 registers starting at register 64 (decimal 100).

MM num org lim cntrl [end_or_count]
The MM command followed by four arguments sets the MMU information for segment 'num'. The Origin, Limit, and control bits can be changed.

```
MM 70 100 ff 7
```

sets the Origin of segment 70 to 100 and the control bits to 7 (a regular segment). The segment limit of -1 makes the segment 512 bytes long.

8.3.7 Timing Functions.

The debugger allows you to create up to 10 timing buckets for measuring execution times. Using the microsecond timer in Drivers, time is accumulated in each bucket and saved along with a count of the number of times the bucket was entered.

Typically, this would be done as follows:

1. Enter the debugger for a given process and create one or more timing buckets with the TB command.
2. Set a break point to stop execution at some point.
3. Go.
4. When the breakpoint is reached, print the timing summary with the PT command.
5. Use the End Timing (ET) command to remove all timing buckets.

The timing commands are as follows:

BT expr
Begin timing. Expr specifies the process number. If the BT command is not given, the current process is assumed. A process number of 0 can be used to indicate domain 0.

TB addr1 addr2
A timing bucket is created from addr1 to addr2.

PT
Print timing summary. There are five columns printed:

1. Bucket number
2. Total time in this bucket.
3. Number of times this bucket was entered.
4. Starting address for this bucket.
5. Ending address for this bucket.

ET
End timing. This command prints the timing summary and removes all the timing buckets.

KB expr
Kill Bucket. This can be used to remove a single bucket. Expr is the number of the bucket to remove.

RT
Reset timers. This resets the timing and count tables while leaving the bucket definitions intact.

Note that all addresses are in the same process. The process number is defined by either the BT command or the first TB, PT, KB, or RT command. If the process number is not given in the BT command the current process is assumed.

8.3.8 Utility functions.

including:

o symbol and base conversion

- o moving the debugger window
- o Setting the NMI Key

8.3.8.1 Symbols and Base Conversion

SY
Display the values of all symbols

SY name
Display the value of the symbol name

SY name expr
Assign expr to the symbol name

CV exprlist
Display the value of each expression in hex and decimal.

SH
Set the default radix to hex

SD
Set the default radix to decimal

8.3.8.2 Moving the Debugger Window:

P expr
Set port number to expr. Valid port numbers are:

0	Lisa Keyboard and screen (default)
1	UART Port A (farthest from Power Supply)
2	UART Port B

If you move the port to a UART, you must have a modem eliminator connected to that port.

RS
Display the patch Return address Stack

8.3.8.3 Setting the NMI Key:

NM
Displays the Key code for the NMI Key.

NM expr
Sets the NMI Key to be Key code expr. A value of zero disables the NMI Key.

For example:

```
>NM $21
```

Sets the NMI Key to be hex 21, which is the "-" Key in the top row of the numeric Keypad.

8.4 Summary of the Debugger Commands.

procedure name	Call the procedure.
register	Display the current value of the register.
register expr	Set the register to expr
A expr statement	

A expr	Assemble one statement (instruction) at expr.
BR	Display the breakpoints currently set.
BR exprlist	Set each breakpoint in exprlist.
BT expr	Begin timing process expr
CL	Clear all breakpoints
CL exprlist	Clear each breakpoint in exprlist
CV exprlist	Display the value of each expression in hex and decimal.
DB expr	Display memory as bytes.
DL expr	Display memory as long words.
DM expr1 expr2	Display memory.
DO expr	Set the SEG1/SEG2 bits.
DR	Display index or ranges of dump RAM.
DW expr	Display memory as words.
ET	End Timing - print summary and remove buckets
FB starting_addr count data	Find Byte.
FL starting_addr count data	Find Long
FM starting_addr count data	Find Memory
FW starting_addr count data	Find Word
G	Start running at the current PC
G expr	Starting running at expr
ID	Disassemble one line at the next address
ID expr	Disassemble one line at expr
IL	Disassemble 20 lines at the next address
IL expr	Disassemble 20 lines starting at expr
IL expr1 expr2	Disassemble expr2 lines starting at expr1
IX statement	Immediate execution of one instruction
KB expr	Kill Bucket expr
LP expr	Convert logical address to physical address.
MM expr1 expr2	Display MMU information
MM num org lim ctrl	Set MMU information
MR	Set a value level #5 interrupt on a word change.
NM	Displays the keycode of the NMI key
NM expr	Sets NMI keycode to expr
P expr	Set port number to expr.
PT	Print timing summary
RB	Reboot.
RS	Display the patch Return address Stack
RT	Reset timers
SB expr1 exprlist	Set memory in bytes with exprlist starting at expr1
SC expr	Stack Crawl.
SD	Set the default radix to decimal
SH	Set the default radix to hex
SL expr1 exprlist	Set memory in long words with exprlist starting at expr1.

SM expr1 exprlist	Set memory with exprlist starting at expr1.
SW expr1 exprlist	Set memory in words with exprlist starting at expr1
SY	Display the values of all symbols
SY name	Display the value of the symbol name
SY name expr	Assign expr to the symbol name
T	Trace one instruction at the current PC
T expr	Trace one instruction at expr
TB addr1 addr2	Create Timing Bucket from addr1 to addr2
TD	Display the Trace Display at the current PC
WP 0 or 1	Diable (0) or Enable (1) Write Protection.

Chapter 9
USING EXEC FILES

Using Exec Files

9.1 Exec Files

Exec files are scenarios of commands to the workshop system. They are contained in a text file, created with the Editor, and are executed with the Run command. They consist of the actual characters you would type to the Workshop to perform the function you want, interspersed with special exec file commands that allow you to use parameters and conditions to vary some portions of the scenario.

In its simplest form, an exec file contains the characters you would press to perform the desired operation. For example, to compile a Pascal program, the exec file would contain:

```
Pmyprog
```

The P invokes the Pascal compiler, myprog is the name of the source file. This could be followed by further lines to Generate, Link, and Run the program.

Special exec file commands allow you to use parameters and conditionally perform the Workshop commands. This would allow you to set up an exec file to compile, Generate, and optionally Link any Pascal program. Such an exec file is shown in Figure 9-1

```
$EXEC
$ { This exec file compiles and Generates a Pascal program. }
$ { If the second parameter is L (or l) the program is Linked }
$IF %0 = '' THEN {no parameter entered}
 $WRITE 'Compile what file?'
 $READLN %0
$ENDIF
P%0
{ no listing file}
{ default I-code file }
G%0
{default object file}
$IF UPPERCASE(%1) = 'L' THEN
 L %0
 IOSPASLIB
 { end of linker input }
 { no list file }
 %0{ output file name }
$ENDIF
$ENDEXEC
```

Figure 9-1. Example Exec File

9.2 Exec File Statements

Exec file statements are contained on one line. There are two types of exec file lines, exec command lines, and normal lines. Normal lines contain commands to be processed by the Workshop system. Exec command lines handle the other features of exec files, such as parameters and conditional statements.

You may use up to 10 parameters in an exec file, numbered as %0 through %9. These receive their values from the invocation of the exec file, or they are assigned values during the exec file execution. When a parameter appears in a normal line, it is replaced by the string value of that parameter. These parameters can be used both as inputs to the exec file and as temporary variables within it.

Exec command lines start with a $; they control the operation of the rest of the exec file. Exec command lines are free format, as long as the order of thier elements is preserved. Any number of blanks can occur before any element of a command line.

Normal command lines contain commands for the Workshop system. These lines are sent to the Workshop exactly as they appear. Any extra blanks will be sent to the Workshop and will be treated exactly as if you had typed in those blanks.

Comments are delimited by curly braces ({and }). They can appear in either a normal or an exec command line. Comments are completely removed from normal lines.

The tilde (~) is used as a literalizing character in normal lines. It passes the following character through without processing it. This allows you to pass $, %, and { to the Workshop system without having them be interpreted as an exec command, a parameter, or a comment. Tilde can be passed as ~~

The following is a description of each exec command line type.

9.2.1 Beginning and Ending Exec Files

The general form of exec files is they must begin with a "**$EXEC**" line and must end with a "**$ENDEXEC**" line. The exceptions to this basic rule (for those miscreants who embed their exec files in their program sources) are: (1) one line of text may preceed the "$EXEC" line if the "I" invocation option is used, and (2) any amount of text may follow the "$ENDEXEC" line, but it will be ignored.

9.2.2 Setting Parameter Values

You can set parameter values in an exec file by using the SET and DEFAULT operations. The REQUEST operation prompts the user for the value of a parameter.

SET and DEFAULT

The SET and DEFAULT commands provide a way of changing the value of a parameter inside of an exec file. The form of these commands is:

```
  $ SET <%n> TO <strexpr>
```

and

```
  $ DEFAULT <%n> TO <strexpr>
```

where <%n> is a parameter reference and <strexpr> is a string expressionas described in the following section.

The effect of the SET command is to change the value of the specified parameter to the value of the given string expression. The effect of the DEFAULT command is similar to that of the SET command, however, the assignment only takes place if the value of the specified parameter is the null string when the DEFAULT command is encountered. Thus, this command can be used to supply default values to parameters that have been leftunspecified or empty in the exec invocation line.

REQUEST
The REQUEST command provides a way to prompt for values from the console. Itsform is:

```
$ REQUEST <%n> WITH <strexpr>
```

The REQUEST command will print the given string expression to the console and will read a line from the console which it will assign to the specified parameter. Thus the <str expr> is the prompt that you will requestwith.

9.2.3 Input and Output

Input to an exec file isrequestedby the READLN of READCH command. The WRITE and WRITELN commands allow you to outputvalues.

READLN and READCH
The READLN and READCH commands allow exec filesto read in text from the console and to assignit to a parameter variable. This mechanism may be used to obtain parameter values, to obtain values to control conditional selection, to pause until the user indicates to continue, or for any other purpose. The form of these commands is:

```
$ READLN <%n>
```

and

```
$ READCH <%n>
```

READLN will read a line from the console and will assignit to the specified parameter. READCH will read a single character from the console (if <return> is typed that character will be a blank).

WRITE and WRITELN
The WRITE and WRITELN commands allow exec files to write text to the console screen. This text may be used forinformatory messages, prompts, or for any other purpose. The form of these commands is:

```
$ WRITE [ <strexpr> [ , <strexpr> ]* ]
```

and

```
$ WRITELN [ <strexpr> [ , <strexpr> ]* ]
```

That is, these commands take an arbitrary number of string expressions, separated by commas, as arguments. The strings are written to the current console line, and in the case of WRITELN a final carriage return is written.

9.2.4 Conditional statements

Conditional statements allow you to perform certain commands depending on the conditions at the time the exec file is run. These conditions can be based on the value of parameters and on the files available to process. The condition is stated in the form of a boolean expression, and can include builtin boolean functions to determine the condition of files.

The IF Statement

The IF, ELSEIF, ELSE and ENDIF commands allow conditional selection in exec files. The syntax of these commands is as follows:

```
 $ IF <bool expr> THEN
   <stuff>
[$ ELSEIF <bool expr> THEN
   <stuff> ]*
[$ ELSE
   <stuff> ]
 $ ENDIF
```

where <bool expr> is a boolean expression as described in the following section and <stuff> is made up of arbitrary normal and command lines (other than commands that would be a part of the current IF construct). The "[...]*" construct above indicates that zero or more ELSEIF commands may appear between the IF and the ENDIF command, while the "[...]" indicates that zero or one ELSE command may appear just before the ENDIF.

The IF construct is evaluated in the usual way. First, the boolean expression on the IF command itself is evaluated; if it is true then the <stuff> between the IF and the next ELSEIF (if any) or ELSE (if any) or ENDIF is selected; otherwise it is not selected. All remaining parts of the IF construct up to the ENDIF will be parsed but will not be selected once one of the <bool expr>s is true and its corresponding <stuff> is selected. To say that <stuff> is selected means that any normal lines will generate text and that any command lines will be processed. Conversely, to say that <stuff> is not selected means that any normal lines will not generate text and that command lines will be parsed (for correctness) but not executed. If the <bool expr> on the IF is not true then the following ELSEIF or ELSE will be processed. If an ELSEIF is next, its <bool expr> will be evaluated, and, if true, its following <stuff> will be selected and the remainder of the IF construct will not be selected. Processing of the IF construct continues until one of the <bool expr>s on an IF or ELSEIF is true or until the ENDIF is reached. If no <bool expr> is true before the ELSE (if any) is reached, its <stuff> will be selected.

IF constructs may be nested within each other to an arbitrary level.

Boolean Expressions--comparison and logical operators

Booleans expressions(<bool expr>s) enable you to test of string values and check properties of files. The grammar for boolean expressions is as follows:

```
<bool expr>         ::=   <bool term> [<binary logic op> <bool expr> ]*

<binary logic op>   ::=   AND
                          | OR

<bool term>         ::=   <bool factor>
                          | ( <bool expr> )
                          | NOT ( <bool expr> )

<bool factor>       ::=   <strexpr> <strop> <strexpr>
                          | <bool function>

<strop>             ::=   =
                          | <>
```

The basic element of a boolean expression (a <bool factor>) is either a boolean function (see the next section) or a string comparison, testing for equality or inequality. These basic elements may be combined using the logical operators AND, OR and NOT, with parentheses used for grouping. All these operators function in the usual way.

Boolean Functions -- EXISTS and NEWER

Several functions returning boolean results are provided for use with the conditional contructs.

The **EXISTS** function allows you to determine whether a file or volume exists. The function has the following form:

```
EXISTS ( <strexpr> )
```

where <str expr> is a string expression whose value is the name of a file. Typically this <str expr> will be an expanded string constant (discussed above), such as "%1.obj".

The **NEWER** function allows you to determine whether one file is newer than another file, that is, whether its last-modified date is more recent than the last-modified date of anther file. The function has the following syntax:

```
NEWER ( <strexpr 1>,<strexpr 2> )
```

where the <str expr>s specify file names. TRUE will be returned if the first file is newer than the second. A preprocessor run-time error will occur if one of the files does not exist.

9.2.5 String Expressions

A string expression (<strexpr>) may specify a string by a number of means, as noted in the following grammar.

```
<strexpr>        ::=   <parameter reference>
                       | <strconstant>
                       | <expanded strconstant>
                       | <strfunction>
                       | <exec function call>
```

A **parameter reference** has the usual "%n" form. A **string constant** has the standard form of text delimited by single quotes ('), with a quote inside the string specified by the double quote rule, as in 'That''s all, folks!'. An **expanded string constant** is similar to a string constant, except that double quotes (") are used as delimiters and parameter references are expanded within the string. A **string function** is a preprocessor function which returns a string value (these are described in the following section). An **exec function call** is an invocation of an exec file which returns a string value (as described in a following section, "Exec Function Calls").

String Functions -- CONCAT and UPPERCASE
The string functions CONCAT and UPPERCASE may be applied to other string expressions to produce new string values.

The **CONCAT** function allows several string expressions to be combined to produce a result which is a single string. The CONCAT function has the form:

```
CONCAT ( <strexpr> [ , <strexpr> ]* )
```

That is, CONCAT takes a list of string expressions, separated by commas.

The **UPPERCASE** function converts any lower case letters in its argument to upper case. It has the following form:

```
UPPERCASE ( <strexpr> )
```

An example of the use of this function is

```
$ SET %0 TO UPPERCASE (%0)
```

which will set parameter 0 to an uppercase version of its previous value.

9.2.6 Nesting exec Files

Exec files can be nested by calling another exec file by using the SUBMIT command. The called file can be a function, which means that it will RETURN a value to a parameter in the calling exec file.

SUBMIT
The SUBMIT command allows nesting of exec files, that is, it allows another exec file to be called from within an exec file. The form of the SUBMIT command is:

```
$ SUBMIT <exec command>
```

where <exec command> is an exec command of the same form as you would have following the "exec/" or "<" at the WorKShop shell command level. This exec command may include parameters and exec options in the usual fashion.

The effect of the SUBMIT command is to process the specified exec file, putting any generated exec output text into the current exec temporary file. Thus, while a single exec file may have several nested sub-exec files, only a single temporary output file is generated which includes the output generated by all of the input files. Exec files may be nested to an arbitrary level.

Within the text of the <exec command>, references to "%n" parameters will be expanded and the literalizing character ("~") will be processed. Be aware that this is the only processing that taKes place within an exec command. Everything up to the first "(" or the end of the line (if no parameter list is present) will be taKen to be the exec file name. If there is a "(" the parameter list will be taKen to be everything between this "(" and the next ")". An <exec command> may not be split across lines.

Note that only the "I" (Ignore first line) and "B" (BlanKs significant) options are valid on a SUBMIT command, while the "R" (ReRun), "S" (Step mode) and "T" (Temporary file saved) options are only applicable from the main exec invocation line.

$RETURN -- Exec Functions

The RETURN command allows exec files to return string values to other (calling) exec files. Thus the RETURN command can turn an exec file into a **function.** The form of the RETURN command is:

```
  $ RETURN  [<strexpr> ]
```

Executing a RETURN command will terminate the current exec file and return to the calling exec file with the specified string value. The method by which exec functions are called is described in the following section.

Exec functions can be used to do such things as determining whether a program file (and its corresponding include files, if any) have been modified since their last compilation, and may thus be used to conditionally submit compiles. If written generally enough, such a function could be used by many exec files.

Exec functions can produce side effects, that is, they may contain normal lines which will get placed in the temporary file. While the intentional use of such side effects is unliKely, inadvertent instances may occur and will be potentially hazardous to your exec files. (An unexpected blanK line in the middle of an exec file can often throw it out of sync.)

Exec Function Calls

Exec function calls return string values, and are thus are one of the basic elements of string expressions. They may also appear in boolean expressions, supplying arguments for string comparisons.. (A typical use of an exec function would be to return a boolean value by returning either the string 'T' or 'F'.) The form of an exec **function call** is:

```
  <  <file name>  [( <arg list> ) ]
```

where "<" is the character that signals a function invocation (just in the way that this character identifies exec files for the WorkShop's Run command). The <file name> and optional <arg list> are the same as in the SUBMIT command.

Due to our liberal conventions concerning what characters (including blanks) may appear in file names, the preprocessor must make some assumptions about how to identify the exec function file name and the argument list. Recognizing the file name is more of a problem in the case of exec functions than it for the SUBMIT command, since exec function calls may appear inside of arbitrary string expressions, while an exec invocation appears by itself in a SUBMIT command. The simple rule the preprocessor uses is: if the exec function invocation has an argument list, the file name is assumed to be everything between the "<" and the "(" beginning the argument list; otherwise, the file name is assumed to be everything between the "<" and the end of the line, which means that you will have to supply an empty argument list to an exec funtion with no arguments if the function call is not the last thing on the command line.

The processing of the text of a function call is the same as that of a SUBMIT command, that is, the only processing that will take place is expansion of "%n" parameters and recognition of the literalizing character ("~"). This means, for instance, that the text of a function call may not contain an embedded function call. Note also that a function call may not be split across lines.

9.3 Using Exec Files

An invocation line for the preprocessor has the following form:

```
  <exec command>  <exec file> [(<parameter list>) [<exec options> ]]
```

The <exec command> can be either "EXEC/" or "<". The <exec file> is the name of the exec file you wish to run. A ".TEXT" extension will be assumed if one is not specified; however, you may override the mechanism which supplies the ".TEXT" extension by ending your <exec file> name with a dot; e.g., using "foo." will cause the preprocessor to look for the file "foo" rather than "foo.text".

The optional <parameter list> is enclosed in parentheses. The parameter list may be empty or it may include up to ten parameters delimited by commas. For example, we may have an exec file to run compiles which takes volume and source file parameters, which we might invoke with "compile(foo,-work-)". Parameters may be omitted (leaving them as null paramters) by specifying them with the null string, as in "compile(foo,)", which omits the volume from our previous example. Alternately, parameters may be left unspecified altogether, as in "compile(foo)", in which case they also get null values. One reason for leaving off parameters is that the exec file may have been set up to supply default values, as is described below.

The <exec options> which follow the closing ")" of the parameter list consist of single letter commands which will modify the behavior of the preprocessor; for example, "S" is used to indicate that you want to step through the exec file as it is being processed, conditionally selecting which commands will be sent to the WorkShop shell. The exec options are discussed in detail in the "Exec Invocation Options" section below.

The preprocessor's output is a temporary file with a "..TEXT" extension. The temporary file is the processed version of your exec commands, that is, all preprocessor-oriented commands will have been processed and removed, leaving only the WorkShop-related commands. This temporary file is passed to the WorkShop shell executive when the preprocessor is done. The WorkShop shell will then run the temporary exec file and delete it automatically when completed.

Note that the preprocessor is not case-sensitive, but it does preserve the case of parameters and strings supplied by the user.

Exec Invocation Options

A number of options are available when running the preprocessor. These options may be specified when invoking the preprocessor or on SUBMIT commands. The options are specified by single letter commands following the exec parameter list. (A null parameter list should be used if you want to use options without parameters, as in "<foo()s".) The options are as follows:

"B" indicates that the preprocessor should not trim blanks on output lines. Normally the preprocessor will trim off leading and trailing blanks on the lines that it outputs to the temporary file. This allows you to indent normal lines (lines which are not exec command lines) without worrying about generating spurious blanks. Thus the preprocessor assumes that leading and trailing blanks are insignificant (which is the case for WorkShop commands, but which may not be true for some perverse programs you may run via exec files). This option will tell the preprocessor not to trim such blanks. The option applies only to the exec file being run or SUBMITted, and not to any nested exec files.

"I" indicates that the first line of the exec file is to be ignored by the preprocessor. This option is intended for deviants who like to embed their exec files in their program sources, in which case the first line of the source should be a "(*" and a "*)" should follow the end of the exec file, thus commenting it out of the program source. (Note that "(*" and "*)" should be used in preference to "{" and "}" since the latter are used as comment characters in the preprocessor.)

"T" indicates that the temporary file which is created (i.e., the expanded form of the exec file) should not be removed after it is run. One reason to use this option is to make it possible to rerun an exec file created with the step option (see below) without going through the stepping prompts a second time by running a previously created expanded exec file. The "R" exec option (described below) is used to run old temporary exec files. Note that the "T" option is not allowed on SUBMIT commands.

"R" indicates that the a exec temporary file which has been saved with the "T" option should be rerun, bypassing the normal processing by which the temporary was created. For example, "foo" may be an exec file which generates a complicated system via a large number of nested exec files which take a significant amount of time for the preprocessor to digest. If we know we are going to run "foo" repeatedly, we may want to generate the temporary file only once but run it several times. The first time we would invoke the preprocessor with "<foo()t" to indicate that the temporary file should not be automatically deleted after it is run. Subsequently, we would invoke the preprocessor with "<foo()r" to rerun the old temporary file. Note that the "R" option will override any others that may be specified, and it is not allowed on SUBMIT commands.

"S" indicates that the exec file should be processed in "Step Mode" which allows selective skipping of output lines and SUBMITs. If this option is used, the following message will appear when you invoke the preprocessor:

```
Step Mode:
-- in response to "Include ?" answer: Y, N, A (Abort), K (Keep rest), or I (Ignore Rest).
-- in response to "Submit ?" answer: Y, N, S (Step), A (Abort), K (Keep Rest), or I
     (Ignore Rest).
More details ? [No]
```

If you repond with "Y" (yes) to the "More details ?" prompt you will get further information on what each of stepping responses means.

When you invoke an exec file with the step option you will be prompted when a line has been generated and is about to go into the temporary file. The line will be displayed followed by "<= Include ?". A response of "Y" will include the line in the expanded exec file. A response of "N" will cause the displayed line to be omitted. A response of "A" will abort out of the exec file preprocessor and no exec file will be run. A response of "K" will keep (include) all the remaining lines of the exec file, leaving step mode, while a response of "I" will ignore the remainder of the exec file.

When a SUBMIT command is encountered when stepping, the SUBMIT line will be displayed followed by "<= Submit ?". A response of "Y" will perform the SUBMIT unconditionally, that is, without stepping through it. A response of "N" will ignore the SUBMIT. A response of "S" will step through the SUBMIT file. A response of "A" will abort out of the exec file preprocessor and no exec file will be run. A response of "K" will keep the rest of the exec file, leaving step mode, while a response of "I" will ignore the remainder of the exec file.

Note that a reponse of "?" to a "Submit ?" or "Include ?" prompt will elicit an explanation of the accepted responses.

Following are some examples of how to use the preprocessor's stepping facility.

Stepping may be used to resume execution of an exec file which did not run to termination. For example, if our example "compile" exec file includes both a compile and a generate step and if we wish to resume with the generate step we could invoke the preprocessor with "compile(foo,-work-)s". Then, in response to the "Include?" prompts for lines corresponding to the compile step we would hit "N" to skip the lines. Upon reaching the firstline of the generate step we would respond with "K" to keep the rest of the file, and the generate step of the exec process would be performed.

The stepping mechanism may be used to run only selected parts of an exec file. Say, for instance, that we have a modular set of exec files which generate a whole system of programs, such as the WorkShop development system, and that one exec file called "make/all" can generate the whole system by SUBMITting exec files for each of the component programs. The exec files for each component program (development system tool) make use of other exec files to perform such standard activities as compiling (and generating) a Pascal unit or program, performing an assembly, installing a library, or manipulating files with the WorkShop's filer. If we are performing a system build and find ourselves constantly having to regenerate parts of the system due to bugs, late deliveries or whatever, then the ability to step by SUBMITs proves to be very useful. Arbitrary parts of the system can be regenerated by running "<make/all()s" (i.e., our master exec file invoked with the stepping option) and selectively submitting the sub-exec files for only those things which we wish to rebuild while stepping over the others.

Stepping in conjuction with the "T" option (for saving the temporary file created by the preprocessor) can be useful when we are going to be regenerating a single component of a program or system a number of times in succession, such as when we are fixing a bug in an element of a system build and we expect that several iterations will be needed to correct the problem. To continue our previous example, suppose that we are having a problem with the "FileIO" unit of the "ObjIOLib" library while building the development system, and that an exec file called "make/ObjIOLib" generates and installs the library, submitting compiles and assemblies for all of its units, linking everything together, and finally performing the installation. By invoking the preprocessor with "make/ObjIOLib()st" we can go into step mode and submit only those things related to the compilation of the "FileIO" unit, the link, and the installation of the library in the Intrinsic Library. Then, after each successive refinement of "FileIO", we could run the saved temporary file by running "<make/ObjIOLib()r" without having to go thru the stepping process. Our alternatives to this procedure

are creating another exec file to generate only the selected parts, or running (and rerunning) the exec file for the whole library, or running each sub-process independently (which requires more of your attention).

Note that typing Apple-period while the preprocessor is running will abort the processing of the exec file.

9.4 Example Exec Files

Example 1 -- an exec file to do a Pascal compile

This exec file does a Pascal compile and generate. Note how comments have been used to make the single-character WorkShop commands more intelligible.

```
$EXEC { "comp" -- perform a Pascal compile
       %0 -- the name of the unit to compile }
 P{Pascal compile}%0{source}
 {no list file}
 {default i-code file}
 G{generate code}%0
 {default obj file}
$ENDEXEC
```

Example 2 -- an exec file to do an assembly

This exec file performs an assembly, and allows for an optional output file name which may be different from the source name.

```
$EXEC { "assemb" -- perform an assembly
       %0 -- the name of the unit to assemble }
       %1 -- (optional) alternate name of OBJ output }
 $DEFAULT %1 TO %0 { use source name if no output name is given }
 A{assemble}%0{source}
 {no list file}
 %1{obj file}
$ENDEXEC
```

Example 3 -- a more flexible exec file to do Pascal compiles

This exec file performs compiles; it allows for an output file with a different name than the souce and permits the use of an alternate intrinsic library.

```
$EXEC { "comp1" -- perform a Pascal compile
       %0 -- the name of the unit to compile
       %1 -- (optional) alternate name for OBJ file
       %2 -- (optional) alternate intrinsic library}
 $DEFAULT %1 TO %0 { if no alternate OBJ name use same name as source }
 $IF %2 <> '' THEN { use alternate intrinsic library }
   P{Pascal compile}?{option flag}
   %2{alternate intrinsic lib}
   %0{source}
 $ELSE
   P{Pascal compile}%0{source}
 $ENDIF
```

```
  {no list file}
  {default i-code file}
  G{generate code}%0
  %1{OBJ file}
 $ENDEXEC
```

Example 4 -- yet another exec file to do Pascal compiles
This compile exec file will only perform the compile if either the object file does not exist or the source file is newer than the object file (i.e., the source has changed since it was last compiled).

```
$EXEC { "comp2" -- perform a Pascal compile (only if really
      required)
       %0 -- the name of the unit to compile
       %1 -- (optional) alternate name for OBJ file
       %2 -- (optional) alternate intrinsic library}
 $DEFAULT %9 TO %1 { set %9 to name of output OBJ file }
 $DEFAULT %9 TO %0
 $IF EXISTS ("%9.obj") THEN
    $IF NEWER ("%0.text","%9.obj") THEN {recomp if source newer
      than object}
    $SUBMIT comp1(%0,%1,%2)
  $ENDIF
 $ELSE { OBJ file does not exist, so generate it }
  $SUBMIT comp1(%0,%1,%2)
 $ENDIF
$ENDEXEC
```

It is left as an exercise as to how to change the above example to take into account the fact that a unit may have an arbitrary number of include files in addition to its main source file, and that the unit will have to be recompiled if one or more of these change.

Example 5 -- exec file "chaining"
This example ("make/Prog") uses the "smart" compile exec file ("comp2") defined in the last example to demonstrate how to "chain" exec file execution. Assume we want to generate a particular program made up of three units (unit1..unit3) and that we have written "link/Prog", a smart exec file which performs a link only when one of the object files for one of the units is newer than the linked program file. Our generation exec file will use these smart exec files to perform the minimal required amount of work, thus it may be used to determine whether we have the latest version of the program without fear of wasting time.

```
$EXEC { "make/Prog"  -- smart version, only recompiles & links when
                                   it has to}
  $SUBMIT comp2(unit1)
  $SUBMIT comp2(unit2)
  $SUBMIT comp2(unit3)
   R<link/Prog                   { run link exec file after compiles have
                                   run so that it will get the correct
                                   file dates }
$ENDEXEC
```

Note that in the lastline of the above exec file we have scheduled an exec file to be run at a later time , as opposed to SUBMITting it now, so that the file dates for the link step will be accessed after the compiles have had a chance to run. The differences between running and submitting and exec files are demonstrated in the following scenario. When an exec file is submitted it is processed immediately by the preprocessor, with its output going to the temporary file, which is then passed back to the WorkShop shell. The then shell runs the commands in the temporary file until it comes to the command to run another exec file, at which point it discards the remainder of the temporary file and runs the preprocessor with the new exec command. This exec file invocation in turn results in another temporary file of commands which is then run by the shell.

Example 6 -- a recursive exec file to do Pascal compiles
This compile exec file will perform up to 10 compiles. It takes an argument list with the names of the units to be compiled .

```
$EXEC { "rcomp" -- perform any number (up to 10) Pascal compiles.
         It calls "comp" on its first argument and then calls itself
         recursively with its arguments shifted left }
 $IF %0 <> '' THEN
   $SUBMIT comp(%0)                              { "comp" the first one }
    $SUBMIT rcomp(%1,%2,%3,%4,%5,%6,%7,%8,%9) { "rcomp" the rest, less

           first }
 $ENDIF
$ENDEXEC
```

Example 7 -- a Basic example
This exec file demonstrates some of the constructs in the preprocessor's meta-language, by generating the BASIC interpreter. The comments in the body of the example should be sufficient to describe what is taking place. The essential idea is that Basic is made out of three components, and that we may want to generate only one or more of them at a time.

```
$EXEC { "make/basic" -- generate the BASIC interpreter.
   There are three parameters -- if a parameter is a "Y" (yes)
   the corresponding part of the system should be generated:
     (0) the b-code interpreter
     (1) the run-time system
     (2) the command interpreter
     If no parameters are specified, the exec file will prompt to see
        what parts of the system should be generated. }
$WRITELN 'Starting generation of the BASIC system'
$IF %0 = '' AND %1 = '' AND %2 = '' THEN {no params supplied -- prompt
        for info}
   $WRITE 'do you want to assemble the b-code interpreter? (y or
        [n])'
  $READCH %0
  $WRITELN { this writeln puts us on a new line for the next prompt }
  $WRITE 'do you want to compile the run-time system? (y or [n])'
  $READCH %1
  $WRITELN
   $WRITE 'do you want to compile the command interpreter? (y or
        [n])'
  $READCH %2
  $WRITELN
$ENDIF
$
$IF UPPERCASE(%0) = 'Y' THEN  {assemble the b-code interpreter }
  $SUBMIT assemb (int.main)
$ENDIF
$
$IF  UPPERCASE(%1) = 'Y' THEN  { compile the run-time unit }
  $SUBMIT comp(b.rtunit)
$ENDIF
$
$IF  UPPERCASE(%2) = 'Y' OR  UPPERCASE(%1) = 'Y' THEN
  ${ compile the command interpreter }
  ${ compile also if the run-time unit has changed }
  $SUBMIT comp(b.basic)
$ENDIF
$
${ link it all together }
  L{link}-p{note that "-p"  gets around a linker bug}
  b.basic
  b.rtunit
  int.main
  hwintl
  iosfplib
  iospaslib


  basic{executable output}
$ENDEXEC
```

Example 8 -- an exec file function

This exec file is a function which will prompt the user for the location of a Profile, returning a string with the name of the device to which the Profile is attached. Note that the function calls itself recursively until a valid device name is specified.

```
$EXEC { "GetProfLoc" -- get location of profile by asking user }
 $REQUEST %9 WITH
                           'Where is the profile attached ?
      (paraport/slot2chan1/slot2chan2) '
 $SET %9 TO UPPERCASE (%9)
   $IF (%9 <> 'PARAPORT') AND (%9 <> 'SLOT2CHAN1') AND (%9 <>
      'SLOT2CHAN2') THEN
  $WRITELN 'That is not a valid device name. Let''s try again.'
  $RETURN <GetProfLoc { recursive call }
 $ELSE
  $RETURN %9
 $ENDIF
$ENDEXEC
```

9.5 Exec File Programming Tips

The following few points may be useful to remember when creating exec files:

Use **modular** exec files. It may helpful to think of **exec files as procedures** which are called via the SUBMIT command. The more modular your exec files are, the easier it will be to use the stepping facility on them.

Create **standard exec files** for common functions; for example, use one exec file to perform all your compilations. One advantage of this is that you only have to edit one file when the interface to the tool changes (as it has in the case of the assembler).

Use **optional parameters** to support features which are not always (or often) used (such as the ability to compile against an alternate intrinsic library in your compile exec file). The parameter mechanism is such that you can remain oblivious to optional parameters if you don't need the functions they support.

Write your exec files to **prompt for information** which was not supplied in parameters. This way you don't need to remember the meaning of a large number of parameters.

9.6 Exec File Errors

The preprocessor can recognize a number of errors during its invocation and execution. The format in which most errors are reported is:

```
ERROR in <err loc>
<curr line>
<err marker>
<err msg>
```

where

<err loc>   is either 'invocation line' or 'line #<n> of file "<file>"'

<curr line> is the current exec line when the error was detected

<err marker> is a line with a question mark indicating where the preprocessor was in <curr line> when the error was detected

<err msg> is one of the messages listed below.

IO errors are followed by an additional line with the text of the OS error raised during the IO operation. The errors detected are as follows:

IO Errors:

```
Unable to open input file "<file>".
Unable to open temporary file "<file>".
Unable to access file "<file>".
Unable to rerun file "<file>".
```

Other Errors:

```
File does not begin with "$EXEC".
End of Exec file before "$ENDEXEC".
$EXEC command other than at start.
No Exec file specified.
More than 10 parameters.
No closing ")" found.
Line buffer overflow (>255 chars).
Invalid Exec option: <option char>.
Invalid Exec option on SUBMIT: <option char>.
End of Exec file in comment.
Invalid percent: not "%n" form.
Garbage at end of command.
No argument to SUBMIT.
ELSE, ELSEIF or ENDIF not in IF.
ELSEIF after ELSE.
File contains unfinished IF.
Nothing following "<tilde>".
Out of memory. Processing aborted.
Bad temp file name generated: "<file>".
No value returned from file called as function.
RETURN with value in file not called as function.
```

and

```
Invalid command. <token> expected.
```

where <token> may be:

```
String value
"%n" parameter
Terminating string delimiter
"=" or "<>"
"<>"
Boolean value
Comma (list delimiter)
"("
")"
```

Valid command keyword
Command

Chapter 10

THE UTILITIES

?

?

THE UTILITIES

10.1 Introduction

how to run utilities

10.2 ByteDiff

BYTEDIFF compares any binary files, but once it finds a difference between the two files, it does not always find where the differences end.

10.3 ChangeSeg

CHANGESEG changes the segment name in the modules in an object file. The first prompt asks for the object file you want to change:

File to change:

Changes are made in place (the file itself is changed). You are next asked:

Map all Names (Y/N)

If you want to change segment names in all modules, respond Y. If you want to be prompted for the new segment name for each module, type N. A response of <cr> accepts the default name.

10.4 CodeSize

10.5 Diff.

DIFF is a program for comparing ".TEXT" files, in the LISA Pascal development environment. DIFF is strongly oriented toward use with Pascal or Assembler source files.

DIFF is not sensitive to upper/lower case differences. All input is shifted to a uniform case before comparison is done. This is in conformance with the language processors, which ignore case differences.

DIFF is not sensitive to blanks. All blanks are skipped during comparison. This is a potential source of undetected changes, since some blanks are significant (in string constants, for instance). However, DIFF is insensitive to "trivial" changes, such as indentation adjustments, or insertion and deletion of spaces around operators.

DIFF does not accept a matching context which is "too small". The current threshold for accepting a match is 3 consecutive matches. The M option allows you to change this number. This has two effects:

Areas of the source where almost "every other line" has been changed will be reported as a single change block, rather than being broken into several small change blocks.

Areas of the source which are "entirely different" are not broken into different change blocks because of trivial similarities (such as blank lines, lines with only "begin" or "end", etc.)

DIFF makes a second pass through the input files, to report the changes detected, and to verify that matching hash codes actually represent matching lines. Any spurious match found during verification is reported as a "JACKPOT". The probability of a JACKPOT is very low, since two different lines must hash to the same code at a location in each file which extends the longest common subsequence, and in a matching context which islarge enough to exceed the threshold for acceptance.

DIFF can handle files with up to 2000 lines.

DIFF firstprompts you for two input file names: the "new" file, and the "old" file. DIFF appends ".TEXT" to these file names, if it is not present. DIFF then prompts you for a filename for the listing file. Type carriage-return to send the listing to the console.

DIFF does not (currently) know about INCLUDE files. However, DIFF does allow the processing of several pairs of files to be sent to the same listing file. Thus, when DIFF is finished with one pair of files, it prompts you for another pair of input files. To terminate DIFF, simply type carriage-return in response to the prompt for an input file name.

The output produced by DIFF consists of blocks of "changed" lines. Each block of changes is surrounded by a few lines of "context" to aid in finding the lines in a hard-copy listing of the files.

There are three kinds of change blocks:

INSERTION -- a block of lines in the "new" file which does not appear in the "old" file.

DELETION -- a block of lines in the "old" file which does not appear in the "new" file.

REPLACEMENT -- a block of lines in the "new" file which replaces a corresponding block of different lines in the old file.

Large blocks of changes are printed in summary fashion: a few lines at the beginning of the changes and a few lines at the end of the changes, with an indication of how many lines were skipped.

DIFF has three options which allow you to change the number of context lines displayed (+C), the number of lines required to constitute a match (+M), and the number of lines displayed at the beginning of a long block of differences (+D). To set one of these numbers, type the option name followed by the new number to the prompt for the first input file name. +D 100, for example, causes DIFF to print out up to 100 lines of a block of differences before using an ellipsis. The maximum number of context lines you can get is 8.

10.6 DumpObj.

DUMPOBJ is a disassembler for 68000 code. It can disassemble either an entire file, or specific modules (procedures) within the file. DUMPOBJ replaces DUMPMCODE.

DUMPOBJ first asks for the input file which should be an unlinked object file. The output (listing) file defaults to CONSOLE:. You are asked whether you want to dump

A(ll, S(ome, or P(articular modules.

If you respond S(ome, DUMPOBJ asks you for confirmation before dumping each module. A response of <ESC> gets you back to the top level. If you respond P(articular, DUMPOBJ asks you for the particular module(s) you want dumped.

The next question is: 'Dump file positions [N]?'The file position is a number of the form [0,000]where the first digit is the block number (decimal) within the file and the second number is the byte number (hexadecimal) within the block at which the module starts. This information can be used in conjunction with the PATCH program. Finally, DUMPOBJ asks if you want the object code disassembled.

10.7 DumpPatch

DumpPatch is a combination of DumpHex and Patch.

DumpHex provides a textual representation of the contents of any file. The file dump is block-oriented with the hexadecimal representation on the left and the corresponding ASCII representation on the right. If a byte cannot be converted to a printable character, a dot is substituted.

When DumpHex is Run, it asks you for the name of the output file. A .TEXT extension is added if necessary. To direct the output to the console, type carriage return. After getting a valid output file name, DumpHex asks for the input file to be dumped. No extensions are appended, so give the full filename. Once a file has been completely dumped, DumpHex asks you for the next file to dump. Type carriage return to exit the program.

After opening the input file, DumpHex asks you which block to dump. The default (carriage return) is block 0. If the output is going to a file, you are asked which block is the last you want dumped. The default here (carriage return) is the last block in the file.

The format of the console output depends on the number of lines your screen has. If fewer than 33 lines are available, the output is displayed only a half block at a time. Between blocks or block halves you have the option to

Type <space> to continue, <escape> to exit.

Escape returns to the prompt for an input file.

Patch allows you to examine and change the contents of any file. The display of the file's contents is exactly like that of DumpHex. With Patch, however, you can use the cursor control keys to move around in the block and change the value of any byte using either the hexadecimal representation on the left or the ASCII representation on the right.

After Running Patch you are asked for the full name of the file to patch. Carriage return exits Patch. No extension is appended to the file name. You are then asked for the number of the block you want to mess around with. Carriage return here returns you to the file name prompt.

The block is displayed with the cursor in the upper left corner at word 0 of the block. The arrow keys can be used to move around in the block. If you move the cursor up from the top line, you get the bottom line of the preceding block. Similarly, if you move down from the bottom line, you move into the top line of the next block.

When the cursor is on the hexadecimal side of the display, you can change any byte by typing the new hexadecimal value. Any non-hex characters are ignored. You can impress your friends by pointing out that the change is reflected automatically in the ASCII portion of the display. When the cursor is on the ASCII side, type any character to replace the value of the byte.

Until you move out of the block you can undo any changes by typing <escape>.

10.8 FileDiv and FileJoin.

It is often necessary to distribute files that are too large to fit onto a single floppy diskette. FILEDIV can be used to break a large file into several diskette-sized pieces. FILEDIV can then be used to rejoin these pieces at the file's destination. These two programs replace the TRANSFER program.

To divide a large text or object file, Run FILEDIV.

Input file: <give the name of the file to be divided>

Output file: <give the name to be used for the output files>

Do not include the suffix in the file name. If, for example, you want to divide TEMP.TEXT, give TEMP as the input file, and TEMP (or whatever) as the output file. FILEDIV will create a group of files named TEMP.1.TEXT, TEMP.2.TEXT, and so on, until TEMP.TEXT is completely divided up. If you use the drive number (#9:, for example), rather than the volume name, the new files can be written to multiple diskettes. When space on a diskette is exhausted, FILEDIV asks you to insert another diskette.

To rejoin the pieces of the file, Run FILEJOIN. Using the example given above, we can rejoin TEMP.1.TEXT and friends into TEMP.TEXT by responding:

Input file: TEMP <will read TEMP.1.TEXT, etc>

Output file: TEMP <will create TEMP.TEXT>

FILEDIV and FILEJOIN use regular directories, so a spurious sex change cannot destroy your file. Files are verified in both directions.

10.9 Grep

10.10 GxRef.

GXREF lists all the modules which call a given procedure, and all the modules which that procedure calls. It provides a global cross reference of subroutines and modules.

10.11 Packseg

10.12 SegMap

SEGMAP produces a segment map of one or more object files. The first prompt:

Files to Map ?

accepts either an object file name or a command file name. A command file must be preceded with a <. SEGMAP adds the .TEXT suffix to the command file name. The next prompt:

Listing File ?

directs the map information to the file given. A response of #1: or CONSOLE:, for example, send the map information to the screen. The map information includes the object file name, the name of the unit in the file, the names of the segments used in that unit (if any), and the new segment names.

10.13 SxRef

THIS MANUAL was produced using LisaWrite, LisaDraw, and LisaList.

ALL PRINTING was done with an Apple Dot-Matrix Printer.

Lisa™

...we use it ourselves.

FOLD

FOLD

FROM:

____

____

____

____

PLACE
STAMP
HERE

APPLE COMPUTER INC.
POS Publications Department
20525 Mariani Avenue, MS 2-0
Cupertino, California 95014

TAPE OR STAPLE

Apple publications would like to learn about readers and what you think about this manual in order to make better manuals in the future. Please fill out this form, or write all over it, and send it to us. We promise to read it.

Is it quick and easy to find the information you need in this manual?
[ ] always [ ] often [ ] sometimes [ ] seldom [ ] never

Comments____

What made this manual easy to use?______________________________

____

What made this manual hard to use?______________________________

____

How are you using this manual?
[ ] learning to use the product [ ] reference [ ] both reference and learning

[ ] other____

Please comment on, for example, accuracy, level of detail, number and usefulness of examples, length or brevity of explanation, style, use of graphics, usefulness of the index, organization, suitability to your particular needs, readability.

____

____

____

What do you like most about the manual?__________________________

____

What do you like least about the manual?__________________________

____

In school have you completed?

[ ] high school [ ] some college [ ] BA/BS [ ] MA/MS [ ] more

Comments____

What is your job title?__________________________________

How long have you been programming?

[ ] 0-1 years [ ] 1-3 [ ] 4-7 [ ] over 7 [ ] not a programmer

Comments____

What languages do you use on your Lisa? (check each)

[ ] Pascal [ ] BASIC [ ] COBOL [ ] other________________________

Comments____

____

What magazines do you read?_______________________________

____

____