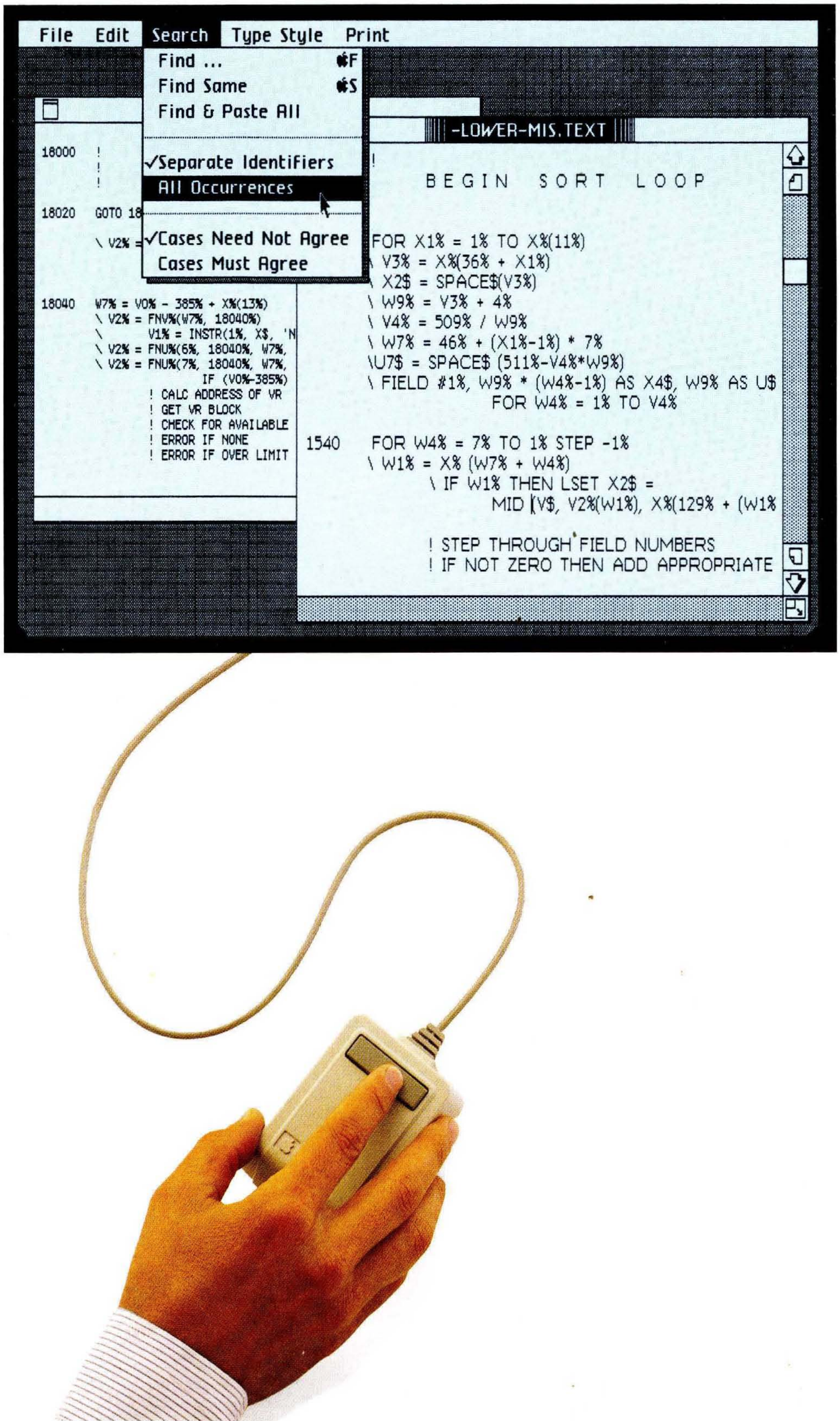
Lisa® BASIC-Plus 2.0 User's Guide



```
18000   !
        !
        !

18020   GOTO 18

        \ V2% =

18040   W7% = V0% - 385% + X%(13%)
        \ V2% = FNV%(W7%, 18040%)
        \       V1% = INSTR(1%, X$, 'N
        \ V2% = FNU%(6%, 18040%, W7%,
        \ V2% = FNU%(7%, 18040%, W7%,
                        IF (V0%-385%)
                ! CALC ADDRESS OF VR
                ! GET VR BLOCK
                ! CHECK FOR AVAILABLE
                ! ERROR IF NONE
                ! ERROR IF OVER LIMIT
```

```
      !
           B E G I N   S O R T   L O O P

      FOR X1% = 1% TO X%(11%)
      \ V3% = X%(36% + X1%)
      \ X2$ = SPACE$(V3%)
      \ W9% = V3% + 4%
      \ V4% = 509% / W9%
      \ W7% = 46% + (X1%-1%) * 7%
      \U7$ = SPACE$ (511%-V4%*W9%)
      \ FIELD #1%, W9% * (W4%-1%) AS X4$, W9% AS U$
                FOR W4% = 1% TO V4%

1540  FOR W4% = 7% TO 1% STEP -1%
      \ W1% = X% (W7% + W4%)
            \ IF W1% THEN LSET X2$ =
                MID (V$, V2%(W1%), X%(129% + (W1%

            ! STEP THROUGH FIELD NUMBERS
            ! IF NOT ZERO THEN ADD APPROPRIATE
```

Workshop User's Guide for the Lisa™

0351-A

Licensing Requirements for Software Developers

Apple has a low-cost licensing program, which permits developers of software for the Lisa to incorporate Apple-developed libraries and object code files into their products. Both in-house and external distribution require a license. Before distributing any products that incorporate Apple software, please contact Software Licensing at the address below for both licensing and technical information.

License and Copyright

Contents

Preface

This manual is intended for experienced Pascal, BASIC, or COBOL programmers. It describes the Workshop system, which is the environment in which these languages are used. We assume you have read the *Lisa Owner's Guide* and are familiar with your Lisa system.

Related Documents

For Pascal programming:

- *Pascal Reference Manual for the Lisa*
- *MC68000 16 Bit Microprocessor User's Manual*
- *Operating System Reference Manual for the Lisa*

For BASIC programming:

- *BASIC-Plus User's Guide for the Lisa*

For COBOL programming:

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

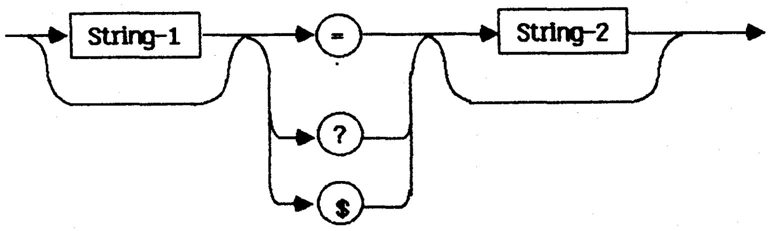
Type and Syntax Conventions

Boldface type is used in this manual to distinguish program text from English text.

Italics are used when technical terms are introduced.

Syntax diagrams are used to describe file specifiers and the syntax of exec files. For example, the following diagram describes a wild-card-spec:

wild-card-spec



0353-A

Start at the left and follow the arrows through the diagram. Several paths are possible. Every path that begins at the left and ends at the arrowhead on the right is valid, and represents a valid way to construct a file specifier. The boxes traversed by a path through the diagram represent the elements that can be used to construct a wild-card-spec. Thus the diagram embodies the following rules:

- A wild-card-spec can begin with a string (string-1) or the string can be omitted.
- A wild-card-spec must contain one of "=", "?", or "$".
- The "=", "?", or "$" can be followed by a string (string-2) or the string can be omitted.

The name contained in a rectangular box is the name for some other syntactic construction that is specified by another diagram. The name in a rectangular box is to be replaced by an actual instance of the construction that it represents.

Symbols such as reserved words, operators, and punctuation, are enclosed in circles or ovals. Text in a circle or oval represents itself, and is to be written as shown (except that capitalization is not required).

NOTES

Chapter 1
Introduction

Introduction

1.1 The Workshop

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.

Command lines provide access to all Workshop functions. The main command line, WORKSHOP, allows you to edit programs, run utilities or user programs, and use the 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. (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.)

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] key and holding it down while you press the [ENTER] key. The System Manager contains the Console command, which can be used to specify where the Workshop should display.

You can currently use the Workshop 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* (if you want to use assembly language or the Debugger)
- *Operating System Reference Manual for the Lisa* (for information on system calls)

For BASIC Programming:

- *BASIC-Plus 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. Specifically, you will not have the Debugger and can disregard the sections that pertain to it. 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, which 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, because hard disks have more space and are faster. See the *Lisa Owner's Guide* for instructions on booting the system.

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 Office System) you can use the Environments window to start up the environment you want, and switch between environments.

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	Display the Environments window 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, for example, to select another environment, use the Quit command from the Workshop command line.

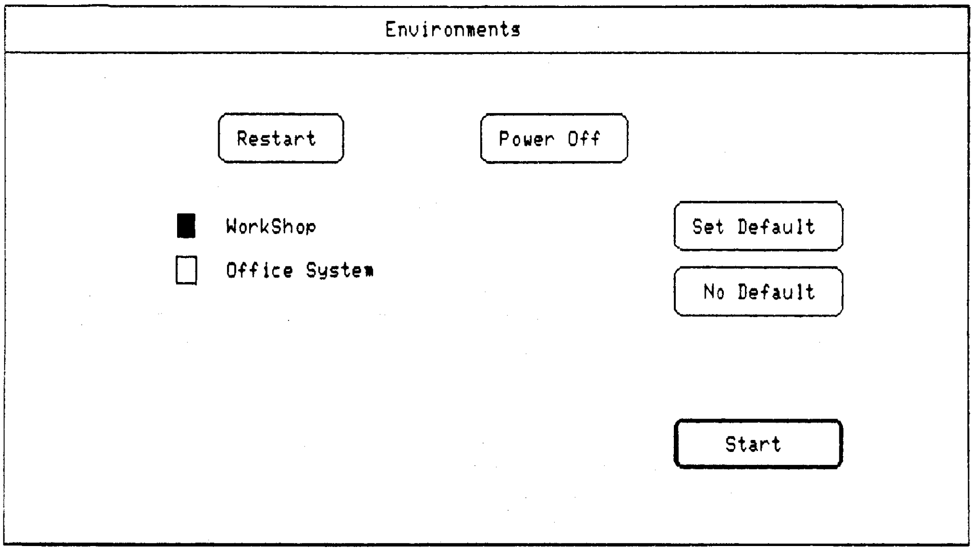


Figure 1-1
The Environments Window

1.3 The Workshop Command Line

When you select the Workshop environment, the Workshop command line appears at the top of the screen. This command line lists all the primary Workshop commands and gives access to several subsystems with additional commands. The Workshop line displayed contains only some of the commands available. You can see the rest of the commands by pressing "?", the last symbol on the line. To return to the original command line, press [RETURN]. Pressing the first letter of a command initiates the command.

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.

Two other subsystems 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.

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, Generate, MakeBackground, Link, TransferProgram

All the main command line commands are described as follows:

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. When you give the Run command a file name with no .OBJ extension, it will first search for that file name. If it is not found, it will search for filename.obj. If you do not specify a volume name, the Run command will search through up to three default volumes for the file. (See Section 2.4.1 for an explanation of volume name.) These defaults can be set by the File Manager's Prefix command. See Chapter 2 for more information on the Prefix command.

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 "<" or "exec/" 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 code. 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.*

Compilation is 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-Plus 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 use it to access the Environments window to start another environment or to turn off your Lisa.

The following prompt line appears after you confirm that you want to leave the shell:

```
WorkShop_shell, Another_shell, Reboot, Power_off
```

Type the first letter of what you want to do, for example, type A to access the Environments window.

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 *MC68000 16 Bit 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.

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.

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 background process will not try to display on the console or require keyboard input.

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.

TransferProgram (T)
The Transfer Program allows your Lisa to communicate with a remote computer. It can be used as a terminal, or to transfer files between the Lisa and the remote computer. The Transfer Program is described in Chapter 10.

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 The Workshop User Interface

This section describes conventions and standards used in the Workshop system. These ways of requesting input from the user are standard throughout the system to make it easier to use.

1.5.1 File Name Prompts

Many of the Workshop prompts are for file names. In the Lisa Operating System, you have few restrictions on what characters you can put in file names. However, you should be aware that the following restrictions exist in the Workshop:

1. You can embed blanks, but leading and trailing blanks and tabs will be removed when the Workshop processes your file prompt input.
2. Cases are preserved as you specify them.

A *pathname* has three parts: a device name, a file name, and an extension. The following conventions apply to a path name:

device (or volume) name	is up to 32 characters long, excluding '-'.
file name	is composed of alphabetic or numeric characters; spaces are permitted.

extension	is composed of alphabetic or numeric characters; spaces are permitted. An extension is optional. If present, it is the final '.' and any characters that follow (there must be at least one) in the pathname.

The combined length of the file name, plus extension, cannot exceed 32 characters.

Prompts often include default values. You do not have to enter parts of file names already supplied by defaults.

If a prompt includes a default extension which you don't want (except if the file name consists of only a logical device name), put a period at the end of the file name. The period will be removed and no extension will be added.

The following sections explain the standard responses allowed to prompts.

1.5.1.1 The CLEAR Key

The [CLEAR] key on the Lisa keyboard is an escape key. You can use it in response to a file name prompt to abort out of the command or program. No [RETURN] is required after pressing the key.

1.5.1.2 Prompts with Single Default Values

When a default value for part of a file name exists, it is shown enclosed in brackets in the prompt message; for example, [.text] indicates that there is a default file name extension value, and that that value is .text. If a default value is present, you need specify only the file name part not supplied by the default.

Extensions will not be added to file specifications consisting of device names only. Therefore, if you want to specify only a device when there is an extension default (for example, when prompted for a listing file with a default extension .TEXT and you want -printer), simply use -printer.

To use the default value for an entire file name, respond with [RETURN]. If you do not want any file to be used, even if a default value exists, respond with a backslash "\".

1.5.1.3 Prompts with Alternate Default Values

Alternate defaults are indicated by a slash. For example:

[-console]/[.text]

means you have a choice of either the console or a ".text" file. To choose the console, simply press [RETURN]. To choose a text file, respond with a file name.

1.5.1.4 Prompts with Separate Default Values

Each of the parts of a file name might have a separate default value, such as [-paraport] [-intrinsic] [.lib]. If each of the defaults is independent:

- a response with no device specification gives you the default device.
- a response with no file name gives you the default file name.
- a response with no extension gives you the default extension.

Sometimes the defaults depend upon each other. For instance, the prompt [-paraport-intrinsic] [.lib] indicates dependency, because the first two components are enclosed in the same set of brackets. When defaults are dependent, if you choose one or the other of them, you will get both. Be sure to look at what has been included in the brackets to see whether the defaults are independent or not.

1.5.1.5 Prompts with No Default Values

If you find that no default value is given in the file name prompt, use [RETURN] or a backslash to specify no file. Sometimes a file is required for the system to perform its function. If this is the case, and you specify no file, the program terminates.

1.5.1.6 Ending a List of Prompts

Some Workshop tools prompt for lists of files, as does the Linker. To indicate that you are finished responding to a prompt for a list of files, use [RETURN].

1.5.1.7 The ? Response

If you need help, or a list of program options, respond to a file name prompt by pressing the ? key followed by [RETURN]. Then proceed according to the information that appears on your screen.

1.5.2 How to Terminate an Operation

You can terminate the operation of most commands and programs by pressing -period, although termination might not be immediate if the program being run does not recognize -period.

NOTE

Note that most Workshop tools check for -period from the keyboard even when running under exec files. This means that you can abort Workshop tools in exec files.

Unless user programs are written to recognize the -period key combination as an abort mechanism, pressing those keys will not terminate the program being run. (See PASLIBCALL, Section 5.4, for information on the function PAbortFlag, which tells whether or not those keys have been pressed.) If this is the case, you can either:

- wait for the user program to terminate so that -period can be recognized by something else, or
- press the NMI key, which forces the system into the Debugger. The NMI key is the "-" key on the numeric keyboard.

See Section 8.2 for instructions on how to stop a user program early.

1.5.3 How to Halt a Screen Display

If you want to temporarily stop the screen display, press the key and type S, which stops the program from running by blocking its current output operation. When you want to restart the screen display, again press -S.

1.5.4 Inserting and Ejecting Diskettes

You can usually insert a diskette at any time. It will be mounted and accessible after you press any key, except the , [CAPS LOCK], [OPTION], or [SHIFT] key, on the keyboard. You can usually eject a diskette by pressing the diskette button and then hitting any key on the keyboard. (When you are in the Editor, the Preferences tool, or TransferProgram, you do not need to hit a key after pressing the diskette button.)

Mounting and unmounting diskettes is handled by the Pascal run-time system in the Workshop. Therefore, the act of inserting a diskette or pressing the eject button is not recognized until Pascal I/O is performed, thus the necessity of hitting a key. If the program you are running does not use Pascal I/O, you must first return to the Workshop command line. Then enter the File Manager and Mount or Unmount your diskette.

1.6 Utility Programs

The Workshop provides various utility programs, which support functions used less often than the functions you obtain through primary commands. The utilities are described in Chapter 10.

You must Run utilities. Choose 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.

1.7 How Do I Install the Pascal Language System?

Because the Lisa Office System is a standard product, you *must* install it before you install any optional language systems.

To install the Pascal language system, start with your ProFile on and your Lisa off.

1. Insert the "Pascal 1" Language System diskette into your Lisa's upper or lower disk drive.
2. Press the on-off button.
3. Hold down the key and type either 1, if you put the diskette in drive 1 (the upper drive), or 2, if you used drive 2 (the lower drive).
4. Wait. It will take about 3 minutes for the Lisa to load in the Operating System and the Workshop shell from the diskette.

NOTE

If you want to stop the loading process at any time after the system has booted, hold down the key while you type a period. The system will stop copying files and you will enter the Workshop environment.

---

5. When the system is finished booting, you will see some information about the cistart.text exec file and about initializing ProFiles. Then the system will ask you a series of questions. Be sure to type [RETURN] to terminate your responses.

    - The system will ask if you want to go ahead with the process. (Type Y for yes.)
    - The system will ask you where the target ProFile is attached. It must be attached to the built-in parallel connector (PARAPORT), or the upper or lower connector of the parallel interface card in expansion slot 2 (SLOT2CHAN2 and SLOT2CHAN1, respectively).
    - The system will then ask you to insert the second Workshop diskette, "Pascal 2".
    - The system will then ask if your ProFile needs to be initialized. Do *not* initialize your ProFile if there is already an Office System on it!
    - If you don't initialize your ProFile, the system will ask you if you have enough space on the target ProFile. "Enough space" for a whole Workshop means about 1500 blocks. If you already have another Workshop Language System on the ProFile, then "enough space" means about 700 blocks. (The language systems share about 800 blocks.)
    - If you do initialize your ProFile, you will be asked if there is now a Lisa OS volume on it. Answer Y if the ProFile has ever been used with a Lisa.

You will now see a lot of text flash by on your screen--don't worry, this is supposed to happen. The commands you generated by answering the questions are now actually being executed.

If you get any error messages, stop the process by typing -period, turn off your Lisa, and start over. If you get the same error again, write it down, and call the Apple® Support Hotline to find out what to do.

When all the files on the "Pascal 2" diskette have been copied, the system will eject the diskette and ask you to insert the "Pascal 3" diskette, then continue to copy files.

When the system is finished copying files, the Workshop command line will appear.

1.8 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 with the Pascal command (press P while the Workshop command line is displayed). 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 with the Link command. In order to be executable, the program must be linked with the Pascal support routines contained in IOSPASLIB.OBJ. If you are using any REAL variables, you must link your program to IOSFPLIB.OBJ. For other applications you can 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.*

1.9 How Do I Write and Run an Assembly Language Program?

Assembly language programs must be called as procedures or 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 1.8 for more information.

5. Use the Link command to link the Pascal object file, the assembly object file, IOSPASLIB.OBJ, and any other needed units or libraries.
6. Use the Run command to run the resulting object file.

1.10 How Do I Install the BASIC Language System?

Because the Lisa Office System is a standard product, you *must* install it before you install any optional language systems.

To install the BASIC language system, start with your ProFile on and your Lisa off.

1. Insert the "BASIC 1" Language System diskette into your Lisa's upper or lower disk drive.
2. Press the on-off button.
3. Hold down the key and type either 1, if you put the diskette in drive 1 (the upper drive), or 2, if you used drive 2 (the lower drive).
4. Wait. It will take about 3 minutes for the Lisa to load in the Operating System and the Workshop shell from the diskette.

NOTE

If you want to stop the loading process at any time after the system has booted, hold down the key while you type a period. The system will stop copying files and you will enter the Workshop environment.

5. When the system is finished booting, you will see some information about the cistart.text exec file and about initializing ProFiles. Then the system will ask you a series of questions. Be sure to type [RETURN] to terminate your responses.
   - The system will ask if you want to go ahead with the process. (Type Y for Yes.)
   - The system will then ask you where the target ProFile is attached. It must be attached to the built-in parallel connector (PARAPORT), or the upper or lower connector of the parallel interface card in expansion slot 2 (SLOT2CHAN2 and SLOT2CHAN1, respectively).
   - The system will then ask you to insert the second Workshop diskette, "BASIC 2".
   - The system will then ask if your ProFile needs to be initialized. Do *not* initialize your ProFile if there is already an Office System on it!

- If you don't initialize your ProFile, the system will ask you if you have enough space on the target ProFile. "Enough space" for a whole Workshop means about 1500 blocks. If you already have another Workshop Language System on the ProFile, then "enough space" means about 700 blocks. (The language systems share about 800 blocks.)
- If you do initialize your ProFile, you will be asked if there is now a Lisa OS volume on it. Answer Y if the ProFile has ever been used with a Lisa.

You will now see a lot of text flash by on your screen--don't worry, this is supposed to happen. The commands you generated by answering the questions are now actually being executed.

If you get any error messages, stop the process by typing -period, turn off your Lisa, and start over. If you get the same error again, write it down, and call the Apple Support Hotline to find out what to do.

When all the files have been copied, the Workshop command line will appear.

1.11 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 immediately 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-Plus User's Guide for the Lisa* for more information on the language.

1.12 How Do I Install the COBOL Language System?

Because the Lisa Office System is a standard product, you *must* install it before you install any optional language systems.

To install the COBOL language system, start with your ProFile on and your Lisa off.

1. Insert the "COBOL 1" Language System diskette into your Lisa's upper or lower disk drive.
2. Press the on-off button.
3. Hold down the key and type either 1, if you put the diskette in drive 1 (the upper drive), or 2, if you used drive 2 (the lower drive).

4. Wait. It will take about 3 minutes for the Lisa to load in the Operating System and the Workshop shell from the diskette.

NOTE

If you want to stop the loading process at any time after the system has booted, hold down the key while you type a period. The system will stop copying files, display "Exec processing aborted", and you will enter the Workshop environment.

---

5. When the system is finished booting, you will see some information about the cistart.text exec file and about initializing ProFiles. Then the system will ask you a series of questions. Be sure to type [RETURN] to terminate your responses.
   - The system will ask if you want to go ahead with the process. (Type Y for Yes.)
   - The system will ask you where the target ProFile is attached. It must be attached to the built-in parallel connector (PARAPORT), or the upper or lower connector of the parallel interface card in expansion slot 2 (SLOT2CHAN2 and SLOT2CHAN1, respectively).
   - The system will then ask you to insert the second Workshop diskette, "COBOL 1".
   - The system will then ask if your ProFile needs to be initialized. Do *not* initialize your ProFile if there is already an Office System on it!
   - If you don't initialize your ProFile, the system will ask you if you have enough space on the target ProFile. "Enough space" for a whole Workshop means about 1500 blocks. If you already have another Workshop Language System on the ProFile, then "enough space" means about 700 blocks. (The language systems share about 800 blocks.)
   - If you do initialize your ProFile, you will be asked if there is now a Lisa OS volume on it. Answer Y if the ProFile has ever been used with a Lisa.

You will now see a lot of text flash by on your screen--don't worry, this is supposed to happen. The commands you generated by answering the questions are now actually being executed.

If you get any error messages, stop the process by typing -period, turn off your Lisa, and start over. If you get the same error again, write it down, and call the Apple Support Hotline to find out what to do.

When all the files have been copied, the Workshop command line will appear.

1.13 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.*

Use the Quit coomand to exit back to the main command line.

1.14 Using the Printer

To use a printer with the Workshop system, you must set up the printer correctly, and configure your system for the printer. If you have more than one printer you will want to set up one of them as the default printer. These operations are explained below.

Setting up the Printer

The procedure for setting up a printer varies with the type of printer. See the instruction manual that came with your printer for directions on how to set it up correctly.

If your printer is an Apple Imagewriter, the default standards which have been factory preset should be satisfactory for normal use. However, if you want to modify the performance of the Imagewriter, you can get the technical specifications from the *Apple Imagewriter User's Manual, Part I: Reference.*

Configure Your Lisa for a Printer

Follow these steps to configure your Lisa for a printer:

1. From the Workshop command line, press S to enter the System Manager subsystem.
2. Then press P for Preferences. The Preferences tool is used to set up the configuration of the Lisa system and the Workshop.
3. Click on Device Connections to display what devices are connected to the Lisa.
4. Select the port to which your printer is connected. When you select the port, all devices that can be connected to that port are displayed.
5. Select printer, and additional configuration options are displayed.
6. When you are finished configuring your printer, select Quit from the Tools menu.
7. Then exit from the System Manager back to the Workshop command line by pressing Q for Quit.

Any changes made with the Preferences tool are made immediately to Parameter Memory, but changes in device connections do not take effect until the next time the Lisa is booted. Therefore, if you want to continue working, it is necessary to reboot your Lisa now. For additional information on the Preferences Tool, refer to Section 3.3.

To reboot, perform the following steps:

1. Press Q for Quit.
2. Select Y in answer to "Are you SURE you want to LEAVE the shell?"
3. Press R for Reboot.

When the system has finished rebooting, the changes you made will be in effect.

Setting a Default with Multiple Printers
If you have multiple printers connected to your Lisa, you can specify which one is to be the default printer. This means that you can establish which printer will be designated by -printer.

First configure all of the devices you want connected to the Lisa. (See the previous section and Section 3.3 for instructions on configuring devices.) After you have rebooted, return to the System Manager command line. Select D for DefaultPrinter, and enter the device name of the default printer. If you do not want to change the device name, because you want the default to remain as it is, press [RETURN] to exit back to the System Manager command line.

Rebooting is not required for the default printer setting to take effect. However, if output redirect to the printer is in effect, you will have to do the output redirection again.

Details on the DefaultPrinter option are available in Section 3.2.

1.15 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. The dependencies of the Lisa Workshop environment are shown in Figure 1-2 on the following page.

More information on the SYSCALL interface and routines can be found in the Lisa Operating System documentation.

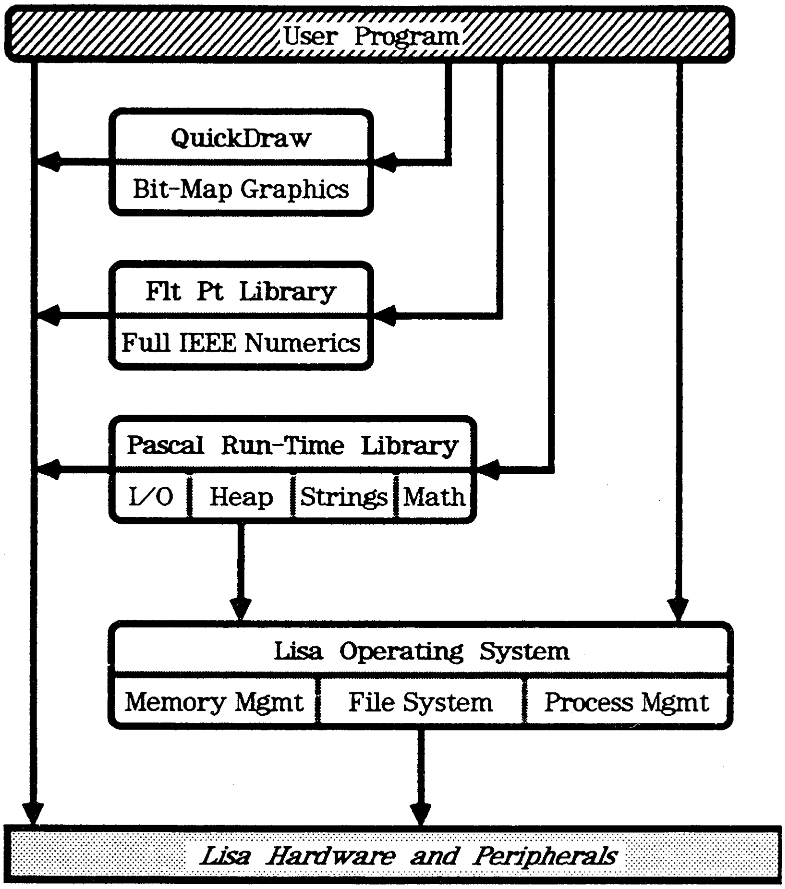


Figure 1-2
Lisa Workshop Environment

NOTES

Chapter 2
The File Manager

The File Manager

2.1 The File Manager

The File Manager is a subsystem of the Workshop. It provides file and device manipulation facilities, and 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.

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:

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

Pressing "?" displays the additional command line:

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 name while the File Manager command line is displayed. Most commands 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 value you want.

To manipulate files with the File Manager you need to address the file with a *file specifier*. 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 example -printer). File specifiers can contain wildcards enabling them to specify a collection of files. See Section 2.5 for more information on wildcards. See Section 2.4 for more information on file specifiers.

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.

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

2.3.1 Backup (B)

The Backup 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). It then compares the source files to the destination files. Whenever the contents of the two files are not equal, the source file is copied. If a source file is missing from the destination, it is copied. Thus it copies only *different* files from the source to the destination.

NOTE

The destination file is temporarily named Workshop.temp, and the source file is automatically copied. If the copy is successful, the destination file is renamed with its original name, and the files are compared. If the files are different, the first file is deleted. Ordering the process this way prevents deletion of the destination file before verification that the source file is good.

Because the file name Workshop.temp is internally involved in the Backup command, do not assign that name to your files.

2.3.2 Copy (C)

The Copy command copies files. It asks for a source file specifier and a destination file specifier. You can 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 is compared to the destination file after the copy operation to ensure that they are the same. The Validate command is described in Chapter 3.

Text files are handled specially when copied to the -printer or -console logical devices. Leading blanks in a line of text might have been replaced by a (DLE,count) pair to save disk space. As such patterns are detected, they are replaced by (count) blanks in the copy of the file sent to the printer or console. All other files are sent byte by byte unchanged.

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 in Section 2.3.13.

- If the file specifier is a file name (for example -MYDISK-example.text) information from only that file is listed.
- If the file 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 file length in blocks (512 bytes).
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 Operating System.
L	File is locked. It cannot be deleted until the file safety switch is turned off. (See FileAttributes command later in this section.)
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
---------                     ---- -----  -------------   -------------  ----
SYSTEM.DEBUG2                14848    29 03/03/83-15:46  06/10/82-21:57
SYSTEM.IUDIRECTORY            7168    14 07/18/83-09:31  02/23/83-10:33
SYSTEM.LLD                    9216    18 06/02/82-00:24  02/23/83-10:24
SYSTEM.LOG                    2992     6 07/18/83-16:56  06/08/83-17:49  O
SYSTEM.OS                   188928   369 05/04/83-10:08  05/04/83-10:08  CO
SYSTEM.SHELL                  8704    17 06/02/82-00:26  03/29/83-15:14  CO
XEJECTEM.OBJ                   512     1 06/02/82-00:27  03/29/83-15:22
```

Figure 2-1
The List Display

2.3.5 Prefix (P)

The Prefix command enables you to set up default volume names to search when you specify a file name without a volume name. You can set 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 file name without a volume name. The second and third prefixes are searched when you try to Run a program without specifying the volume it is on.

NOTE

The second and third prefixes affect the running of programs directly from the Workshop shell. They are not searched for programmatic file operations, such as opening files, or for other File Manager operations.

The last option of the Prefix command asks if you want to initialize the Prefix set at boot time. Answer Y if you want what you have entered to be established as defaults when you boot.

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 that you want. If you want to have no prefix, press [CLEAR] as the prefix for that level.

2.3.6 Rename (R)

The Rename command enables you to change the name of a file. It asks for the file name 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 specified by wild cards. See Sections 2.5 and 2.10 for more information on using wild cards and renaming files.

2.3.7 Transfer (T)

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

2.3.8 Quit (Q)

The Quit command exits from the File Manager subsystem back to the Workshop command line.

2.3.9 Equal (E)

The Equal command compares the contents of two files to determine if 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 and clear file attributes. You can set the safety attribute, which prevents you from accidentally deleting a file. You can also make a file into a protected master (see below).

To use the FileAttributes command press F in response to the File Manager command prompt. It displays the 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 ClearAttributes command clears the C, O, and S attributes on the specified volume, file, or set of files with wildcards. These attributes 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.

See the Scavenge command in Section 2.3.15 for more information.

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

Protect (P)
The Protect command is used to make an executable object file into a protected master. This is a form of copy protection for programs. 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 Lisa machine.
- Copies made will run only on the Lisa that made the *first* copy of the file.

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 from the FileAttributes subsystem to the File Manager.

2.3.11 Initialize (I)

The Initialize command 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, and the volume name. If you want the entire device to be initialized, press [RETURN] for the number of blocks (accepting the default). 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 warns you if you attempt to initialize a disk that already contains a volume, because the contents will be erased. 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)

The Mount 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 in Section 2.3.16, 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 command is 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, with the following information:

DeviceName	The name of the device.
VolumeName	The name of the volume.
VolSize	The number of blocks on the volume.
FreeBlks	The number of blocks still available.
Files	The number of files stored on the volume.
Open	The number of files open on the volume.
Attr	The attributes of the volume:

B	The Boot volume.
P	The Prefix volume (Prefix 1).
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
DeviceName    VolumeName                    VolSize  FreeBlks  Files  Open  Attr
----------    ----------                    -------  --------  -----  ----  ----
PARAPORT      Fred's Workshop                  9690       754    178    16  MBP
SLOT2CHAN2                                        0         0      0     0  M
RS232A                                            0         0      0     0  M
RS232B                                            0         0      0     0  M
MAINCONSOLE                                       0         0      0     1  M
ALTCONSOLE                                        0         0      0     0  M
```

Figure 2-2
The Online Display

2.3.15 Scavenge (S)

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

To scavenge a disk, use the Scavenge command and specify the device name. After the scavenge is complete, use the Mount command to mount it again, and 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 or another ProFile and run the Scavenger from it.

If a file is changed in any way by the Scavenger, the file attributes are set to S, for scavenged. This attribute is displayed by the List command. The changes made to the file might or might not affect the data in the file, depending on what state the file was in when it was scavenged. Examine any file that has the scavenged attribute before relying on its contents. After the file has been checked, you can remove the scavenged attribute with the FileAttribute command.

NOTE

A disk's File System can get into an inconsistent state if the Operating System terminates abnormally, 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 might 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 always returns the disk to a consistent state, but it is possible to lose data when the system crashes. 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 (takes it off line). It asks for a device name. For diskettes, use a volume name to unmount, or a device name to unmount and eject, the diskette. Always unmount a device before disconnecting it from a running machine.

2.4 The Workshop View of Files

Workshop users are provided with a view of files and devices that is actually a composite of what is provided by the Lisa Operating System, the Pascal run-time system, and the File Manager itself. Each contributes a specific set of facilities:

- The Lisa Operating System provides support for a variety of input and output devices, including both *block-structured devices* (disks and diskettes) and *sequential devices* (RS232 ports, consoles).
- The Pascal run-time system provides support for several *logical-devices* (console, printer, keyboard) which are not provided by the OS.
- The File Manager provides wild-card facilities which enable many File Manager commands to be applied to a whole set of files, rather than just one at a time.

2.4.1 OS Volumes on Disk

Every block-structured device is organized as a single volume with a flat directory structure. Volumes can be initially created on a disk by using the File Manager's Initialize command. The Initialize command:

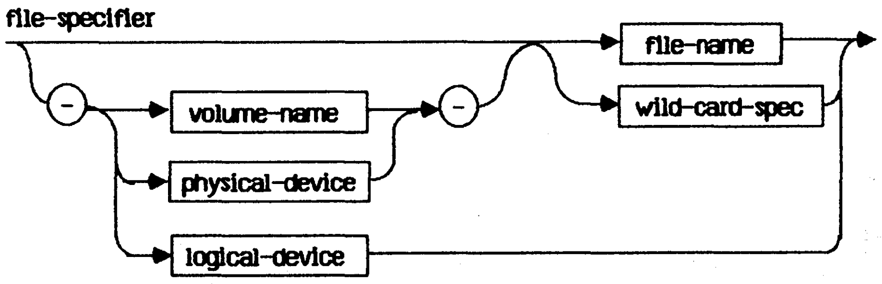
1. Formats the disk (if necessary).
2. Records its assigned volume name of up to 32 characters.

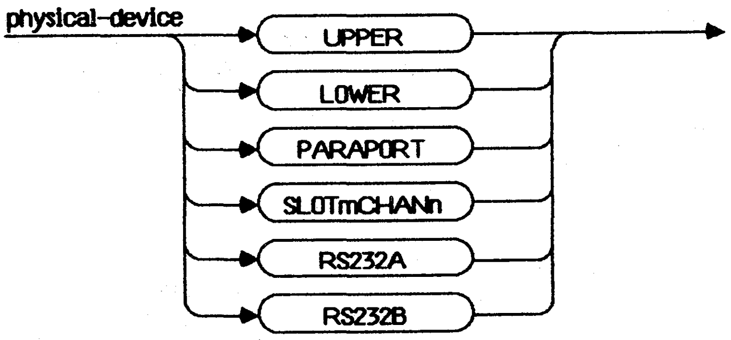
3. Creates its initial, empty directory (also called a *catalog*).
4. Mounts the initialized disk.

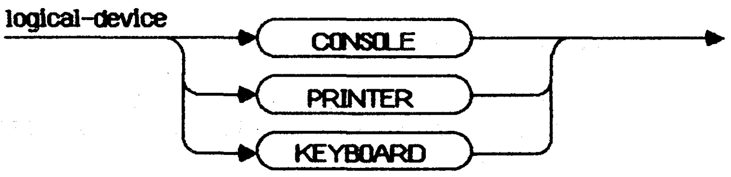
When an object is created on a disk, its file name of up to 32 characters is entered in the disk's directory. File names must be unique within a volume so that every object can be clearly identified.

2.4.2 File Specifiers

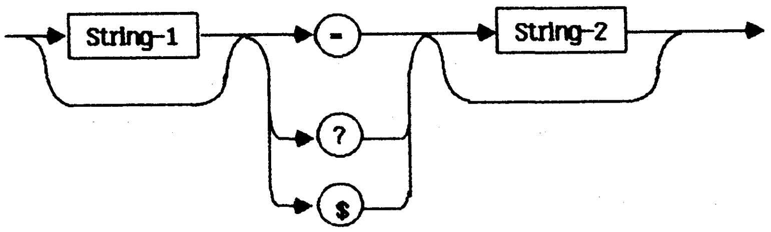
Within the Workshop, file specifiers are used to identify the volume, device, file, or set of files an operation applies to. The diagrams that follow show the makeup of a file specifier and its components.







wild-card-spec



A physical device name refers to a specific hardware device or port, whether or not there is actually anything connected or mounted there. When a device is block-structured and mounted, its physical device name can be used in a file specifier instead of the disk or diskette's volume name. Since sequential devices are not mass storage devices, they never have volume names. The only way to specify them is to use their physical device names followed by dummy file names; for example, "-RS232A-X". Logical devices are also not mass storage devices and do not have volume names. They can be referred to by their logical device names only.

2.4.3 The Working Directory and the Prefix

Sometimes, specifying the same volume name or physical device name again and again is inconvenient. With the File Manager's Prefix command you can establish a particular volume as the OS's working directory. Otherwise, the default working directory is the volume the system was booted from. If a file specifier omits the volume or physical device name, the file or set of files is assumed to be in the working directory. For example, if the working directory is -MYDISK, the file specifier PROGRAM1.OBJ refers to the same file as -MYDISK-PROGRAM1.OBJ.

-UPPER	The upper diskette; drive 1.
-LOWER	The lower diskette; drive 2.
-PARAPORT	ProFile attached to the parallel connector.
-SLOTmCHANn	ProFile attached to the Parallel Interface Card in slot m, channel n (where m is a slot between 1-3, and n is channel 1 or 2).

NOTE

To avoid confusion within the system, do not assign a device name to a volume.

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 that is used as the console can be changed by the Console command in the System Manager. See Section 3.2 for information on the Console command.
-PRINTER	Used to output to the printer. The physical connector that the printer is connected to is set by the Preferences tool, described in Section 3.3.3. If you have more than one printer, the one that will be used is specified by the DefaultPrinter command described in Section 3.2.
-KEYBOARD	Used as a nonechoing 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 extensions 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 converts an intermediate file into an object code file.
.LIB	This indicates a library directory.

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 "$". Only one wild card character can appear in a file specifier. These characters are used as follows:

string1=string2

The "=" character stands for any sequence of zero or more characters that can be ignored in the search. 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 enables 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 begin with bu/.
qd.=	to	quickdraw.=	Change all files starting with qd to begin with quickdraw.

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 in 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 after confirmation.

When using the List command, you cannot use the "?" wildcard in response to the prompt for a volume name.

string1$string2

The "$" character can stand for part of a destination file name only. It is replaced by the entire source file 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=.text, which results in:

bkfmgr.text
bksmgr.text

Hint: You can adopt conventions for naming files that pretend there is a hierarchical file system: for example,

```
Source/F1.text
Source/F2.text
Source/XYZ.text
```

2.6 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 "?" wildcard cannot be used in response to the List command prompt for a volume name.) If you want a listing of the default volume, press [RETURN].

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

You can send a copy of the directory to a file by following the specification with a comma and then the name of the file to send the directory to. For example,

```
-paraport-bk/=,foo.text
```

sends the directory to foo.text.

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

2.7 How Do I Copy a File?

You can Copy a file and leave the original file intact, or you can Transfer a file, which copies the file, then deletes the original file. To copy a file:

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 is 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.

The following are 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.)

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.

```
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.)

The Backup command is another way to copy files. It is selective, in that only different files will be copied. You use the same procedure to backup a file as to copy a file. See Section 2.3.1 for more information on the Backup command.

2.8 How Do I Delete a File?

To delete a file:

1. If you are not in the File Manager subsystem, enter it by typing F in response to the Workshop command prompt.
2. Invoke 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 Section 2.5 for more information on using wildcards.

2.9 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 the disk 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:
   a. The device name (upper or lower for a diskette, slot2chan2 or paraport for a ProFile, and so forth)
   b. The number of pages to initialize; the default is to initialize the whole device.
   c. The volume name.
   d. 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 warns you if you are initializing a device that has an existing volume on it, and gives you a chance to change your mind before destroying the existing volume.

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

2.10 How Do I Change the Name of a File or Volume?

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

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 same device name is assumed for a file.)

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.

NOTES

Chapter 3
The System Manager

The System Manager

3.1 The System Manager

The System Manager allows you to set system defaults and specify the system configuration. Using it, you can:

- 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 startup device.
- Set processes to be resident or nonresident, for performance tuning your Workshop system.
- Set which 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 is:

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

The System Manager command line works the same as the main Workshop command line. Pressing "?" shows you the additional line of commands:

Console, FilesPrivate, Validate, DefaultPrinter

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. A resident process will not be removed from memory when it terminates, so it will not have to be reloaded when it is run again. It also allows you to display the status of all currently existing processes, and remove processes. The process managment subsystem is described in Section 3.4.

OutputRedirect (O)
This 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 (which is the default).

Preferences (P)
This command starts the Preferences tool which allows you to set up the configuration of the Lisa system and the Workshop. The Preferences tool is described in Section 3.3.

Time (T)
This command allows you to set the hardware clock/calendar's date and time. See the *Lisa Owners Guide* for more information on the system clock and calendar. The date and time values are used for the creation and modification dates on your files, so they should be kept correct.

Quit (Q)
This command exits from the System Manager and returns 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, which is the default, on the alternate screen, where the Debugger displays, or on an external terminal connected to the RS232A or RS232B connector. When the main or alternate screen is used for the console, output can be stopped and restarted by pressing -S. If an external terminal is used with XOn/XOff processing enabled, then control-S stops output and control-Q restarts it.

The console can be moved to the alternate screen when you run a graphics program to prevent output from writelns from appearing on the graphics screen (the main screen). You can display either the alternate or the main screen by pressing OPTION-ENTER. When the console is moved to the alternate screen, both the console output (writelns) and the Debugger output will be mixed together on the same screen.

FilesPrivate (F)
This command enables or disables the selection of private system files. The Lisa Office System uses file names beginning with the "{" character for its tools and documents, and the Workshop user should rarely be concerned with such files. These files are called "private". When selection of private files is disabled (the default), the Workshop File Manager's wild card mechanism will exclude them from its selections unless the file specifier explicitly includes the leading "{".

There are just a few private files which are used by the Workshop (for example, {T11}menus.text). You must enable the selection of private files if you want a single file specifier to refer to the entire set of Workshop system files.

Validate (V)
This command is used to set up how much verifying you want the Workshop to do for you. There are two values you can set with this command. The first is whether or not to verify file copies. The system verifies a copy by comparing the original file with the copy to be sure they are the same. The default is to never verify. You should have no reason to verify unless you suspect something is wrong with your disk. The second value you can set is whether or not your selections for File Manager commands are verified. Selections are verified by listing the file names and asking you to confirm the operation.

DefaultPrinter (D)
This command is used when you have more than one printer connected to your Lisa. It tells the system which one will be the -printer logical device. It first gives you a list of all the physical devices that have been configured by the Preferences tool as printers, then asks you for the device name of the printer you wish to refer to as -printer.

3.3 The Preferences Tool

Start the Preferences tool by pressing P in response to the System Manager command line. It displays a window with four checkboxes and a tools menu. The Preferences display is shown in Figure 3-1.

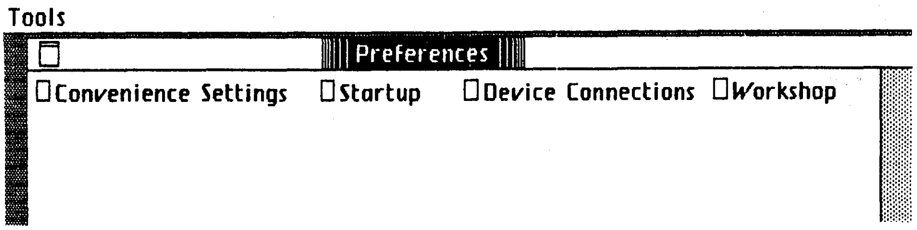


Figure 3-1
The Preferences Window

After you have finished with the Preferences tool, 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 regulate screen contrast, the speaker volume, and repeat delays.
- Device Connections that tell the Lisa system what external devices are connected.

- Startup, which tells the Lisa what device to use as a startup device.
- Workshop which sets up defaults for the Workshop.

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, and the parameter memory is lost, the preference settings will be restored from information on the startup disk.

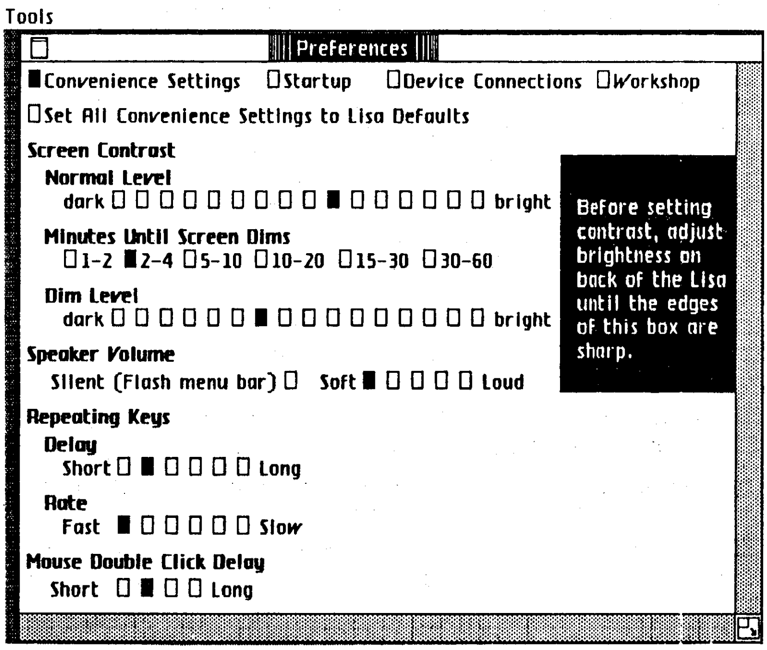
Any changes made with the Preferences tool change parameter memory immediately, but some of them, such as device connections and startup options, have no effect until the system is booted again.

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.

Four areas of preferences are described briefly below. More information on the first three areas can be found in the *Lisa Owners Guide,* Section D, Desktop Manager Reference Guide. 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 3-2.



■ 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

Speaker Volume

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

Repeating Keys

Delay

Short ☐ ■ ☐ ☐ ☐ ☐ Long

Rate

Fast ■ ☐ ☐ ☐ ☐ ☐ Slow

Mouse Double Click Delay

Short ☐ ■ ☐ ☐ Long

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

Figure 3-2
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 Minutes Until Screen Dims 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 demonstrates the volume by causing 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.

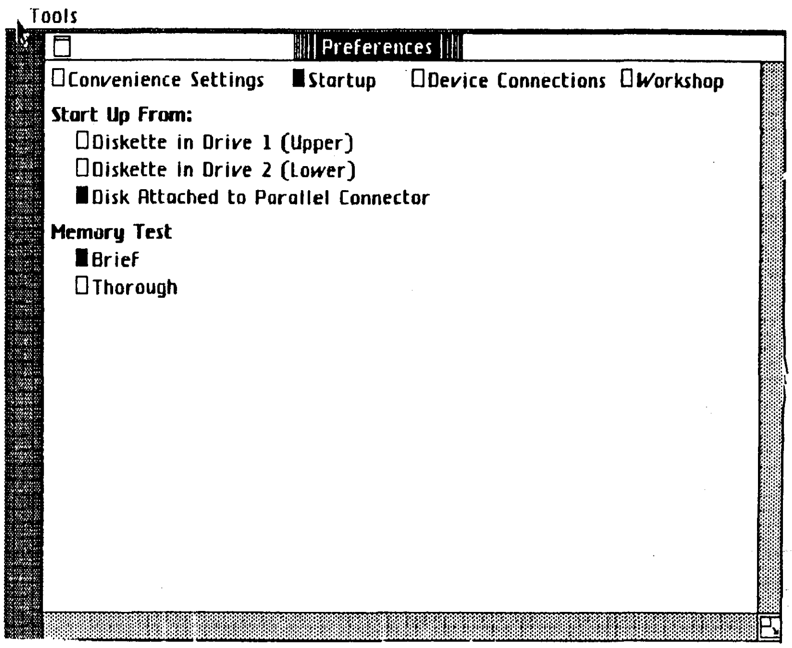
3.3.2 Startup

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

The Startup 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 Startup display also allows you to select a long or short memory test. The brief test takes about 20 seconds, the long test takes about 40 seconds.

Changes made to the Startup display are put into parameter memory immediately, but have no effect until the system is booted again.



☐Convenience Settings ■Startup ☐Device Connections ☐Workshop

Start Up From:

☐Diskette in Drive 1 (Upper)
☐Diskette in Drive 2 (Lower)
■Disk Attached to Parallel Connector

Memory Test

■Brief
☐Thorough

Figure 3-3
The Startup Display

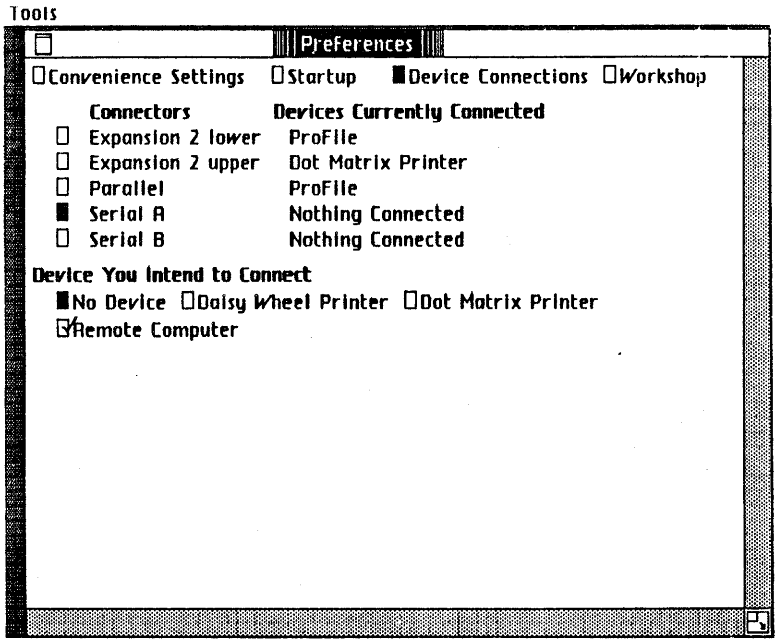
3.3.3 Device Connections

The Device Connections display allows you to specify what external devices are attached to the Lisa. When you choose Device Connections, the Lisa displays a table showing all the connectors available, and the device (if any) that is attached to it.

To tell the Lisa that you are attaching, removing, or changing an external device, check the box for the connector you are using. The Lisa will display a list of all devices that can be attached to that connector. Check the correct device. If you are removing a device, check No Device.

For some devices, such as printers, another set of specifications appears. Check the appropriate boxes for the device you are attaching.

Any changes made to the device connections are made immediately to parameter memory, but they do not take effect until the Lisa is rebooted. For the two serial ports, see the PortConfig utility in Section 11.10. A typical device connections display is shown in Figure 3-4.


Tools
Preferences
☐ 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

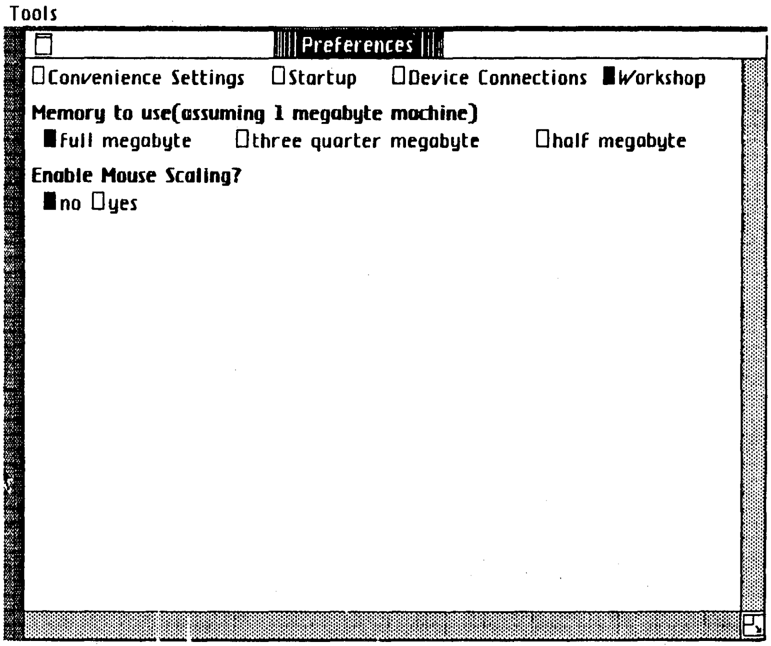
Figure 3-4
A Device Connections Display

3.3.4 Workshop

The Workshop display, shown in Figure 3-5, allows you to set parameters of the Workshop system. These parameters will not go into effect until you reboot the system. Then they are stored in parameter memory and will stay in effect until you change them.

Note that changes to the memory size affect all other systems (for example, the Office System) and will prevent large programs from running.

With mouse scaling, equivalent X and Y movements of the mouse cause diagonal cursor movement on the rectangular Lisa screen. Without scaling, the cursor would move at a true 45-degree angle on the screen when X and Y movements of the mouse are the same.



☐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 3-5
The Workshop Display

3.3.5 The Tools Menu

The tools menu provides you with two functions: Set all of PM to defaults, and Quit. Set all of PM to defaults resets parameter memory to the standard Lisa defaults. Quit exits from the Preferences tool, and puts a copy of the current settings of parameter memory on the disk.

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:

Manage Process: AddResident, DeleteResident, KillProcess, ProcessStatus, Quit ?

This subsystem is used to control which processes will be resident. After a resident process runs to completion, it is suspended and retained in memory, if possible, rather than terminated and removed from memory. This allows it to restart faster, because the process does not have to be recreated. 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 command removes a process from the list of resident processes, but does not kill the process if it is currently running.

KillProcess (K)
The KillProcess command terminates a currently existing process, including a background process, but does not remove it from the list of resident processes.

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

Pathname	The name of the processes object file.
Process_ID	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
The Quit command exits from the process management subsystem back to the System Manager command line.

NOTES

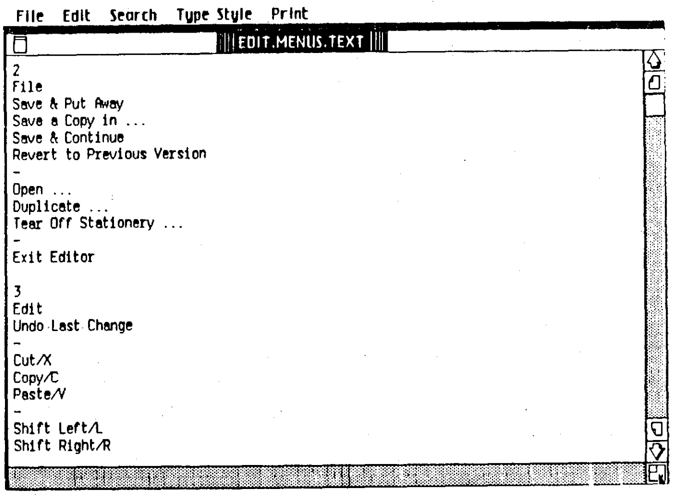
Chapter 4
The Editor

The Editor

4.1 The Editor

The Editor is used to create and modify text files. These files can 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 is shown in Figure 4-1.



```
2
File
Save & Put Away
Save a Copy in ...
Save & Continue
Revert to Previous Version
-
Open ...
Duplicate ...
Tear Off Stationery ...
-
Exit Editor

3
Edit
Undo Last Change
-
Cut/X
Copy/C
Paste/V
-
Shift Left/L
Shift Right/R
```

Figure 4-1
The Editor Display

The basic editing operations are inserting characters, cutting a portion of the text, and pasting text into a new location. Text that is cut goes 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 are inserted at this point. This is true even if the insertion point is not currently displayed in the window. The window is automatically scrolled to show the insertion point.

NOTE

The Editor is *memory based.* This means that there is a physical 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 11.

The mouse is used to scroll the text in the window, move the insertion point, select text to be cut or copied, point to menus, and select items on menus.

4.2 Using the Editor

Start the Editor by pressing E in response to the Workshop command prompt. The Editor prompts you for a text file name. If you want to edit an existing file, enter its name. If you want to create a new file, choose Tear Off Stationery from the File menu. The Editor prompts you for the stationery name. Press [RETURN] for the default, which is blank paper, or enter a name. For more information on stationery, see Section 4.2.3.

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

To leave the Editor, select Exit from the file menu, and you will return to the Workshop command line.

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 for complete information on selecting text. Text that is selected and then 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 can be pasted at any point in the active document by placing the insertion point where you want the text inserted and choosing 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 documents and stationery, to put away files, and to exit the Editor. 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 in the sections that follow.

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. Choose the menu item by moving the mouse down until the item you want appears in reverse video. Releasing the mouse 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 text file containing the desired text. To use any stationery other than the default blank paper, choose 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 text you want on the stationery. Save this document on the disk. To use this stationery, choose 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 document can 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, choose Open from the File menu. It asks you for the document name. The new document is read into a window on the screen and becomes the active document. To make another document that is already open the active document, use the mouse to move the pointer into a portion of that document and click the mouse button. If you have several documents open, you might have to move some out of the way.

This capability of working with more than one document at a time can be used to copy text from one document 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 choose 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.
- Place the insertion point at the place you want the text to be inserted, or select the text you want to replace.
- Choose Paste, which copies the text from the Clipboard to the active document.

Further information on each of these operations can be found in the sections that follow.

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, you place the insertion point where you want the text to be placed. You select text and place the insertion point by using the mouse to move the pointer on the screen.

Within an active document, the pointer will have one of three shapes:

Text pointer in a document

Arrow pointer for menus and scroll bars

Hourglass when an operation will take over 20 seconds

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

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

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

4.3.1 Moving 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 pointer to where you want it to be and click. Note that the insertion point moves when you select text.

4.3.2 Selecting Characters

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

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 to select and click the mouse button. Then move the pointer to the end of the text you want selected and shift click. Shift click means to hold down the shift key on the keyboard and click the mouse button. You can 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 Selecting Words and Lines

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

To select multiple words or lines, click the mouse button the required number of times, and hold. Move the pointer to the last word or line you want selected and release. If you double-click, and hold down the mouse button while you move the insertion point to the left or right, the selection expands or contracts by words. If you triple-click, and move the insertion point up or down, the selection expands or contracts by lines.

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 becomes selected.

4.3.4 Adjusting 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 enables you to have multiple documents displayed on the screen. These operations are done using the title bar and size control box as explained in Section 4.4.2.

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 grey portion of the scroll bar indicates the relative position of the currently displayed 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 in the window is near the middle of the document, and so on. To change the position of the text window, you can move the pointer into the elevator, click and hold the mouse 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 show 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 scroll bar. If you move the pointer to a view button and click, the display moves one windowful 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 Window

You can move the window on the screen and change its size. This lets you display multiple documents 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 is 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 reading in and writing out documents, updating documents, copying documents, and exiting the Editor. The File menu is shown in Figure 4-2. Each function is explained below.

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 document name. You are prompted for the name of the document 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 document from the disk.

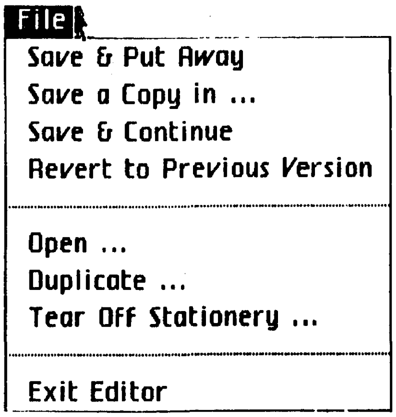


Figure 4-2
The File Menu

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 supplies the .TEXT extension on the file name. If the file name that you want does not end in .TEXT, you must end the file name with a period. See Section 1.5, The Workshop User Interface.

Duplicate . . .
This enables you to read in a copy of an existing document to edit into a new document. 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 for more information on stationery. 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. If you make the Editor resident, you can exit and restart the Editor without losing any information between invocations. Section 3.4, Process Management, gives instructions on how to make the Editor resident.

4.6 The Edit Functions

The Edit menu provides 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.

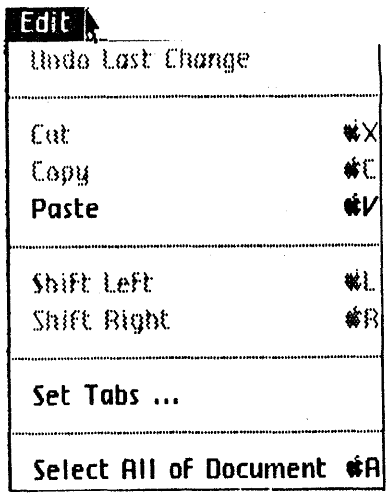


Figure 4-3
The Edit Menu

For example, to move text from one place in a document to another:

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

The Edit menu also enables you to adjust selected text left or right by inserting or deleting spaces, and to set tabs.

Some edit functions can also be done by holding down the key and pressing another key. The key that corresponds to each function is shown in the Edit menu, as you can see in Figure 4-3.

Undo Last Change
This command puts the document back to the way it was before the previous operation, if possible. You will receive a warning message if the last operation cannot be undone.

Cut
Cut places a copy of the currently selected text onto the Clipboard and removes the text from the active document. You can also Cut by pressing the X key while holding down the key.

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 the C key while holding down the key.

Paste
Paste inserts a copy of the text on the Clipboard at the insertion point in the active document. If a section of text is selected, Paste replaces it. You can also Paste by pressing the V key while holding down the key.

Shift Left
Shift Left moves selected text left by deleting a single space from the left of each line. It does 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 the L key while holding down the key.

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 the R key while holding down the key.

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

Select All of Document
This command selects the entire document. You can also select the entire document by pressing the A key while holding down the key.

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 replaces 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.

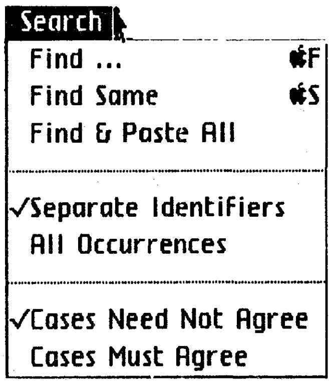


Figure 4-4
The Search Menu

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

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 is selected. If not, the system tells you. The Find command can also be executed by pressing the F key while holding down the key.

Find Same
Find Same repeats a previously specified Find, and selects the next occurence of the string. You can do a Find Same by pressing the S key while holding down the key.

Find & Paste All
Find & Paste All 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 if case must be matched. The options currently in effect have a check mark in front of them. To change the option, you choose a new one.

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

Separate Identifiers
When Separate Identifiers is chosen, the search operation looks for a "token" or word to match the search string. A token is a word bounded by spaces.

All Occurrences
When All Occurrences is chosen, the search operation matches 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 Cases Need Not Agree is chosen, any string with the same characters is a match, regardless of whether they are in uppercase or lowercase.

Cases Must Agree
When Cases Must Agree is chosen, the string with the same characters, and matching case, is selected.

4.8 The Type Style Functions

The Type Style menu enables you to change the display font. The Type Style menu is shown in Figure 4-5. A check appears in front of the font in which the document is currently displayed. You can change the font by selecting another font from the menu.

The font selected affects how many characters can be displayed on a line, and whether or not the display is proportionally spaced. When a document is printed, it is printed in the same type style it is displayed in, if that type style is available on your printer.

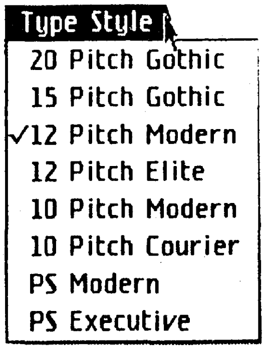


Figure 4-5
The Type Style Menu

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.

The Print functions are as follows:

Print All of Document
The Print All of Document command prints the entire document.

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

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

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

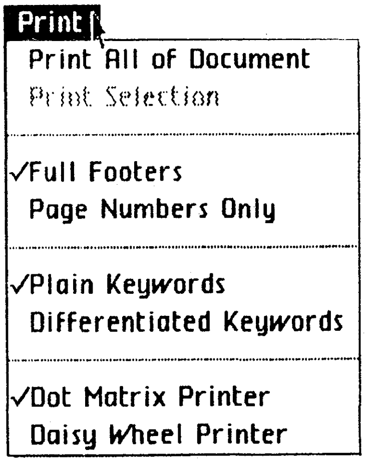


Figure 4-6
The Print Menu

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

Full Footers
When Full Footers is chosen, each page printed has a footer consisting of the document name, the page number, and the date. If the document is less than one page long, no footer will be printed.

Page Number Only
Choosing Page Number Only results in only a page number on the bottom of each printed page. If the document is less than one page long, no page number will be printed.

The next options are used for printing Pascal programs.

Plain Keywords
Choosing Plain Keywords causes Pascal keywords to print as normal text.

Differentiated Keywords
Choosing Differentiated Keywords causes Pascal keywords to print with underlining. In addition, the read procedure, write procedure, and other standard Pascal procedures and functions are underlined.

You choose the type of printer to print on with the next options. Select the type of printer you have attached to your Lisa: Dot Matrix Printer or Daisy Wheel Printer.

NOTES

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*. A Pascal program can call assembly language routines. More information on assembly language is in Chapter 6 of this manual.

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*.

The Pascal run-time support routines are in the library IOSPASLIB.OBJ. The support routines for floating point operations are in IOSFPLIB.OBJ. After generating the object code, it is necessary to link the program with IOSPASLIB.OBJ before you can run it. If you are using real numbers, you must also link with IOSFPLIB.OBJ. 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:

```
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:

```
List file[.TEXT]
```

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 are displayed in the listing file, or on the console if there is no listing file. When a message is displayed on the console, you are given a choice of aborting the compile by pressing [CLEAR], or continuing the compilation to look for more errors by pressing the space bar. A few errors give additional information after you press the space bar. Errors can also be placed in a separate error file by using the $E Compiler command.

5.2.1 Using the Code Generator

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

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

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

```
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.OBJ, and IOSFPLIB.OBJ for floating point operations. 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.

5.3 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. All the Compiler commands are listed in Table 5-1. A complete explanation of the Compiler commands is found in the *Pascal Reference Manual for the Lisa*.

Table 5-1
Pascal Compiler Commands

Command	Meaning
$I filename	Include contents of filename in this compilation.
$U filename	Search filename for units used.
$C+ or $C-	Turn code generation on (+) or off (-) for a procedure. Default $C+.
$R+ or $R-	Turn range checking on (+) or off (-). Default $R+.
$S segname	Start putting code modules into segment segname.
$X+ or $X-	Turn automatic stack expansion on (+) or off (-). Default $X+.
$D+ or $D-	Turn procedure name generation for Debugger on (+) or off (-). Default $D+.
$E filename	List Compiler errors in filename.
$L filename	Produce Compiler listing in filename.
$L+ or $L-	Turn source listing on (+) or off (-). Default $L+.
$DECL list	Declare compile time variables.
$SETC	Assign a value to a compile time variable.
$IFC	Begin conditional compilation section.
$ELSEC	Begin ELSE clause of conditional compilation. $ELSEC is optional.
$ENDC	End of conditional compilation section.

5.4 The Pascal Run-Time Environment

The Pascal run-time environment provides a unit PASLIBCALL which allows you to use some special system functions. It also provides special heap manipulation functions.

5.4.1 The PASLIBCALL Unit

The unit PASLIBCALL provides you with some additional system functions. In order to access the PASLIBCALL routines, you must use the units SYSCALL and PASLIBCALL:

```
USES
    {$U syscall} SYSCALL,
    {$U paslibcall} PASLIBCALL;
```

This gives you access to the routines listed below. These routines are contained in IOSPASLIB.OBJ, so programs using them require no additional inputs to the Linker.

function PAbortFlag : boolean

This function tells whether or not the -period key combination has been pressed. It enables programs to exit out of long operations. The flag is cleared when PAbortFlag is called. If you want your program to stop when you press -period, you must use this function in the program to detect that the key combination has been pressed. For example:

```
{This program fragment hangs in an infinite loop until -period
is pressed}

aborted :=false

Repeat {Wait for -period.  You might want to do other things
       here}

   aborted :=PAbortFlag;

until aborted.
```

procedure ScreenCtr (contrfun : integer);

This procedure provides standard screen control functions, and enables programs to perform screen control without having to to use escape sequences. Escape sequences are explained in Appendix C. The parameter specifies the screen control function. It is defined in the constants as follows, in the PASLIBCALL unit:

		Value	
Function	Constant	Decimal	Hex
clear screen	CclearScreen	1	1
clear to the end of screen	CclearEScreen	2	2
clear to end of line	CclearELine	3	3
move cursor to home position	CgoHome	11	B
cursor left one position	CleftArrow	12	C
cursor right one position	CrightArrow	13	D
cursor up one line position	CupArrow	14	E
cursor down one line position	CdownArrow	15	F

Screen control example:

```
{This program fragment clears the screen, and positions the
 cursor on the third line}

   ScreenCtr (CgoHome);
   ScreenCtr (CclearScreen);
   ScreenCtr (CdownArrow);
   ScreenCtr (CdownArrow);
```

procedure GetGPrefix (var prefix : pathname);

This procedure provides your program with the first level prefix setting in the File-Mgr in the Workshop.

procedure GetPrDevice (var PrDevice : e_name);

This procedure returns the corresponding default printer device name so that you can perform additional device control functions using DEVICE_CONTROL. (The *Operating System Reference Manual for the Lisa* explains the device control call.) The default printer device name is the one corresponding to the logical device '-printer'. Note that the device name returned contains a leading '-'.

```
procedure PLINITHEAP  (var  errnum,refnum:integer;
                            size,delta:longint
                            ldsn:integer;
                            swapable:boolean);
```

where:

errnum	is the error number returned if the procedure has any problems making a data segment having a mem_size of size bytes. Appendix A contains an explanation of the error codes for the Workshop.
size	is the number of bytes in the heap.
refnum	is the refnum of the heap.
delta	is the amount you want the data segment to increase when the current space is used up. If you use a large heap, use a large number for delta.
ldsn	is the logical data segment number used for the heap. The default is 5. For more information see the *Operating System Reference Manual for the Lisa.*
swapable	is the boolean that determines if the system can swap the heap data segment out to disk if it needs to.

This procedure can be used when you have special needs; for example, when you want to specify your own **ldsn** or heap size. When you use PLINITHEAP, you must call it before calling other heap routines. For more information on the heap, see Section 5.5.

5.4.2 The Pascal Heap

The Pascal heap is one contiguous piece of memory, a data segment, which works automatically without any initialization call. See Chapter 11 of the *Pascal Reference Manual for the Lisa* for information on the normal heap functions.

When a Pascal program starts execution, no heap space is allocated (no data segment made). On the first call to one of the heap routines or on the first PLINITHEAP call, the heap is created with either a default size of 16k bytes or the size specified in the PLINITHEAP call.

PLINITHEAP makes the heap as a private data segment so that the Operating System removes it when the process calling PLINITHEAP terminates. Note that when the heap is initialized, size and delta are put on 512 byte block boundaries. Therefore, if you use the PLINITHEAP call and specify values for size and delta that do not fall on block boundaries, the procedure increases the values to the next block boundary.

If the heap runs out of space while it is being used, the size of the heap is increased by the default of 16k or the delta specified in PLINITHEAP. The default **ldsn** used is 5. If you want a different **ldsn** for the heap data segment, call PLINITHEAP. Remember that the size of a data segment is limited by the **ldsn** you use. For **ldsn** 16, you can get only 128k (actually 96k safely), for **ldsn** 15 you can get only 256k, for **ldsn** 14 you can get only 384k, and so forth. See the *Operating System Reference Manual for the Lisa* for more information on **ldsn**'s and data segments.

If swapable is true, the heap is made with disc_size equal to size so the data segment is not memory resident. This uses up disc_size bytes on the startup disc. The default for swapable is false. When swapable is false, the procedure creates a data segment that has a disc_size of 0 (zero), which makes it memory resident.

The built-in Pascal heap routines are NEW, MEMAVAIL, MARK, RELEASE, and HEAPRESULT.

- If you call NEW and not enough space is available, the size of the heap is increased by either the default of 16k or the delta size specified in PLINITHEAP.
- MEMAVAIL provides the maximum number of words you could ever expect to get, taking into account the **ldsn** you used as well as the amount of free space the Operating System currently has available. If another process is using memory concurrently, its use of memory also affects MEMAVAIL. MEMAVAIL does not show the amount of memory left in the heap's data segment alone, since the heap's data segment can grow and shrink over time.
- MARK sets a pointer to the lowest free area on the heap. It is used with RELEASE to deallocate variables from the heap.
- RELEASE deallocates variables from a marked area of the heap. If you release the heap to a point within the original size of the heap data segment, the heap data segment is reduced to its original size. More information on MARK and RELEASE can be found in the *Pascal Reference Manual for the Lisa.*

- HEAPRESULT returns a 0 if the last heap operation was successful, otherwise it contains the Operating System error number indicating what failed. A list of the Operating System errors is in Appendix A.

NOTES

Chapter 6
The Assembler

The Assembler

6.1 The Assembler

The Assembler is a program that translates assembly language source statements into object code. The Assembler accepts a text file containing the source statements 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 can be found in the Motorola MC68000 Reference 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: an optional listing file and the object file (the file that will contain the object code produced by the Assembler).

The input file must be a text file containing assembly language source statements. You can create 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.

Any errors in the program will be indicated by messages on the console or in the listing file. A complete list of Assembler error messages is found in Appendix A of this manual.

6.2.1 Assembler Options

When you start the Assembler, the option settings are displayed. You can enter the options selection mode by responding to the input file prompt with "?". There are two Assembler options:

P	Pretty listing.
S	Print information about available space.

Each option may be set to + or -:

+	On
-	Off

When pretty listing is on, the forward referenced labels or offsets are filled in with the correct values in the listing.

After setting options, 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 object 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. File naming is explained in Chapter 2.

6.2.3 The Object File

The object file produced by the Assembler contains a machine code version of your source program. The name of an object file ends with .OBJ . A raw assembly object file is not executable; it must be linked with a Pascal program that calls it. See Section 6.6 for further information.

The output file will be an object file which must be linked with a Pascal main program before it can be executed. The object file goes to the same volume as the input text file was on unless another volume is specified.

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 referenced labels will be presented in the listing file as asterisks (*). If pretty listing is on, the actual values will be filled in.

Source statement errors are flagged in the listing. Refer to the Appendix for a list of Assembler error messages.

An example of an Assembler listing file is shown in Figure 6-1. Figure 6-2 shows the same file listed with the pretty list option.

```
0000| 0000 0001          one       .equ     1
0000| 0000 0020          label2    .equ     $20
0000|
0000| 303C 0020                    move     #label2, d0
0004| 4240                         clr      d0
0006| 5240               @2        add      #one, d0
0008| 6700 ****                    beq      @1        ; show listing patching
000C| 60F8                         bra      @2        ; address filled in
                                                      ; for backward branching
000E|
000E|
000A* 0004
000E| 41FA ****          @1        lea      data, a0
0012| 60**                         bra.s    done
0014|
0014|                              ; some more code ...
0014|
0014| 4E71                         nop
0012* 02
0016| 4E75               done      rts
0018|
0010* 0008
0018| 19 30 13           data      .byte    25, $30, 19    ; odd number of bytes
001B| 00                           .align   2     ; make sure next instruction
                                                  ; is on even
```

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.

NOTE

If you want pretty listing, the listing output must be sent to a file, not to a device. Pretty listing is done by making an additional pass through the listing file to patch in the forward references. There must be enough disk space for two listing files for this operation to succeed.

```
0000|                                 .proc     example
0000|
0000| 0000 0001           one         .equ      1
0000| 0000 0020           label2      .equ      $20
0000|
0000| 303C 0020                       move      #label2, d0
0004| 4240                            clr       d0
0006| 5240                @2          add       #one, d0
0008| 6700 0004                       beq       @1      ; show listing patching
000C| 60F8                            bra       @2      ; address filled in
                                                        ; for backward branching
000E|
000E|
000E| 41FA 0008           @1          lea       data, a0
0012| 6002                            bra.s     done
0014|
0014|                                 ; some more code ...
0014|
0014| 4E71                            nop
0016| 4E75                done        rts
0018|
0018| 19 30 13            data        .byte     25, $30, 19     ; odd number of bytes
001B| 00                              .align    2      ; make sure next instruction
                                                       ; is on even
```

Figure 6-2
Pretty Listing

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 that are more indicative of how the instruction is being used after unsigned comparisons. These variants are BHS (Branch on High or Same) for BCC, and BLO (Branch on Low) for BCS. 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 (an 8-bit displacement rather than a word displacement), append a .S to the instruction. The default branch size is Word.

Note that the TAS (test and set) instruction is not implemented on the Lisa hardware. Using this instruction may cause timing problems.

Note 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. The structure of an assembly language program is shown in Section 6.4.1. Rules for forming constants, identifiers, labels, expressions, and addressing modes are provided in the following sections.

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 is shown in Figure 6-3. The source file contains an (optional) section of operations that doesn't generate code. Constants or macros are usually defined here. Next it conains one or more procedures (.PROC) or functions (.FUNC). These each contain a sequence of directives and code generating operations. A procedure or function ends when the Assembler encounters the next .PROC or .FUNC. The .END directive is the last statement that is processed by the Assembler. Any text beyond the .END is ignored.

non code generating operations

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

.PROC
...

.FUNC
etc.

.END

Figure 6-3
Structure of an Assembly Language Program

The directives that don't generate code 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**. Numeric constants are expressed as follows:

Decimal
Decimal numbers are formed with the decimal digits (0-9). Examples:

```
10
13
137
```

Hexadecimal
Hex numbers can be expressed in two ways:

1. Preceed the number with a "$". Examples:

   ```
   $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 number 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 can be alphanumeric, period, underbar, or percent sign.

Examples of identifiers are:

```
LOOP
EXIT_PRC
NUM
num64%
```

6.4.4 Labels and Local Labels

Labels begin in column one. They can be followed by an optional colon.

Local labels can be used to avoid using up the storage space required by regular labels. The local label stack can handle 50 labels at a time. It is cleared every time a regular label is encountered. 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 long unless constrained by the instruction. Expressions are evaluated from left to right with *no operator precedence*. Angle brackets can be used to control expression evaluation. The operators are:

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

There is no operator precedence in expressions. For example, in the expression 2 + 9 * 4, the addition is performed first. To perform the multiplication first, rewrite the expression with angle brackets to show precedence: 2 + <9 * 4>; or reorder the operands: 9 * 4 + 2.

6.4.6 Addressing Modes

Refer to the Motorola 68000 manual for detailed information on the addressing modes supported by the 68000 microprocessor. Table 6-1 gives a summary of the addressing modes including their syntax.

Table 6-1
Summary of Addressing Modes

Mode	Register	Syntax	Meaning	Extra Words
0	0..7	Di	Data direct	0
1	0..7	Ai	Address direct	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)	Offset indexed	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:

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

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

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

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

NOTE

All programs that run under the Lisa OS must be relocatable. Addresses should not be absolute.

6.4.7 Miscellaneous Syntax

Comments

A comment in an assembly language program begins with a semicolon. The Assembler ignores all characters after a semicolon in a line. Examples are:

```
;   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 infinitely
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

Assembler directives tell the Assembler to do various functions besides generating executable code. These functions include defining symbols and constants, defining macros, doing conditional assembly, and controlling listing options.

The Assembler directives (pseudo-ops) are shown in Table 6-2.

Table 6-2
The Assembler Directives

Directive	Operands	Meaning
.PROC	<identifier>	begin procedure
.FUNC	<identifier>	begin function
.DEF	<identifier-list>	make identifiers externally available
.REF	<identifier-list>	declare external identifiers
.SEG	'<name>'	put code of next .PROC in segment 'name'
.END		end of entire assembly
.ASCII	'<char-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> relative to beginning of assembly
.RORG	<value>	same as .ORG
.EQU	<value>	set label equal to <value>
.MACRO	<identifier>	begin macro definition
.ENDM		end macro definition

Table 6-2 (continued)
The Assembler Directives

Directive	Operands	Meaning
.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 macro expansion listing
.PATCHLIST		turn on patchlist
.NOPATCHLIST		turn off patchlist
.INCLUDE	<filename>	include contents of <filename> in assembly

6.5.1 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   "don't"        ; string containing single quote
    .ASCII   'a "glitch"'   ; string containing 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 zeroes 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.

For example,

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

creates the assembled output:

```
0000
FFFF
FFFE
0011
```

.LONG <values>
Allocates two words of space for each value in the list. For example,

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

creates the output:

```
00000000
0000FFFF
FFFFFFFE
00000011
```

<label> .EQU <value>
Assigns <value> to <label>. <value> can be an expression containing 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 addressing must be PC relative.*

6.5.2 Macro Directives

A macro consists of a macro name, optional arguments, and a macro body. When the Assembler encounters the macro name, it substitutes the 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 string is 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 listing created would be:

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

6.5.3 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 the conditional expression.

.IF <expression>
Identifies the beginning of a block of source statements that is assembled only under certain conditions. If <expression> is false, the Assembler ignores all statements until a .ELSE or .ENDC is found. The statements between the optional .ELSE and .ENDC are assembled if <expression> is evaluted to be false at the time of assembly. Otherwise they are ignored.

<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. 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
```

6.5.4 External Reference Directives

Separate routines can share data structures and subroutines by linkage between assembly routines using .DEF and .REF. These directives generate link information that allows separately assembled routines to be linked together.

.DEF and .REF directives 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 the data as parameters in the procedure or function call. Information on parameter passing between Pascal and assembly language routines 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
```

This example 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 offset in the jump table. If the referenced procedure is in an intrinsic unit (and therefore in a different segment), the IUJSR, IULEA, IUJMP, and IUPEA instructions are used. The Linker blindly assumes that the word immediately before the word of zeros is an opcode in which the low order 6 bits are 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". The .SEG directive takes effect when the next .PROC or .FUNC is reached. Thus it is not possible to split one procedure into two segments. This is an example of how the .SEG directive works:

```
.SEG 'name1'
     .PROC    A
     ...
     {code in .PROC A}
     ...
     .SEG      'name2'
     ...
     {code still in .PROC A}    {this code will still be in segment 'name1'}
     ...
     .PROC    B                 {code of .PROC B will be in segment 'name2'}
```

6.5.5 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's prompt, the listing directives are ignored.

The default for the Assembler is for .LIST, .MACROLIST, and .PATCHLIST to be in effect when the Assembler starts. .TITLE defaults to blank.

.LIST and .NOLIST
Can be used to select portions of the source to be listed. The listing goes 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
Causes the next line of the listing file to be printed on the next page.

.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. For example:

```
    .TITLE   'Interpreter'
```

places the word, "Interpreter", at the head of each page of the listing.

.PATCHLIST
Patches the forward referenced labels in the listing. It must be on if you want pretty listing. See Section 6.2.4 for more information on pretty listing.

.NOPATCHLIST
Turns off patching of forward references.

.MACROLIST
Turns on listing of the expanded code from a macro.

.NOMACROLIST
Turns off listing of macro expansion. See Figure 6-4 for examples of macro listing.

```
0000|                           ;    2 parameters in INC:
0000|                           ;      %1 - the amount to add to
                                ;           register that is passed as %2
0000|                           ;      %2 - register name
0000|                           .macro    INC
0000|                           add       #%1,%2
0000|                           .endm
0000|
0000|
0000|                           ;    parameters passed to DEC:
0000|                           ;      %1 - amount to subtract
                                ;           from register %2
0000|                           ;      %2 - register name
0000|                           .macro    DEC
0000|                           sub       #%1,%2
0000|                           .endm
0000|
0000|
0000|                           .proc     MacroExample
0000|                           INC       2,d0
0000| 5440            #           ADD       #2,d0
0002|                           INC       1,a4
0002| 524C            #           ADD       #1,a4
0004|                           DEC       $ff,d3
0004| 0443 00FF       #           SUB       #$ff,d3
0008|                           .end
```

Figure 6-4
Macro Listing

6.5.6 File Directive

.INCLUDE <filename>

Causes the contents of <filename> to be assembled at the point of the .INCLUDE. You need not specify the .TEXT suffix. An included file cannot itself contain an .INCLUDE statement.

6.6 Communication with Pascal

Assembly language routines must be called from a Pascal program. In order to call an assembly language routine, the Pascal program declares the assembly language procedure or function to be EXTERNAL. If the assembly routine does not return a value, declare the assembly routine as a PROCEDURE in the Pascal program. If a function result is to be returned from the assembly routine, declare it as a FUNCTION in Pascal and space for the returned value is allocated (by the Pascal Compiler) on the stack just before the function parameters, if any. The amount of space allocated depends on the type of the function. A Longint or Real function result takes two words, a Boolean result takes one word with the result in the high order byte, and other types take one word. A Boolean result of 0 indicates false, any non zero value indicates true.

NOTE

Assembly language programs are in read only memory segments. Thus they have no data space to write into. Any data space needed must be allocated by the Pascal Compiler. A pointer to the space is then passed to the assembly language routine. "Writes" to the data space are done by pointer references using modes like (Ax), i(Ax), etc. For examples of this technique see Section 6.7.5

In the following example, an assembly language routine is linked to a Pascal program. The assembly language routine accepts two integers and returns the logical AND of them. The Pascal host file 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
    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 little attempt has been made 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 memory management hardware signals a bus error that causes the 68000 to generate a hardware exception and pass control to an exception handler. See the *Lisa Hardware Manual* for more information on the memory management hardware. 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 reexecuted 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. In this case, the Compiler currently emits code that in some cases looks 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 memory area following the final RTS or JMP (A0) contain the procedure name, in upper case (produced by the Pascal Compiler). The Debugger gets the procedure name from this block, allowing you to use procedure names in the Debugger. The following example shows how an assembly language programmer can provide the Debugger with information it needs to perform symbolic low level debugging. Note that all procedure names must be in upper case to be compatible with the Debugger.

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

These two macros, HEAD and TAIL, can be used to make it easier to debug assembly language routines called from Pascal programs.

Upon entry to the assembly routine, the stack is as shown in Figure 6-5.

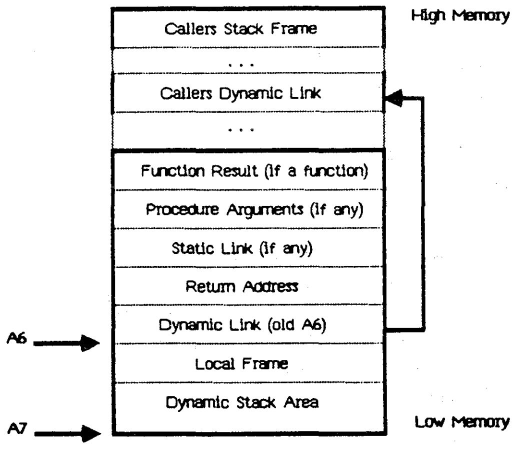


Figure 6-5
The Pascal Run-Time Stack

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.

Procedure arguments are present only if parameters are passed from Pascal. They are pushed on the stack in the order of declaration. All reference parameters (parameters declared as VAR's in the Pascal Procedure or Function declaration) are represented as 32-bit addresses. Value parameters less than 16 bits long always occupy a full word. A boolean parameter passed by value occupies a word with the value in the most significant byte (the lower-addressed byte). All non-set value parameters larger than 4 bytes are passed by reference.

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

6.6.2 Register Conventions

The following are the register conventions used in the Lisa system. It is your responsibility to preserve 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 (must be preserved)
A5:	Pointer to user globals (must be preserved)
A6:	Pointer to base of stack (must be preserved)
SP:	Top of stack

Registers D3 and A2 may be used at some time in the future by the Compiler for code optimization, so you should preserve them also.

6.6.3 Parameter Passing Between Pascal and Assembly Language

Parameters are passed between Pascal and assembly language routines in the following ways:

by value:

boolean	a word on the stack with the boolean value in the most significant byte of the word (lower, or even address).
integer	a word
longint	two words
data structure	by address (4 bytes). It is the responsibility of the assembly language routine to interpret the data structure correctly.

by reference (VAR parameters):

all types	by address (4 bytes on the stack)

6.7 Assembly Language Examples

6.7.1 Using .REF and .DEF Directives

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

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
```

6.7.2 String Parameters

The following program illustrates how to pass a Pascal string to an assembly language program, modify the string, and return it. Pascal strings have their length stored as the first byte in the string.

NOTE

Assembly language routines are in read only segments and do not have their own data (read/write) area. All read/write data should be declared in Pascal and passed to the assembly routines using pointers.

The Pascal source file is:

```
program  pasStr;

type     strType = string[80];

var      str : strType;
         ch : char;

procedure AsmStr (var str : strType);
  external;

begin
  str := 'initial string in Pascal main program';
  writeln (str);
  AsmStr (str);
  writeln (str);
  writeln;
  write ('press any key to continue');
  read (ch);
end.
```

The assembly language file is:

```
     .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'
     .align   2             ;get on a word boundary
```

6.7.3 Writing a Function

The following example shows how to write a function in assembly language. This function returns a boolean value.

The Pascal program is:

```
program booleanFunction;

var    int : integer;
       ch : char;

function swapBytes (var int : integer) : boolean;
  external;

        { if a parameter is passed by reference
          (a var parameter) its addesss is passed
          to the assembly routine on the stack }

begin
  int := 256;
  writeln ('the initial value of int = ', int:1);
  repeat
    if swapBytes(int) then
      writeln ('int = ', int:1)
    else writeln ('int = 0, function value is false');
    int := int - 1;
  until (int < 0);
  write ('press any key to continue');
  read (ch);
end.
```

The assembly language function is:

```
     .func    swapBytes

     move.l   (A7)+,A0        ; pop return address
     move.l   (A7)+,A1        ; get address of word to swap

     move     (A1),D0         ; get the number
     ror      #8,D0           ; swap the bytes
     move     D0,(A1)         ; put it back

     bne      @1
     clr      (A7)            ; number = 0 so return false (0)
     bra      @2

@1   move     #$FFFF,(A7)     ; return result true (non zero)
@2   jmp      (A0)            ; return to calling program

     .end
```

6.7.4 Calling Pascal I/O Routines

The following example illustrates how to call Pascal routines from assembly language to do I/O. Note the use of macros for calling the Pascal routines.

```
program AsmIO;

type    strType = string[80];

var     str:strType;
        f1,f2: text;
        ch: char;

procedure main;
  external;

{THE FOLLOWING FUNCTIONS ARE CALLED FROM THE ASSEMBLY LANGUAGE
 PROGRAM MAIN TO PERFORM I/O}

function f_rewrite (f_num: integer; f_name: strType):integer;
begin
  case f_num of
    1: rewrite (f1,f_name);
    2: rewrite (f2,f_name);
  end;
  f_rewrite := ioresult;
end;

function f_reset (f_num: integer; f_name: strType): integer;
begin
  case f_num of
    1: reset (f1,f_name);
    2: reset (f2,f_name);
  end;
  f_reset := ioresult;
end;

procedure writeLine (f_num: integer; var S: strType);
begin
  case f_num of
    0: write (s);       {file id = 0 means write to -console}
    1: write (f1,s);
    2: write (f2,s);
  end;
end;


procedure writeLF (f_num: integer; var S: strType);
begin
```

```
  case f_num of
    0: writeln (s);
    1: writeln (f1,s);
    2: writeln (f2,s);
  end;
end;

procedure f_close (f_num: integer; lock_file: boolean);
begin
  case f_num of
    1: if lock_file then
         close (f1,lock)
       else
         close(f1);
    2: if lock_file then close(f2,lock)
       else close(f2);
  end;
end;

{THE MAIN PROGRAM CALLS THE ASSEMBLY LANGUAGE MAIN}

begin
  writeln ('test program - using assembly main routine to do I/O');
  writeln;
  main;
  write ('press any key to continue');
  read (input,ch);
end.
```

The assembly language file is:

```
            .proc   main
;========================================
;   EXTERNAL REFERENCES AND CONSTANTS
;========================================

            .ref    writeLF
            .ref    writeLine
            .ref    f_rewrite
            .ref    f_reset
            .ref    f_close

first_file  .equ    1               ; id # of file one
printerId   .equ    2               ; id # of file '-printer'

; return address to the Pascal main routine is left on the stack
```

```
;======================================
;   MACROS TO CALL PASCAL FUNCTIONS
;======================================

            .macro   open_write_file
            ;    %1 --- file #
            ;    %2 --- file name
            clr      -(a7)            ; reserve space for function
                                      ; result from f_rewrite
            move     #%1,-(a7)        ; file id # as first param
            lea      %2,a0            ; second param is file name
            move.l   a0,-(a7)
            jsr      f_rewrite
            move     (a7)+,a0         ; pop IOresult
            ble      @1
            error    %2               ; IOresult > 0 -> error
                                      ; (nested macro call)
@1          .endm

            .macro   open_read_file
            ;    %1 --- file #
            ;    %2 --- file name
            clr      -(a7)            ; reserve space for function
                                      ; result of f_reset
            move     #%1,-(a7)
            lea      %2,a0
            move.l   a0,-(a7)
            jsr      f_reset
            move     (a7)+,a0         ; pop IOresult
            ble      @1
            error    %2               ; IOresult > 0 -> error
@1          .endm

            .macro   write_file       ; write a line (with no linefeed)
            ;    %1 --- file #
            ;    %2 --- label of string to be written
            move     #%1,-(a7)
            lea      %2,a1
            move.l   a1,-(a7)         ; push string address onto stack
            jsr      writeLine        ; write it out
            .endm


            .macro   writeln_file     ; write a line of text with
                                      ; linefeed
            ;    %1 --- file #
            ;    %2 --- label of string to be written
```

```
          move      #%1,-(a7)
          lea       %2,a1
          move.l    a1,-(a7)        ; push string address onto stack
          jsr       writeLF         ; write it out
          .endm

          .macro    close_file
          ;    %1 --- file #
          ;    %2 --- close status code
          ;            0 - $00ff          normal close
          ;            $0100 - $ffff      lock
          move      #%1,-(a7)
          move      #%2,-(a7)
          jsr       f_close
          .endm

          .macro    error
          ;    %1 --- file name
          write_file  0,errStr      ; write error message
                                    ; to -console
                                    ; (file id # 0)
          writeLn_file 0,%1         ; output file name also
          rts                       ; quit
          .endm

;======================================
;   MAIN ASSEMBLY LANGUAGE PROGRAM
;======================================

      open_write_file  first_file,file1    ; open IO/record.text
      open_write_file  printerId,printer

      writeLn_file     0,openstr           ; write the openstr
                                           ; to -console (file # 0)
      writeLn_file     first_file,string   ; write string to
                                           ; first_file
      writeLn_file     printerId,str1      ; write str1 to printer

      close_file       first_file,$0100    ; lock first_file
      close_file       printerId,0         ; do not lock the printer

      open_read_file   1,file1             ; no error should occur
      close_file       1,$ffff             ; preserve file1

      open_read_file  2,errFile            ; no errFile around, should
                                           ; cause error.
```

```
        rts                                   ; back to Pascal main
                                              ; program

;===============
;   CONSTANTS
;===============

file1       .byte    14
            .ascii   'IO/record.text'
            .align   2

printer     .byte    8
            .ascii   '-printer'
            .align   2

string      .byte    38
            .ascii   'this string is from the Lisa Assembler'
            .align   2              ; make sure on even memory


str1        .byte    34
myStr       .ascii   'another string from Lisa Assembler'
            .align   2

openstr     .byte    26
            .ascii   'opened file IO/record.text'
            .align   2

errStr      .byte    22
            .ascii   'error in opening file '
            .align   2

errFile     .byte    6
            .ascii   'noFile'
            .align   2

            .end
```

6.7.5 Using Pascal Data Areas

Assembly language routines are in read only segments and do not have a data area. Any data area that must be written into must be declared in the Pascal program and referenced in the assembly language program by pointers. The following two examples illustrate the correct and incorrect ways of doing this. The correct example illustrates how to do a READLN from an assembly language program.

The first example illustrates the "obvious" and *incorrect* way of doing a READLN from an assembly language program. The Pascal program is as follows:

```
program ASMDemo;

{ BAD EXAMPLE: Note that this example does not work, because
  it tries to write into a memory space reserved by the
  Assembler. Data space must be set up in the Pascal program
  and referenced by a pointer variable. The following example
  illustrates the correct way of doing this. }

  type
    PasStr = string[255];
  var
    ch: char;

  procedure w_write(S: PasStr);
    begin
      write(s);
    end;

  procedure w_writeln;
    begin
      writeln;
    end;

  procedure w_readln(var s: PasStr);
   { read a line from -CONSOLE and put it into
      (write to) string s }
    begin
      readln(s);
    end;

  procedure main; external;

  begin {ASMDemo}
    main;          { call to assembly language routine }
    write('That''s all folks, type space to continue');
  repeat read(ch); until ch = ' ';
end. {ASMDemo}
```

This is the corresponding *incorrect* assembly language program:

```
.proc       main

.ref        w_write, w_writeln, w_readln

.macro      a_write         ; (s: passtr)
                            ; %1 = string label
lea         %1, a0
move.l      a0, -(a7)
jsr         w_write
.endm

.macro      a_writeln       ; no parameters

jsr         w_writeln
.endm

.macro      a_readln        ; (var s: passtr)
                            ; %1 = string label
            ; ===================================
            ; Put the address of the string into
            ; which a line is to be read on the
            ; stack and call Pascal routine to
            ; read the string.
            ; ===================================
lea         %1, a0          ; This space has been
                            ; reserved for the string.
move.l      a0, -(a7)
jsr         w_readln
.endm

; ==============================
; MAIN ASSEMBLY LANGUAGE PROGRAM
; ==============================

a_write     string1         ; This will write a string
a_writeln                   ; and a newline.
a_write     hello
```

```
                                         ; ========================
            a_readln   stringspace       ; NOTE: this will fail
                                         ; with a bus error
                                         ; because stringspace is
                                         ; in program space (read
                                         ; only), not in read/write
                                         ; memory space.
                                         ; ========================
            a_writeln  stringspace
            rts

hello       .byte      13
            .ascii     'Type a line: '
            .align     2

StringSpace .block     256               ; Save some space for a
                                         ; readln.  This block of
                                         ; memory is in program
                                         ; space, therefor it is
                                         ; read only.
            .align     2

String1     .byte      39
            .ascii     'This string is from the Lisa Assembler.'
            .align     2

            .end
```

This is the *correct* way of doing a READLN from an assembly language program. Note that the string "s", declared in the Pascal program, is used in the w_readln function and passed to the assembly language program by pointer.

```
program ASMDemo;

{ GOOD EXAMPLE: This example does a readln by using a pointer
  variable as a parameter.  This allows the string to be
  reserved by the Pascal Compiler. }

 type
   PasStr = string[255];
   ByteP  = ^PasStr;

  var
   s: PasStr;  { this string is allocated in read/write
                 memory by the Pascal Compiler }
```

```
    ch: char;

  procedure w_write(S: PasStr);
    begin
      write(s);
    end;

  procedure w_writeln;
    begin
      writeln;
    end;

  function w_readln: ByteP;
    { This function reads a line into the string s (space
      allocated by the Pascal Compiler in read/write memory
      segment) and returns address of s to assembly routine }
    begin
      readln(s);
      w_readln := pointer (@s);
    end;

  procedure main; external;

  begin {ASMDemo}
    main;           { call to assembly language routine }
    write('That''s all folks, type space to continue');
    repeat read(ch); until ch = ' ';
  end. {ASMDemo}
```

This is the *correct* assembly language program:

```
            .proc     main

            .ref      w_write, w_writeln, w_readln

            .macro    a_write        ; (s: passtr)
                                     ; %1 = string label
            lea       %1, a0
            move.l    a0, -(a7)
            jsr       w_write
            .endm

            .macro    a_writeln      ; no parameters

            jsr       w_writeln
            .endm
```

```
                .macro   a_readln       ; function w_readln: ByteP;
                         ; ========================================
                         ; This function expects the Pascal routine
                         ; w_readln to return the pointer to the
                         ; string in which a line has been read
                         ; ========================================
                clr.l    -(a7)
                jsr      w_readln
                .endm


                a_write  string1        ; this will write a string
                a_writeln               ; and a newline
                a_write  hello
                a_readln                ; leaves the address of
                                        ; string read at top of
                                        ; stack
                jsr      w_write        ; takes top of stack as
                                        ; parameter
                a_writeln
                rts

hello           .byte    13
                .ascii   'Type a line: '
                .align   2

String1         .byte    39
.ascii          'This string is from the Lisa Assembler.'
.align          2

                .end
```

NOTES

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.OBJ 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.

When a program is compiled into an object file, it contains the following sorts of things:

- Object code, in the form of relocatable machine language, that expresses the algorithm of the program.
- Symbolic (named) references to all locations that were not known at compile time. These include externally compiled routines (units and intrinsic units) and the Pascal library support routines (IOSPASLIB.OBJ).
- Other information to be used by the Linker.

The purpose of the Linker is to resolve all the symbolic references (link references to definitions), and output an object file that can be executed. The Linker also sorts the code modules into named segments. These segments are swapped into memory at run time by the Operating System.

The Linker does its work in two phases. In the first phase, it reads all the input files, and finds all symbolic references and their corresponding definitions. Errors such as duplicate and missing references are detected during phase one. In the second phase, the Linker copies code from the input files into the output files in executable format.

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. 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.

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

- The object file from a main Pascal program.
- IOSPASLIB.OBJ to provide the standard Pascal procedures and functions.

- IOSFPLIB.OBJ, if you are using any floating point variables.
- Object files for any other external procedures referenced by the main program. These can be Pascal units, assembly language routines, or intrinsic units defined in INTRINSIC.LIB.

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 displays an error message. If there is an error, the object file is not produced.

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

While it is linking a main 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 regular units 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 can be entered after entering "?" in response to the input file prompt. After all file names and options are entered, the link begins. Hence the set of options in effect is 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 as needed for input files.

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 a blank line exists 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. It is not necessary to enter the .TEXT extension; the Linker will provide that as needed for all input command files. Create the text file by using the Editor.

The default listing is -console. You can send the listing to a text file by entering its name in response to the listing file prompt. When sending the listing to a text file, you do not need to provide the .TEXT extension, since the Linker provides it.

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

To enter the Linker options mode, type "? [RETURN]" in response to the prompt for an input file. To leave options mode and return to entering input files, press [RETURN] in response to the options prompt. 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 or an option.

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 initial disk space allocated to the program's stack. The default is to automatically include space for the program variables and the value specified in the +S option.
+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 segmentation decisions 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. Also, you cannot use this option to map segments to or from the blank segment.

+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:

- A compiled Pascal PROGRAM object file.
- Object files for any other 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.OBJ, and IOSFPLIB.OBJ (if any real variables are used).

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 displays a header and asks you for an input file.
2. Enter any desired options. To enter the options mode, press "? [RETURN]" in response to the request for an input file. See Section 7.3 in this chapter for information on Linker options. Press [RETURN] after each option entered. When you have entered all the options, press [RETURN] to begin entering input file names.
3. Enter the file names for all the object files, pressing [RETURN] after each one. The file names can be entered in any order. You do not need to enter the .OBJ extension; the Linker will automatically append 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. You do not need to enter the .OBJ extension; it is 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 are displayed.

7.5 Regular and Intrinsic Units

The two types of units are regular units and intrinsic units. Each is a separately compiled code module 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 matter how many main programs use it.

In addition to being shared on the disk, an intrinsic unit is also shared in memory.

NOTE

The current implementation has no provision for users to create new intrinsic units. All intrinsic units are supplied by Apple Computer.

7.5.1 How Do I Link with a Regular Unit?

A regular unit is a separately compiled segment of code. It is written in Pascal, and compiled like a regular program. 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 can be accessed from any other program or regular unit you write. The Linker combines 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.
- IOSPASLIB.OBJ, and IOSFPLIB.OBJ (if any floating point variables are used).

The Linker combines all these object files into an executable object file. It also does a dead code analysis to eliminate any routines that are not used, to reduce the size of the object file.

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 gives 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.

Errors are reported as they are found, and you are told whether or not the output is executable.

If you requested optional listings, they are also printed. An example of a Linker listing with no options requested is shown in Figure 7-1. Linker listings are mainly used for debugging at the machine code level. See Chapter 8 for more information on the Debugger.

```
Beginning memory -   262488
After static allocation, memory -   106815
Input file [.OBJ] ? TRANSVOL
Input file [.OBJ] ? IOSPASLIB
Input file [.OBJ] ?
Listing file [CONSOLE:]/[.TEXT] -
Output file [.OBJ] - TRANSFER_FLS
Reading file: TRANSVOL.OBJ
Reading file: IOSPASLIB.OBJ
Read     2 files, max =   100
         4 segments, max =   128
        16 modules, max =  1450
        32 entries, max =  2000
        38 ref. lists, max =  8000
       124 references, max = 16000
Linking Main Program.
Active: 4 of 16 read.
Visible: 1 of 32 read.
Global data: $00067C
Common data: $000000
Linking segment #: 0 :          file (JT) seg: 1 size:     2908
  Beginning memory -   104487
  Ending memory -   104032
   0 Errors detected.

The output is executable.
Elapsed time: 298 and 304/1000  seconds.
That's all Folks !!! . . .
```

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. Local names are generated by the Pascal compiler. A *global name* is any name that is not a local name.

The scope of a global name is the entire program being linked. Unsatisfied references to global names are not allowed. Only one definition of a given global name can occur in a given link. The one exception to this is that the Linker accepts 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. All references to a given local name must occur within the same input file. 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.

7.8 Module Inclusion

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 called.

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 affects its performance.

Segmentation is controlled by three things:

- The $S Compiler command and the .SEG Assembler option, which assign segment names to source code modules.
- The +M Linker option, which enables you to remap compiler segment names into new segment names.
- The ChangeSeg utility, which enables changing the segment names prior to linking. See Chapter 10 for information on ChangeSeg.

NOTES

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 contains 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. Section 6.6 in this manual contains more information about the run-time environment of programs.

When you enter the Debugger, the Debugger screen is made visible by the Debugger. You can display the main screen by pressing [OPTION] and [ENTER] to see the state of the program before the Debugger was entered. Redisplay the Debugger screen (by pressing [OPTION]-[ENTER] again) to continue with debugging.

8.2 Inadvertent Entry into the Debugger

Accidental entry into the Debugger can be caused by a bug in the program you are running or by some malfunction in the system. A message from the Debugger will suggest the type of problem. The messages and the actions you can take for program bugs are described in Section 8.2.1 below. System malfunctions are described in Section 8.2.2.

8.2.1 Program Bugs

You can enter the Debugger while your program is executing for any of the following reasons. More information on these conditions can be found in the *MC68000 16 Bit Microprocessor User's Manual.*

- A value range error
- An illegal string index
- A bus error or address error
- An illegal instruction or a privilege violation
- Integer division by zero
- Spurious interrupt or unexpected exception
- Overflow when TRAPV is executed
- Line 1111 Emulator

- System malfunction
- Intentionally, by pressing the NMI key. This is the way to terminate an infinite loop (when -period doesn't stop your program). Do not use NMI when running system programs.

Usually the system will tell you the most appropriate action to take, for example, "type g to continue". Follow these instructions unless you have a special reason for doing something different.

Programming errors are described in Section 8.2.1.1 below. Stopping an infinite loop is described in Section 8.2.1.2 below.

8.2.1.1 Program errors

If you have an error in your program it will drop into the Debugger and display one of the following messages:

If a range check error occurs in application code, the message displayed is:

```
VALUE RANGE ERROR in process gid <gggg>
value to check = <vvvv>  lower bound = <nnnn>  upper bound = <uuuu>
return pc = <pppppp>  caller a6 = <cccccc>
Going to Lisabug, type g to continue.
```

or:

```
ILLEGAL STRING INDEX in process of gid <gggg>
value to check = <vvvv>  lower bound = <nnnn>  upper bound = <uuuu>
return pc = <pppppp>  caller a6 = <cccccc>
Going to Lisabug, type g to continue.
```

where:

<gggg>	is the global process ID of the process that incurred the exception.
<vvvv>	is the value that is outside the range.
<nnnn>	is the lower bound of the range.
<uuuu>	is the upper bound of the range.
<pppppp>	is the address of the statement after the call to the range check routine in Paslib.
<cccccc>	is the address of the link field at the time of the call to Paslib.

During execution applications can field hardware exceptions. Refer to the *MC68000 16 Bit Microprocessor User's Manual* for definitions of these hardware exceptions. If such an exception occurs, the system displays one of the following messages:

Bus error or address error exception:

```
EXCEPTION in process of gid <gggg>
Process is about to be terminated.
access address = <aaaaaaaa> = mmu# <mmm> (segment name), offset
<oooo>
inst reg = <rrrr>        sr = <ssss>       pc = <pppppp>
saved registers at <xxxxxxxx>
Going to Lisabug, type g to continue
```

Any other hardware exception:

```
EXCEPTION in process of gid <gggg>
Process is about to be terminated.
sr = <ssss>      pc = <pppppp>
saved registers at <xxxxxxxx>
Going to Lisabug, type g to continue
```

where:

EXCEPTION	is one of: BUS ERROR ADDRESS ERROR ILLEGAL INSTRUCTION PRIVILEGE VIOLATION SPURIOUS INTERRUPT UNEXPECTED EXCEPTION ZERO DIVIDE CHK RANGE ERROR OVERFLOW LINE 1111 EMULATOR
<gggg>	is the global ID of the process that incurred the exception.
<aaaaaaaa>	is the address that caused the bus or address error
<mmm>	is the segment number represented by <aaaaaaaa> and
<oooo>	is the offset within that segment
<rrrr>	is the value of the instruction register at the time of the exception
<ssss>	is the value of the status register at the time of the exception
<pppppp>	is the value of the program counter at the time of the exception
<xxxxxxxx>	is the address of the saved register information

All numbers displayed are decimal; the segment name is displayed only if the segment number makes sense to the Operating System.

If the exception is divide by zero, overflow, or CHK out of bounds, the process is not terminated and the line to that effect is not shown. If the process has declared an exception handler for this exception, control passes to the handler after you type g to LisaBug, and the process then continues execution. If no handler has been declared, the system default handler terminates the process. If the exception is a bus error and the segment name is 'stack seg', a stack overflow has probably occurred. To find your bug you can do a SC (stack crawl) and IL (immediate disassemble) to find where you are in the program. The instruction register tells you the exact instruction being executed. The PC might be 2 to 10 bytes ahead.

You can declare an exception handler in your program to handle divide by zero, overflow, or CHK out of bounds exceptions. Then your process will not be terminated by the system if this type of exception occurs. You can also declare an exception handler for the "SYS_TERMINATE" exception in your program. This exception handler will then get executed if your process has a fatal error as described above. This allows you to clean up your program, close your files, etc. (in this exception handler) before your program is terminated. See the *Operating System Reference Manual* for the Lisa for how to declare an exception handler.

8.2.1.2 Terminating an Infinite Loop

NOTE

The following procedure should be used on user programs only. To terminate a systems program use ⌘-period.

If your program is in an infinite loop, or appears to be doing nothing, you can enter the Debugger by pressing the NMI key (the - key on the numeric keypad). This will put you into the Debugger and show the trace display, which looks something like:

```
Level 7 Interrupt
aaaaaaaa           bbbb                        <instr>
PC=xxxxxxxx SR=xxxxxxxx US=xxxxxxxx SS=xxxxxxxx D0=d PROC=yyy
D0=xxxxxxxx D1=xxxxxxxx D2=xxxxxxxx D3=xxxxxxxx
D4=xxxxxxxx D5=xxxxxxxx D6=xxxxxxxx D7=xxxxxxxx
A0=xxxxxxxx A1=xxxxxxxx A2=xxxxxxxx A3=xxxxxxxx
A4=xxxxxxxx A5=xxxxxxxx A6=xxxxxxxx A7=xxxxxxxx
>
```

where:

aaaaaaaa	is the current address
bbbb	is the contents of the current address
<instr>	is the current instruction disassembled
xxxxxxxx	is the contents of the specified register
d	is the current domain (0 - 3)
yyy	is the process ID of the interrupted process

This information is used in debugging your program. If your program is in an infinite loop, proceed as follows:

1. Check the domain (D0=d). If the domain is zero, you are currently executing in system code. You must be executing user code before you can work on your program (domain 1 - 3). See Section 8.2.1.3 "User Break" below for a procedure to get you into user code.
2. Make sure you are in your own process, instead of another process that may be running in the background. If the current address does not show the name of one of your procedures, type SC (stack crawl). The procedure names displayed should be from your program.
3. If you are in a tight loop you can step the PC beyond it by using other Debugger commands. In order to do this you must be familiar with 68000 assembly language and the Debugger commands. Most often you will just want to stop your program. This is explained below.
4. First make sure the domain is not zero. Type "PC 0" and press [RETURN]. This will cause an exception when you restart your program.
5. Type "G" and press [RETURN]. Your program will restart, cause an exception, and immediatly drop back into the Debugger with an exception message that includes the instructions "Type g to continue".
6. Type "G" and press [RETURN]. Your program will be terminated.

8.2.1.3 User Break

The user break facility stops processing in user process code. Use this procedure if the trace display indicates that the domain is zero. (Either DOMAIN=0 or DOMAIN = n OVERRIDDEN TO 0.) The UBR command will set a breakpoint at the next instruction to be executed in the user process. To stop your program in user process code, proceed as follows:

1. Type "UBR" and press [RETURN].
2. The system will continue executing until it returns to user process code, then it will drop back into the Debugger. You can now proceed to work on your code.

NOTE

There are two cases when UBR will not set a breakpoint. The first is if the system is interrupted while a system process is running (PROCESS = 0, 1, or 2). The second is if the system is interrupted while the scheduler is running and it has not chosen a process to run. If UBR does not seem to be working, check for this as follows:

Type "ID PC-4" and press [RETURN]. If the STOP instruction is displayed, you are in the scheduler. You must press "G" and return to start the system running again and press NMI again.

If your program is doing a READ or READLN, the system will display the STOP instruction. The only way to continue execution is to press "G" and enter something from the keyboard to satisfy the read.

8.2.2 System Malfunctions

If there is a system malfunction, the system will enter the Debugger with a message indicating a system error or an EXCEPTION display with the domain zero. The message will include instructions telling you what command to type. Ususally it will tell you to type OSQUIT. It may be necessary to type this command several times.

If you are having problems with system malfunctions, call your support hotline for more information. It will be useful to have copies of the messages that were displayed. If you have a printer connected to the lower or upper port, use PL or PU to generate a bug report.

8.3 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 is run with a breakpoint set at the first instruction and drops 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 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
  - Print Debugger information

8.3.1 Examples of Using the Debugger

This section gives examples of how to use the Debugger. An explanation of all Debugger commands is in Section 8.4. A summary of all Debugger commands is in Section 8.5.

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 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 breakpoint 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.

8.3.2 A Pascal Example: Range Errors

The Debugger can be used for run-time debugging of Pascal programs. Its displays and commands reference Pascal procedure names to make it easier to debug programs. If your program has a fatal run-time error, it will drop into the Debugger and give you a trace display. The trace display will include the name of the procedure that was executing.

One common reason for dropping into the Debugger is if you get a range error. Range errors can be caused by array indexes, string value parameters, and assignments to variables of a subrange type. If you get a range error, you will drop into the Debugger with the RANGE ERROR exception message.

To help find the error in your program, give the Debugger an IL PC-20 command. This will give you a display of the previous 20 lines of assembly code. You should see an instruction of the form:

```
CHK       #<lim>,<data reg>
```

where <lim> is an integer, and <data reg> is a data register (D0 - D7). Lim is the allowable value. The contents of the data register is the actual value that was out of range. The contents of all the registers can be displayed with the TD (trace display) command.

Figure 8-1 shows a Pascal program that produces a check range error. Figure 8-2 shows the resulting Debugger display, with an explanation of what the display means.

```
program check;
var ch:char;

procedure localproc;
var
   i:integer;
   a:array[0..10] of 1..7;
begin
  i := 9;
  a[3] := i;
end;

begin
  writeln('press space to run...');
  read(ch);
  localproc;
end.
```

Figure 8-1
Pascal Program that Produces a Check Range Error

```
     CHK RANGE ERROR in process of gid      25
     sr =         0  pc =  2359330
(1)   saved registers at 13369278
     Going to Lisabug, type g to continue.

     Level 7 Interrupt                  (2)
     LOCALPRO+001A 1D40 FFF5        PC        MOVE.B  D0,$FFF5(A6)
     PC=00240022 SR=0000  0  US=00F7FBEC SS=00CBFEE0 DO=1 P#=00019
     D0=00100009 D1=00000008 D2=000000C0 D3=000264A7
     D4=00000001 D5=4EF90084 D6=12CC4EF9 D7=00840000
     A0=00F8126E A1=00CCA22A A2=00240060 A3=00CCA22A
     A4=00CCA22A A5=00F7FC44 A6=00F7FBFA A7=00F7FBEC
(3)  >il pc-20
     00240002      00A4 0024 0000 4A6F  EFF2 4E56 FFF2 3D7C ...$..Jo..NV..=|
     LOCALPRO+0000 4A6F EFF2        LOCALPRO TST.W   $EFF2(A7)
     LOCALPRO+0004 4E56 FFF2                 LINK    A6,#$FFF2
     LOCALPRO+0008 3D7C 0009 FFFE            MOVE.W  #$0009,$FFFE(A6) (5)
     LOCALPRO+000E 302E FFFE                 MOVE.W  $FFFE(A6),D0
     LOCALPRO+0012 3200                      MOVE.W  D0,D1       (7)
     LOCALPRO+0014 5341                      SUBQ.W  #$1,D1          (6)
     LOCALPRO+0016 43BC 0006                 CHK     #$0006,D1
     LOCALPRO+001A 1D40 FFF5        PC   (4) MOVE.B  D0,$FFF5(A6)
     LOCALPRO+001E 4E5E                      UNLK    A6
     LOCALPRO+0020 4E75                      RTS
     >pl
```

Figure 8-2
Check Range Debugger Display

Notes:

1. Debugger display produced by check range error.
2. Actual value in D1. This is the value that was checked and found out of range.
3. Disassembly command typed in to display the assembly language display of the program causing the error.
4. Look for the CHK instruction near the PC.
5. Note that the previous identifier is LOCALPRO, therefore the error occurred near the beginning of LOCALPRO.
6. Value in register D1 was supposed to be in range 0..6.
7. Pascal lower limit (#$1) was subtracted from D1. Therefore the range in the Pascal type was 1..7.

More information on the run time environment of a Pascal program is found in Chapter 6.

8.4 The Debugger Commands

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

8.4.1 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+ was 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.

8.4.2 Display and Set Memory Locations

The following commands 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. Expr1 must be an even address, or the address will be rounded down to the nearest even address.

SL expr1 exprlist
Set memory in long words with exprlist starting at expr1. Expr1 must be an even address or it will be rounded down to the nearest even address. 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. Expr must be an even address or it will be rounded down to the nearest even address.

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. Expr can be any byte address.

DW expr
Display memory as words. Expr must be an even address or it will be rounded down to the nearest even address.

DL expr
Display memory as long words. Expr must be an even address or it will be rounded down to the nearest even address.

8.4.3 Finding Patterns in Memory

FB expr1 expr2 exprlist
Find Byte. Find the byte or bytes 'exprlist' in the address range specified. If expr 1 < expr2 then search the range from expr1 to expr2. Otherwise search for expr2 bytes starting at expr1.

FM expr1 expr2 exprlist
Find Memory.

FW expr1 expr2 exprlist
Find Word.

FL expr1 expr2 exprlist
Find Long word.

8.4.4 Set and Display Registers

TD
Display the Trace Display at the current PC. An example of the trace display is shown in Figure 8-3. 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.

```
█
Level 7 Interrupt
LOCALPRO+001A 1D40 FFF5             MOVE.B  D0,$FFF5(A6)
PC=00240022 SR=0000  0  US=00F7FBEC SS=00CBFEE0 DO=1 P#=00010
D0=013C0009 D1=00000008 D2=000000C0 D3=00199752
D4=00000001 D5=53656750 D6=78487A20 D7=00000000
A0=00F8126E A1=00CCB614 A2=00240060 A3=00CCB614
A4=00CC75FC A5=00F7FC44 A6=00F7FBFA A7=00F7FBEC
```

Figure 8-3
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.

8.4.5 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. Press just [RETURN] to exit the immediate assembler. Note that the immediate assembler cannot assemble any intrinsic unit instructions, but they are correctly disassembled. Code segments can be write protected, which prevents 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 user's PC is not changed by this operation.

8.4.6 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. Breakpoints 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 you can:

```
BR Ralph+4
```

if Ralph is a known symbol.

Expressions can be of the form:

```
pp:aaaaa
```

where pp is the process ID, and aaaaa is the address in that process where you want the breakpoint set. If the process ID 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. The Stack Crawl display is shown in Figure 8-4. More information on the Pascal stack can be found in Section 6.6.

```
>sc
At LOCALPRO+001A
Stack frame at 00F7FBFA called from CHECK+0038
Stack frame at 00F7FC44
>
```

Figure 8-4
The Stack Crawl Display

procedure name
This calls a user procedure or function. It is your 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 breakpoint 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) or CLR.L -(A7).

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

UBR
UBR is a procedure that sets a breakpoint in the user code so that you will drop into the Debugger as soon as you reenter user code. UBR is explained in Section 8.2.1.3.

8.4.7 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*.

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. Note that if you change domain, you should restore the original domain before you type g.

WP 0 or 1
Disable (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.

8.4.8 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 and enter the process number that you want to time using the BT command.
2. Create one or more timing buckets with the TB command.
3. Set a breakpoint to stop execution at some point.
4. Go.
5. When the breakpoint is reached, print the timing summary with the PT command.
6. 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 expr 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.

8.4.9 Utility functions

The utility functions include:

- Symbol and base conversion
- Moving the Debugger window
- Setting the NMI key
- Printing Debugger displays
- Dumping memory to a diskette

8.4.9.1 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.

8.4.9.2 Moving the Debugger Window

CS
The CS command clears the Debugger screen.

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

0	Lisa keyboard and screen (default)
1	Serial A
2	Serial B

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

RS
Display the patch Return address Stack

8.4.9.3 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.

NOTE

This affects the entire system. If the NMI key is disabled, you cannot use it to stop an infinite loop, or a system hang.

For example:

```
>NM $21
```

Sets the NMI key to be hex 21, which is the "-" key in the top row of the numeric keypad. This is the default NMI key.

8.4.9.4 Printing from the Debugger

The following commands allow you to print information from the Debugger on the dot matrix printer.

PR expr
The PR command enables or disables printing to the two-port card. When printing is enabled, all Debugger output to the screen is printed.

expr = 1	enable printing upper port
expr = 2	enable printing lower port
expr = 0	disable printing

NOTE

The Debugger only supports printing to a printer connected to the lower or upper port. The serial printer is not supported. If the printer is not connected the Debugger will hang when you try to print with the PL, PU, or PS command.

PS expr
The PS command prints the entire primary or alternate screen. Printing must be enabled (the PR command) before PS is used. Expr tells which screen to print:

expr = 1	print primary screen
expr = 0	print alternate screen

FF
The FF command sends a form feed to the printer if printing is enabled.

PL and PU
The PL and PU commands print a bug report on the lower and upper ports respectivly. The bug report consists of the following:

Dump of the primary screen
Dump of the alternate screen
Description of the exception
Trace Display
Stack Crawl
Disassemble of 20 lines from PC-$20
Display words from RA6-$20 for $80 bytes

8.4.9.5 Dumping Memory to Diskette

The following commands allow you to create a copy of the contents of memory on a diskette.

ML and MU
The ML and MU commands dump a copy of memory to the lower and upper diskette respectivly. This information can be used to reconstruct the conditions at the time of a crash, for example. These commands work as follows:

- If there is a disk in the drive, it is ejected.
- You are prompted to insert a disk.
- The disk is formatted and all necessary information is copied to it. This process takes about 3 1/2 minutes.

8.5 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	Assemble statement at expr.
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 expr1 expr2 exprlist	Find Byte.
FF	Send form feed to printer
FL expr1 expr2 exprlist	Find Long word
FM expr1 expr2 exprlist	Find Memory
FW expr1 expr2 exprlist	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.
ML	Dump memory to lower diskette
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.
MU	Dump memory to upper diskette
NM	Displays the keycode of the NMI key
NM expr	Sets NMI keycode to expr
OSQUIT	Exits from the operating system *
P expr	Set port number to expr.
PL	Print bug report on lower port
PR expr	Enable printing. 0=disable, 1=upper port, 2=lower port.

PS expr	Print screen. 0=aletmate, 1=primary
PT	Print timing summary
PU	Print bug report on upper port
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
UBR	User break*
WP 0 or 1	Disable (0) or Enable (1) Write Protection.

* These are procedure calls to Operating System procedures. They are explained in Section 8.2.

NOTES

Chapter 9
Exec Files

Using Exec Files

9.1 Exec Files

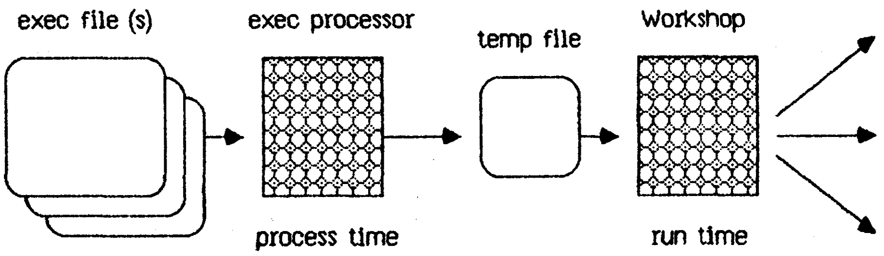
Exec files are scenarios of commands to the Workshop system. They are contained in text files, created with the Editor, and are executed with the Run command. Exec files consist of characters you type to the Workshop to perform the functions you want, and special exec file commands, which enable you to use parameters and conditions to vary portions of the scenario.

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

```
$EXEC
        Pmyprog
                { You need to enter two blank lines here }
                { to run the Compiler }
$ENDEXEC
```

where P is the command to invoke the Pascal Compiler, and myprog is the name of the source file. Further lines to Generate, Link, and Run the program might follow.

Two separate activities occur while running an exec file: processing and running. First, during *process time*, the exec processor creates a *temporary file*, which consists of a stream of Workshop commands. This temporary file is then sent to the Workshop, which executes the command stream at *run time*. A simple diagram of this procedure follows:



With special exec file commands, you can use parameters and conditionally perform the Workshop commands. An example of an exec file for a simple Pascal program is shown in Figure 9-1.

```
$EXEC { "makeprog" -- This exec file compiles, generates, and
          links a Pascal program.  }
   P%0
   { no listing file}
   { default I-code file }
   G%0
   {default object file}
   L%0
     IOSPASLIB
     { end of linker input }
     { no list file }
     %0{ output file name }
$ENDEXEC
```

Figure 9-1
Example Exec File

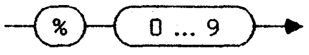
You have several options available to you when running the exec file processor. The Step Mode option, which enables you to selectively skip command lines going to the temp file, could be used in the above example to choose whether to do only the compile, generate, or link. Section 9.3.1 contains additional information on the exec file options.

9.2 Exec File Statements

Exec file statements are line oriented. Two types of exec file lines exist: *exec command lines* and *normal lines*. Normal lines contain Workshop commands. Exec command lines handle the other features of exec files, such as parameters and conditional statements.

You can use up to 10 parameters in an exec file, numbered %0 through %9.

parameter



You can pass parameters when you invoke an exec file and use them during the execution of the exec file. For example, if you wanted to pass a parameter in the Example Exec File shown in Figure 9-1, you would Run:

<makeprog (myprog)

The value "myprog" would then be assigned at each reference to %0.

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 **$** (dollar sign). They control the operation of the rest of the exec file. Exec command lines are free format, as long as the order of their elements is preserved. You can have any number of spaces before or after any element of a command line. These can go on to more than one line. The processor will look on the next line if it does not have a complete command at the end of a line.

Normal lines contain commands for the Workshop system. These lines are sent to the Workshop as they appear, with the following exceptions:

1. Leading and trailing blanks are removed from these lines unless the "B" option is in effect. See section 9.3.1 for more on the "B" option.
2. Comments are removed.
3. Parameters are expanded.
4. The tilde (~) literalizing character is processed.

Comments are delimited by brackets { }, and can appear in either a normal or an exec command line. These can cross line boundaries. They can be used to comment out carriage returns in normal lines.

The "~" is used as a literalizing character in normal lines, meaning it passes the character following it through without processing. With a tilde you can pass the character **$**, **%**, or { to the Workshop system without having it be interpreted as part of an exec command, a parameter, or a comment. To represent a tilde, use a double tilde (~ ~).

Note that while the exec file processor is not case sensitive, it does preserve the case of parameters and strings supplied by the user.

A description of each exec command follows.

9.2.1 Beginning and Ending Exec Files

Generally, exec files must begin with an **EXEC** line and must end with an **ENDEXEC** line. The exceptions to this basic rule, for those who embed exec files in their program sources, are: (1) one line of text can preceed the **EXEC** line if the I (Ignore) invocation option is used, and (2) any amount of text can follow the **ENDEXEC** line, but it is ignored.

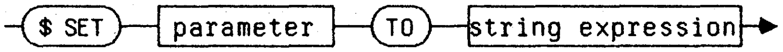
9.2.2 Setting Parameter Values

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

9.2.2.1 The SET and DEFAULT Commands

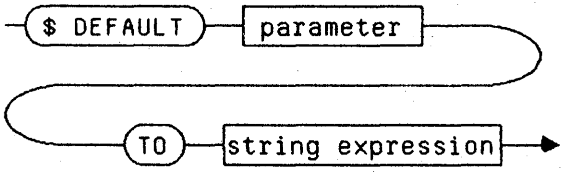
The **SET** and **DEFAULT** commands provide ways to change the value of a parameter inside of an exec file. The forms of these commands are:

`set statement`



and

`default statement`



"String expression" is described in Section 9.2.5.

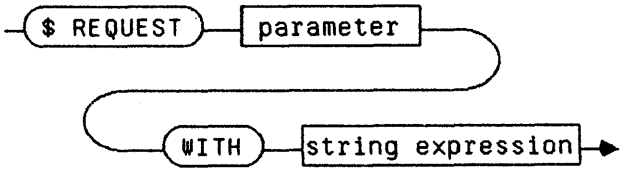
The **SET** command changes the value of the specified parameter to the value of the given string expression. The **DEFAULT** command is similar to the **SET** command, except that the assignment takes place only if the value of the specified parameter is the null string when the **DEFAULT** command is encountered. Thus, you can use this command to supply default values to parameters that have been left unspecified or empty in the exec invocation line.

These commands also allow you to use unused parameters as variables within the exec file.

9.2.2.2 The REQUEST Command

The **RFQUEST** command provides a way to prompt for values from the console. The form of this command is:

`request statement`



The REQUEST command prints the given string expression to the console, and reads a line, which it assigns to the specified parameter, from the console. Thus, "str expr" prompts the user for the value.

9.2.3 Input and Output

You can request input to an exec file with the **READLN** and **READCH** commands. You can output values by using the **WRITE** and **WRITELN** commands.

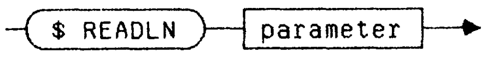
9.2.3.1 The READLN and READCH Commands

The **READLN** and **READCH** commands enable exec files to read in text from the console, and to assign it to a parameter variable. You can use these commands to:

- obtain parameter values
- obtain values to control conditional selection
- pause until the user indicates to continue

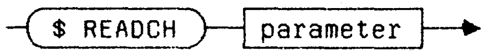
The forms of these commands are:

readln statement



and

readch statement

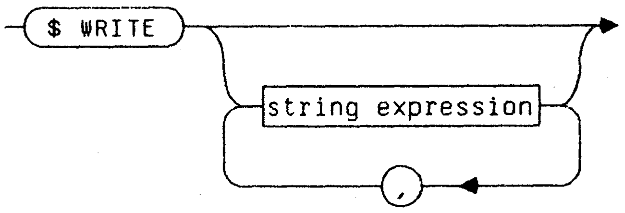


The **READLN** command reads a line from the console and assigns it to the specified parameter. The **READCH** command reads a single character from the console. If you press [RETURN], **READCH** will interpret it as a space.

9.2.3.2 The WRITE and WRITELN Commands

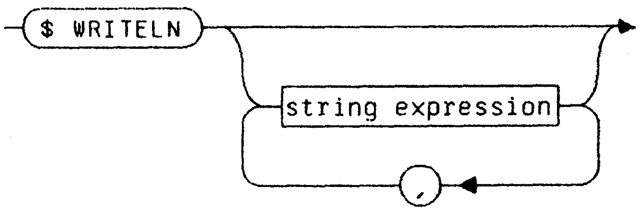
The **WRITE** and **WRITELN** commands enable exec files to write text to the console screen. You can use this text for informatory messages or prompts. The forms of these commands are:

write statement



and

writeln statement



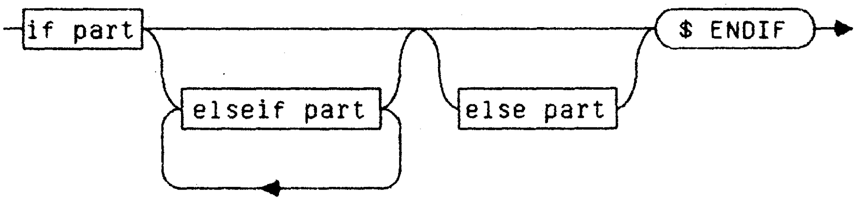
These commands take an arbitrary number of string expressions, separated by commas, as arguments. The strings are written to the current console line. The **WRITELN** command adds a final carriage return.

9.2.4 Conditional Statements - the IF Statement

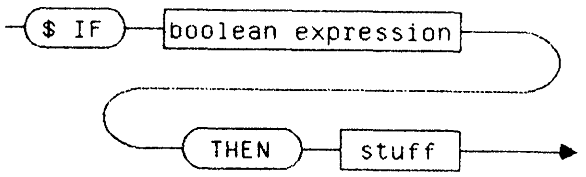
Conditional statements enable you to perform commands depending on conditions existing at process time (when the temporary file is created). The condition is stated in the form of a boolean expression, and can include built-in boolean functions.

The **IF**, **ELSEIF**, **ELSE**, and **ENDIF** commands enable conditional selection in exec files. The forms of these commands are:

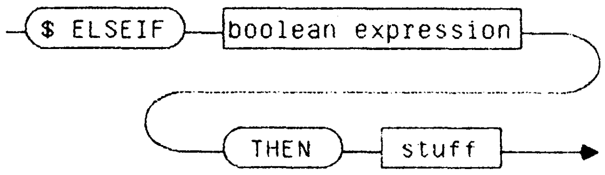
if statement



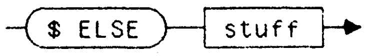
if part



elseif part



else part



where "boolean expression" is described in Section 9.2.4.1, and "stuff" is composed of arbitrary normal and command lines, other than commands that would be a part of the current **IF** construct. The **IF** statement is multiline, meaning that the components **IF**, **ELSEIF**, **ELSE**, **ENDIF**, and "stuff" each need to be on separate lines.

The **IF** construct is evaluated in the usual way. First, the boolean expression on the **IF** command itself is evaluated. If it is true, the "stuff" between the **IF** and the next **ELSEIF** (if any), or **ELSE** (if any), or **ENDIF** is selected; otherwise, it is not selected. The remaining parts of the **IF** construct, up to the **ENDIF** command, are parsed, but are not selected once one of the boolean expressions is true and its corresponding "stuff" is selected. Selecting "stuff" means that any normal lines are processed by the Workshop, and any command lines are processed. Conversely, if "stuff" is not selected, any normal lines and command lines are not executed. However, the command lines are parsed for correctness.

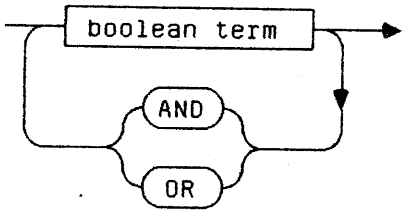
If the boolean expression on the **IF** construct is not true, the **ELSEIF** or **ELSE** command that follows is processed. If an **ELSEIF** command is next, its boolean expression is evaluated. If true, its corresponding "stuff" is selected and the remainder of the **IF** construct is not selected. Processing the **IF** construct continues until one of the boolean expressions on an **IF** or **ELSEIF** command is true, or until the **ENDIF** is reached. If no boolean expression is true before the **ELSE** (if any) is reached, the "stuff" corresponding to the **ELSE** command is selected.

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

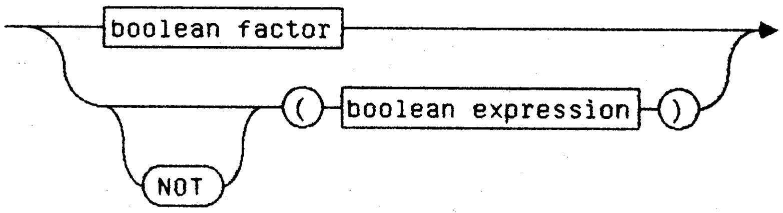
9.2.4.1 Boolean Expressions -- Comparison and Logical Operators

Boolean expressions enable you to test string values and check properties of files. The syntax for boolean expressions is:

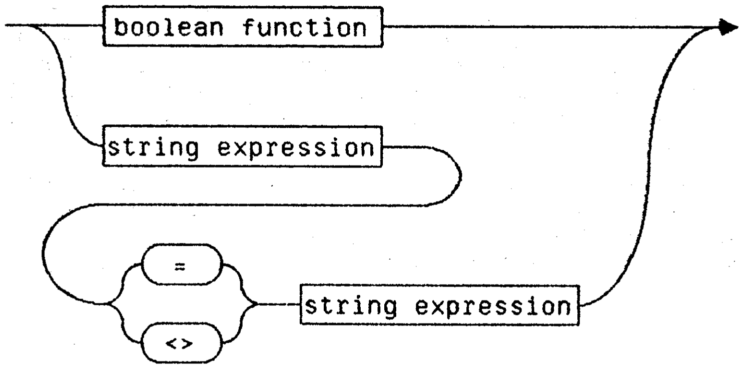
boolean expression



boolean term



boolean factor

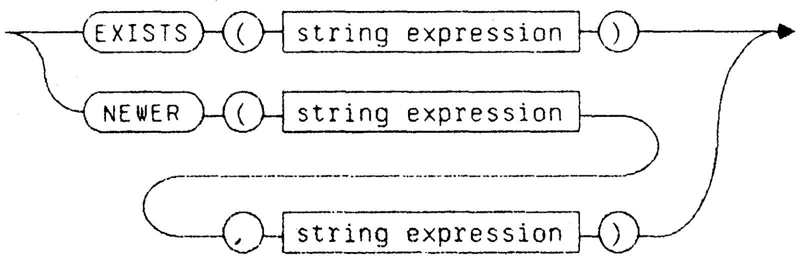


The basic element of a boolean expression, a "bool factor", is either a boolean function (see Section 9.2.4.2) or a string comparison, testing string expressions for equality or inequality (see Section 9.2.4.3). The basic elements can be combined with the logical operators **AND**, **OR**, and **NOT**, with parentheses for grouping. These operators function in the usual way.

9.2.4.2 Boolean Functions -- EXISTS and NEWER

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

boolean function

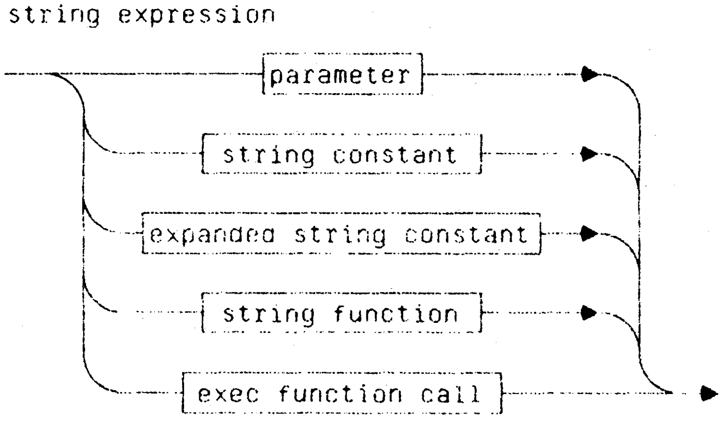


The **EXISTS** function enables you to determine whether or not a file, volume, or device exists. If you specify a device, the function will return a value of TRUE if the device has a volume mounted on it. The string expression arguments to these functions should specify names of files. Typically these string expressions will be expanded string constants, discussed in Section 9.2.4.3, such as "%1.obj".

The **NEWER** function enables you to determine if one file is newer than another file; that is, whether or not its last-modified date is more recent than the last-modified date of another file. A value of TRUE is returned if the first file is newer than the second. During processing, an error will occur if one of the files does not exist.

9.2.4.3 String Expressions

A string expression can specify a string in a variety of ways, as noted in the following:



- A *parameter* has the form %n.
- A *string constant* has the standard form of text delimited by single quotes ', with an embedded quote 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 an exec file processor function that returns a string value. A detailed description of string functions is provided in the following section.
- An *exec function call* is an invocation of an exec file that returns a string value, as described in Section 9.2.5.3.

9.2.4.4 String Functions — CONCAT and UPPERCASE

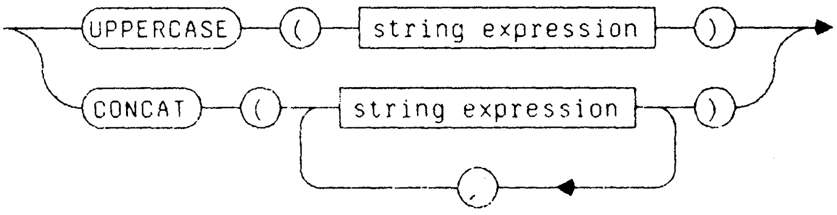
The string functions **CONCAT** and **UPPERCASE** can be applied to other string expressions to produce new string values.

The **CONCAT** function enables you to combine several string expressions to produce a single string result. The **CONCAT** function takes a list of string expressions, separated by commas, as arguments.

The **UPPERCASE** function converts any lowercase letters in its argument to upper case.

The form of these functions is:

string function



An example of the use of the UPPERCASE function is

$ SET %0 TO UPPERCASE (%0)

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

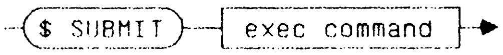
9.2.5 Nesting Exec Files

Exec files can be nested in two ways. One is to use the **SUBMIT** command to call another exec file in the same way that you would call a procedure. Alternately, you can call exec files as functions (returning string values to a string expression), as explained in Section 9.2.5.3.

9.2.5.1 The SUBMIT Command

The **SUBMIT** command enables you to nest exec files; that is, you can call one exec file within another exec file. The form of the **SUBMIT** command is:

submit statement



where "exec command" is an exec command of the same form as would follow the exec/ or < at the Workshop command level. This exec command can include parameters and exec options in the usual fashion (see Section 9.3).

The **SUBMIT** command processes the specified exec file, putting any generated exec output text into the current exec temporary file. Thus, while a single exec file can have several nested subexec files, only one temporary output file is generated. This file contains the output generated by all of the input files. Exec files can be nested to an arbitrary level.

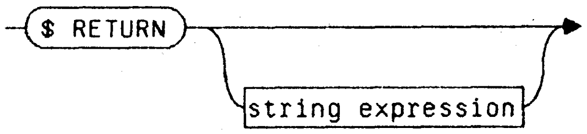
Within the text of the exec command, references to %n parameters are expanded, and the literalizing character tilde (~) is processed. Be aware that this is the only processing that takes place within the exec command. Everything up to the first left parenthesis, or the end of the line if no parameter list is present, is taken to be the exec file name. If a left parenthesis exists, the parameter list is taken to be everything between this parenthesis and the next right parenthesis. The exec command cannot be split across lines.

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

9.2.5.2 The RETURN Command

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

```
return statement
```



Executing a **RETURN** command terminates the current exec file, and returns to the calling exec file with the specified string value. (Section 5.2.5.3 describes how exec functions are called.) You can use a **RETURN** command without a string expression to exit from exec files which are *not* used as functions.

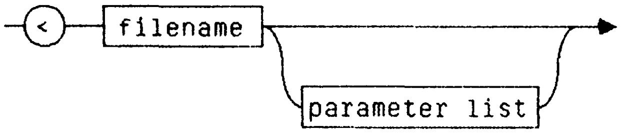
One way you can use exec functions is to determine if a program file, including any corresponding include files, has been modified since its last compilation. This function can then be used to conditionally submit compiles. If written generally enough, such a function could be used by many exec files.

Exec functions can produce side effects; that is, they can contain normal lines that get placed in the temporary file. While the intentional use of such side effects is unlikely, inadvertent instances can occur and are potentially hazardous to your exec files. An unexpected blank line in the middle of an exec file can often throw it out of sync.

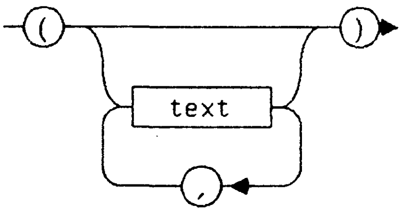
9.2.5.3 Exec Function Calls

Exec function calls return string values, and are thus one of the basic elements of string expressions. They can also appear in boolean expressions, supplying arguments for string comparisons. A typical use of an exec function is to return a boolean value by returning either the string T or F. The form of an exec function call is:

exec function call



parameter list



where < is the character that signals a function invocation, in the same way that this character identifies exec files for the Workshop's Run command. The "file name" and optional "parameter list" are the same as described in the **SUBMIT** command section, Section 9.2.5.1.

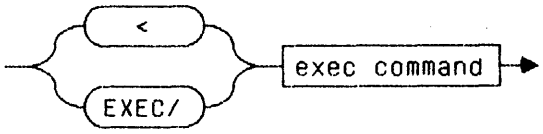
Due to the liberal conventions concerning what characters, including blanks, can appear in file names, the exec file processor must make some assumptions about how to identify the exec function file name and the argument list. The following rule is used: if the exec function invocation has an argument 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. This means that if the function call is not the last thing on the command line, you must supply an empty argument list to an exec function with no arguments.

Processing the text of a function call is the same as with a **SUBMIT** command; that is, the only processing that takes place is the expansion of %n parameters and recognition of the literalizing character "~". This means that the text of a function call cannot contain an embedded function call. Note also that a function call cannot be split across lines.

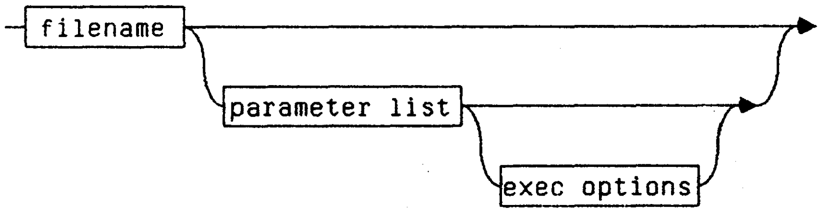
9.3 Using Exec Files

You invoke the exec file processor in response to the Workshop Run command prompt. An invocation line for the exec file processor has the form:

`exec invocation line`



`exec command`



The "exec file" is the name of the exec file you want to run. An extension of ".TEXT" is assumed if no extension is specified. However, you can override the mechanism that supplies the ".TEXT" extension by ending your exec file name with a period; for example, using "foo." causes the exec file processor to search for the file "foo" rather than "foo.text".

The optional "parameter list" is enclosed in parentheses. The parameter list can be empty or it can include up to ten parameters separated by commas. For example, an exec file to run compiles, which takes volume and source file parameters, might be invoked with "compile(foo,-work)". You can omit parameters, leaving them as null paramaters, by specifying them with the null string, as in "compile(foo,)". The volume that was present in the previous example has been omitted. Alternately, parameters can be left unspecified altogether, as in "compile(foo)". In this case, they also get null values. One reason to omit parameters is that the exec file might have been set up to supply default values, as described in Section 9.2.2.1.

The exec options that follow the closing right parenthesis of the parameter list consist of single-letter commands, which change the behavior of the exec file processor; for example, you use the letter S to indicate that you want to step through the exec file as it is being processed, conditionally selecting which commands are to be sent to the Workshop. The exec options are discussed in detail in Exec Invocation Options, Section 9.3.1.

The exec file processor's output is a temporary file with a ". .text" extension. The temporary file is the processed version of your exec commands; that is, all exec command lines have been processed and removed, leaving only the resulting Workshop commands. This temporary file is passed to the Workshop when the processing is completed. The Workshop then runs the temporary exec file, and automatically deletes it when finished.

NOTE

To terminate the processing of the exec file while the exec file processor is running, you press -period.

9.3.1 Exec Invocation Options

Several options are available when running the exec file processor. You can specify these options when invoking the exec file processor 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 exec file processor should not trim blanks on output lines. Normally the exec file processor trims off leading and trailing blanks on the lines that it outputs to the temporary file. Trimming enables you to indent normal lines (lines that are not exec command lines) without worrying about generating spurious blanks. In other words, the exec file processor assumes that leading and trailing blanks are insignificant. While this assumption is true for Workshop commands, it might not be true for some other programs you can run with exec files. Using this option tells the exec file processor not to trim such blanks. The option applies to only the exec file being run or **SUBMIT**ted, and not to any nested exec files.

I indicates that the first line of the exec file is to be ignored by the exec file processor. This option is intended for those who embed exec files in their program sources. When using this option, you should begin the first line of the source with a "(*", and follow the end of the exec file with a "*)", thus commenting it out of the program source. Note that you should use "(*" and "*)" instead of "{" and "}", since the latter are comment delimiters in exec files.

T indicates that the temporary file, which is created (i.e., the expanded form of the exec file), should *not* be automatically deleted after it is run. This option enables you to 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 next, is used to run old temporary exec files. Note that the T option is not allowed on **SUBMIT** commands.

R indicates that the an exec temporary file, saved with the T option, should be rerun, bypassing the normal processing by which the temporary was created. For example, "foo" might be an exec file that generates a complicated system using a large number of nested exec files that take a significant amount of time for the processor to digest. If you know you are going to run "foo" repeatedly, you might want to generate the temporary file only once but run it several times. The first time you would invoke the exec file processor with "<foo()t" to indicate that the temporary file should not be automatically deleted after it is run. Subsequently, you would invoke the exec file processor with "<foo()r" to rerun the old temporary file. Note that the R option overrides any others that might be specified; since, if you are rerunning an old exec temporary file, all the processing has been performed and the other options make no sense. Using the R option 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**.

9.3.1.1 Using the Step Function

If you use the step option, the following prompts appear when you invoke the exec file processor:

```
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 ?  (Y or N)  [No]
```

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

When you invoke an exec file with the step option, you are prompted when a line has been generated and is about to go into the temporary file. The line is displayed followed by "<- Include ?".

- A response of Y includes the line in the expanded exec file.
- A response of N omits the displayed line.
- A response of A aborts out of the exec file processor, and no exec file is run.
- A response of K keeps (includes) all the remaining lines of the exec file, leaving step mode.
- A response of I ignores the remainder of the exec file. No more lines are included.

When a **SUBMIT** command is encountered in stepping, the **SUBMIT** line is displayed followed by "<= Submit ?".

- A response of Y performs the **SUBMIT** unconditionally; that is, without stepping through it.
- A response of **N** ignores the **SUBMIT**.
- A response of S steps through the **SUBMIT** file.
- A response of **A** aborts out of the exec file processor, and no exec file is run.
- A response of **K** keeps the rest of the exec file, leaving step mode.
- A response of I ignores the remainder of the exec file.

NOTE

A reponse of ? to a "Submit ?" or "Include ?" prompt elicits an explanation of the accepted responses.

Some examples of how to use the exec file processor's stepping facility follow.

Stepping can be used to resume execution of an exec file that did not run to termination. For example, if your "compile" exec file includes both a compile and a generate step, and if you want to resume with the generate step, you invoke the exec file with "compile(foo,-work)s". Then, in response to the "Include?" prompt for lines corresponding to the compile step, you hit **N** to skip the lines. Upon reaching the first line of the generate step you respond with **K** to keep the rest of the file. Thus the generate step of the exec process would be performed.

The stepping mechanism can be used to run only selected parts of an exec file. Say, for instance, that you have a modular set of exec files, which generates a whole system of programs, such as the Workshop, and that one exec file called "make/all" can generate the whole system by **SUBMIT**ting 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 you perform a system build and find yourself constantly having to regenerate parts of the system, the ability to step by **SUBMIT**s proves very useful. You can regenerate arbitrary parts of the system by running "<make/all()s" (our master exec file invoked with the stepping option), and selectively submitting the subexec files for only those things that you want to rebuild, while stepping over the others.

Stepping in conjuction with the T option, for saving the temporary file created by the exec file processor, can be useful when you are going to be regenerating a single component of a program or system a number of times in succession; for example, when you are fixing a bug in an element of a system build and you expect that several iterations will be needed to correct the problem. To continue the previous example, suppose that while building the development system, you have a problem with the "fileio" unit of the "objiolib" library. Suppose also 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 exec file processor with "make/objiolib()st", you 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", you can run the saved temporary file by running "<make/objiolib()r" without having to go through the stepping process. The alternatives to this procedure are: to create another exec file to generate only the selected parts, to run (and rerun) the exec file for the whole library, or to run each subprocess independently (which requires more of your attention).

9.4 Example Exec Files

9.4.1 An Exec File to Do a Pascal Compile

This exec file does a Pascal compile and generate. Note how comments are 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
```

9.4.2 An Exec File to Do an Assembly

This exec file performs an assembly, and allows for an optional output file name which can 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
```

9.4.3 A More Flexible Exec File to Do Pascal Compiles

This exec file performs compiles, allowing for an output file with a different name than the souce.

```
$EXEC { "comp1" -- perform a Pascal compile
             %0 -- the name of the unit to compile
             %1 -- (optional) alternate name for OBJ file }
$DEFAULT %1 TO %0 { if no alternate OBJ name use same name as
                    source}
P{Pascal compile}%0{source}
   {no list file}
   {default i-code file}
G{generate code}%0
   %1{OBJ file}
$ENDEXEC
```

9.4.4 A "Smart" Exec File to Do Pascal Compiles

This compile exec file only performs the compile if either the object file does not exist or the source file is newer than the object file; that is, the source has changed since it was last compiled. It uses the comp1 exec file shown in Section 9.4.3 above.

```
$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 }
  $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)
     $ENDIF
  $ELSE  { OBJ file does not exist, so generate it }
     $SUBMIT comp1(%0,%1)
  $ENDIF
$ENDEXEC
```

9.4.5 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 you want to generate a particular program composed of three units (unit1, unit2 ,unit3), and that you 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. Your generation exec file uses these smart exec files to perform the minimal required amount of work. Thus it can be used to ensure that you have the latest version of the program without performing a full regeneration.

```
$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 gets the correct file
                        dates. This is one example of when you
                        should note the difference between
                        process time and run time.}
$ENDEXEC
```

Note that in the last line of the above exec file you 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 are 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 exec file processor. Its output goes to a temporary file, which is then passed back to the Workshop. The Workshop runs the commands in the temporary file until it comes to the command to Run another exec file. At this point it discards the remainder of the temporary file, and runs the exec file processor with the new exec command. This exec file invocation results in another temporary file of commands, which is then run by the Workshop. This means that some exec processing has been scheduled to follow some exec running, rather than all of the processing taking place first.

9.4.6 A Recursive Exec File to Do Pascal Compiles

This compile exec file performs 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 }
   ${ "rcomp" the rest, less  first }
   $SUBMIT rcomp(%1,%2,%3,%4,%5,%6,%7,%8,%9)
 $ENDIF
$ENDEXEC
```

9.4.7 A BASIC Example

This exec file demonstrates, by generating the BASIC Interpreter, some of the constructs in the exec file processor's meta language. 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 of three components and that you might want to generate only one or two 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 prompts 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} b.basic
   b.rtunit
   int.main
   hwintl
   iosfplib
   iospaslib
   basic{executable output}
$ENDEXEC
```

9.4.8 An Exec File Function

This exec file is a function which prompts the user for the location of a ProFile, and returns 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 function call }
  $ELSE
    $RETURN %9
  $ENDIF
$ENDEXEC
```

9.5 Exec File Programming Tips

The following points might be useful to remember when creating exec files.

1. Use *modular* exec files. Think of *exec files* as *procedures* that are called by the **SUBMIT** command. The more modular your exec files are, the easier it is to use the stepping facility on them.
2. Create *standard exec files* for common functions; for example, use one exec file to perform all your compilations. Therefore, if changes become necessary, you have only one place to change.
3. Use *optional parameters* to support features of your exec files that you do not always use. The parameter mechanism enables you to ignore optional parameters if you do not need the functions they support.

4. Write your exec files to *prompt for information* not supplied in the parameters. Thus you do not need to remember the meaning of a large number of parameters.

9.6 Exec File Errors

The exec file processor can recognize a number of errors during its invocation and execution. The format in which 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 text of the current exec line where the error was detected.
<err marker>	is a line with a question mark indicating where the exec file processor was in <curr line> when the error was detected.
<err msg>	is one of the messages listed below.

I/O errors are followed by an additional line with the text of the OS error raised during the I/O operation. The errors detected are listed below.

9.6.1 I/O Errors

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

9.6.2 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> might be:
      String value
      "%n" parameter
      Terminating string delimiter
      "=" or "<>"
      "<>"
      Boolean value
      Comma (list delimiter)
      "("
      ")"
      Valid command keyword
      Command
```

NOTES

Chapter 10
The Transfer Program

The Transfer Program

10.1 Introduction

The transfer program is a data communications package that allows you to transfer text files from your Lisa to another computer. You can also receive text from the remote computer and store it in a text file, which can then be read by the Editor.

To use the transfer program, you must either:

- Get the necessary modem and attach it to the Serial A or Serial B connector on the back of your Lisa. Then tell the Preferences tool in the System Manager the you are attaching to a Remote Computer.
- Or, get the necessary modem eliminator cable and attach it to the Serial A or Serial B connector on your Lisa. Then attach the other end to a serial port on another computer, and tell the Preferences tool that you are attaching to a Remote Computer.

When you have completed either action, set the Transfer Program characteristics to match the requirements of the remote computer.

These operations are explained in Sections 10.2 and 10.3 below. Section 10.4 explains how to use the Transfer Program to send and receive data.

10.2 Hardware Connections and Configuration

In order for the Lisa to communicate to a remote computer the Lisa can be connected to a modem or a modem eliminator cable through either the Serial A or the Serial B connector on the back of the Lisa.

In addition to connecting the hardware, you must configure the software To do this, use the Preferences tool from the System Manager command line. Access the Device Connections display, and set either Serial A or Serial B to Remote Computer. More information on the Preferences tool can be found in Section 3.3.

You must also set the active Transfer Program to access the correct connector. Do this by selecting either Serial A or Serial B from the Connector menu. The default is Serial A.

10.3 Setting Transfer Program Characteristics

In order to communicate with a remote computer, the Transfer Program must be set up so that it transmits and receives data in the same way as the host. These settings are made by using the Baud Rate, Parity, Handshake, Duplex, and Control menus. These settings are explained below.

Baud Rate
The baud rate is the speed at which data passes to and from the remote computer. The baud rate must be set to agree with the remote computer and modem you are using. The baud rate menu is shown in Figure 10-1. The default is 1200 baud. See the note in Section 11.10, PortConfig, for the valid baud rate settings for each Serial port.

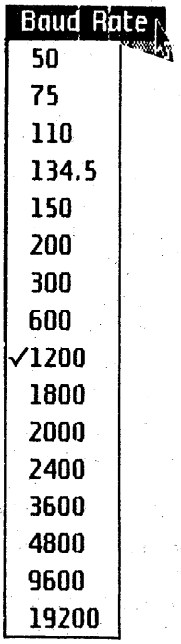


Figure 10-1
The Baud Rate Menu

Parity
Parity refers to the process of checking that data was not damaged in transmission. Parity should be set to agree with the host computer. Parity can be even, odd, or turned off (none). Select the option desired from the Parity menu. The default is none. The parity menu is shown in Figure 10-2.

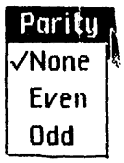


Figure 10-2
The Parity Menu

Handshake
The handshake menu, shown in Figure 10-3, selects either an XOn/XOff protocol, or no handshake. The XOn/XOff protocol allows the remote computer and the Transfer Program to tell each other whether they are ready to receive more information. Using this protocol, the Lisa can stop transmission from the host by sending XOff, and start it again by sending XOn. The host can start and stop transmission from the Transfer Program by sending XOn and XOff to the Lisa. The XOn character is a control-Q, XOff is control-S. The default is for handshaking to be turned on.

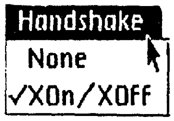


Figure 10-3
The Handshake Menu

Duplex
This menu allows you to select Full or Half duplex. Full duplex sends all characters typed from the Lisa keyboard to the remote computer, but does not display them on the Lisa screen. All characters sent from the host are displayed on the screen. Using full duplex, you will only see what you type if the remote computer sends back the characters you type. Most hosts you are likely to use with a Lisa do send back the characters they receive to be displayed.

Half duplex displays the characters typed on the keyboard, bacause it does not expect the host to send them back. The default is full duplex. The duplex menu is shown in Figure 10-4.

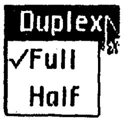


Figure 10-4
The Duplex Menu

Control
The control menu allows you to set two delay times, if needed. The first is a delay between each character sent, the second is the delay between each line. Both are in milliseconds. Delays are used to simulate typing speeds when transmitting to a remote computer that can not keep up with full speed transmission. The default is for no delay. The control menu is shown in Figure 10-5.

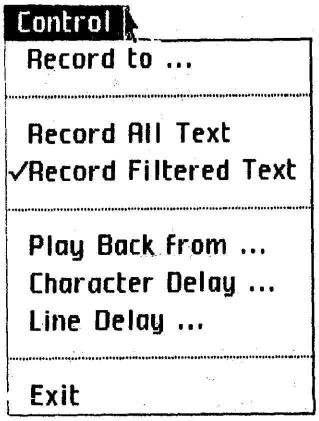


Figure 10-5
The Control Menu

10.4 Using the Transfer Program

Start the Transfer Program by pressing T in response to the Workshop command line. The Transfer Program will display a window on the screen with menus at the top. You must configure the Transfer Program to match the remote computer you wish to communicate with. Information on configuring it can be found in Section 10.3 earlier in this chapter.

After the Transfer Program comes up, it is ready to act as a terminal emulator. Evrything you type on the keyboard will be transmitted through the modem to the remote computer.

The Transfer Program can also be used to transfer files back and forth between the Lisa and the remote computer. The functions for doing this are in the Control menu. The control menu is shown in Figure 10-6.

To transfer a file from the Lisa to the remote computer, select "Play Back From . . ." from the control menu. It will ask you for the file name to play back. It expects a .TEXT file. The contents of that file will be transmitted to the remote computer.

To transfer a file from the remote computer to the Lisa, select "Record to ..." from the control menu. It will ask you for the name of the file to record to. After you have set up the remote computer to transmit the file you want (by typing commands at the keyboard) select "Record All Text" from the control menu. When you tell the remote computer to transmit the file, it will be recorded in the file you specified. This command will record the file exactly as transmitted, including all control characters. If you don't want the control characters, select "Record Filtered Text". This option changes carriage returns to newlines and replaces tabs by the appropriate number of spaces. All other control characters are thrown away. The filtering option affects only the disk file, not what is displayed on the screen. The default is "Record Filtered Text".

To transmit control characters from the keyboard, hold down the key and press the character. Other special purpose characters can be transmitted as shown in Table 10-1. Option keys are treated as no-ops.

Table 10-1
Transmitting Special Characters from the Keyboard

Keyboard	*Transmits*
Apple backspace	del
clear	esc
ENTER (alpha keyboard)	break
ENTER (numeric keypad)	return
arrow keys	their symbols
Apple Q	XOn
Apple S	XOff

NOTES

Chapter 11
The Utilities

The Utilities

11.1 ByteDiff

Synopsis

ByteDiff compares the contents of two files and reports which bytes (words) are different.

Dialog

```
Source file?
Target file?
```

Description

ByteDiff compares the source file to the target file and reports on their differences. This utility is useful for finding the first differences between files or for finding a small number of differences.

The program prompts for an input file and an output file. The two files can be in any format: .text, .obj, .i, and so forth.

The output is of the form:

```
    Bytes $xxxxxx differ aaaa bbbb
```

where:

xxxxxx is the byte address in hex
aaaa is the word (two bytes) from the source file
bbbb is the word from the target file

After 20 lines of output the user can either terminate by pressing [CLEAR] or continue by pressing the space bar.

See Also

Diff, E(qual command of the File Manager

Notes

ByteDiff compares any binary files, but once it finds a difference between the two files, it does not try to resynchronize. This utility does block-at-a-time I/O. The program stops at the first end-of-file and has no termination message. ByteDiff is nonstandard user interface.

11.2 ChangeSeg

Synopsis

ChangeSeg changes the segment name in the modules in an unlinked object file.

Dialog

File to change:
Map all Names (Y/N)

Description

The first prompt asks for the unlinked object file you want to change.

You are next asked if you want to map all names. 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 [RETURN] accepts the default name.

Notes

Changes are made in place (the file itself is changed).

11.3 CodeSize

Synopsis

Determines the code size and code segmentation for a unit, a program, or a library.

Dialog

Input file [.OBJ] -
Resident file [.TEXT] -
Output file [-CONSOLE]/[.TEXT] -

The *resident file* is the file that contains the segemnt names that are considered resident. The names in the file must be the same case as in the code file itself. The resident information is used in the summary reports to automatically sum the resident and swapping code.

At any time when specifying the file names, the run-time options can be turned on or off. The run-time options are:

+%	turns the mapping of calls to *system externals* on or off. System externals are procedures whose names begin with a "%". Using this option, the system will count the number of procedures that call a particular system external. This option is used to determine which system routines are being used, for example, if WRITELNs are left in the code.
+E	turns the mapping of calls to *nonsystem externals* on or off. Nonsystem externals are procedures in a segment other than the calling procedure. Using this option, the system will count the number of procedures that call a particular nonsystem external. This option is used to determine which routines are being used, for example, which library routine the code is using.
+M	tells CodeSize that a particular segment is mapped onto another segment. This information generates the segment mapping summary and the segment summary. This option is used when smaller segments are mapped into larger segements, and the sizes of the smaller and resulting larger segements are needed.
+S	turns the main report on and off. Sometimes the summary report is all that is needed. Use this option to print only the summary report.

Description

CodeSize generates two types of reports depending on the type of input file(s): main report and summary report. The input file can be an execution file, a library, or an object file. For each file, the report format will be:

Type of File	Main Report	Summary Report
Execution file	segment information	segment summary main summary
Library file	unit information segment information	unit summary segment summary main summary
Object file	unit information procedure information	external summary(+E or +%) unit summary segment mapping summary(+M) segment summary main summary

The contents of the report section are:

Segment information	
segment type	intrinsic, nonintrinsic, main program
segment name	first eight charcters of the segment's name
segment size	size of the segment in decimal or hex
Unit information	
unit name	first eight characters of the unit name
unit global size	how much global space the unit uses
unit type	intrinsic, shared intrinsic, regular
Procedure information	
procedure name	first eight characters of the procedure's name
associated segment	first eight characters of its segment's name
procedure size	size of the procedure in decimal or hex
interface information	is the procedure in the interface of the unit?
external references	list of all the external calls the procedure makes. This is triggered by the +E or +% options
External summary	
external procedure name	name of the procedure
# of occurrences	how many different procedures called the procedure. This is triggered by the +E or +% options.
Unit summary	
unit name	first eight characters of the unit's name
unit size	size of the unit in decimal or hex
unit type	intrinsic or not
unit global size	how much global space the unit uses

Segment mapping summary	
original segment name	name of the original segment
new segment name	name the segment is being mapped into
segment size	size of the segment being mapped. This is triggered by the +M option.
Segment summary	
segment type	swapping or resident. Resident segment is specified to CodeSize by the "resident file".
segment name	first eight characters of the segment's name
segment size	size of the segment in decimal or hex
Main summary	
total code size	summation of the code size
total resident code	summation of the code that is considered resident all the time. Resident code is specified to CodeSize by "resident file".
total swapping code	summation of the code that is considered swapping all the time. Swapping code is specified to CodeSize by "resident file."
total data globals	summation of the global space for data
total main prog globals	summation of the global space in the main program
total globals	sum of main program globals plus data globals
total jump table	size of the jump table

11.4 Diff

Synopsis

Diff is a program for comparing .TEXT files, in the Workshop. Diff is designed to be used with Pascal or Assembler source files.

Dialog

(Type '?' to change or display options.)

```
New file name [.TEXT] -
Old file name [.TEXT] -
Listing file  [.TEXT] (<CR> =  -CONSOLE) -
```

Description

Diff first prompts 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. Press [RETURN] to send the listing to the console.

Diff does not know about INCLUDE files. However, Diff does enable 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 press [RETURN] in response to the prompt for a new 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:

C	change the number of context lines displayed.
M	the number of lines required to constitute a match.
D	the number of lines displayed at the beginning of a long block of differences.

To set one of these numbers, type the option name and [RETURN], followed by the new number to the prompt for the first input file name. An entry of D [RETURN] 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. You can get a display of the current option settings by pressing "?" in response to the first file prompt.

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:

1. 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.
2. 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, and so forth)

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 is large enough to exceed the threshold for acceptance.

See Also
ByteDiff

Notes
Diff can handle files with up to 2000 lines.

11.5 DumpObj

Synopsis

DumpObj is a disassembler for 68000 code. This option provides a symbolic and formatted listing of the contents of object files. It can disassemble either an entire file, or specific modules within the file.

Dialog

```
Input file? [.OBJ]
Output file? [-CONSOLE]

Dump A(ll, S(ome, or P(articular modules [S]?
Dump file positions [N]?
Dump selected object code [N]?
```

Description

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 All, Some, or Particular modules.

If you respond S, DumpObj asks you for confirmation before dumping each module. A response of [CLEAR] gets you back to the top level. If you respond P, DumpObj asks you for the particular module(s) you want dumped.

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 DumpPatch program.

If you want the selected object code to be dumped, respond Y to the final prompt. The default for this prompt is N.

See Also

DumpPatch

Notes

DumpObj displays only the low order 24 bits of longint fields, which are interpreted as addresses. This is consistent with the hardware, but causes some bytes of the file not to be displayed.

11.6 DumpPatch

Synopsis

Dump and/or patch a file

Dialog

DumpPatch - Hexadecimal Dump and Patch

File: - Output: [-CONSOLE] [.TEXT] -

If you want to select the default of [-CONSOLE], press [RETURN] and select the block number you want to start with; for example, 2.

If you type a file name, the following prompt appears:

Would you like to access (input file name) interactively? (Y or N)

If you respond Y, you will be prompted for the block number you want to start with. If you respond N, you will be prompted for starting and ending block numbers. The default values are 0 for the starting block number and EOF for the ending block number.

Description

DumpPatch provides a textual representation of the contents of any file and the ability to change its contents in either the ASCII character or hexadecimal form. 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. The patch facility uses the arrow keys to move around within the displayed block and change the value of any byte.

When DumpPatch is Run, you will be asked for the full name of the input file. No extensions are appended. Pressing [RETURN] will exit DumpPatch. If the input file can be found, you will be asked where you want to direct the output. The default for the output file is [-printer]. If you type an output file name, a .TEXT extension will be added if necessary. If you type a device name; for example, -printer, no extension will be appended.

If an output file name or a valid device name was entered, you will be asked if you would like to access the input file interactively. If you answer No, you will get a quick dump of the input file and will be prompted for the starting block to dump. The default [RETURN] for the last block to be dumped is the last block of the input file. If you specify a block that is beyond the end-of-file, you will be given the block number of the last block in the file. Pressing [CLEAR] enables you to exit with no dumping.

Once a file has been completely dumped, DumpPatch asks you for the next input file. Press [RETURN] to exit the program.

If you access the input file interactively, you will be asked for the block to dump. The output will be dumped to the screen with the option of dumping it to the output file when you are ready to leave that block. Press the space bar to look at the next halfblock. Press [CLEAR] to go into patch mode. Press [RETURN] to quit the present block.

When you are in patch mode, the cursor will be found in the upper left corner at word 0 of the block. The arrow keys are used to move the cursor around in the current block and to previous or successive blocks. Press [TAB] to toggle between the hexadecimal and the ASCII portions of the display. A change made on one side of the display is automatically updated on the other side as well. Until you get ready to move out of the current block you may undo any changes by pressing [CLEAR]. When leaving a block in which you made changes, you will be asked if you want to write the changed block back to the input file. This is your last chance to undo any unwanted changes! If you specified output to something other than the console, you will also be asked if you want to dump the current block to the output file when you try to leave that block. To exit patch mode press [RETURN].

See Also
DumpObj

11.7 FileDiv and FileJoin

Synopsis

FileDiv can be used to break a large file into several smaller pieces. FileJoin can then be used to rejoin these pieces into one file. These functions are most useful when saving and restoring very large files, or when you want to break a large text file into smaller ones to be viewed in the Editor.

Dialog

Is this a .TEXT file? (Y or N)

Infile name : [.text]
Outfile name : [.text]

You might want to keep portions of a file on more than one disk. To give you an opportunity to do that, FileDiv contains the following additional prompts:

Another disk? (Y or N)

Have you inserted the next disk? (Y or N)

Description

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.

To rejoin the pieces of the file, Run FileJoin. The dialog is the same as for FileDiv.

11.8 Find

Synopsis

Find searches a text file for a pattern.

Dialog

type "?" to display or change options
Enter input file name [.TEXT] (name of the file to be searched)
Enter output file name [-CONSOLE]/[.TEXT] (default is the console)
Enter pattern: (pattern to be matched)

Description

Find searches text files for lines which match a string pattern. Lines found are printed to the console. The following options are recognized:

+C	Matches are case sensitive
+S	Matches are space sensitive.
+D	Print dots as lines which do not match are scanned.
+L	As lines are reported, print out the relative line numbers.
+T	Report the files that are being scanned.

Typing ? in response to any of the input prompts will display a description of the options available and read in the options. You can leave Find by typing [RETURN] or [CLEAR] in response to the input or pattern prompts.

More than one file can be input at a time. Find supports the same wildcard scheme as the Workshop File Manager. So submitting "-paraport-ch=" will direct Find to search all of the text files beginning with "ch" on the paraport directory. Find can also search predefined lists of files; suppose the file "foobar.text" contained:

```
" hooha.text
  grok.text
  bruhaha.text"
```

Then submitting "<foobar.text" will direct Find to search, sequentially, "hooha.text", "grok.text", and then "bruhaha.text". If you type "foobar.text" (without the leading '<') then Find will search "foobar.text", not the files listed therein, for the pattern.

Notes

Find truncates output lines to 256 characters.

11.9 GXRef

Synopsis

Global Cross Reference.

Dialog

```
Input file [.OBJ] ?
Listing file [CONSOLE:]/[.TEXT] -
```

Description

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.

GXRef accepts any number of object file as input. When you have entered all the object files, press [RETURN] in response to the input file request.

11.10 PortConfig

Synopsis

PortConfig enables you to configure the RS232 ports.

Dialog

First you must supply information on how to configure the port.

```
Which RS232 port do you want to configure ? (A or B)

What parity setting ?
   0) No parity
   1) Odd parity; no input parity checking
   2) Odd parity; input parity errors = 00
   3) Even parity; no input parity checking
   4) Even parity; input parity errors = $80
Enter selection (0 - 4)   [0]

What output handshake protocol ?
   0) None
   1) DTR handshake
   2) XON/XOFF handshake
   3) Delay after CR,LF
Enter selection (0 - 3)   [0]

What baud rate ?  [9600]

Receive and buffer input how ?
   0) Buffer input until full request is satisfied
   1) Return whatever is received
Enter selection (0 - 1)   [1]

What input handshake protocol ?
   0) None
   1) DTR handshake
   2) XON/XOFF handshake
Enter selection (0 - 2)   [0]

Adjust type-ahead buffer how ?
   0) Flush only
   1) Flush and re-size
   2) Flush, re-size, and set thresholds
Enter selection (0 - 2)   [0]

What form of disconnect detection ?
   0) None
   1) BREAK detected means disconnect
Enter selection (0 - 1)   [0]

Timeout on output after how many seconds (0 = no timeout) ?   [0]
```

```
Automatic linefeed insertion ?
   0) Disabled
   1) Enabled
Enter selection (0 - 1)   [0]
```

```
We are now ready to configure the port.  Shall we proceed? (Y or N)
```

PortConfig contains a series of questions. After you answer one, you will be prompted for an answer to the next one. The default values for each question are shown in brackets.

Description
With the PortConfig utility, you can configure the RS232 ports, and establish such things as the parity setting, handshake protocol, baud rate, disconnect detection, and so forth. If you are using Pascal and want additional information on port configuration, see Section 2.10.12 in *Operating System Reference Manual for the Lisa.*

NOTE

For Serial A and Serial B ports, the baud rate can be set to 50, 75, 110, 150, 200, 300, 600, 1200, 1800, 2000, or 2400. Serial A can also be set to 4800 or 9600.

For output *only,* Serial B can also be set to 3600, 4800, 7200, 9600, or 19200.

11.11 SegMap

Synopsis

SegMap produces a segment map of one or more object files.

Dialog

Files to Map ? [.OBJ]
Listing File ? [-CONSOLE]

Description

SegMap accepts either an object file name or a command file name, which enables you to include predefined lists of files.

A command file must be preceded with a "<". SegMap adds the .TEXT suffix to the command file name.

For example, if the file "Apple.text" contains:

```
"code"
"pascal"
"basic"
```

Submitting "<Apple" directs SegMap to accept, sequentially, "code.obj", "pascal.obj", and "basic.obj".

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.

11.12 SXRef

Synopsis

Pascal cross reference utility

Dialog

Source File ? [.TEXT]
Output file for Listing ? [-CrossRef] [.TEXT]
Do you want a numbered listing of the source ? (Y or N)
Flag the declarations and assignments of each indentifier ? (Y or N)
Declaration Character ? [*]
Assignment Character ? [=]
Text file of words to Omit ? [SXRef.Omit] [.TEXT]

Description

SXRef gives a numbered listing of the source files and an alphabetical listing of identifiers found. For each identifier, all references to the identifier are listed in the order in which the references were encountered. Procedure and Function names along with all references to them will be found at the end of the cross reference listing.

Identifiers follow current Lisa Pascal conventions: the first eight characters, without regard to case sensistivity. Case insensitivity is achieved by shifting identifiers to lower case, within the Cross Reference section.

INCLUDE files are automatically processed. User interfaces are not processed. Comments and strings are recognized and skipped. There is no conditional compilation processing or elimination of code controlled by boolean constants.

SXRef will accept multiple source files. This can be used to get a cross reference of a set of Main Programs together with the Units which the programs use. References are given by file number and line number within the file. A directory of files read is printed at the end of the source listing, and before the cross reference section.

SXRef attempts to read a file for a list of words to omit from the cross reference. The default name is SXRef.omit.text, but other names can be given. If the file cannot be opened, execution proceeds normally without omitting any identifiers.

SXRef will optionally flag where all identifiers are declared and assigned values. The default flag characters are: [*] for declaration and [=] for assignment.

If SXRef runs short of storage, an error message is given and the program aborts.

See Also

GXRef, UXRef

11.13 UXRef

Synopsis

Show unit dependencies of one or more Pascal source programs

Dialog

Type "?" to see current options
Source File ? [.TEXT]
Output file for Listing ? [-Cross Ref] [.TEXT]
Text File of unit names with unexpected pathnames ? [UXRef.UMap] [.TEXT]

Description

UXRef gives an alphabetical listing of programs and units. Each program or unit listed includes two parts: 1) alphabetically lists all programs and units that USE that program or unit, and 2) alphabetically lists all units that ARE USED BY that program or unit.

UXRef recognizes conditional compilation and will determine the truth value of any {$ifc ...} expression. Compile-time variables can be of both boolean and integer types and a {$setc ...} can change a variable to a new type. Warnings will be sent to the console if a syntactical or semantic error is found in an {$ifc ...} expression.

Warnings about units that can't be found are sent to the console. Even though a unit cannot be found it will still show up on the Cross Reference listing.

Options may be turned on or off during file name prompt stage of UXRef. Four options are included:

+C	You will be asked to manually clarify a compile-time expression or variable that cannot be evaluated correctly. Enter 'T' for true and 'F' for false. If this option is off, the entire expression will be treated as false.
+F	As each file is opened, a message will be printed on the -console specifying the file name and the unit name being read.
+I	"Include Files" will be treated as units and will show up on the Cross Reference listing. Only those "include files" that are found between the beginning of the program/unit and the end of the uses section will be listed.
+W	All warnings will be written at the beginning of the Cross Reference listing as well as on the console.

By entering ? during the file name prompt stage a short description of each option will appear along with their current values. The default values of the options are: -C, +F, -I, and -W.

UXRef provides a facility to map a unit to an unexpected pathname. For example, the unit "FOO" might not be compiled yet (e.g., "FOO.OBJ" does not exist) and the source is named "UNIT/FOO.TEXT". UXRef will attempt to read a file for a list of logically connected units and pathnames and if FOO,-UPPER-UNIT/FOO.TEXT is an entry in that file then "UNIT/FOO.TEXT" will be located and searched on the UPPER diskette when the unit FOO is referenced. The unit name and the pathname must be separated by a comma with no extra spaces between. In addition this same facility can be used to shut off unnecessary warnings that occur when an inaccessable unit is referenced. Normally warnings will be printed when a unit cannot be found, but if the unit name followed by a comma appears on UXRef.Omit.TEXT (or some other name provided by the user) the warnings for that unit will be bypassed. Example entries are:

```
FOO,-UPPER-UNIT/FOO.TEXT

SYSCALL
```

See Also
GXRef, SXRef

NOTES

Appendix A
Error Messages

Error Messages

A.1 Assembler Errors

The following errors can be produced by the Assembler.

1 Undefined label
2 Operand out of range
3 Must have procedure name
4 Number of parameters expected
5 Extra garbage on line
6 Input line over 80 characters
7 Not enough .IFs
8 Illegal use of .REF label
9 Identifier previously declared
10 Improper format
11 .EQU expected
12 Must .EQU before use if not to a label
13 Macro identifier expected
14 Word addressed machine
15 Backward .ORG currently not allowed
16 Identifier expected
17 Constant expected
18 Invalid structure
19 Extra special symbol
20 Branch too far
21 Variable not PC relative
22 Unexpected .ENDM
23 Not enough macro parameters
24 Operand not absolute
25 Illegal use of special symbols
26 Ill-formed expression
27 Not enough operands
28 Too many undefined lables in this expression
29 Constant overflow
30 Illegal decimal constant
31 Illegal octal constant
32 Illegal binary constant
33 Invalid key word
34 Macro stack overflow - 5 nested limit
35 Include files cannot be nested
36 Unexpected end of input
37 This is a bad place for an .INCLUDE file
38 Only labels and comments may occupy col 1
39 Expected local label
40 Local label stack overflow

41 String constant must be on one line
42 String constant exceeds 80 characters
43 Illegal use of macro parameter
44 Illegal use of .DEF label
45 Expected key word
46 String expected
47 Nested macro definitions illegal
48 '=' or '<>' expected
49 Cannot .EQU to undefined labels
50 Not even a register
51 Not a Data Register
52 Not an Address Register
53 Register expected
54 Right paren expected
55 Right paren or comma expected
56 Unrecognizable operand
57 Odd location counter
58 Comma expected
59 One operand must be a Data Register
60 Dn,Dn or -(An),-(An) expected
61 No longs allowed
62 First operand must be immediate
63 First operand must be Dn or #E
64 (An+),(An+) expected
65 Second operand must be an An
66 Second operand must be a Dn
67 #<data>,Dn expected
68 First operand must be a Dn
69 An,#<displacement> expected
70 An is not allowed with byte
71 Only alterable addressing modes allowed
72 Only data alterable addr modes allowed
73 An is not allowed
74 USP, SR, and CCR not allowed
75 Cannot move from CCR
76 Dx,d(Ay) or d(Ay),Dx expected
77 Only memory alterable addr modes allowed
78 Only control addressing modes allowed
79 Must branch backwards to label
80 Patch out of code buffer boundaries
81 Code buffer overflow
82 Segment name must be in a string
83 Cannot .DEF macro
84 MACRO defined already
85 Illegal use of MACRO
86 ERROR while WRITING SYMBOL TABLE FILE
87 Not enough ENDCs

88 Must have an <EA> (effective address)
89 Unimplemented Motorola directive
90 Operand size must be a word
91 No undefined or forward label in .BLOCK
92 Only byte-size displacement value allowed

A.2 Linker Errors

Linker errors are either Warnings, Errors, or Fatal Errors. All Linker errors are listed below, along with a brief description of their probable cause. The Linker can also produce errors from ObjIOLib. These errors are listed in Section A.3.

A.2.1 Warnings

A warning message is an indication of a potential error. However, the link is allowed to continue normally and may produce a valid output file. Warnings cannot be ignored! You must make sure that the conditions indicated by the warning are what was intended. When in doubt, attempt to remedy the conditions which caused the warning message to occur.

No Starting Location:
The file containing the main Pascal program has probably been omitted.

Duplicate entry definitions:
An entry name has been found in a library file which is the same as a name in the main program. References to the name are interpreted as referring to the main program entry. (NOTE: this can be an error if a Unit in the link was trying to reference the library entry.)

Conflict with Intrinsic Unit Name:
A regular Unit in the link has the same name as a library Intrinsic Unit.

Also an IU segment:
A segement in the link has the same name as as a library segment.

A.2.2 Errors

A error message is an indication of a condition which prevents the production of a valid output file. The link is allowed to continue, in order to detect any other errors. However, the output file will not be produced.

Multiple start locations.
More than one main program file has been provided as input to the Linker.

Duplicate definition of Unit Name
Doubly defined Global Data area:
Two units of the same name have been provided as input to the Linker.
Duplicate entry definitions.
Two entries of the same name have been found in the Linker input files.

Undefined entry:
The entry name has been referenced, but not defined. Either an input file has been omitted or a spelling error was made in a procedure name.

Undefined Code Module:
The module name has been referenced, but not defined. Either an input file has been omitted or a spelling error was made in a procedure name.

Undefined data area:
The unit name has been referenced, but not defined. Either an input file has been omitted or a spelling error was made in a unit name.

Segment name not found in Intrinsic.lib:
A name which occurs in an intrinsic library file does not appear in the directory file. Probably indicates an "architecture" consistency error; that is, the library file was not linked against the same directory as the current directory.

Bad block in Library file.
The library file being read does not have valid contents.

Relocation Block.
Common Definition Block.
The IULinker does not support these object blocks. Either the object file is very old, or an error has occured in the object file format.

Bad block, start of file:
Bad block type
The object file does not have valid contents. Most likely a disk error has caused to object file to be damaged. You should regenerate the object file or obtain a copy from a backup disk.

Bad Module type:
This indicates an internal Linker error, or perhaps an undetected memory error.

IU Code with main program.
The input contains both unlinked intrinsic units and an unlinked main program. Link the intrinsic units into a library file. Then link the main program, using the intrinsic library as input.

More than 32K of globals
The globals required by the main program and regular units exceeds the current limitation of 32K. You will need to recompile the program or the units, moving some large variables to the heap.

Code Size too big:
The code in the segment being linked exceeds the current limitation of 32K. You will need to resegment the program either using the +M Linker option, or by recompiling with different $S compiler options.

Segs 1-16 are Reserved:
The directory indicates that a segment name has been associated with one of the segments reserved for physical addresses.

A2.3 Fatal Errors

A fatal error indicates a condition which prevents the link from continuing.

Linker error -
Indicates an error in internal Linker logic, perhaps caused by an undetected disk or memory error.

Inconsistent Intrinsic.lib.
Probably indicates an I/O error, such as bad media, which has corrupted the directory file, or the specification of a bad directory.

Can't re-open inFile: xxxxxxx
An I/O error has occured which prevents the opening of file 'xxxxxxx' for phase 2 processing. Examine the file using the File Manager, or regenerate the file. Then attempt to do the link again.

Too many code segments.
The program has too many small segments. The current limitation is for segments numbered 17 through 105. Reduce the number of segments by combining small segments with the +M option in the Linker.

Regular unit during Intrinsic Link.
Intrinsic unit during Regular Link.
MainProg as part of Intrinsic Library Link:
The Linker has detected an unlinked regular unit or main program mixed with unlinked intrinsic units.

Regular unit in Intrinsic Seg File:
The Linker has detected an unlinked regular unit in an intrinsic library file.

Not Main or Intrinsic Link:
The Linker has not seen a valid input file to decide what type of link is desired.

No Starting location, linking Main Program:
The file containing the Pascal main program has been omitted from the input list, or is damaged.

One or more IU Segs not in Intrinsic.Lib:
An intrinsic segment name does not appear in the directory file. Probably indicates an architecture consistency error; that is, the library file was not linked against the same directory as the current directory.

Bad Unit Block (Old .OBJ file?):
Either this is a very old object file, not supported by this Linker, or a disk error has occured.

A.3 Messages Generated by ObjIOLib

The IULinker uses a number of units from the ObjIOLib intrinsic library file. These units are also used by the Compiler, Code Generator, and object file utility programs. These units detect some error conditions and issue messages.

A.3.1 Warnings

No Code Block found in input .LIB file.

For the O.S. Loader, there should be a Code Block in the directory file. Perhaps this is an old directory file, or a directory for another operating system.

Errors detected: No Output .LIB file written.

When the error count is nonzero, the directory file is not rewritten.

A.3.2 Errors

Bad Peek
Bad Peek2:

Indicates an internal error in the ObjIOLib library, perhaps caused by a disk or memory error. Check your hardware then retry the link.

I/O error, can't write last buffer:

Either the volume does not have enough space for the file or a hardware error has occurred.

MemMan Error:

An error has occurred in the managing of storage elements. Usually this error is due to insufficient initial space (Allocation error) or due to exhaustion of available space (Memory Full). The cause of the error is indicated on the next output line.

Attempt to delete vertex with arcs.
Argument to OppositeVertex is not an endpoint:

These are errors reported by the Graphs unit. If they occur while the Linker is executing, there has been an internal logic error, perhaps caused by an undetected I/O or memory error.

A.3.3 Fatal Errors

I/O error.

An I/O error has occurred within FileIO. Usually this is the result of a volume being almost full or a hardware failure. The previous message line indicates whether the error occurred during reading or writing and at what position within the file the error occurred.

No VersionControl Block.
No Unit Table.
No Segment Table.
No File Names Table:

Indicates a bad format for the directory file. The indicated block is missing from the directory, but is required.

Errors during Installation:
Indicates errors during the installation of an object file library.

SetObjInvar: VarSize is not divisible by variant size:
Indicates an internal logic error in ObjIO. Either initialization was not called, or ObjIO globals have been clobbered.

File Buffer less than 2 blocks:
Indicates an internal logic error in FileIO. Perhaps initialization was not called.

Attempt to delete item not on list:
This is an error reported by the Lisats unit. If it occurs while the Linker is executing, there has been an internal logic error, perhaps caused by an undetected I/O or memory error.

A.4 Operating System Errors

-6081	End of exec file input
-6004	Attempt to reset text file with typed-file type
-6003	Attempt to reset nontext file with text type
-1885	ProFile not present during driver initialization
-1882	ProFile not present during driver initialization
-1176	Data in the object have been altered by Scavenger
-1175	File or volume was scavenged
-1174	File was left open or volume was left mounted, and system crashed
-1173	File was last closed by the OS
-1146	Only a portion of the space requested was allocated
-1063	Attempt to mount boot volume from another Lisa or not most recent boot volume
-1060	Attempt to mount a foreign boot disk following a temporary unmount
-1059	The bad block directory of the diskette is almost full or difficult to read
-696	Printer out of paper during initialization
-660	Cable disconnected during ProFile initialization
-626	Scavenger indicated data are questionable, but may be OK
-622	Parameter memory and the disk copy were both invalid
-621	Parameter memory was invalid but the disk copy was valid
-620	Parameter memory was valid but the disk copy was invalid
-413	Event channel was scavenged
-412	Event channel was left open and system crashed
-321	Data segment open when the system crashed. Data possibly invalid.
-320	Could not determine size of data segment
-150	Process was created, but a library used by program has been scavenged and altered
-149	Process was created, but the specified program file has been scavenged and altered
-125	Sepcified process is already terminating
-120	Specified process is already active

-115	Specified process is already suspended
100	Specified process does not exist
101	Specified process is a system process
110	Invalid priority specified (must be 1..225)
130	Could not open program file
131	File System error while trying to read program file
132	Invalid program file (incorrect format)
133	Could not get a stack segment for new process
134	Could not get a syslocal segment for new process
135	Could not get sysglobal space for new process
136	Could not set up communication channel for new process
138	Error accessing program file while loading
141	Error accessing a library file while loading program
142	Cannot run protected file on this machine
143	Program uses an intrinsic unit not found in the Intrinsic Library
144	Program uses an intrinsic unit whose name/type does not agree with the Intrinsic Library
145	Program uses a shared segment not found in the Intrinsic Library
146	Program uses a shared segment whose name does not agree with the Intrinsic Library
147	No space in syslocal for program file descriptor during process creation
148	No space in the shared IU data segment for the program's shared IU globals
190	No space in syslocal for program file description during List_LibFiles operation
191	Could not open program file
192	Error trying to read program file
193	Cannot read protected program file
194	Invalid program file (incorrect format)
195	Program uses a shared segment not found in the Intrinsic Library
196	Program uses a shared segment whose name does not agree with the Intrinsic Library
198	Disk I/O error trying to read the intrinsic unit directory
199	Specified library file number does not exist in the Intrinsic Library
201	No such exception name declared
202	No space left in the system data area for Declare_Excep_Hdl or Signal_Excep
203	Null name specified as exception name
302	Invalid LDSN
303	No data segment bound to the LDSN
304	Data segment already bound to the LDSN
306	Data segment too large
307	Input data segment path name is invalid
308	Data segment already exists
309	Insufficient disk space for data segment
310	An invalid size has been specified
311	Insufficient system resources

312	Unexpected File System error
313	Data segment not found
314	Invalid address passed to Info_Address
315	Insufficient memory for operation
317	Disk error while trying to swap in data segment
401	Invalid event channel name passed to Make_Event_Chn
402	No space left in system global data area for Open_Event_Chn
403	No space left in system local data area for Open_Event_Chn
404	Non-block-structured device specified in pathname
405	Catalog is full in Make_Event_Chn or Open_Event_Chn
406	No such event channel exists in Kill_Event_Chn
410	Attempt to open a local event channel to send
411	Attempt to open event channel to receive when event channel has a receiver
413	Unexpected File System error in Open_Event_Chn
416	Cannot get enough disk space for event channel in Open_Event_Chn
417	Unexpected File System error in Close_Event_Chn
420	Attempt to wait on a channel that the calling process did not open
421	Wait_Event_Chn returns empty because sender process could not complete
422	Attempt to call Wait_Event_Chn on an empty event-call channel
423	Cannot find corresponding event channel after being blocked
424	Amount of data returned while reading from event channel not of expected size
425	Event channel empty after being unblocked, Wait_Event_Chn
426	Bad request pointer error returned in Wait_Event_Chn
427	Wait_List has illegal length specified
428	Receiver unblocked because last sender closed
429	Unexpected File System error in Wait_Event_Chn
430	Attempt to send to a channel which the calling process does not have open
431	Amount of data transferred while writing to event channel not of expected size
432	Sender unblocked because receiver closed in Send_Event_Chn
433	Unexpected File System error in Send_Event_Chn
440	Unexpected File System error in Make_Event_Chn
441	Event channel already exists in Make_Event_Chn
445	Unexpected File System error in Kill_Event_Chn
450	Unexpected File System error in Flush_Event_Chn
530	Size of stack expansion request exceeds limit specified for program
531	Cannot perform explicit stack expansion due to lack of memory
532	Insufficient disk space for explicit stack expansion
600	Attempt to perform I/O operation on non I/O request
602	No more alarms available during driver initialization
605	Call to nonconfigured device driver
606	Cannot find sector on floppy diskette (disk unformatted)
608	Illegal length or disk address for transfer

609	Call to nonconfigured device driver
610	No more room in sysglobal for I/O request
613	Unpermitted direct access to spare track with sparing enabled on floppy drive
614	No disk present in drive
615	Wrong call version to floppy drive
616	Unpermitted floppy drive function
617	Checksum error on floppy diskette
618	Cannot format, or write protected, or error unclamping floppy diskette
619	No more room in sysglobal for I/O request
623	Illegal device control parameters to floppy drive
625	Scavenger indicated data are bad
630	The time passed to Delay_Time, Convert_Time, or Send_Event_Chn has invalid year
631	Illegal timeout request parameter
632	No memory available to initialize clock
634	Illegal timed event id of -1
635	Process got unblocked prematurely due to process termination
636	Timer request did not complete successfully
638	Time passed to Delay_Time or Send_Event_Chn more than 23 days from current time
639	Illegal date passed to Set_Time, or illegal date from system clock in Get_Time
640	RS-232 driver called with wrong version number
641	RS-232 read or write initiated with illegal parameter
642	Unimplemented or unsupported RS-232 driver function
646	No memory available to initialize RS-232
647	Unexpected RS-232 timer interrupt
648	Unpermitted RS-232 initialization, or disconnect detected
649	Illegal device control parameters to RS-232
652	N-port driver not initialized prior to ProFile
653	No room in sysglobal to initialize ProFile
654	Hard error status returned from drive
655	Wrong call version to ProFile
656	Unpermitted ProFile function
657	Illegal device control parameter to ProFile
658	Premature end of file when reading from driver
659	Corrupt File System header chain found in driver
660	Cable disconnected
662	Parity error while sending command or writing data to ProFile
663	Checksum error or CRC error or parity error in data read
666	Timeout
670	Bad command response from drive
671	Illegal length specified (must = 1 on input)
672	Unimplemented console driver function
673	No memory available to initialize console
674	Console driver called with wrong version number

675	Illegal device control
680	Wrong call version to serial driver
682	Unpermitted serial driver function
683	No room in sysglobal to initialize serial driver
685	Eject not allowed this device
686	No room in sysglobal to initialize n-port card driver
687	Unpermitted n-port card driver function
688	Wrong call version to n-port card driver
690	Wrong call version to parallel printer
691	Illegal parallel printer parameters
692	N-port card not initialized prior to parallel printer
693	No room in sysglobal to initialize parallel printer
694	Unimplemented parallel printer function
695	Illegal device control parameters (parallel printer)
696	Printer out of paper
698	Printer offline
699	No response from printer
700	Mismatch between loader version number and Operating System version number
701	OS exhausted its internal space during startup
702	Cannot make system process
703	Cannot kill pseudo-outer process
704	Cannot create driver
706	Cannot initialize floppy disk driver
707	Cannot initialize the File System volume
708	Hard disk mount table unreadable
709	Cannot map screen data
710	Too many slot-based devices
724	The boot tracks do not know the right File System version
725	Either damaged File System or damaged contents
726	Boot device read failed
727	The OS will not fit into the available memory
728	SYSTEM.OS is missing
729	SYSTEM.CONFIG is corrupt
730	SYSTEM.OS is corrupt
731	SYSTEM.DEBUG or SYSTEM.DEBUG2 is corrupt
732	SYSTEM.LLD is corrupt
733	Loader range error
734	Wrong driver is found. For instance, storing a diskette loader on a ProFile
735	SYSTEM.LLD is missing
736	SYSTEM.UNPACK is missing
737	Unpack of SYSTEM.OS with SYSTEM.UNPACK failed
801	IOResult <> 0 on I/O using the Monitor
802	Asynchronous I/O request not completed successfully
803	Bad combination of mode parameters
806	Page specified is out of range

809	Invalid arguments (page, address, offset, or count)
810	The requested page could not be read in
816	Not enough sysglobal space for File System buffers
819	Bad device number
820	No space in sysglobal for asynchronous request list
821	Already initialized I/O for this device
822	Bad device number
825	Error in parameter values (Allocate)
826	No more room to allocate pages on device
828	Error in parameter values (Deallocate)
829	Partial deallocation only (ran into unallocated region)
835	Invalid s-file number
837	Unallocated s-file or I/O error
838	Map overflow: s-file too large
839	Attempt to compact file past PEOF
841	Unallocated s-file or I/O error
843	Requested exact fit, but one could not be provided
847	Requested transfer count is <= 0
848	End of file encountered
849	Invalid page or offset value in parameter list
852	Bad unit number
854	No free slots in s-list directory (too many s-files)
855	No available disk space for file hints
856	Device not mounted
857	Empty, locked, or invalid s-file
861	Relative page is beyond PEOF (bad parameter value)
864	No sysglobal space for volume bitmap
866	Wrong FS version or not a valid Lisa FS volume
867	Bad unit number
868	Bad unit number
869	Unit already mounted (mount)/no unit mounted
870	No sysglobal space for DCB or MDDF
871	Parameter not a valid s-file ID
872	No sysglobal space for s-file control block
873	Specified file is already open for private access
874	Device not mounted
875	Invalid s-file ID or s-file control block
879	Attempt to postion past LEOF
881	Attempt to read empty file
882	No space on volume for new data page of file
883	Attempt to read past LEOF
884	Not first auto-allocation, but file was empty
885	Could not update filesize hints after a write
886	No syslocal space for I/O request list
887	Catalog pointer does not indicate a catalog (bad parameter)
888	Entry not found in catalog
890	Entry by that name already exists

891	Catalog is full or is damaged
892	Illegal name for an entry
894	Entry not found, or catalog is damaged
895	Invalid entry name
896	Safety switch is on--cannot kill entry
897	Invalid bootdev value
899	Attempt to allocate a pipe
900	Invalid page count or FCB pointer argument
901	Could not satisfy allocation request
921	Pathname invalid or no such device
922	Invalid label size
926	Pathname invalid or no such device
927	Invalid label size
941	Pathname invalid or no such device
944	Object is not a file
945	File is not in the killed state
946	Pathname invalid or no such device
947	Not enough space in syslocal for File System refdb
948	Entry not found in specified catalog
949	Private access not allowed if file already open shared
950	Pipe already in use, requested access not possible or dwrite not allowed
951	File is already opened in private mode
952	Bad refnum
954	Bad refnum
955	Read access not allowed to specified object
956	Attempt to position FMARK past LEOF not allowed
957	Negative request count is illegal
958	Nonsequential access is not allowed
959	System resources exhausted
960	Error writing to pipe while an unsatisfied read was pending
961	Bad refnum
962	No WRITE or APPEND access allowed
963	Attempt to position FMARK too far past LEOF
964	Append access not allowed in absolute mode
965	Append access not allowed in relative mode
966	Internal inconsistency of FMARK and LEOF (warning)
967	Nonsequential access is not allowed
968	Bad refnum
971	Pathname invalid or no such device
972	Entry not found in specified catalog
974	Bad refnum
977	Bad refnum
978	Page count is nonpositive
979	Not a block-structured device
981	Bad refnum
982	No space has been allocated for specified file
983	Not a block-structured device

985	Bad refnum
986	No space has been allocated for specified file
987	Not a block-structured device
988	Bad refnum
989	Caller is not a reader of the pipe
990	Not a block-structured device
994	Invalid refnum
995	Not a block-structured device
999	Asynchronous read was unblocked before it was satisfied
1021	Pathname invalid or no such entry
1022	No such entry found
1023	Invalid newname, check for '-' in string
1024	New name already exists in catalog
1031	Pathname invalid or no such entry
1032	Invalid transfer count
1033	No such entry found
1041	Pathname invalid or no such entry
1042	Invalid transfer count
1043	No such entry found
1051	No device or volume by that name
1052	A volume is already mounted on device
1053	Attempt to mount temporarily unmounted boot volume just unmounted from this Lisa
1054	The bad block directory of the diskette is invalid
1061	No device or volume by that name
1062	No volume is mounted on device
1071	Not a valid or mounted volume for working directory
1091	Pathname invalid or no such entry
1092	No such entry found
1101	Invalid device name
1121	Invalid device, not mounted, or catalog is damaged
1128	Invalid pathname, device, or volume not mounted
1130	File is protected; cannot open due to protection violation
1131	No device or volume by that name
1132	No volume is mounted on that device
1133	No more open files in the file list of that device
1134	Cannot find space in sysglobal for open file list
1135	Cannot find the open file entry to modify
1136	Boot volume not mounted
1137	Boot volume already unmounted
1138	Caller cannot have higher priority than system processes when calling ubd
1141	Boot volume was not unmounted when calling rbd
1142	Some other volume still mounted on the boot device when calling rbd
1143	No sysglobal space for MDDF to do rbd
1144	Attempt to remount volume which is not the temporarily unmounted boot volume

1145	No sysglobal space for bit map to do rbd
1158	Track-by-track copy buffer is too small
1159	Shutdown requested while boot volume was unmounted
1160	Destination device too small for track-by-track copy
1161	Invalid final shutdown mode
1162	Power is already off
1163	Illegal command
1164	Device is not a diskette device
1165	No volume is mounted on the device
1166	A valid volume is already mounted on the device
1167	Not a block-structured device
1168	Device name is invalid
1169	Could not access device before initialization using default device parameters
1170	Could not mount volume after initialization
1171	'-' is not allowed in a volume name
1172	No space available to initialize a bitmap for the volume
1176	Cannot read from a pipe more than half of its allocated physical size
1177	Cannot cancel a read request for a pipe
1178	Process waiting for pipe data got unblocked because last pipe writer closed it
1180	Cannot write to a pipe more than half of its allocated physical size
1181	No system space left for request block for pipe
1182	Writer process to a pipe got unblocked before the request was satisfied
1183	Cannot cancel a write request for a pipe
1184	Process waiting for pipe space got unblocked because the reader closed the pipe
1186	Cannot allocate space to a pipe while it has data wrapped around
1188	Cannot compact a pipe while it has data wrapped around
1190	Attempt to access a page that is not allocated to the pipe
1191	Bad parameter
1193	Premature end of file encountered
1196	Something is still open on device--cannot unmount
1197	Volume is not formatted or cannot be read
1198	Negative request count is illegal
1199	Function or procedure is not yet implemented
1200	Illegal volume parameter
1201	Blank file parameter
1202	Error writing destination file
1203	Invalid UCSD directory
1204	File not found
1210	Boot track program not executable
1211	Boot track program too big
1212	Error reading boot track program
1213	Error writing boot track program
1214	Boot track program file not found
1215	Cannot write boot tracks on that device

1216	Could not create/close internal buffer
1217	Boot track program has too many code segments
1218	Could not find configuration information entry
1219	Could not get enough working space
1220	Premature EOF in boot track program
1221	Position out of range
1222	No device at that position
1225	Scavenger has detected an internal inconsistency symptomatic of a software bug
1226	Invalid device name
1227	Device is not block structured
1228	Illegal attempt to scavenge the boot volume
1229	Cannot read consistently from the volume
1230	Cannot write consistently to the volume
1231	Cannot allocate space (Heap segment)
1232	Cannot allocate space (Map segment)
1233	Cannot allocate space (SFDB segment)
1237	Error rebuilding the volume root directory
1240	Illegal attempt to scavenge a non-OS-formatted volume
1296	Bad string argument has been passed
1297	Entry name for the object is invalid (on the volume)
1298	S-list entry for the object is invalid (on the volume)
1807	No disk in floppy drive
1820	Write-protect error on floppy drive
1822	Unable to clamp floppy drive
1824	Floppy drive write error
1882	Bad response from ProFile
1885	ProFile timeout error
1998	Invalid parameter address
1999	Bad refnum
6001	Attempt to access unopened file
6002	Attempt to reopen a file which is not closed using an open FIB (file info block)
6003	Operation incompatible with access mode with which file was opened
6004	Printer offline
6005	File record type incompatible with character device (must be byte sized)
6006	Bad integer (read)
6010	Operation incompatible with file type or access mode
6081	Premature end of exec file
6082	Invalid exec (temporary) file name
6083	Attempt to set prefix with null name
6090	Attempt to move console with exec or output file open
6101	Bad real (read)
6151	Attempt to reinitalize heap already in use
6152	Bad argument to NEW (negative size)
6153	Insufficient memory for NEW request

6154	Attempt to RELEASE outside of heap

Operating System Error Codes

The error codes listed below are generated only when a nonrecoverable error occurs while in Operating System code.

10050	Request block is not chained to a PCB (Unblk_Req)
10051	Bld_Req is called with interrupts off
10100	An error was returned from SetUp_Directory or a Data Segment routine (Setup_IUInfo)
10102	Error > 0 trying to create shell (Root)
10103	Sem_Count > 1 (Init_Sem)
10104	Could not open event channel for shell (Root)
10197	Automatic stack expansion fault occurred in system code (Check_Stack)
10198	Need_Mem set for current process while scheduling is disabled (SimpleScheduler)
10199	Attempt to block for reason other than I/O while scheduling is disabled (SimpleScheduler)
10201	Hardware exception occurred while in system code
10202	No space left from Sigl_Excep call in Hard_Excep
10203	No space left from Sigl_Excep call in Nmi_Excep
10205	Error from Wait_Event_Chn called in Excep_Prolog
10207	No system data space in Excep_Setup
10208	No space left from Sigl_Excep call in range error
10212	Error in Term_Def_Hdl from Enable_Excep
10213	Error in Force_Term_Excep, no space in Enq_Ex_Data
10401	Error from Close_Event_Chn in Ec_Cleanup
10582	Unable to get space in Freeze_Seg
10590	Fatal memory parity error
10593	Unable to move memory manager segment during startup
10594	Unable to swap in a segment during startup
10595	Unable to get space in Extend_MMlist
10596	Trying to alter size of segment that is not data or stack (Alt_DS_Size)
10597	Trying to allocate space to an allocated segment (Alloc_Mem)
10598	Attempting to allocate a nonfree memory region (Take_Free)
10600	Error attempting to make timer pipe
10601	Error from Kill_Object of an existing timer pipe
10602	Error from second Make_Pipe to make timer pipe
10603	Error from Open to open timer pipe
10604	No syslocal space for head of timer list
10605	Error during allocate space for timer pipe, or interrupt from nonconfigured device
10609	Interrupt from nonconfigured device
10610	Error from Info about timer pipe
10611	Spurious interrupt from floppy drive #2
10612	Spurious interrupt from floppy drive #1, or no syslocal space for timer list element
10613	Error from Read_Data of timer pipe

10614	Actual returned from Read_Data is not the same as requested from timer pipe
10615	Error from open of the receiver's event channel
10616	Error from Write_Event to the receiver's event channel
10617	Error from Close_Event_Chn on the receiver's pipe
10619	No sysglobal space for timer request block
10624	Attempt to shut down floppy disk controller while drive is still busy
10637	Not enough memory to initialize system timeout drives
10675	Spurious timeout on console driver
10699	Spurious timeout on parallel printer driver
10700	Mismatch between loader version number and Operating System version number
10701	OS exhausted its internal space during startup
10702	Cannot make system process
10703	Cannot kill pseudo-outer process
10704	Cannot create driver
10706	Cannot initialize floppy disk driver
10707	Cannot initialize the File System volume
10708	Hard disk mount table unreadable
10709	Cannot map screen data
10710	Too many slot-based devices
10724	The boot tracks do not know the right File System version
10725	Either damaged File System or damaged contents
10726	Boot device read failed
10727	The OS will not fit into the available memory
10728	SYSTEM.OS is missing
10729	SYSTEM.CONFIG is corrupt
10730	SYSTEM.OS is corrupt
10731	SYSTEM.DEBUG or SYSTEM.DEBUG2 is corrupt
10732	SYSTEM.LLD is corrupt
10733	Loader range error
10734	Wrong driver is found. For instance, storing a diskette loader on a ProFile
10735	SYSTEM.LLD is missing
10736	SYSTEM.UNPACK is missing
10737	Unpack of SYSTEM.OS with SYSTEM.UNPACK failed
11176	Found a pending write request for a pipe while in Close_Object when it is called by the last writer of the pipe
11177	Found a pending read request for a pipe while in Close_Object when it is called by the (only possible) reader of the pipe
11178	Found a pending read request for a pipe while in Read_Data from the pipe
11180	Found a pending write request for a pipe while in Write_Data to the pipe
118xx	Error xx from diskette ROM (See OS errors 18xx)
11901	Call to Getspace or Relspace with a bad parameter, or free pool is bad

Appendix B
Workshop Character Set

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	NUL	DLE	SP	0	@	P	`	p	▨	▨	▨	▨	▨	▨	▨	▨
1	SOH	DC1	!	1	A	Q	a	q	▨	▨	▨	▨	▨	▨	▨	▨
2	STX	DC2	"	2	B	R	b	r	▨	▨	▨	▨	▨	▨	▨	▨
3	ETX	DC3	#	3	C	S	c	s	▨	▨	▨	▨	▨	▨	▨	▨
4	EOT	DC4	$	4	D	T	d	t	▨	▨	▨	▨	▨	▨	▨	▨
5	ENQ	NAK	%	5	E	U	e	u	▨	▨	▨	▨	▨	▨	▨	▨
6	ACK	SYN	&	6	F	V	f	v	▨	▨	▨	▨	▨	▨	▨	▨
7	BEL	ETB	'	7	G	W	g	w	▨	▨	▨	▨	▨	▨	▨	▨
8	BS	CAN	(	8	H	X	h	x	▨	▨	▨	▨	▨	▨	▨	▨
9	HT	EM	)	9	I	Y	i	y	▨	▨	▨	▨	▨	▨	▨	▨
A	LF	SUB	*	:	J	Z	j	z	▨	▨	▨	▨	▨	▨	▨	▨
B	VT	ESC	+	;	K	[	k	{	▨	▨	▨	▨	▨	▨	▨	▨
C	FF	FS	,	<	L	\	l	|	▨	▨	▨	▨	▨	▨	▨	▨
D	CR	GS	-	=	M	]	m	}	▨	▨	▨	▨	▨	▨	▨	▨
E	SO	RS	.	>	N	^	n	~	▨	▨	▨	▨	▨	▨	▨	▨
F	SI	US	/	?	O	_	o	DEL	▨	▨	▨	▨	▨	▨	▨	▨

The first 32 characters and DEL are nonprinting control codes.

The shaded area is reserved for future use.

Appendix C
Screen Control Characters

To perform standard screen control functions in Pascal, use the ScreenCtr procedure of PASLIBCALL as detailed in Section 5.4. For an alternative method of screen control, you can use **WRITE** or **WRITELN**'s with the corresponding character string from Table C-1.

In BASIC, you should use **PRINT** with the **CHR$** function and the argument that corresponds to the desired action. For example:

```
10    print chr$(27); chr$(42); chr$(10); chr$(10)
20    end
run
```

should erase the screen, and position the cursor on the third line.

Table C-1
Screen Control Character Strings

	single-character string			2-character string		
Desired function	ASCII Char	HEX	Decimal	ASCII Char	HEX	Decimal
position to home		1E	30			
one position left	BS	8	8			
one position right	FF	C	12			
position up one line	VT	B	11			
position down one line	LF	A	10			
erase to end of line				ESC-T	1B-54	27-84
erase to end of screen				ESC-Y	1B-59	27-89
erase screen				ESC-*	1B-2A	27-42

Appendix D
Common Problems

Common Problems

This section presents the most common problems that programmers seem to have with the Workshop with suggestions for handling them.

D.1 What to Do When You Find Yourself in the Debugger

You can tell you have entered the Debugger when you suddenly end up with cryptic looking numbers and symbols on your screen. You are actually viewing the alternate screen, and the numbers and symbols are a disassembly of the code where you have stopped and the values of the machine registers. To return to the normal screen to see where you were before you entered the Debugger, hold down the [OPTION] key and press the [ENTER] key. Additional information on the alternate screen is available in Section 3.2.

Often the Debugger display will include suggestions for what to do next, such as "Press g to continue". Figure D-1 is an example of what appears on the screen when you enter the Debugger.

```
Level 7 Interrupt
LOCALPRO+001A 1D40 FFF5      PC       MOVE.B  D0,$FFF5(A6)
PC=00240022 SR=0000  O  US=00F7FBEC SS=00CBFEE0 DO=1 P#=00019
D0=00100009 D1=00000008 D2=000000C0 D3=000264A7
D4=00000001 D5=4EF90084 D6=12CC4EF9 D7=00840000
A0=00F8126E A1=00CCA22A A2=00240060 A3=00CCA22A
A4=00CCA22A A5=00F7FC44 A6=00F7FBFA A7=00F7FBEC
>
```

Figure D-1
Debugger Screen Display

You can enter the Debugger in a number of ways, most commonly by having an error in your program, pressing the NMI (nonmaskable interrupt) key, or having a memory parity error. The NMI key is the "-" key on the numeric keypad.

More information on handling the Debugger is given in Chapter 8. Section 8.2 will help you handle accidental entry into the Debugger. Section 8.3.2 contains information aboout Pascal run-time errors, particularly range errors.

D.2 How to Stop Your Program

If your program has been running for longer than you think it needs to, it might be in an infinite loop. Before you stop the program, you should:

- Check the alternate screen. Maybe your program is waiting for input.
- Try -period to see if it responds.

If neither of these actions works, press the NMI key, which stops your program in the Debugger. See Section 8.2 for information about what you can do from the Debugger.

D.3 What to Do When a Diskette Won't Eject

The eject request buttons are only recognized after the Workshop system does a Pascal I/O operation. Thus when you press an eject button, nothing will happen until you press a key, or I/O happens for some other reason. (When you are in the Editor, the Preferences tool, or TransferProgram, you do not need to hit a key after pressing the diskette button.)

In general, if a diskette will not eject, it means that the file system still has some file open on it. Use the Online command to check the open count, which will tell you if any files are still open. Then use the List command from the File Manager to list the contents of the diskette. If some files are open, there is probably a resident process that has a file open or a data segment open that has been mapped to the disk. Use the ManageProcess subsystem in the System Manager to kill the process. This will close the files and the disk will eject.

Further information on the List command can be found in Sections 2.3 and 2.6. The ManageProcess subsystem is described in Section 3.4.

D.4 What to Do When You Get a Range Error

A range error drops you into the Debugger. Instructions for handling range errors are in Section 8.3.2.

D.5 What to Do When the System Does Not Respond

Some of the reasons your Workshop might not respond are:

1. You might be running a program with an infinite loop.
2. You might have stopped console output by pressing -S.
3. You might have the alternate screen showing.
4. You might have altered the NMI character.

Press the NMI key (the "-"key on the numeric keypad) to drop into the Debugger. See Section 8.2 for further instructions.

If pressing the NMI key does not work, power off your Lisa and reboot the system.

D.6 What to Do with a Runaway Exec File

If you think that your exec file has gone wild, how do you stop it?

When the exec file processor has finished processing your exec file (s), it has created a temporary file with the stream of characters that are to perform the actions in the exec file. The Workshop then sets the run-time environment so that standard input comes from the temporary file, and begins executing the commands in the temporary file. While they are executing, the Workshop ignores the keyboard, although the characters you type will be remembered.

You can terminate standard Workshop programs by pressing -period, although termination might not be immediate if the program being run does not recognize -period.

NOTE

Note that most Workshop tools check for -period from the keyboard even when running under exec files. This means that you can abort Workshop tools in exec files.

Unless user programs are written to recognize the -period key combination as an abort mechanism, pressing those keys will not terminate the exec file if a user program is being run. (See PASLIBCALL, Section 5.4, for information on the function PAbortFlag, which tells whether or not those keys have been pressed.) If this is the case, you can either:

- wait for the user program to terminate so that -period can be recognized by something else, or
- press the NMI key, which forces the system into the Debugger.

If the user program does recognize -period, pressing it will terminate the program but not the exec file. To terminate the exec file, wait until the Workshop prompt appears and press -period again.

See Section 8.2 for instructions on how to stop a user program early.

NOTES

Index

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

ALL PRINTING was done with an Apple Dot Matrix Printer.

the Lisa™
...we use it ourselves.

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.

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

[ ] other ______________________________

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

Comments ______________________________

What makes this manual easy to use? ______________________________

____

What makes this manual hard to use? ______________________________

____

What do you like most about the manual? ______________________________

____

What do you like least about the manual? ______________________________

____

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 languages do you use on your Lisa? (check each)

[ ] Pascal [ ] BASIC [ ] COBOL [ ] other ______________________________

How long have you been programming?

[ ] 0-1 years [ ] 1-3 [ ] 4-7 [ ] over 7 [ ] not a programmer

What is your job title? ______________________________

Have you completed:

[ ] high school [ ] some college [ ] BA/BS [ ] MA/MS [ ] more

What magazines do you read? ______________________________

____

____

Other comments (please attach more sheets if necessary) ______________________________

____

____

____

FOLD

FOLD

PLAC
STAN
HERL

apple computer

POS Publications Department

20525 Mariani Avenue

Cupertino, California 95014

TAPE OR STAPLE