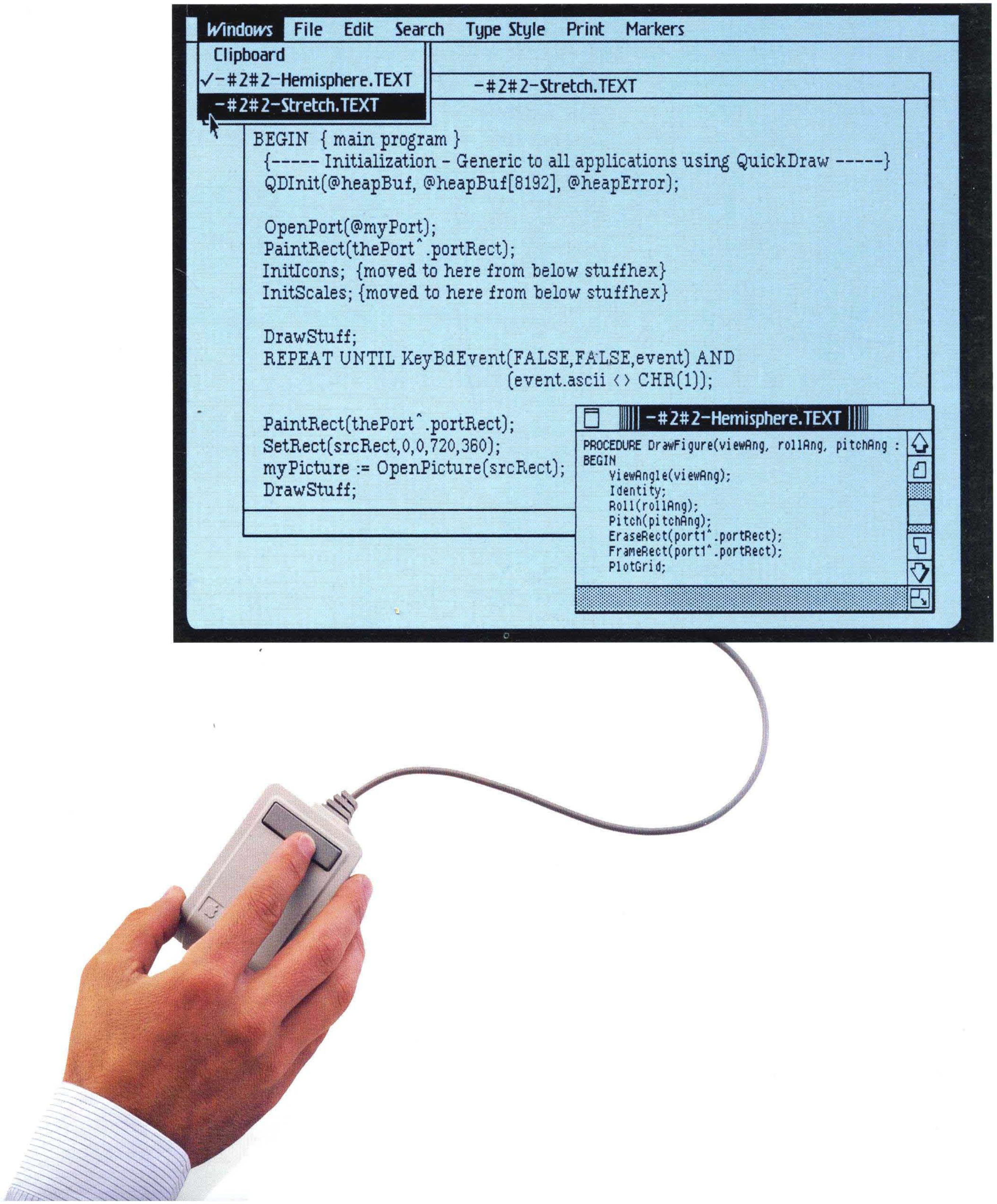
Lisa Workshop User's Guide



```
BEGIN { main program }
 {----- Initialization - Generic to all applications using QuickDraw -----}
 QDInit(@heapBuf, @heapBuf[8192], @heapError);

 OpenPort(@myPort);
 PaintRect(thePort^.portRect);
 InitIcons; {moved to here from below stuffhex}
 InitScales; {moved to here from below stuffhex}

 DrawStuff;
 REPEAT UNTIL KeyBdEvent(FALSE,FALSE,event) AND
                          (event.ascii <> CHR(1));

 PaintRect(thePort^.portRect);
 SetRect(srcRect,0,0,720,360);
 myPicture := OpenPicture(srcRect);
 DrawStuff;
```

```
PROCEDURE DrawFigure(viewAng, rollAng, pitchAng :
BEGIN
    ViewAngle(viewAng);
    Identity;
    Roll(rollAng);
    Pitch(pitchAng);
    EraseRect(port1^.portRect);
    FrameRect(port1^.portRect);
    PlotGrid;
```

Workshop User's Guide for the Lisa

-6148-B

Simultaneously published in the U.S.A. and Canada.

Reorder Apple Product #620-6148-B.

Contents

Preface

The *Workshop User's Guide for the Lisa* describes the Workshop environment for developing, testing, and running programs written in assembly language, Pascal, and other high-level languages.

This manual is written for programmers who are familiar with the Lisa® system.

Related Documents

For all programmers:

- *Lisa 2 Owner's Guide*

For Pascal and assembly-language programmers:

- *Pascal Reference Manual for the Lisa*
- *M68000 16/32-Bit Microprocessor: Programmer's Reference Manual*
- *Operating System Reference Manual for the Lisa*

For BASIC programmers:

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

For Macintosh programmers:

- *Inside Macintosh*

What This Manual Contains

The contents of the *Workshop User's Guide* are summarized below.

- **Chapter 1, Introduction**, describes the Workshop environment for program development and discusses the conventions used by the Workshop tools. It tells you how to install the Workshop and how to use the main Workshop command line.
- **Chapter 2, The File Manager**, describes file-naming conventions; tells you how to list directories; how to copy, rename, and delete files; and how to mount, unmount, initialize, and repair volumes.
- **Chapter 3, The System Manager**, tells you how to set system defaults and specify device connections.
- **Chapter 4, The Editor**, tells you how to create, modify, search, save, and print text files.
- **Chapter 5, The Pascal Compiler**, tells you how to use the Compiler and the Code Generator to turn a Pascal source program into an object file.

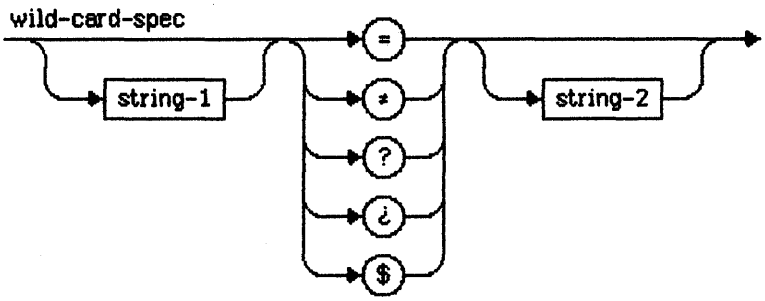
- **Chapter 6, The Assembler**, tells you how to assemble a 68000 assembly-language source program into an object file; it also describes how your assembly-language program can communicate with a Pascal program.
- **Chapter 7, The Linker**, tells you how to combine compiled or assembled object files into a single executable object file. It discusses regular and intrinsic units, external names, and segmentation.
- **Chapter 8, The Debugger**, describes how to set breakpoints in your program; how to display memory and registers; how to trace the program flow; and other run-time debugging functions.
- **Chapter 9, Exec Files**, tells you how to create a file of commands to run programs under the Workshop automatically; the commands consist of Workshop and program commands plus a special high-level exec language.
- **Chapter 10, The Transfer Program**, describes a data communications package for transferring keyboard input or text files between the Lisa and a remote computer.
- **Chapter 11, The Utilities**, documents a set of utility programs that perform file comparing, file searching, cross-referencing, Lisa-Macintosh communication, and various other functions.
- **Appendix A, Error Messages**, provides the text of error messages from the Assembler, the Linker, ObjIOLib, SULib, PasLib, the Exec Processor, and the Lisa Operating System.
- **Appendix B, Lisa Extended Character Set**, is a table of ASCII character codes and special characters.
- **Appendix C, Screen Control Characters**, contains information on screen control in Pascal and BASIC.
- **Appendix D, Common Problems**, contains troubleshooting suggestions.

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 show how to enter filenames and other syntactic constructions. For example, the following syntax diagram from Chapter 2 describes a wild-card-spec:



Start at the left and follow the arrows through the diagram. Alternate paths are possible. Every path that begins at the leftmost arrow and ends at the rightmost arrow is valid.

Circles and ovals contain reserved words, operators, or punctuation symbols that must be written as shown, except that capitalization is not required.

Boxes contain the name of a syntactic construction that is described by another syntax diagram. Replace the name with an instance of the construction.

The wild-card-spec diagram embodies the following rules:

- A wild-card-spec can begin with an optional string (String-1).
- A wild-card-spec must contain =, ?, $, *, or ¿.
- The =, ?, $, *, or ¿ can be followed by an optional string (String-2).

Here are some examples that conform to the wild-card-spec syntax:

```
-vol-$.text
?.obj
=
```

Chapter 1
Introduction

Introduction

1.1 Introduction to the Workshop

The Workshop contains a collection of tools for preparing and running programs. These tools allow you to

- Configure the Lisa and set system defaults.
- Write, compile or assemble, link, and run programs.
- Debug programs that run under the Lisa Operating System.
- Create and run files of Workshop and program commands using a high-level exec language.
- Initialize, list, copy, rename, delete, compare, search, cross-reference, and otherwise view and modify files, catalogs, and volumes.
- Transfer data between the Lisa and a remote computer.

The Workshop lets you develop Macintosh programs on the Lisa. You can also transfer files between Lisa and Macintosh by running the MacCom utility program. With MacWorks, you can even run Macintosh programs on the Lisa. Several programming languages are available, including 68000 assembly language, Pascal, BASIC, C, and others.

The Workshop tools run under the Lisa Operating System (OS). The OS enables programs to do file handling, process management, and memory management; it provides some facilities for which there are no parallels in the Workshop. If you are writing programs to run under the Lisa OS, you should be familiar with the *Operating System Reference Manual for the Lisa*. If you are writing programs to run under the Macintosh OS, you should be familiar with *Inside Macintosh*.

You use the main Workshop features by typing a single character response to a *command line* that lists available programs. The main command line is described in this chapter. The File Manager and the System Manager have their own command lines, described in chapters 2 and 3.

1.2 Starting the Workshop

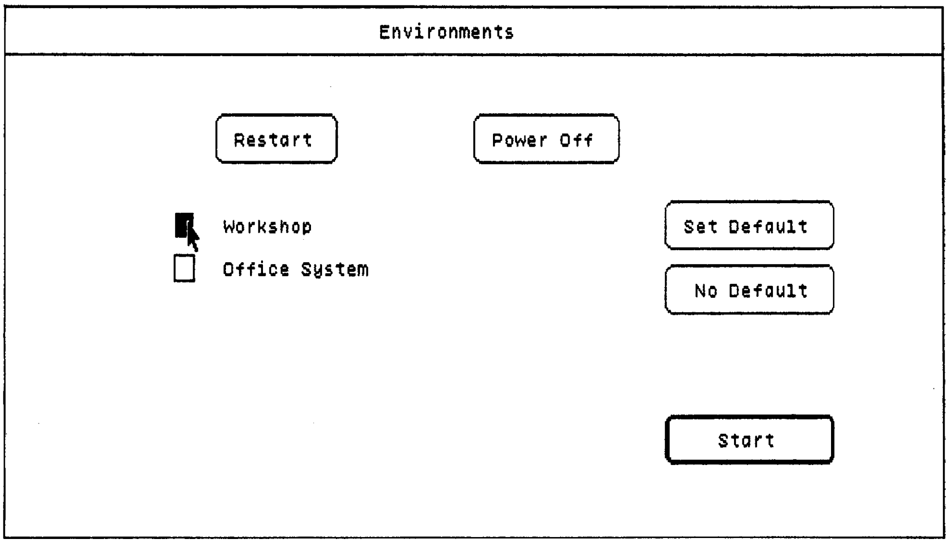
If you have already installed the Workshop from micro diskettes, boot from the Workshop startup disk. Either the Workshop command line or the Environments window appears on the screen. For a description of the Workshop command line, see Section 1.5.1. The Environments window allows you to start the Workshop or another environment such as the Office System. For a description of the Environments window, see Section 1.2.1.

If you have not yet installed the Workshop software onto a startup disk, follow the instructions in Section 1.2.2.

1.2.1 The Environments Window

If your startup disk contains only the Workshop environment, booting automatically starts the Workshop and its command line appears. If the startup disk contains more than one environment, the Environments window, below, lets you select which environment you want. The window displays a checkbox for each environment plus the following 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	Always display the Environments window on startup.



To start the Workshop or another environment from the Environments window, move the pointer to the check box of the environment you want to start and click the mouse button. Then move the pointer to the Start button and click.

To return to the Environments window from the Workshop, use the Quit command in the Workshop command line. Reply **Y** when asked if you really want to leave the Workshop. Then type **A** for Another_shell.

To go to the Workshop or another environment automatically at startup time, select the environment's check box and click Set Default. To go to the Environments window automatically at startup time, click No Default.

To go to the Environments window when booting the system, press any key while the Lisa is starting up.

You can create your own environments. Any object file named SHELL.filename will appear in the Environments window as an alternate environment.

1.2.2 Installation Overview

The Workshop Pascal software comes on nine micro diskettes, "Workshop Pascal 1-9." Installing the Workshop involves transferring copies of files from these micro diskettes to a hard disk that you designate. This hard disk will then be called a *startup disk* or *boot disk*.

Here is an overview of the steps you must follow to properly install the Workshop Pascal software. The actual instructions are in the next section.

- Physically hook up the Lisa and any peripheral devices, such as printers and external hard disks. If you have not yet physically set up the Lisa hardware, turn to Appendix A, Setting Up Your System, in the *Lisa 2 Owner's Guide*.
- Insert the "Workshop Pascal 1" micro diskette and use it to install the startup software from the first six micro diskettes ("Workshop Pascal 1-6") onto a hard disk that you designate. These diskettes produce a startup disk containing the minimum Pascal Workshop, which is capable of editing, assembling, compiling, linking, running programs, managing files, and configuring various hardware and software options.

  It will take about 15 minutes to install the minimum Pascal Workshop, a little over 2 minutes per micro diskette.
- Start the Workshop and use the System Manager's Preferences tool to describe your Lisa's particular configuration of disks, printers, and other devices.
- Use the Workshop's File Manager to copy to the hard disk any additional files that you need from the remaining micro diskettes ("Workshop Pascal 7-9").

1.2.3 Installing the Workshop Pascal Software

Follow these step-by-step instructions to install the Workshop Pascal software.

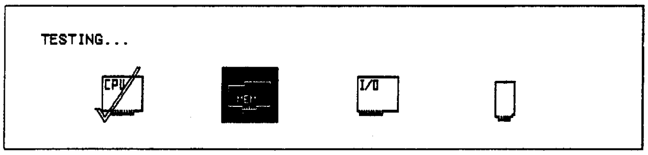
It's a good idea to read through this entire procedure before starting.

1. If the Lisa is on, turn it off by pressing the on-off button.
2. Have the nine micro diskettes handy ("Workshop Pascal 1-9"). If your system has an external hard disk, be sure it is on and the ready light is steady.

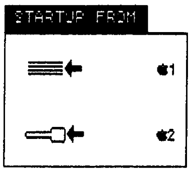
   Do not write-protect the micro diskettes. If you try to install the software from a write-protected diskette, the Lisa will fail, try to boot, and continue to fail.

3. Insert the "Workshop Pascal 1" micro diskette into the drive. Make sure the arrow embossed on the diskette points toward the drive.
4. Turn the Lisa on by pressing the on-off button once. About four seconds later, after you hear a click from the cabinet, hold down the key and type a 2 on the main keyboard. *Do not use the 2 on the numeric keypad.*

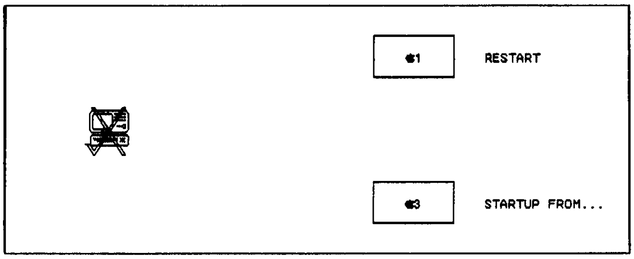
   If you type -2 correctly, the Lisa will go through a self-test that checks to make sure the Central Processing Unit (CPU), Memory (Mem), Input-Output (I/O), and expansion slots are working properly. Proceed to step 5.



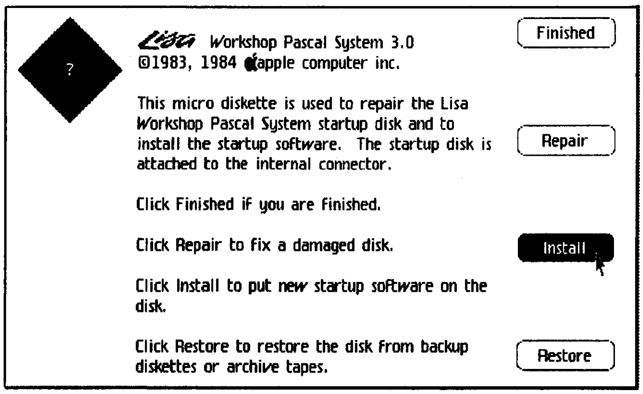
If you type a 2 without pressing the key, the startup menu will appear. Hold down the key and type *2* again.



If you type -2 late--after you hear a second click from the cabinet--this screen will appear. Hold down the key and type *3*. The startup menu described in the preceding paragraph appears. Type -2.



5. When the Main Menu appears, use the mouse to move the pointer to the Install box. Click the mouse button once. The box will darken, indicating that you have selected it.


Lisa Workshop Pascal System 3.0
©1983, 1984 apple computer inc.

This micro diskette is used to repair the Lisa Workshop Pascal System startup disk and to install the startup software. The startup disk is attached to the internal connector.

Click Finished if you are finished.

Click Repair to fix a damaged disk.

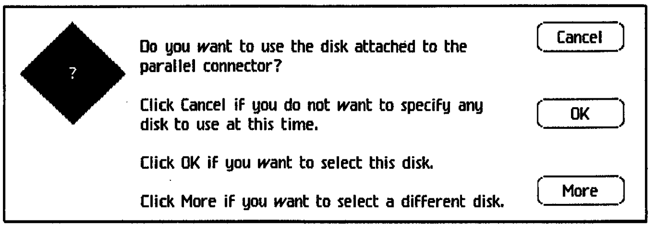
Click Install to put new startup software on the disk.

Click Restore to restore the disk from backup diskettes or archive tapes.

Clicking the Repair box performs the same function as the Scavenge command in the File Manager (see Chapter 2); select it only if your hard disk is damaged and you want to try to preserve its files. The Restore box is of interest only to Office System users.

6. A message will tell you that the Lisa is looking for any attached hard disks. It will only find disks that are attached and powered on.

   Then you will be asked to select the *startup disk*: that is, the disk on which you want to install the Workshop Pascal software.



Do you want to use the disk attached to the parallel connector?

Click Cancel if you do not want to specify any disk to use at this time.

Click OK if you want to select this disk.

Click More if you want to select a different disk.

To select the hard disk identified by the first paragraph of the screen message, click OK.

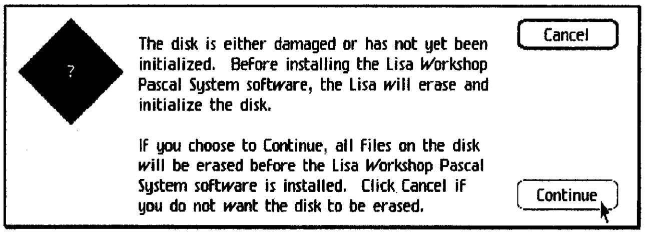
To select a different hard disk, click More. The Lisa will continue looking for other attached disks.

You can keep clicking More each time a disk is presented for approval. If the screen notifies you that no more disks are available, click Retry to start over or click Main Menu to return to the screen shown in Step 5.

7. After you select your startup disk, one of two screens will appear.

   a. If the disk you have chosen is a new disk that has never before been used, or if it is a disk containing software unusable by the Lisa, you will see the message shown below.

   Click Continue.

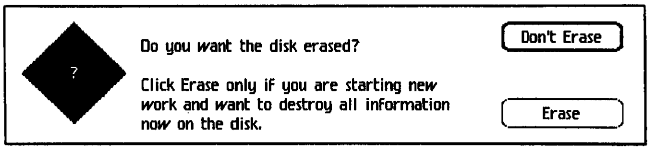


The disk is either damaged or has not yet been initialized. Before installing the Lisa Workshop Pascal System software, the Lisa will erase and initialize the disk.

If you choose to Continue, all files on the disk will be erased before the Lisa Workshop Pascal System software is installed. Click Cancel if you do not want the disk to be erased.

Or

   b. If the hard disk you select as your startup disk already has Lisa files on it, you will be asked if you want to erase it. You can't share the disk with MacWorks unless you initialize or erase it. Click Erase to erase everything on the disk.

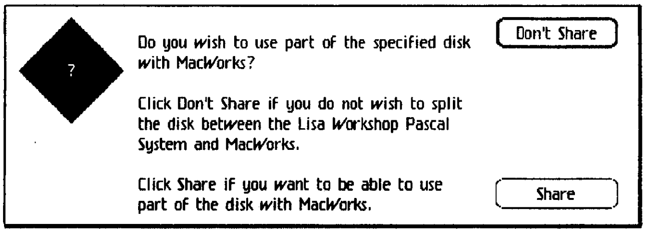

Do you want the disk erased?

Click Erase only if you are starting new work and want to destroy all information now on the disk.

If the hard disk contains files that you want to keep, click Don't Erase; go to step 9.

8. You will be asked if you want to use part of the disk with MacWorks (the Macintosh environment for the Lisa system).

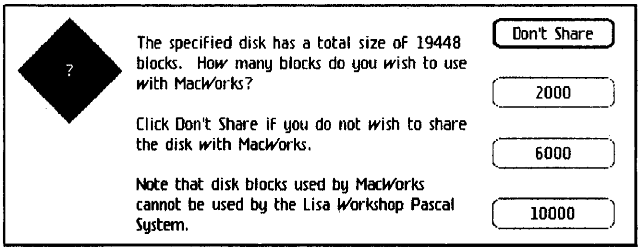
   If you don't want to store Macintosh files on this disk, click Don't Share.


Do you wish to use part of the specified disk with MacWorks?

Click Don't Share if you do not wish to split the disk between the Lisa Workshop Pascal System and MacWorks.

Click Share if you want to be able to use part of the disk with MacWorks.

If you plan to use part of the hard disk for Macintosh files, click Share. You will be asked how much space you want to reserve. The only choice possible for a 5-MegaByte ProFile is 2000 blocks (1 MegaByte). Click one of the buttons.


The specified disk has a total size of 19448 blocks. How many blocks do you wish to use with MacWorks?

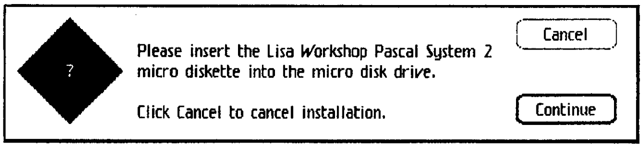
Click Don't Share if you do not wish to share the disk with MacWorks.

Note that disk blocks used by MacWorks cannot be used by the Lisa Workshop Pascal System.

9. A wait message will appear while the startup disk is erased and initialized. It takes a few minutes for the disk to be formatted and initialized.

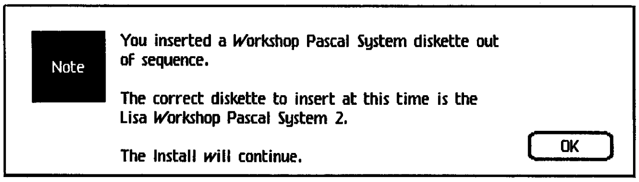
   When the startup disk has been initialized, the Lisa will automatically begin installing the Workshop software from the "Workshop Pascal 1" micro diskette. You will see a message telling you that the startup software is being installed. The first micro diskette will soon be ejected from the drive.

10. When the message on the screen tells you to insert the next micro diskette, remove the "Workshop Pascal 1" micro diskette from the drive and insert the "Workshop Pascal 2" micro diskette. The installation process will automatically continue. You do not need to click either of the boxes shown.


Please insert the Lisa Workshop Pascal System 2 micro diskette into the micro disk drive.

Click Cancel to cancel installation.

If you insert a micro diskette out of sequence, the diskette will be ejected, and this message will appear.


You inserted a Workshop Pascal System diskette out of sequence.

The correct diskette to insert at this time is the Lisa Workshop Pascal System 2.

The Install will continue.

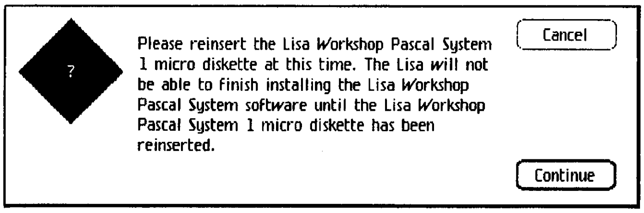
Replace the diskette with the correct one. Click OK; the installation procedure will continue automatically.

11. Insert the remaining diskettes when you are prompted for them.

    If you cancel the automatic installation process before installing software from the first six diskettes, you will have to repeat this procedure beginning with the "Workshop Pascal 1" diskette. You will not be able to use the Workshop until software from the first six diskettes have been installed.

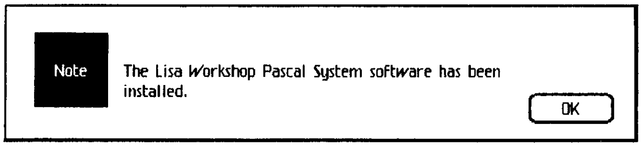
12. After the Lisa has installed the software from the sixth ("Workshop Pascal 6") micro diskette, you will be asked to reinsert the "Workshop Pascal 1" diskette.

    Insert the diskette.



Please reinsert the Lisa Workshop Pascal System 1 micro diskette at this time. The Lisa will not be able to finish installing the Lisa Workshop Pascal System software until the Lisa Workshop Pascal System 1 micro diskette has been reinserted.

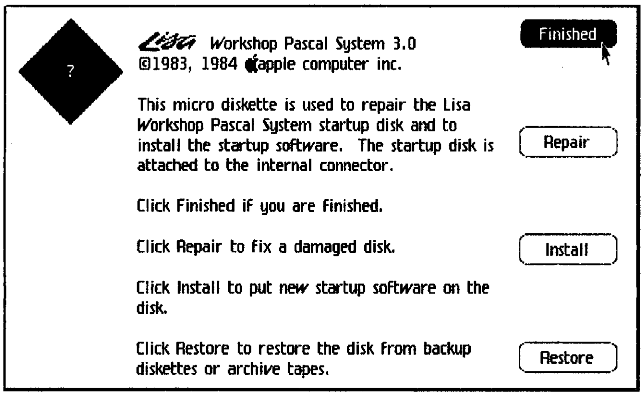
13. When the message informs you that the Workshop Pascal software has been installed, click OK.



The Lisa Workshop Pascal System software has been installed.

14. The Main Menu will reappear. Click Finished.

    The minimum Pascal Workshop is now installed.



Lisa Workshop Pascal System 3.0
©1983, 1984 apple computer inc.

This micro diskette is used to repair the Lisa Workshop Pascal System startup disk and to install the startup software. The startup disk is attached to the internal connector.

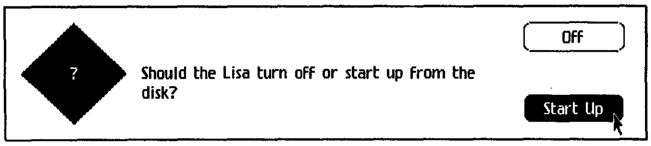
Click Finished if you are finished.

Click Repair to fix a damaged disk.

Click Install to put new startup software on the disk.

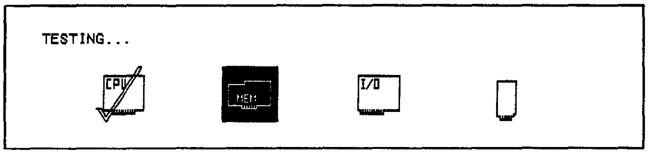
Click Restore to restore the disk from backup diskettes or archive tapes.

15. When this message appears, click Start Up.


Should the Lisa turn off or start up from the disk?

The "Workshop Pascal 1" diskette will be ejected. Put it in a safe place.

Your Lisa will go through a series of self-tests similar to those that occurred when you first turned it on.



16. The Workshop will be started from the startup hard disk. If the Lisa's clock/calendar has not been set, the Workshop asks you to set the correct time and date. You should do this now, because some functions in the Lisa applications require a correct date and time.

17. Use the Workshop System Manager's Preferences tool to tell the Lisa what peripheral devices are physically connected and what default settings to use each time you turn on the Lisa. Starting the Workshop is described in Section 1.3. The System Manager subsystem and the Preferences tool are described in Chapter 3.

18. Decide what files you need from the remaining diskettes ("Workshop Pascal 7-9"). Use the Workshop's File Manager to copy these files to the startup disk (or any hard disk, if you have more than one). See Chapter 2, The File Manager, for instructions. Unless you plan to use a file frequently, you may prefer to access it directly from the micro diskette rather than take up space on the hard disk.

Your Workshop Pascal software is now fully installed.

1.3 Hardware Configuration

In order to use a device with the Workshop, you must do two things: first, tell the Workshop it exists; and second, connect it to the Lisa. Telling the software about the hardware is known as *configuring the system.* The configuration information you provide is saved on the boot disk and in the

Lisa's parameter memory, so you only have to reconfigure if you change to a different boot disk or if you connect or disconnect devices.

If you have just installed new software on your boot disk, you should check its configuration now.

1.3.1 Specifying Hardware Connections

The File Manager's Online command tells you what hardware the Workshop software thinks is connected to your Lisa. If a device that is connected to the Lisa is not listed by Online, use the Preferences command in the System Manager command line to tell the Workshop about the device. If Online pauses unexpectedly while listing devices, or if it reports an error, it is probably looking for a device that the software thinks is connected. If the device is not present, use Preferences to detach it.

Preferences also lets you specify various defaults such as which device to boot from, which printer to associate with the logical printer device (-printer), normal and dimmed brightness levels for the screen, and so on.

1.3.2 Printer Configuration

Before using a printer with the Workshop, you must set up the printer and tell the Workshop where it is connected.

Refer to the instruction manual that came with your printer for directions on how to set it up. If you have more than one printer you will want to configure one of them as the default printer, as described in Section 1.3.2.3.

1.3.2.1 Setting up a Printer

The procedure for setting up a printer varies with the type of printer. Follow the manufacturer's instructions.

During startup, or when you attach a printer using the Preferences tool, the Workshop sends a control sequence to set the printer to 9600 baud, auto line feed, DTR handshake, and no parity. If your printer is an Apple Imagewriter, the default standards which have been factory preset should be satisfactory. However, if you want to modify the performance of the Imagewriter, see the technical specifications in the *Apple Imagewriter User's Manual, Part I: Reference*.

1.3.2.2 Configuring the Workshop 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. Press *P* for Preferences. The Preferences tool is used to set up the configuration of the Lisa system and the Workshop; refer to Section 3.3 for more information on Preferences.
3. Click on the Connect Device Software box to see what devices are connected to the Lisa.

4. Select the connector to which your printer is connected. All devices that can be connected to that connector are displayed.
5. Select Printer; additional configuration options are displayed.
6. When you are finished configuring your printer, select Quit from the File menu.
7. Exit from the System Manager back to the Workshop command line by pressing *Q* for Quit.

1.3.2.3 Specifying a Default Printer

If you have more than one printer connected to your Lisa, you can specify a default printer--the one you can refer to as -printer. First use the Preferences tool to configure the printers and other devices connected to the Lisa. Then choose Select Defaults in Preferences. See Section 3.3, The Preferences Tool, for more information.

Another way to specify the default printer is to type *D* for DefaultPrinter in the System Manager command line and enter the device name of the default printer (for example, **#10#1** or its alias **RS232A**). Or, if you want to keep the current default, press [RETURN]. See Section 3.2, The System Manager Command Line, for more information on the DefaultPrinter command.

The default printer you specify using the Preferences tool is also the default for the Office System if you have it installed; the default printer you specify using the DefaultPrinter command affects only the Workshop environment.

1.4 The Workshop Shell

The Workshop shell is the highest-level program in the Workshop environment. Programs you run from command lines or using the Run command return control to the Workshop shell when they're finished. The shell provides an exec file mechanism and performs a number of automatic actions; its Command Interpreter communicates with you at the level of the *main command line*. The Workshop uses the command line to provide you with access to system functions at a single keystroke.

1.4.1 Exec Files

Exec files let you automate Workshop utilities and user programs, make programmed decisions (for example, whether to recompile a source program), modify the Workshop environment, automate test procedures, and more. Exec source files can contain a high-level command language, Workshop commands, and input to user programs. Common uses of exec files include standard compile procedures and standard application runs. See Chapter 9, Exec Files, for more information.

1.4.2 The Main Command Line

When you enter the Workshop environment, the Workshop's main command line appears at the top of the screen. It shows:

- Two subsystems, the File Manager and the System Manager, that have their own command lines.
- The Run command, which lets you run Workshop utilities and any program that you or someone else wrote to run under the Workshop.
- The main tools provided by the Workshop.

The main command line actually comes in two parts because the screen isn't wide enough to show all the commands on a single line. The first part of the main command line looks like this:

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

You can see the rest of the commands by pressing ?, the last symbol on the line. To return to the first part of the command line, press [RETURN]. The second part looks like this:

Assemble, Generate, MakeBackground, Link, TransferProgram

Type the first letter of a command to use a tool. For example, type *E* or *e* to run the Editor. You can use all of the commands no matter which part of the main command line is showing when you type the command letter. The Workshop looks for the tool on the boot volume; if it doesn't find it there, it looks on the Prefix volumes.

Some commands will ask for additional information. Default values are displayed in square brackets ([default]). To accept a default value, press [RETURN]. If you don't want the default value, type in the value you want and *then* press [RETURN]. If you make a mistake, press [CLEAR] to escape. See Section 1.5.1.2, Defaults in File Name Prompts, for more information.

The main command line commands are described below. The letter you type to access the command is shown in parentheses.

FILE-MGR (F)
The FILE-MGR command give you access to the File Manager subsystem, described in Chapter 2. This subsystem is used to manipulate files, catalogs, and volumes.

SYSTEM-MGR (S)
The SYSTEM-MGR command gives you access to the System Manager subsystem, described in Chapter 3. This subsystem provides various configuration, process management, and utility functions.

Edit (E)
The Edit command gives you access to the Editor in order to create, modify, and print text files. You can use the Editor to write exec files, data files, and programming language source files, as well as memos or other documents. The Editor is described in Chapter 4.

Run (R)
The Run command has two functions. You can use it to execute an *object program* (a Workshop utility program, a user-written program, or any other

software designed to run under the Workshop), and you can use it to cause an *exec file* to be processed and executed.

The Run command asks you what file you want to run. The default is the last program or exec file you ran. To run the same file again, just press [RETURN]. To run a different file, type the program or exec file name followed by [RETURN]. The name of an exec file must be preceded by < or **exec/**.

If the Run command doesn't find the file under the name you supplied, it adds the standard extension if you didn't give one (.OBJ for program files, .TEXT for exec files) and looks for the file again. If you don't specify a volume name, the Run command searches through the first Prefix volume for an exec file or through up to three Prefix volumes for a program file; then, if necessary, it looks on the boot volume. Prefixes can be set through the File Manager's Prefix command.

Debug (D)
The Debug command inserts a breakpoint at the first instruction in your program, so you can use the Debugger. Then it executes the program just as the Run command does. More information on the Debugger can be found in Chapter 8.

Pascal (P)
The Pascal command starts the Pascal Compiler, described in Chapter 5. More information on the Pascal language can be found in the *Pascal Reference Manual for the Lisa.*

Basic (B)
The Basic command starts the BASIC Interpreter. More information on BASIC programming can be found in the *BASIC-Plus User's Guide for the Lisa.*

Quit (Q)
The Quit command lets you leave the Workshop. If you opened files in the Editor and didn't save them, you'll receive a reminder. The following prompt line appears after you confirm that you want to leave the shell:

WorkShop_shell, Another_shell, Reboot, Power_off

Type *W* to return to the Workshop environment.

Type *A* to go to the Environments window. If you have the Lisa Office System installed, you can go from the Workshop to the Office System by means of the Environments window.

Type *R* to reboot the Lisa.

Type *P* to turn off the Lisa.

Assemble (A)

The Assemble command starts the Assembler, described in Chapter 6. Additional information on assembly language can be found in the *M68000 16/32-Bit Microprocessor* manual.

Generate (G)
The Generate command, described in Chapter 5, converts intermediate code files produced by the Pascal Compiler into object code. (The Compiler performs this step automatically unless you specify otherwise.)

MakeBackground (M)
The MakeBackground command lets you run a program as a background process while you continue using the Workshop for other functions. The background process should not display on the console or request keyboard input.

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

TransferProgram (T)
The TransferProgram command starts the Transfer program, described in Chapter 10. This program allows your Lisa to communicate with a remote computer.

1.4.3 Automatic Actions Taken by the Shell

Certain actions are automatically performed by the Workshop shell. These include running a user exec file during startup and shutdown, mounting disks, and establishing the logical console and default printer devices.

1.4.3.1 User Startup and Shutdown Procedures

During startup, the Workshop shell looks for a user exec file named **CISTART.TEXT** and runs it if it exists. You can create your own CISTART (Command Interpreter startup) file to modify the Workshop environment or set up a user application. Any commands that are valid in a normal exec file are valid in CISTART. See Chapter 9 for more information on exec files.

The following CISTART file sets the Validate command so that file transfers are not verified and file selections are not confirmed with messages like "Are you SURE you want to copy...?" (see Section 3.2, The System Manager Command Line, for more information):

```
$EXEC
   S{ys-Mgr}V{alidate}N{o}N{o}Q{uit}
$ENDEXEC
```

You can also create an exec file named **CIFINISH.TEXT** that will be run automatically when you leave the Workshop shell.

1.4.3.2 Automatic Mounting of Disks

Devices must be mounted before you can read from them or write to them. At startup time, the Workshop mounts any physically attached disk that has been logically connected using the Preferences tool. (See Section 3.3.3, Device Connections, for more information.)

1.4.3.3 Automatic Setting of Prefixes

Prefixes tell the Workshop where to look for a file when you don't specify a full pathname. The File Manager's Prefix command lets you specify three levels of prefixes that remain in effect until you change them or until you power off.

You can also set the prefixes so that they are automatically reestablished during startup, by answering *Y* to the Prefix command's question

Initialize this Prefix Set at boot time? (Y or N)

1.4.4 The Main Screen and the Alternate Screen

The Lisa can show you two different displays; they are known as the *main screen* (-MAINCONSOLE) and the *alternate screen* (-ALTCONSOLE). By convention, the Workshop (except for the Debugger) displays output on the main screen; that is, the logical console, -console, is normally set to the main screen. The Debugger uses the alternate screen so that its messages are not intermingled with program output. The Console command in the System Manager lets you choose which screen is the logical console.

To switch to whichever screen is not currently viewed, hold down the Option key on the right side of the Lisa keyboard while you press the Enter key on the numeric keypad.

Your program can direct output to the alternate screen by opening and writing to a file named "-ALTCONSOLE-x", where x is any file name.

1.5 Workshop Conventions and Standards

This section describes file name conventions and other standards used in the Workshop. In general, these features are not available in user programs unless you specifically provide for them. (Refer to The StdUnit Unit in the third binder of this set for more information on how to program these features.)

1.5.1 File System Conventions

This section introduces file naming conventions and tells you how to respond to prompts that ask for a file name. Most of the Workshop tools follow these conventions.

A more complete description of the File System can be found in Chapter 2 and in the *Operating System Reference Manual for the Lisa.*

1.5.1.1 File Names

When the Workshop prompts you for a file, you must supply a valid pathname; the following rules apply:

- A pathname has three parts:

Device, volume, or catalog name	Starts with "-"; defaults to Workshop Prefix 1 if not supplied.
File name	Composed of alphabetic and/or numeric characters; spaces are permitted.
Extension	Composed of alphabetic and/or numeric characters; spaces are permitted. If present, it is the final "." and any characters that follow. The standard extensions are .TEXT, .OBJ, .I, and .LIB.

- The length of the full pathname must not exceed 255 characters. The length between dashes (-) or between a dash and the end of the pathname must not exceed 32 characters.
- Leading and trailing blanks or tab characters will be discarded by the Workshop.
- Uppercase and lowercase are usually preserved as you specify them and are ignored in distinguishing between file names.

When entering a list of files, indicate that you are finished by pressing [RETURN].

1.5.1.2 Defaults in File Name Prompts

Prompts may display default values, shown in square brackets ([ ]). If a file name prompt contains no default value, enter [RETURN] or a backslash (\) if you don't want to specify a file.

To accept a default extension, type the file name without an extension. For example, when a prompt displays

[.text]

and you do not enter an extension, ".text" will be added to the file name you enter.

To prevent an extension from being added, enter the file name with a period at the end. The Workshop won't add an extension to a device, volume, or catalog name, so you don't have to follow these with a period.

To accept a default file name, respond with [RETURN]. If you do not want the default file or any other file, enter a backslash (\).

Alternate defaults are indicated by a slash (/). For example,

[-console]/[.text]

lets you default to either the console device or a text file. This option is made available in cases where you may want to display output on the screen or save it in a file. Press [RETURN] if you want the Workshop to use the console. Enter a file name if you want the Workshop to use a file. (If you don't supply an extension, ".text" is added.)

A separate default may be shown for each part of a pathname. For example,

[-paraport] [-intrinsic] [.lib]

shows a default value for the device, file name, and extension. If you leave out any part of the pathname, the Workshop supplies the default value for that part. Sometimes parts of a pathname are shown within the same set of brackets if the parts cannot be accepted independently of one another; for instance

[-paraport-intrinsic] [.lib]

1.5.2 Getting Help

If you need help or want to see a list of program options, respond to a file name prompt by typing *?* followed by [RETURN]. Help information appears on your screen if it is available. (Not all programs provide help screens.)

1.5.3 Getting Out

You may want to stop what you're doing--cancel a program that's running, cancel a command prompt, or temporarily stop a screen display. This section tells you how.

1.5.3.1 Canceling a Program

You can terminate the operation of most Workshop tools and utilities by pressing the *-period key combination*. Most Workshop tools check for -period even when running under exec files.

Unless a user program was written to recognize the -period key combination, pressing those keys will not terminate the program. (The function PAbortFlag tells a program whether or not -period has been pressed. For more information, see PASLIBCALL, Section 5.4.) If -period doesn't work, you can do one of the following:

- Wait for the program to terminate.
- Press the NMI (nonmaskable interrupt) key, which forces the system into the Debugger. The minus (-) key on the numeric keyboard is normally set to be the NMI key. See Section 8.2.1.2, Terminating an Infinite Loop, for further instructions.

1.5.3.2 Canceling a Prompt

The Clear key on the numeric keypad is an escape key. You can use it in response to a file name prompt. For example, if you're in the File Manager and you type *D* for Delete by mistake, press the Clear key to return to the File Manager command line. You don't have to press [RETURN] after pressing [CLEAR].

1.5.3.3 Halting a Screen Display

To stop the screen display while a program is running, press the -S key combination. The program temporarily halts. To restart the screen display, just press -S again. This feature works for all programs that do screen output through the Pascal run-time system.

1.5.4 Standard Error Messages

Every error reported by the Operating System or the Workshop has a number associated with it. If the file containing the text of the error message is available at the time of the error, the full message is displayed; if the error file is not available, only the error number is displayed.

The error files are:

OSErrs.ERR	Errors reported by the Operating System
PasErrs.ERR	Compile errors reported by the Pascal Compiler
WorkshopErrs.ERR	Errors reported by the Exec Processor

For a list of all error numbers and their associated message text, see Appendix A.

Chapter 2
The File Manager

See also the Release 3.0 Notes for this chapter.

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:

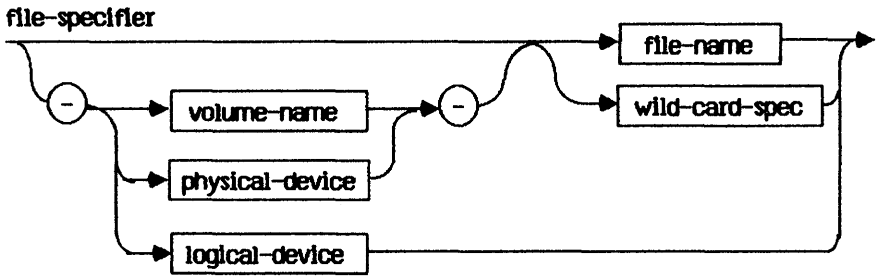
1. Formats the disk (if necessary).
2. Records its assigned volume name of up to 32 characters.

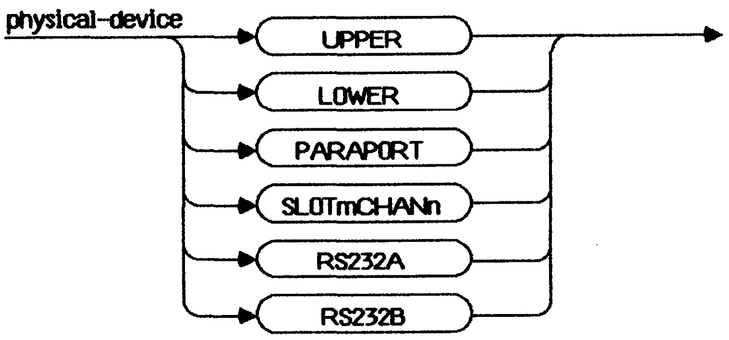
3. Creates its initial, empty directory (also called a *catalog*).
4. Mounts the initialized disk.

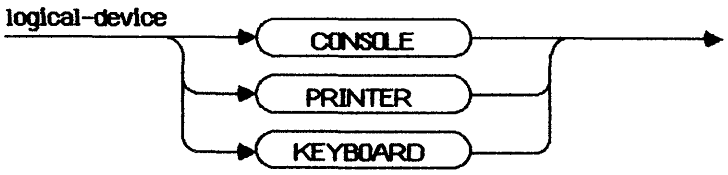
When an object is created on a disk, its file name of up to 32 characters is entered in the disk's directory. File names must be unique within a volume so that every object can be clearly identified.

2.4.2 File Specifiers

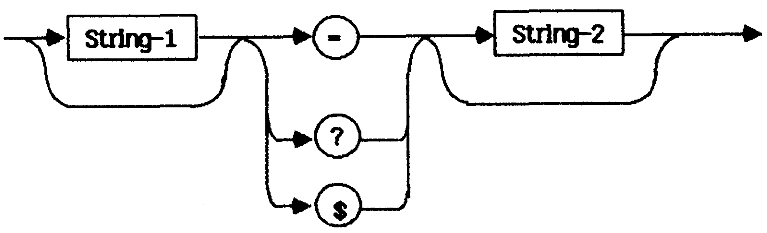
Within the Workshop, file specifiers are used to identify the volume, device, file, or set of files an operation applies to. The diagrams that follow show the makeup of a file specifier and its components.







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.

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.
- Inquire or set the hardware clock's time and date.
- Set the configuration of external devices such as disks and printers.
- Set the default startup device.
- 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 Command Line

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 display the status of all currently existing processes, and to remove processes. The process management 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).

NOTE

Console output frequently contains control characters and escape sequences for such things as positioning the cursor and clearing the screen; these special characters will be part of the redirected output as well. If the output has been redirected to a printer, the control characters may cause "special effects" such as overprinting. If the output has been redirected to a text file, the characters will be embedded within the text file (the Editor will show such characters as inverted question marks).

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 -S stops output and -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). To display the screen not currently displayed, hold down the right-hand OPTION key, and press ENTER on the numeric keypad. 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 wild-card 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}BUTTONS). 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 before it is performed.

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

The Preferences tool lets you specify what disks, printers, and other devices are connected to your Lisa, which should be the defaults, and how you want the special convenience settings adjusted. When you start Preferences (by pressing **P** in response to the System Manager command line), it displays a window with five checkboxes:

Set Conveniences is used to customize the Lisa's screen brightness, key delays, etc. See Section 3.3.1, Set Conveniences.

Select Defaults is used to specify your default printer and startup disk, and the length of the automatic memory test. See Section 3.3.2, Select Defaults.

Connect Device Software is used to connect your Lisa to peripheral devices. See Section 3.3.3, Peripheral Device Connections.

Install Device Software is used to install a device software driver. This is used in cases where your Lisa does not already have the software it needs for a particular peripheral device you wish to use. See Section 3.3.3, Peripheral Device Connections.

Remove Device Software is used to erase a software driver from your startup disk when you no longer want to use the peripheral device. See Section 3.3.3, Peripheral Device Connections.

After you have finished with Preferences, you can get back to the System Manager by choosing Quit from the File menu.

3.3.1 Set Conveniences

When you first check the Set Conveniences box of Preferences, each convenience setting already has one option marked. These prechosen options are known as default settings. Whenever you click the first item in the list, Set All Convenience Settings to Lisa Defaults, the default settings are chosen.

3.3.1.1 Screen Brightness and Contrast

Always adjust the screen brightness before setting the contrast. The brightness is adjusted through the brightness control knob while contrast is set through Preferences.

To set the screen brightness and contrast:

1. Find the brightness control knob (the higher of the two white knobs extending from the back of the cabinet).
2. Turn the brightness control down until your screen is entirely black.
3. Turn the knob back up just until the black rectangle turns to gray.
4. Slowly turn the knob back down, just until the rectangle is distinctly black, with no video scan lines visible, and there is a clean line on all borders.
5. Set the Normal Level (Contrast) by clicking different boxes under Normal Level until the screen is at a comfortable contrast level for you.

3.3.1.2 Screen Dim

In order to protect the screen from prolonged high-intensity illumination, the screen dims when not in use. If a period of time passes without the mouse being moved or any keys struck on the keyboard, the screen automatically dims to a lower level of illumination. As soon as the mouse is moved or a key struck it returns to the normal brightness. You can set the amount of time that passes before dimming with Minutes Until Screen Dims, and you can adjust the level it dims down to with Dim Level.

3.3.1.3 Speaker Volume

From time to time, the Lisa communicates by sounding various beeps and tones. The meanings of these signals are explained in the *Lisa Owner's Guide*. The Speaker Volume setting controls the loudness of these beeps and tones.

Each time you click one of the boxes under Speaker Volume, the Lisa sounds two tones, at the low and high extremes of the level you have chosen. Experiment with different settings until you find one you like.

3.3.1.4 Repeating Keys

Most of the Lisa keys repeat automatically when held down. The line of stars that appear when you click one of the Delay boxes demonstrates how long you have to hold a key down before it starts repeating. The line of stars that appear when you click one of the Rate boxes demonstrates how fast the characters will be repeated.

The correct settings depend on your typing speed and the way you use the Lisa. If you find that the Lisa often generates multiple letters when you intended to type only one, change the repeat delay to a setting nearer the long end of the scale. If you use the repeating keys often, you probably want to specify a short delay and a fast repeat speed.

3.3.1.5 Mouse Double-Click Delay

Some of the desktop functions are accomplished by double-clicking the mouse button (rapidly clicking the button twice). The Mouse Double Click Delay setting determines the time lag between release of the first click and the start of the second click that the Lisa interprets as one double- click.

Like the keyboard repeat delays, this setting should reflect your habits and work style. If the Lisa often treats your double clicks as two single clicks, try adjusting the delay to a longer setting. If the Lisa often interprets two single clicks as one double-click, try adjusting the delay to a shorter setting.

3.3.2 Select Defaults

When you check Select Defaults, Preferences lists all the printers and disks currently connected to the Lisa, and the length of the memory test it performs at startup time, indicating the current defaults.

To select your defaults:

1. **Default Printer:** If you have connected any printers to your Lisa with Connect Device Software (see Section 3.3.3, Peripheral Device Connections), the printer(s) you specified will be listed under Use This Printer as Default. Check the printer you normally wish to use. If you have connected more than one printer, the default printer will be the one identified as the logical device **-printer**, and used by the Editor. If you wish to use some other printer for a particular document, you can specify the physical device name of the printer (such as **-RS232A-p** or **-#10#1-p**). (You can also change your default printer with Preferences at any time.) Note that if you have the Office System on the same disk as the Workshop, the printer you specify will be the default printer in the Office System, too.

2. **Default Startup Disk:** When the Lisa is turned on, it looks to the startup disk for its initial instructions. Under Start Up From, check the disk you wish to use as a startup disk. For the Workshop this will be a hard disk. The startup disk should be the disk containing all of your Workshop software.

   Note: If you wish a new disk to be the startup disk, you will need to install the Workshop on the new disk. Once you have finished setting your defaults and device connections, you will also have to turn off the Lisa before the new disk becomes the startup disk.

3. **Default Memory Test:** Under Test Memory, check either Briefly or Thoroughly. The memory test setting determines how thoroughly the Lisa's memory is tested during the automatic startup test. If you check Briefly, the test takes about 20 seconds. If you check Thoroughly, the test takes about 40 seconds.

3.3.3 Peripheral Device Connections

A peripheral device can be an external hard disk, a printer, a graphics plotter, or any other mechanism connected to your Lisa. In order to use a peripheral device, your Workshop software must know where the device is connected, how to communicate with it, and how to operate it. In software terminology, the set of codes and instructions that tell your computer how to operate a device is called a *driver*. Your Workshop already includes software drivers for some of the most common devices, and if you wish to connect some other mechanism, you can install the proper software driver from the micro diskette supplied with that particular device.

To establish the necessary software connection between your Lisa and a peripheral device, the proper software driver must be linked to the hardware connection that the device is attached to. This is accomplished through use of the Preferences tool.

To connect device software:

1. Enter Preferences and click the Connect Device Software box. You will see a screen that lists all the possible connectors (or ports) at the back of your Lisa, and what, if anything, is attached to them.

   If you are using a Lisa 2/5, your screen will display an additional box labelled Parallel.

2. To link a peripheral device to your Workshop, click the box of the connector to which the device is physically attached. The screen will then display a listing of what devices your Lisa currently has software to link to from that connector.

3. If the peripheral device you wish to use is now listed, simply click the box next to its name. The device name will then appear opposite the appropriate connector and you are finished with the linking process. The device can be used immediately.

If the device you wish to use is not listed after you click the connector box in step 2, it means that the connector you have clicked cannot be used with that peripheral or your Lisa lacks the software driver needed to link up with the device. If lack of software is the problem, first install the necessary driver as described below under Install/Remove Device Software, and then repeat these steps 1 through 3.

Expansion Cards

When you attach an expansion card to one of your three expansion slots (see your *Lisa Owners Guide,* Attaching an Expansion Card), the card itself will contain one or more connectors which you can use to attach peripheral devices. Because more than one connector may be part of an expanion card, an additional level of Preferences becomes necessary to link a device to the Workshop through an expansion card.

To connect an expansion card:

1. Enter Preferences and click **Connect Device Software.**
2. When you click one of the Expansion Slot boxes, the screen will display a listing of the cards that your Lisa can currently link to. (If the card you wish to use is not listed, it means that your Lisa lacks the driver needed to link up with that card, and you will need to first install the necessary software as described below under Install/Remove Device Software and then return to step 1.)
3. Click the box of the card you wish to use, and a list of the connectors on that card will be displayed (Connector 1, Connector 2, etc.). Connectors on cards are numbered from the bottom up, thus, the bottom connector is always number 1.
4. Click the box of the connector that you are going to use, and a list of the devices that the Lisa can link to for that type of connector will be displayed. (If the device you wish to use is not listed, it means that your Lisa lacks the software needed to link up with that device, and you will need to first install the necessary driver as described below under Install/Remove Device Software and then return to step 1.)

   Note: In some cases there are different types of connectors on a single expansion card, or a certain connector can only accept a specific device. If a particular connector cannot accept a certain device, the device will not be listed when you click that connector's box even if you have installed the appropriate software driver on your Lisa.
5. Click the box of the device you wish to use, and it will be named opposite the connector number. This indicates that you are finished with the linking process. The device can be used immediately.

Disconnecting or Changing Device Software
If you wish to attach a different device to a connector, simply repeat the procedure described above and click the name of the new device. If you wish to disconnect a device simply click Nothing in the list of devices.

NOTE

Deferred Detachment: If you click Nothing or some new device driver for a connector that is linked to a disk storage device (such as a ProFile), or a connector linked to a peripheral that is currently in operation (a printer that is in the midst of printing a document, for example), you will get a message telling you that the device cannot be disconnected and asking if you wish to Defer Detachment. If you answer Yes, the new driver (or Nothing) will not go into effect until the Workshop is restarted (either by turning the Lisa off and then back on or by going to the Environments window and clicking Restart) and you will be unable to use that connector for anything else until then.

Install/Remove Device Software
If Preferences does not list the device you wish to use, you have to install the device software driver. This is done with the Install Device Software choice in Preferences and the micro diskette you received with the device.

To install device software:

1. Click the Preferences choice Install Device Software.
2. Insert the micro diskette in the disk drive. A list of the drivers contained on the disk will be displayed.
3. Click the box beside the name(s) of the device(s) you wish to install. The software driver will be copied from the micro diskette and installed on your Lisa.

   Note: If you install from a micro diskette a driver that is already on your Lisa (that is, a driver that already shows up under Connect Device Software), the version on the micro diskette will replace the version that had been on the Lisa.
4. When you have installed the device driver(s) you wish, click Connect Device Software to leave the Install Device Software menu. When you are finished with Preferences, the micro diskette can be ejected in the normal manner (-E).

To remove device software:

1. Click Remove Device Software. A list of all the software drivers currently installed on your Lisa will be displayed.
2. Click the device drivers that you want to erase.

Note: You cannot erase a driver for a device that is in use. You must cease using the device, and in some cases disconnect it, before you can erase its driver.

3.3.3.1 Linking to Expansion Cards

As with any other device, your software must be told about an expansion card connected to one of your Lisa's expansion slots. For a discussion of using Preferences to link peripheral devices to your Workshop, see Section 3.3.3, Peripheral Device Connections.

To hook up an expansion card:

1. With the power to your Lisa turned off, insert the expansion card in the appropriate slot at the rear of your Lisa as explained in Appendix 1 of your *Lisa Owner's Guide* and the documentation included with the card.
2. When the card has been properly attached, turn on your Lisa.
3. Enter Preferences.
4. Click the Connect Device Software box.
5. Click the box for the appropriate expansion slot number (Expansion Slot 1, for example).
6. If the type of expansion card you are installing is listed, click the appropriate box and go to step 8.
7. If the expansion card you wish to install is not listed, take the micro diskette that came with the expansion card and insert it in the micro drive. Click in the Preferences box to reactivate the Preferences window. Now click Install Device Software. The software driver(s) on the disk will be displayed.

   Click the box beside the name of the expansion card you wish to install.
8. If you are installing other expansion cards, repeat steps 1 through 6 or 7. If you are ready to connect a peripheral device to the expansion card, follow the appropriate instructions in the sections below.

3.3.3.2 Linking to Printers

For a discussion of using Preferences to link peripheral devices to your Workshop, see Section 3.3.3, Peripheral Device Connections.

To connect a printer:

1. Connect the printer to the appropriate connector at the back of your Lisa, as explained in the documentation that comes with the printer and *Lisa Owners's Guide*. If you intend to attach the printer to an expansion card, you will first have to attach the card and link it to the Workshop as explained in the instructions that came with the card and section 3.3.3.1 above.
2. Enter Preferences.
3. Click the Connect Device Software box.
4. Click the box for the connector you are using for your printer.
   - a. If you are connecting the printer to one of the built-in serial ports, click either Serial A or Serial B. (Serial A is the port next to the mouse port.)
   - b. If you wish to connect your printer to an expansion card, click the appropriate box. (If the expansion card is not named in the options list, you will have to install the card's software driver.)

     When you select an expansion card, the Lisa asks which of the two or three connectors on the card you wish to use. Connectors on cards are numbered from the bottom up, thus, the bottom connector is always number 1.
5. Once you have selected the connector you wish to use, the Lisa displays a list of devices that can be attached to that connector. Click the appropriate printer (Apple Imagewriter or Apple Daisywheel, for example).

   If the printer you wish to install is not listed, take the micro diskette that came with the printer and insert it in the micro drive. Now click the Install Device Software box. A series of printer choices appears; click the name of the printer you wish to use. The printer software driver on the diskette will be automatically installed, and the name of the printer listed under Connect Device Software for you to select.

   Note: Expansion cards usually have two or three connectors. In some cases these connectors are different, and each one can only accept specific devices. If, after installing the software driver as explained above, you do not see the peripheral you want listed in the menu, try one of the other connectors on the expansion card.
6. If you are installing additional devices or expansion cards, you can set Preferences for them before proceeding to the next step.
7. Load the paper into the printer as explained in the documentation that came with the printer.

8. Turn on the printer and run the printer's self-test to make sure the printer, independent of the Lisa, will run correctly. See the manual that came with the printer for instructions on running the self-test.

3.3.3.3 Linking to External Hard Disks

In addition to the 5-megabyte ProFile or 10-megabyte internal hard disk you are already using, you can attach other hard disks to your Lisa for additional storage.

For a general discussion of using Preferences to link peripheral devices to your Workshop, see Section 3.3.3, Peripheral Device Connections.

To connect an external hard disk:

1. After installing an expansion card at the back of your Lisa, connect the disk to the card's connector, as explained in the documentation that comes with the disk and your *Lisa Owners's Guide*.
2. Enter Preferences.
3. Click the Connect Device Software box.
4. Click the box for the expansion card you are using. If the expansion card is not listed, you will have to install the card software driver as explained above.

   When you select an expansion card, you will be asked to click which of the two or three connectors on the card you wish to use. Connectors on cards are numbered from the bottom up, thus the bottom connector is always number 1.
5. Once you have selected the connector you wish to use, you will see a list of devices that can be attached to that connector. Click the appropriate hard disk.

   If the hard disk you wish to install is not listed, take the micro diskette that came with the disk and insert it in the micro drive. Now click the Install Device Software box and then click the name of the hard disk you wish to use when it is displayed. The disk software driver on the diskette will be automatically installed, and the name of the hard disk listed under Connect Device Software for you to select.

   Note: If the hard disk comes with an expansion card, you may also have to install the driver for the card. Check the instructions that accompany the hard disk for the names of the drivers needed to operate the hard disk.
6. If you are installing additional devices or expansion cards, you can set Preferences for them before proceeding to the next step.

If you wish the new disk to be the startup disk, you will need to install the Workshop on the new disk.

3.3.3.4 Linking to Other Devices

Other devices may be connected to your Lisa, and selected with Preferences. For a general discussion of using Preferences to link peripheral devices to your Workshop, see Section 3.3.3, Peripheral Device Connections.

It is recommended that only devices approved for use with the Lisa, and supplied with Lisa software drivers, be purchased for connection to the Lisa. However, if you wish to use a peripheral that does not have a Lisa software driver, but is connected to the Lisa by means of a parallel or serial cable, you may be able to operate it by selecting the software driver for Parallel Cable or Serial Cable.

For example, by connecting a device's serial cable to Connector B, clicking Serial B under Connect Device Software, and then Serial Cable, you may be able to use the device. Depending on the particular peripheral device, this may or may not be adequate.

In technical terms, to use the Serial Cable driver the device must use an RS232C serial cable, no more than 9600 baud, and either asynchronous 8-bit or 7-bit with parity-checking communications. In case of doubt, consult your dealer.

In technical terms the Parallel Cable driver is for devices using the standard Centronics™ Parallel Interface Protocol.

3.4 Process Management

The process management subsystem is used to monitor and kill suspended and background processes. It is started by pressing **M** (for ManageProcess) in response to the System Manager command line. (See the *Operating System Reference Manual* for information on processes.)

The process manager displays the following command line:

ManageProcess: KillProcess, ProcessStatus, Quit

KillProcess (K)
The KillProcess command terminates a currently existing process, including a background process.

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

Pathname	The name of the process's object file.
Process_ID	The unique identifier assigned to the process.
State	The current state of the process: Active, Suspended, or Waiting.

Quit (Q)
The Quit command exits from the process management subsystem back to the System Manager command line.

Chapter 4
The Editor

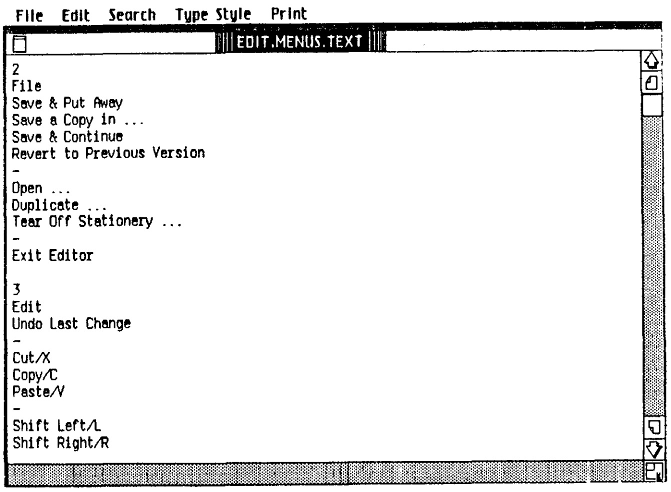
See also Release 3.0 Notes for this chapter.

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.

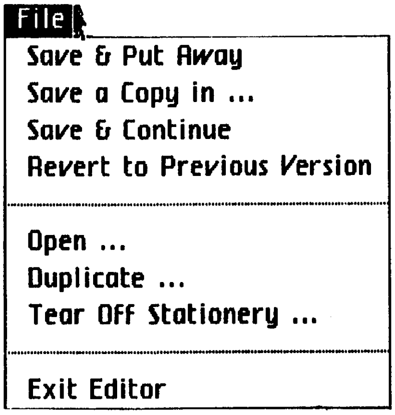


Figure 4-2
The File Menu

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.

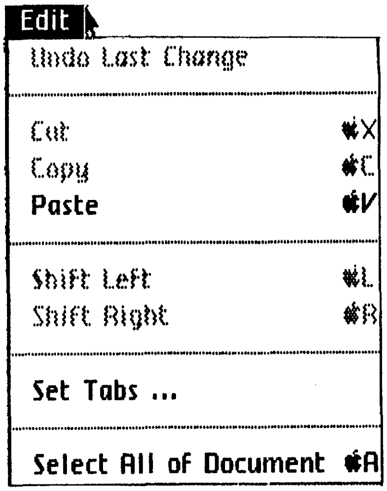


Figure 4-3
The Edit Menu

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.

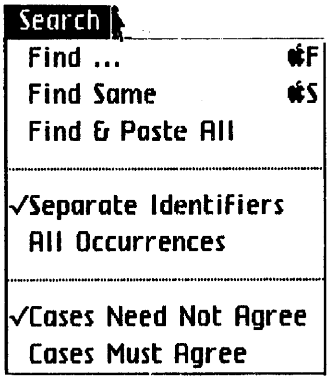


Figure 4-4
The Search Menu

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.

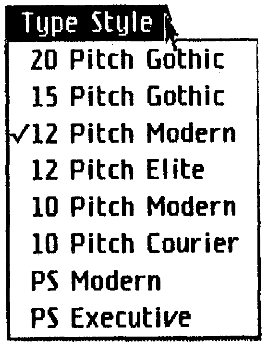


Figure 4-5
The Type Style Menu

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.

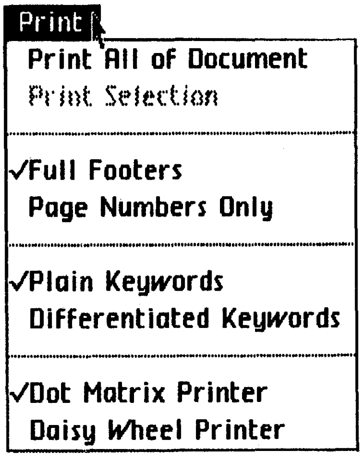


Figure 4-6
The Print Menu

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.

Chapter 5
The Pascal Compiler

See also the Release 3.0 Notes for this chapter.

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  ernum,refnum:integer;
                            size,delta:longint
                            ldsn:integer;
                            swapable:boolean);
```

where:

ernum	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.

Chapter 6
The Assembler

See also the Release 3.0 Notes for this chapter.

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  DO                ;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.

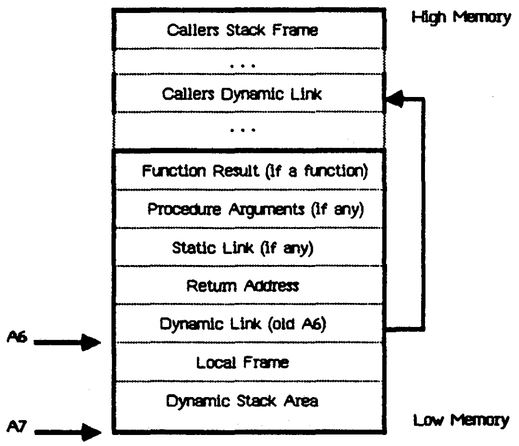


Figure 6-5
The Pascal Run-Time Stack

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
```

Chapter 7
The Linker

See also the Release 3.0 Notes for this chapter.

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 -   262400
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
        30 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.

Chapter 8
The Debugger

See also the Release 3.0 Notes for this Chapter.

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# <mmmm> (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
<mmmm>	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 DO=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.

Chapter 9
Exec Files

Exec Files

9.1 Introduction to Exec Files

Sitting at your computer and typing Workshop commands is like driving a car yourself. Using exec files is like teaching a chauffeur the route, then saying, "Take me there again" and sitting back while the chauffeur drives. With exec files you can execute Workshop commands automatically, without retyping them each time.

An exec file is actually a program. You can pass parameters to the exec file, and you can execute its statements conditionally. Its programming language consists of the *exec commands* described in this chapter plus *Workshop commands* you already know.

For instance, you can create a test procedure called **TESTEXEC** that runs a set of application programs. Then each time you modify a program you can rerun the entire test simply by typing the Workshop Run command

```
R<TESTEXEC
```

Here's what **TESTEXEC** looks like:

```
$EXEC
      Rsales
      Rexpenses
      Rgenledger
$ENDEXEC
```

The first and last lines of **TESTEXEC** are exec commands. The other lines each contain a Workshop Run command.

Like other programs, an exec program doesn't run directly from its source statements--it has to be processed first. You use the Editor in the Workshop to create an *exec source file*. Then at *process time* you invoke the Exec Processor to create an *exec run file* which is run by the Workshop at *run time*, as shown in Figure 9-1.

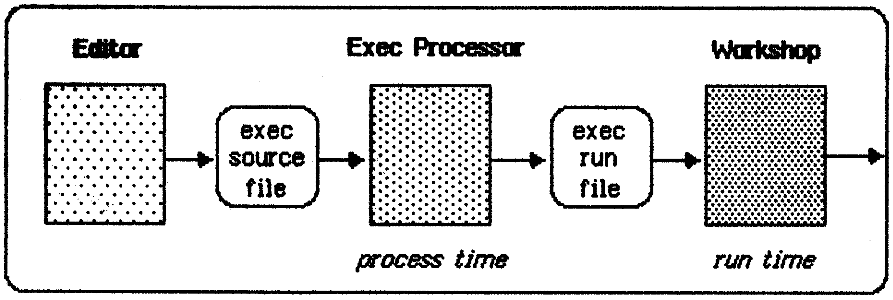


Figure 9-1. Overview of Exec Files

9.1.1 The Exec Processor

The Exec Processor operates under the Workshop Run command. When you type

R<pathname

or

REXEC/pathname

in the Workshop command line, the < or **EXEC/** (upper or lower case) tells the Exec Processor to process an input file. The input file is usually an exec source file but it may be a previously created exec run file.

The exec run file--the output of the Exec Processor--contains only Workshop commands. The Exec Processor looks at variables in the exec source file and determines their process-time value; then, based on conditional execution of exec commands, it determines which Workshop lines to place in the exec run file. The Exec Processor's final step is to give the exec run file to the Workshop, which runs it.

An exec source file normally has a file name with a ".text" extension. An exec run file always has the same file name with a "..text" extension:

Exec Source File		Exec Run File
myexec.text	==>	**myexec..text**
myexec	==>	**myexec..text**
my.exec.text	==>	**my.exec..text**
my.exec	==>	**my.exec..text**

The normal Exec Processor function is "process-and-run," but a number of commands and options are provided for greater flexibility. For instance, the **DOIT** command tells the Exec Processor to give the current contents of the exec run file to the Workshop to run immediately; the Workshop then returns control to the Exec Processor so it can continue processing the the source file. The **Keep** and **Rerun** Processor options allow you to save the exec run file and run it again without reprocessing. (For more information on exec run files, see Section 9.3, Running an Exec Program.)

NOTE

To terminate processing when the Exec Processor is running, press -period.

9.1.2 Distinguishing between Exec Lines and Workshop Lines

There are two kinds of exec source lines: exec lines and Workshop lines.

Exec lines contain exec commands, written in a language similar to Pascal; these commands are described in Section 9.2, Writing an Exec Program. Exec commands allow you to change variable values, skip over Workshop lines under exec control, perform I/O, and control the Exec Processor.

Each exec command must begin on a new text line; it can occupy more than one text line. The Exec Processor looks for a continuation line only if the command is syntactically incomplete. In the following example, line 5 completes a valid command, so line 6 is flagged as an error:

```
1.   repeat
        ...
5.   until reply = 'YES'
6.   or count > 5
```

To notify the Exec Processor that the command continues, rewrite line 5 so that it is syntactically incomplete:

```
5.   until reply = 'YES' or
6.   count > 5
```

Workshop lines contain either responses to the Workshop command line--such as File-Mgr and Linker commands--or input to any programs you run under the Workshop. Workshop lines should be typed in the exec source file just the way you would enter them from the keyboard, following the dollar-sign convention you have chosen (see below).

9.1.2.1 The Dollar-Sign Convention

The dollar-sign ($) convention allows the Exec Processor to distinguish between exec lines and Workshop lines.

If the first line of your exec program, the **EXEC** command, begins with a dollar sign, the Exec Processor considers every line that begins with a dollar sign to be an exec line, except if the line is the continuation of a comment; other lines are considered Workshop lines.

If the **EXEC** command does *not* begin with a dollar sign, every line *without* an initial dollar sign is considered an exec line, and you must precede every Workshop line with a dollar sign.

NOTE

The command formats in this chapter are shown without an initial dollar sign, but many of the examples use the dollar sign on exec lines. If you write an exec that calls another exec, the two exec files need not use the same dollar-sign convention.

9.1.3 Introduction to Variables and Parameters

A *variable* is a string whose contents can change during execution of an exec program. Variables allow you to generalize an exec program so that you can use the same exec in a variety of situations.

Variables in exec language are *local* to the exec that declares them. If you give a variable the same name in two different exec files, you will still have two separate local variables.

A *parameter* is a variable to which you expect to give an initial text value from outside the exec program. You can pass a list of parameter values to an exec when you invoke it either from the main command line or from another exec.

Variables and parameters are identified either by a name or by a number. They are written in one of the following ways:

%n	where **n** is a variable number (0-9):	**%3**
x	where **x** is a variable name in an exec line:	**var3**
[x]	where [x] is a variable name in a Workshop line or an exec invocation:	**[param3]**

9.1.3.1 Variable Names and Numbers

You can declare up to twenty named variables or parameters. The first ten of them can be referred to by number as well. Numbered variables have the advantage of not having to be declared; named variables provide more meaningful documentation. Values are assigned to variables in the same way whether you use names or numbers. You can use both numbers and names in a given exec program, and you can refer to the first ten variables either by name--if you declared them--or number.

When you use variable numbers, these rules apply:

- The numbered variables are shown as a percent sign followed by a number from 0 to 9: **%0** through **%9**.

- You can supply initial values for numbered variables in an invocation parameter list (see Section 9.3.1). The value for **%0** must be first in the list, the value for **%1** next, and so on in numeric order.

When you use variable names, these rules apply:

- Variable names must be declared in the EXEC command's variable declaration list. The list is enclosed in parentheses and contains variable names separated by commas; for instance, **(payday,paytype,profitshare,bank)**.
- You can supply initial values for named variables in an invocation parameter list. The values must be listed in the order in which the corresponding variable names were declared; for instance, **(05/04/85,hourly,.0276,FirstState)** supplies values for the variable declaration list above.
- A variable name must be alphanumeric and must begin with an alphabetic character; the name can be as long as you like, but only the first eight characters are significant.
- A variable name in a Workshop line, an expanded string constant, or an exec invocation must be enclosed in square brackets (**[ ]**) to distinguish it from ordinary text.
- You can refer to the first ten named variables either by name or by number. If you declare five variable names, they correspond to variables **%0** through **%4**.

The two examples shown below function identically. The first example uses numbered variables:

```
exec       {NUMBERED VARIABLES
            %0=source file name, %1=counter}
...
if exists("%0.text") then
   set %1 to eval(%1-1)
   $F{iler}D{elete}%0.text
   $Q{uit the Filer}
endif
...
endexec
```

The second example uses variable names (note the variable declaration list following the **EXEC** command):

```
exec (oldsource,counter) {NAMED VARIABLES}
...
if exists("[oldsource].text") then
   set counter to eval(counter-1)
   $F{iler}D{elete}[oldsource].text
   $Q{uit the Filer}
endif
...
endexec
```

9.1.3.2 Setting Variable Values

You can *alter the value* of a variable by using the **SET**, **DEFAULT**, and **REQUEST** exec commands described in Section 9.2.1.

The *initial values* of variables at process time are supplied in an *invocation parameter list*--a list in parentheses following the exec name in a Workshop Run command, a **SUBMIT** command, or a user function. Values in the list consist of text separated by commas. If a parameter value is not provided for a given variable, its inital value is the null string.

Whether you use numbered or named variables, the invocation parameter list is the same. For instance, the following Run command will work with either example in Section 9.1.3.1 above. The value of the **%0** or **oldsource** variable is "-backup-oct6"; the value of the **%1** or **counter** variable is "4".

```
R<files(-backup-oct6,4)
```

You can supply initial values for some, none, or all of the variables your exec program uses. The Workshop Run command below contains initial values for parameters **%0** and **%2**--or their named equivalents. A value for **%1** is not supplied.

```
R<exsales(-lower-customers,,Accounts Receivable)
```

To demonstrate the use of variables to make an exec more versatile, let's generalize the **makeoneprog** exec, shown in the next example.

```
$EXEC { "makeoneprog" -- This exec file compiles and
         links a specific Pascal program named ONEPROG. }
   P{ascal compile}ONEPROG
     { no listing file }
     { default object file }
   L{ink}ONEPROG
     IOSPASLIB
     { end of linker input }
     { no list file }
     ONEPROG{ output file name }
$ENDEXEC
```

If you want to compile and link a Pascal program named **OTHERPROG**, you can't use the **makeoneprog** exec file. To compile and link any Pascal program, change the name of the Pascal program in the exec from **ONEPROG** to **ANYPROG** and declare it as a variable. We'll call the new exec **makeanyprog**.

```
$EXEC (ANYPROG) { "makeanyprog" -- This exec file
                  compiles and links any Pascal
                  program.}
   P{ascal compile}[ANYPROG]{program-name variable}
     { no listing file }
     { default object file }
   L{ink}[ANYPROG]
     IOSPASLIB
     { end of linker input }
     { no list file }
     [ANYPROG]{ output file name }
$ENDEXEC
```

You can run **makeanyprog** to compile and link the **ONEPROG** program. The initial value "ONEPROG" replaces every occurrence of the variable **ANYPROG** when you use the following Workshop Run command:

```
R<makeanyprog(ONEPROG)
```

To compile and link the **OTHERPROG** program, you can run **makeanyprog** again, simply changing the Run command:

```
R<makeanyprog(OTHERPROG)
```

9.1.4 Syntax of Exec Lines and Workshop Lines

This section contains rules for writing exec lines and Workshop lines. You can use it first as a general introduction and later as a reference tool.

The Exec Processor places a Workshop line in the exec run file after performing the following processing:

- Removing the inital dollar sign, if any.
- Processing tildes.
- Substituting the current values of variables.
- Removing comments.
- Eliminating leading and trailing spaces (unless the Blanks process-time option was specified).

Special characters are used as delimiters and as signals to evoke special processing by the Exec Processor; they include

$	**Exec/Workshop distinguishing character**
%	**Numbered variable character**
[ ]	**Variable name delimiters (W)**
{ }	**Comment delimiters**
' '	**Simple string constant delimiters (E)**
" "	**Expanded string constant delimiters (E)**
~	**Tilde literalizing character (W)**
<	**Exec invocation character**

(W) means this character has special significance only in Workshop lines. (E) means this character has special significance only in exec lines.

Comments can be included in exec or Workshop lines. A line that consists of nothing but comments is considered an exec line or a Workshop line depending on the dollar-sign convention. A comment can extend over more than one line, as in the exec program examples in Section 9.1.3.2. Any information in braces ({ }) is considered a comment and is ignored by the Exec Processor. For example, the Workshop line

```
A%0
```

can be documented with comments; in the example shown below, the first line is a Workshop line and the second line is an exec line:

```
A{ssemble}%0{source file}
${Use a separate line if more comments are necessary.}
```

It's good practice to write all separate comment lines as exec lines because a Workshop line with nothing but comments causes a [RETURN] to be placed in the exec run file.

Upper and lower case in Workshop lines is passed intact to the exec run file. In exec lines, case is significant only in string constants; that is, **var1** is the same variable name as **VAR1**, but **'YES'** is not equal to **'yes'**.

Spaces are delimiters in exec lines (extra spaces are ignored). In Workshop lines, leading and trailing spaces are *removed* before the line goes into the exec run file unless you specify the Blanks option at process time; whether spaces are significant within Workshop lines depends upon the program you are running.

The dollar sign **($)** is used to distinguish between Workshop lines and exec lines. See Section 9.1.2.1, The Dollar-Sign Convention, for more information.

The tilde **(~)** is used as a *literalizing character* (Workshop lines only). The special character that follows the tilde is not interpreted by the Exec Processor but is placed in the exec run file as is; for example,

```
~$40,723.78    ~{Cost of Sales}    35.5~%
```

In this example the dollar sign is *not* interpreted according to the dollar-sign convention. The information within braces, which would normally be discarded as a comment, *is* placed in the exec run file. (Because a tilde cancels the effect of the left brace as a comment delimiter, the right brace has no meaning and does not require a tilde.) The **%** is *not* interpreted as the first character of a numbered variable.

To represent the tilde itself in a Workshop line, use two tildes in a row.

The exec invocation character (<) should be followed by the pathname of an exec file. This character is used to call a user exec function; in the following example, **devname** is an exec file that returns a function result:

```
$set checkvol to <devname([checkvol],-myvol)
```

The invocation character can also be used in a Workshop Run command to cause chaining to another exec program (see Section 9.4.1, Exec File Chaining):

```
R<linkif
```

Simple string constants (exec lines only) consist of text surrounded by *single quotation marks*: **'text'**.

NOTE

The maximum length of any string is 255 characters.

Expanded string constants (exec lines only) consist of text and/or variables surrounded by *double quotation marks*: **"Text [var] text"** or **"Text %0 text"**. The Exec Processor places the current value of each variable in the string before executing the exec command.

String functions (exec lines only) are built-in functions or user-defined functions that return a string value. See Section 9.2.4; compare *Boolean functions*.

String expressions (exec lines only) contain one or more of the following: simple string constants, expanded string constants, string functions, and variables.

Boolean constants (exec lines only) are **true** and **false**.

Boolean functions (exec lines only) are built-in functions that return a Boolean result **true** or **false**. You cannot write an exec that returns a Boolean value.

Boolean expressions (exec lines only) are expressions that return a Boolean result **true** or **false**. They can contain the following: Boolean constants, Boolean functions, comparisons of string expressions or numeric expressions, and combinations of the preceeding with logical operators. See Section 9.2.3.1, String and Numeric Comparisons in Boolean Expressions, for more information.

Numeric constants (exec lines only) are integers, not enclosed in quotes; for example: **0**, **-255**, **1984**. Numeric constants are permitted in numeric expressions, where they are treated as numbers, and in string expressions, where they are treated as strings.

Numeric expressions (exec lines only) are resolved arithmetically, not as strings; they return a numeric result that can be used only where specified in the syntax for each exec command. To produce a string containing the result of a numeric expression, make the numeric expression the argument of the **EVAL** function. Numeric expressions consist of numeric constants, string functions that yield a numeric value, variables that contain a numeric value, and numeric operators (see Section 9.2.4.5, String Arithmetic Using the EVAL Function).

9.2 Writing an Exec Program

This section describes all the available exec commands. These commands are executed by the Exec Processor, before the exec run file is run.

9.2.1 Declaring and Setting Variables

The commands described in this section tell you how to declare named variables and how to change the value of a variable.

9.2.1.1 The EXEC and ENDEXEC Commands

Every exec program must begin with the **EXEC** command and end with the **ENDEXEC** command. The **EXEC** command is where you identify the names of any named variables you use in the exec.

The first line of an exec program has the format

```
EXEC (variable-declaration-list)
```

where **(variable-declaration-list)** is a list in parentheses containing variable names separated by commas. It is required only if you use variable names for parameters or internal variables (see Section 9.1.3.1, Variable Names and Numbers). For example,

```
$exec (leapyear,debits)
```

declares variables named **leapyear** and **debits**.

The last line of an exec program has the format

```
ENDEXEC
```

The last line in the exec program doesn't have to be the last line in the file. It's possible to imbed an exec program in a Pascal program or another programming language source file by using the Imbed Processor option; see Section 9.3.2, Processor Options, for more information.

9.2.1.2 The SET and DEFAULT Commands

The **SET** and **DEFAULT** commands let you assign a value to a variable within the exec program.

The **SET** command replaces the current value of a variable with a new value; it overrides an initial value specified in the invocation parameter list. The format of the **SET** command is

```
SET variable TO string-expression
```

The format of string expressions is described in Section 9.1.4, Syntax of Exec Lines and Workshop Lines. Examples of the **SET** command follow:

```
$set %0 to '-backup-oct6'
$set counter to '4'
```

The **DEFAULT** command is executed only if the specified variable has the null string as its value. **DEFAULT** does not override an initial value supplied in the invocation parameter list. The format of the **DEFAULT** command is

```
DEFAULT variable TO string-expression
```

If **execA** contains these commands

```
$exec (vol,month,day)
$default vol to "-paraport"
$set month to "July"
$default day to "17"
```

and is run with this invocation parameter list

```
R<execA(,September,23)
```

then--after the commands are executed--the results are as follows:

```
The value of month becomes "July"
The value of vol becomes "-paraport"
The value of day becomes "23"
```

9.2.1.3 The REQUEST Command

The **REQUEST** command prompts the user for keyboard input. Like the **SET** command, the **REQUEST** command replaces the current value of the variable.

REQUEST causes the Exec Processor to wait until [RETURN] is typed. The format of this command is

```
REQUEST variable WITH string-expression
```

For instance,

```
$request %0 with "month?"
```

String-expression is displayed on the console as a prompt. **Variable** is set to whatever value the user types in response to the prompt.

9.2.2 Input and Output

In addition to using the invocation parameter list and the **REQUEST** command, you can provide input to an exec program and create output from it through the commands discussed in this section. You can

- Read a character or a line from the keyboard or a text file (**READCH** and **READLN** commands).
- Write to the screen or a text file (**WRITE** and **WRITELN** commands).
- Open and close a text file (**RESET**, **REWRITE**, and **CLOSE** commands).
- Obtain filenames from a directory (**RESETCAT** command and **NEXTFILE** function).
- Check for successful completion of I/O (**IORESULT** function). *All of the above commands set the IORESULT function.*

9.2.2.1 The RESET, REWRITE, and CLOSE Commands

Use these exec commands when reading or writing text files. **RESET** opens a file for input; **REWRITE** opens a file for output; **CLOSE** closes an open file. These commands set the **IORESULT** function, which is described in Section 9.2.2.5.

The format of the commands is

```
RESET file-id, filename
REWRITE file-id, filename
CLOSE file-id
```

File-id is associated with a file when the file is opened; it identifies the file for subsequent read, write, and close commands. It is a global file identifier that is allocated when the file is opened. It is deallocated either when the file is closed or when the exec program finishes running. It is not a string variable and can be used only where a file identifier is expected. It does not have to be declared in the EXEC command. Its name follows the rules for variable names. Its identifier can be any number of alphanumeric characters, but only the first eight characters are significant; the first character must be alphabetic.

Filename is any string expression that yields a valid pathname; it must refer to a text file.

Here are some examples of exec commands that open and close files:

```
$reset fileone, "execdata.text"
$rewrite errmsg, "-pay-mfg-address.update.errs.text"
$close errmsg
```

9.2.2.2 The READCH and READLN Commands

With the **READCH** and **READLN** commands you can read data from the keyboard or from a textfile and assign it to a variable. **READCH** reads one character. **READLN** reads one line--up to and including the next [RETURN]. These commands set the **IORESULT** function. The format of the commands is

```
READCH (file-id) variable
READLN (file-id) variable
```

(File-id) associates the read command with the pathname specified in the **RESET** command. If **(file-id)** is not specified, the **READCH** or **READLN** command reads from the keyboard. This causes your exec to pause until a value for **variable** is typed; for **READLN**, the value must be followed by [RETURN]. When reading from the keyboard, it's a good idea to prompt the user using **WRITE** or **WRITELN** to indicate what information the exec program is waiting for.

Variable identifies the variable that will hold the information to be read. If end-of-file is encountered while reading, **variable** is set to 'EOF'. In the first example below, a character is read from a file into a numbered variable, **%3**; in the second example, a line is read from the keyboard into a named variable, **title**.

```
$readch (fileone) %3 {read one character from a file}
$readln title {halt/read a title from the keyboard}
```

9.2.2.3 The WRITE and WRITELN Commands

With the **WRITE** and **WRITELN** commands you can write data to the screen or to a textfile. **WRITELN** ends its output with [RETURN] and **WRITE** does not; otherwise the commands are identical. These commands set the **IORESULT** function. The format of the commands is

```
WRITE   (file-id) string1, string2, ... stringN
WRITELN (file-id) string1, string2, ... stringN
```

(File-id) is required only if you are writing to a textfile; it associates the write command with the pathname specified in the **REWRITE** command for the same file.

String1 through **stringN** represent any number of string expressions separated by commas. The strings are written consecutively--at the current cursor location in the case of screen output, or at the current location in the file in the case of textfile output. Here are some examples:

```
$write 'Ready to stop? Type Y or N ... '
$writeln (percentage) "value of %4 is %7~%"
$writeln 'Finished writing to file ',outfile
```

9.2.2.4 The RESETCAT Command and NEXTFILE Function

The **RESETCAT** command opens an OS directory; **NEXTFILE** is a string function that returns the name of a file in the open directory. These commands set the **IORESULT** function. The format of the **RESETCAT** command is

```
RESETCAT directoryname
```

Directoryname is a string expression that specifies the pathname of a volume, catalog, or file; the wildcard character = may be used in the filename part only. For instance,

```
$resetcat "-[vol]"
$resetcat '=.obj'
```

If **directoryname** includes a filename part but no wildcard, the filename part is used as a prefix. In other words, RESETCAT **"-[vol]-[cat]-[file]"** is equivalent to RESETCAT **"-[vol]-[cat]-[file]="**.

When RESETCAT is executed, the value of **NEXTFILE** is set to the first pathname in the directory that meets the criterion specified in **directoryname**. (In searching a directory, **NEXTFILE** returns catalog names as well as filenames.) In the examples shown above, after the first execution of RESETCAT the value of **NEXTFILE** is as described below:

Directoryname	Value of NEXTFILE
"-[vol]"	-first file in [vol] directory
'=.obj'	-first file with .obj suffix on the default (prefix) volume

When **NEXTFILE** is called again, it contains the name of the next file (or catalog) in the directory that meets the **directoryname** criterion. When no such file exists, or if the directory is empty, **NEXTFILE** returns an empty string. Here's an example of an exec routine that checks for a blank volume and lists filenames:

```
exec (vol,ior,savefile,count)
resetcat "-[vol]"
if ioresult <> '' then
   set ior to ioresult
   writeln 'Bad volume'
   writeln ior
else
   set savefile to nextfile
   if savefile = '' then
      writeln vol,' has an empty directory.'
   else
      set count to 1
      while savefile <> '' do
         writeln "File [count] on volume [vol] is ",
                savefile
         set count to eval(count+1)
         set savefile to nextfile
      endwhile
   endif
endif
```

9.2.2.5 The IORESULT Function

IORESULT is a string function that tells you if an error occurred during a previous **RESET**, **REWRITE**, **READCH**, **READLN**, **WRITE**, **WRITELN**, **RESETCAT**, or **NEXTFILE** operation. If the I/O operation was successful, the value of the **IORESULT** function is an empty string. If an error occurred, **IORESULT** contains an Operating System error message in the form

```
Error <number>: <message>
```

You can display the error message as follows:

```
$reset infile
$set errmsg to ioresult
$if errmsg <> '' then
   $writeln errmsg
$endif
```

This example demonstrates the need for an intermediate variable to save the contents of **IORESULT** before displaying it because the **WRITELN** command also sets the **IORESULT** variable.

9.2.2.6 The Program Communication Buffer

Programs that run under the Workshop can communicate with each other by writing and reading in a 1K-byte communication buffer made available by the ProgComm unit. (See The ProgComm Unit in the third binder of this set.) You can open and close the communication buffer and write to or read from it from an exec program by using the exec I/O commands (**REWRITE**, **RESET**, **CLOSE**, **READCH**, **READLN**, **WRITE**, and **WRITELN**) with a special keyword file identifier, **COMMBUFR**.

Some of the I/O commands require an *access key* that limits access to the buffer. **Access-key** is a string expression. Since several applications can share the buffer, programs within each application must agree upon a value for **access-key**. The format of the I/O commands for use with the program communication buffer is

```
RESET COMMBUFR, access-key
REWRITE COMMBUFR, access-key
CLOSE COMMBUFR, access-key
READCH (COMMBUFR) variable
READLN (COMMBUFR) variable
WRITE (COMMBUFR) string1, string2, ... stringN
WRITELN (COMMBUFR) string1, string2, ... stringN
```

These formats correspond to the formats described earlier in Section 9.2.2 except for the **CLOSE** command, which requires an access key when used with the communication buffer.

NOTE

Do not close **COMMBUFR** after a write command. The communication buffer should be closed after reading, in order to empty it. **CLOSE** flushes the buffer for the specified access key; **REWRITE** flushes the buffer unconditionally.

The following exec program demonstrates communication buffer I/O:

```
exec (key, line, ior, n, ch)
repeat {do one cycle of writing, then reading}
   clear screen
   request key with 'Open CommBufr for write...key ? '
   rewrite commbufr, key
   request line with 'Write what to buffer ? '
   while line <> '' do {terminate input with empty line}
      writeln (commbufr) line
      request line with 'Write what ? '
   endwhile
   writeln
   repeat {try opening until we succeed}
      request key with 'Open CommBufr for read...key ? '
      reset commbufr, key
      set ior to ioresult
      if ior <> '' then
         writeln 'CommBufr open failed: ', ior
      endif
      until ior = ''
      set n to '1'
      repeat {write out CommBufr lines}
         readln (commbufr) line
         if line <> 'EOF' then
            writeln 'CB(', n, '): ', line
            set n to eval(n + 1)
         endif
      until line = 'EOF'
      writeln
      write 'Do you want to try another test ? (Y or [N]) '
      readch ch
   until uppercase(ch) <> 'Y'
   halt 'Done'
   endexec
```

9.2.3 Conditional Statements

Like other programming languages, exec language allows you to execute commands under some circumstances but not others. The **IF**, **WHILE**, and **REPEAT** statements described in this section are similar to their Pascal counterparts, but the conditions they test are examined *at process time*, not run time.

The example that follows below and on the next page demonstrates the use of **IF**, **WHILE**, and **REPEAT** statements to prompt for a series of directories and list their contents:

```
EXEC (cat,ioerr,file)
REWRITE text,'catlist.text'
IF IORESULT = '' THEN {successful list file open}
   REPEAT
      REQUEST cat WITH 'Search what directory?  '
      IF cat = '' OR LOWERCASE(cat) = 'quit' THEN
         CLOSE text
         HALT 'Done'
      ENDIF
      RESETCAT cat
      IF IORESULT = '' THEN  {successful catalog open}
         SET file to NEXTFILE
         WHILE file <> '' DO
            WRITELN (text) file
            SET file to NEXTFILE
         ENDWHILE
      ELSE
         SET ioerr TO IORESULT
         WRITELN 'Could not open ',cat
         WRITELN 'OS error: ',ioerr
      ENDIF
      UNTIL FALSE  {endless loop}
   ELSE
      SET ioerr TO IORESULT
      WRITELN 'Could not open output file'
   ENDIF
   ENDEXEC
```

9.2.3.1 String and Numeric Comparisons in Boolean Expressions

The condition tested by a conditional statement is in the form of a *boolean expression*--an expression whose value is either **true** or **false**. The constants **true** and **false** may also be used in boolean expressions. In the boolean expression

```
uppercase(answer) = 'NO'
```

uppercase(answer) is a string function with its argument, = is a string comparison operator, and **'NO'** is a string constant; the value of the expression is **true** if the value of **answer** is any one of the following: NO, No, nO, no.

Use the *string comparison operators* in a boolean expression to compare string expressions:

=	**{equal}**
<>	**{not equal}**
>	**{greater than}**
>=	**{greater than or equal}**
<	**{less than}**
<=	**{less than or equal}**

Use the *numeric comparison operators* in a boolean expression to compare string expressions that yield a numeric result:

EQ	**{equal}**
NE	**{not equal}**
GT	**{greater than}**
GE	**{greater than or equal}**
LT	**{less than}**
LE	**{less than or equal}**

String comparisons proceed character by character; numeric comparisons cause two numeric values to be compared. The results may be the same either way: **COUNT = 1** (string comparison) is equivalent to **COUNT EQ 1** (numeric comparison). Usually, however, the results are not the same. For example, the string comparison **1006 < 509** is true (because '1' is less than '5'), while the corresponding numeric comparison **1006 LT 509** is false.

You can use the following *logical operators* in a boolean expression:

```
AND     {expression is true if both terms are true... A AND B }
OR      {expression is true if either term is true.... A OR B  }
NOT     {expression is true if the term is false...... NOT (A)  }
```

The expression following **NOT** must be enclosed in parentheses. The default sequence of evaluation of a boolean expression is left to right. You can also use parentheses to control the sequence according to the rules of algebra. For instance,

```
not (A) or B {true if A is false or B is true }
not (A or B) {true if A and B are both false  }
```

9.2.3.2 The IF Statement

The **IF** statement lets you choose an action depending on conditions evaluated at process time; it consists of the **IF**, **ELSEIF**, **ELSE**, and **ENDIF** commands. Each command must begin on a new line and may occupy more than one line. **ENDIF** always ends an **IF** statement. **ELSEIF** and **ELSE** are optional. More than one **ELSEIF** may be present in an **IF** statement. *Nesting* is permitted; that is, any number of **IF** statements can be contained within an **IF** statement.

The format of the **IF** statement is shown below.

```
IF boolean-expression THEN
   Workshop and exec commands
ELSEIF boolean-expression THEN
   Workshop and exec commands
ELSEIF ...
ELSE
   Workshop and exec commands
ENDIF
```

The **IF** statement is evaluated in the order it appears in the exec source file. When the first true boolean expression in an **IF** or **ELSEIF** command is encountered, its corresponding **THEN** clause is *selected* --that is, its Workshop commands are processed and placed in the exec run file, and its exec commands are executed. If no **true** condition is encountered, the **ELSE** Workshop and exec lines, if present, are selected. Exec lines that are not selected are examined for correct syntax. Here is an example of an **IF** statement that submits a different exec file depending on the day of the week:

```
exec (date,ledger,payroll,payable,bankbal,personnel)
if date = 'FRIDAY' then
   submit endweek([ledger],[payroll])
   writeln 'Have a good weekend!'
elseif date = 'MONDAY' then
   submit startwk([payable],[payroll])
else {tuesday, wednesday, thursday}
   submit midweek([bankbal],[personnel])
endif
endexec
```

Here are the Workshop Run commands needed to run this exec file on three different days of the week:

```
R<weekday(FRIDAY,-ledger.march,-payroll.hourly)
R<weekday(MONDAY,,-payroll.exempt,-payable.march)
R<weekday(MIDWEEK,,,,-bankbal.march,-personnel)
```

9.2.3.3 The WHILE and REPEAT Statements

The **WHILE** statement lets you repeat an action *while a condition remains true*; the condition is tested *before* the action is performed. The **REPEAT** statement lets you repeat an action *until a condition becomes false*; the condition is tested *after* the action is performed. The condition is in the form of a boolean expression. Each command in a **REPEAT** or **WHILE** statement must begin on a new line and may occupy more than one line.

The **WHILE** statement consists of the **WHILE** and **ENDWHILE** commands. **ENDWHILE** always ends a **WHILE** statement. The format of the **WHILE** statement is

```
WHILE boolean-expression DO
   Workshop and exec commands
ENDWHILE
```

When the boolean expression in a **WHILE** command is true, the Exec Processor selects the corresponding **DO** clause by executing its exec commands and placing its Workshop lines in the exec run file. Then the Exec Processor reevaluates the **WHILE** command. If the expression is still true, the **DO** clause is selected again. When the expression becomes false, processing continues at the command following **ENDWHILE**. Commands that are not selected are examined for correct syntax. Here is an example of a **WHILE** statement that deletes a series of object files named **fileN**, **fileN-1**, and so on:

```
F{iler}
$while inputval > '0' do
   D{elete}-[vol]-[file][inputval].obj
   $set inputval to eval(inputval-1)
$endwhile
Q{uit}
```

The **REPEAT** statement consists of the **REPEAT** and **UNTIL** commands. The format of the **REPEAT** statement is

```
REPEAT
   Workshop and exec commands
UNTIL boolean-expression
```

The example shown above for the **WHILE** statement can be rewritten using the **REPEAT** statement:

```
F{iler}
$if inputval > '0' then
   $repeat
      D{elete}-[vol]-[file][inputval].obj
      $set inputval to eval(inputval-1)
   $until inputval = '0'
$endif
Q{uit}
```

9.2.3.4 The EXISTS and NEWER Boolean Functions

The **EXISTS** function returns a value of **true** if the specified file, catalog, volume, or device is online at process time; otherwise the value **false** is returned. A volume or device is online if it is mounted; a file is online if it exists on a mounted device. The format of the function is

```
EXISTS (pathname)
```

Pathname is any string expression that yields a valid file, volume, or device name. Some examples follow:

```
$if exists ("-slot2chan1") then          {device}
$if exists ("-[vol]") then               {volume}
$if exists ("-paraport-%1.obj") then     {file}
```

The **NEWER** function returns a value of **true** if the Last-Mod-Date of the first file specified is more recent than that of the second file; otherwise the value **false** is returned. Both files must be online at process time or an error will be reported. The format of the function is

```
NEWER (file1, file2)
```

File1 and **file2** can be any string expressions that yield a valid pathname. Some examples follow:

```
$if newer ("-[fed]-taxes", "-[state]-taxes")
   then {calc state}
$if not (newer ("-%3.obj", "-%3.backup")) then {backup
   is current}
$if newer("[pgm].TEXT", "[pgm].OBJ") then {recompile}
```

9.2.4 Built-In String Functions

A string function is a function whose result is a string (text) value. The types of strings used in exec files are described in Section 9.1.4, Syntax of Exec Lines and Workshop Lines. Even a function result that is a number (for example, **ORD**) is returned as a string. Since the **ORD**, **POS**, **LENGTH**, and **EVAL** functions always return a number, they may always be used in a numeric expression even though the function result is a string. In fact, any string function can be used in a numeric expression as long as it returns a number.

Several built-in string functions are included as part of the exec language. In addition, you can write your own functions (see Section 9.2.6.3, Calling a User Function). The built-in string functions provided by the Exec Processor are

CONCAT	Combines strings
UPPERCASE	Converts a string to uppercase
LOWERCASE	Converts a string to lowercase
LENGTH	Gives the length of a string
COPY	Copies all or part of a string
POS	Gives the position of a string within another string
CHR	Translates a number into its corresponding ASCII character
ORD	Translates an ASCII character into its corresponding number
EVAL	Provides string arithmetic
RETSTR	Returns the ProgComm return string.
TRIMBLANKS	Trims leading and trailing blanks.
NEXTFILE	Refer to Section 9.2.2.4.
IORESULT	Refer to Section 9.2.2.5.

9.2.4.1 The CONCAT Function

The **CONCAT** function lets you combine string expressions and functions to produce a single string result. The format of the **CONCAT** function is

CONCAT (string1, string2, ... stringN)

String1 is a string expression. **String2** through **stringN** are optional string expressions. The function result is a string containing the string parameters in the order they were given. Here's an example that combines two string variables and three string constants:

```
exec (vol,file,pathname)
...
set pathname to concat('-',vol,'-',file,'.text')
```

Note that you can accomplish the same result by using an expanded string constant:

```
set pathname to "-[vol]-[file].text"
```

9.2.4.2 The UPPERCASE and LOWERCASE Functions

The **UPPERCASE** function converts any lowercase letters in a string to uppercase. The **LOWERCASE** function converts any uppercase letters in a string to lowercase. Nonalphabetic characters remain unchanged. For instance, **UPPERCASE** converts **ABc,dEf.3$gh** to **ABC,DEF.3$GH**. The format of the functions is

```
UPPERCASE (string-expression)
LOWERCASE (string-expression)
```

You can save the result of the function in the same variable it converts:

```
$set pathname to uppercase(pathname)
```

You can also convert a string in order to compare it. In the following example, the expression is true whether the value of **reply** is **YES**, **yes**, or any other uppercase and lowercase combination of these three characters.

```
$while lowercase(reply)='yes' do
```

9.2.4.3 The LENGTH, COPY, and POS Functions

LENGTH gives the number of characters in a string, **COPY** duplicates part or all of a string, and **POS** gives the location of a substring within a string.

The **LENGTH** function returns the length of a string in its function result. (The length of a null string is '0'.) The format of the **LENGTH** function is

```
LENGTH (string-expression)
```

For example,

```
$if length(word) GT 24 then
   $writeln word,' is even longer than  ',
             'disestablishmentarianism!'
$endif
```

The **COPY** function copies all or part of a string into the result string. The format of the **COPY** function is

```
COPY (source, position, count)
```

Source is the string expression containing the *substring* (part of a string) to be copied. **Position** is a numeric expression indicating the place in **source** of the first character to be copied; the first position in **source** is 1. **Count** is a numeric expression indicating the number of characters to be copied. If fewer than **count** characters are found at **position**, those that are found are placed in the function result. (Note that this differs from the Pascal Copy function.) If **position** is beyond the end of the source string, **COPY** returns a null function result. The following example copies **establishment** out of **disestablishmentarianism**:

```
set %8 to copy('disestablishmentarianism',4,13)
```

The **POS** function returns the position of a substring within a string. If the substring does not appear in the string, the function result is '0'. The format of the **POS** function is

```
POS (substring, source)
```

Substring and **source** are string expressions. In the **COPY** example above, you can use the **POS** function if you don't know the position of **establishment** in the source string:

```
set %7 to 'disestablishmentarianism'
set %8 to copy(%7,pos('establishment',%7),13))
```

9.2.4.4 The CHR and ORD Functions

The **CHR** function returns a one-character string that represents the character value of a number. The **ORD** function returns a string that represents the numeric value of an ASCII character or any other character in the Lisa's extended character set. For any character x, **CHR(ORD(x))** is x.

The format of the **CHR** function is

```
CHR (numeric-expression)
```

Numeric-expression must result in a whole number; it is taken MOD 256, producing an intermediate result in the range 0..255. **CHR** returns the character that corresponds to the intermediate result.

You can use the **CHR** function to generate a nonkeyboard character. The following example writes a BEL character:

```
$if ioresult<>'' then {there's an error}
  $write chr(7) {ring bell}
$endif
```

The format of the **ORD** function is

```
ORD (string-expression)
```

String-expression must not be an empty string, or a process-time error will occur. If **string-expression** yields a string longer than one character, the numeric value of the first character is placed in the **ORD** function result.

9.2.4.5 String Arithmetic Using the EVAL Function

The **EVAL** function lets you do long-integer arithmetic. It evaluates a numeric expression and returns an integer value in the function result string. The format of the function is

```
EVAL (numeric-expression)
```

Numeric-expression consists of numeric (decimal) constants, variables that contain integer values, string functions that yield integer values (such as **LENGTH**, **POS**, and **ORD**), string constants with integer values (such as '25'), and the operators listed below. You can use parentheses to control the sequence of operations as in algebra. The numeric operators are:

+	Addition
-	Subtraction
*	Multiplication
/	Division
MOD	Modulo

Here is an example of an exec routine that takes a word and writes it vertically, one character per line:

```
set count to length(word)
while count GT 0 do
   writeln copy(word,1,1)
         {write first char}
   set count to eval(count-1)
         {reduce count by 1}
   set word to copy(word,2,count)
         {remove first char from word}
endwhile
```

9.2.4.6 The RETSTR Function

The **RETSTR** function returns a string containing whatever is currently in the ProgComm unit's return string. The format of the **RETSTR** function is

```
RETSTR
```

The value of **RETSTR** can be set by any program that uses the ProgComm unit's **PCSetRetStr** procedure. (Refer to the System Software Manuals binder of this set for more information about the ProgComm unit.) If you run a program containing **PCSetRetStr** from an exec file, you can check the results using the **RETSTR** function. For example,

```
exec
   run 'comm.prog.obj'
   if retstr <> 'SUCCESS' then
      abort 'Program failed.'
endexec
```

9.2.4.7 The TRIMBLANKS Function

The **TRIMBLANKS** function strips leading and trailing blanks and tab characters from a string. The format of the **TRIMBLANKS** function is

```
TRIMBLANKS (string-expression)
```

9.2.5 Controlling the Screen Display

When you write to the screen with **WRITE** or **WRITELN**, the information is displayed at the current cursor location. Three commands--**GOTOXY**, **CLEAR**, and **CURSOR**--are provided to let you do custom formatting of a screen display by moving the cursor and/or clearing the screen.

9.2.5.1 The CLEAR Command

The **CLEAR** command erases all or part of the screen. The format of the **CLEAR** command is

```
CLEAR option
```

Option is one of the following keywords:

SCREEN	Clears screen, moves cursor to home position.
ENDSCREEN	Clears screen from current cursor position to end.
ENDLINE	Clears current line from cursor position to end.

For instance, the following exec program demonstrates the use of all three forms of the **CLEAR** command, plus the **GOTOXY** command, to display text on a diagonal across the screen:

```
exec (DisplayStr, x, y)
   clear screen
   repeat
      gotoxy 0,0 {move cursor to home}
      clear endline {clear for input, but don't destroy
                     previous display}
      write '>' {prompt}
      readln DisplayStr {get text}
      if lowercase(DisplayStr)='quit' then
         halt 'Done'
      else {display text on diagonal,one char at a time}
         set x to 20
         set y to 6
         clear endscreen {leave prompt, clear display}
         while DisplayStr <> '' do
            {display high-order char, then delete}
            write copy (DisplayStr,1,1)
            set DisplayStr to copy (DisplayStr,2,255)
            {move cursor to next point on the diagonal}
            set x to eval (x+2)
            set y to eval (y+1)
            gotoxy x, y
            endwhile
         endif
      until false
   endexec
```

9.2.5.2 The CURSOR Command

The **CURSOR** command lets you move the cursor relative to its current location. (To move the cursor to an absolute coordinate, use the **GOTOXY** command described in the next section.) The only change the **CURSOR** command makes to the screen display is to relocate the cursor. The format of the **CURSOR** command is

```
CURSOR option
```

Option is one of the following keywords:

HOME	Cursor moves to location 0,0 (upper left corner).
UP n	Cursor moves n positions up from current location.
DOWN n	Cursor moves n positions down from current location.
LEFT n	Cursor moves n positions left from current location.
RIGHT n	Cursor moves n positions right from current location.

N is an optional numeric expression; if you don't give it a value, it defaults to 1. Here is an example of the **CURSOR** command where the value of **n** is determined by the **EVAL** function:

```
$write %6
$cursor left eval(length(%6)-1)) {move cursor to start
                                   of previous write}
```

9.2.5.3 The GOTOXY Command

The **GOTOXY** command moves the cursor to the screen coordinates you specify. (To move the cursor relative to its current position, use the **CURSOR** command.) The format of the **GOTOXY** command is

```
GOTOXY x,y
```

X and **y** are numeric expressions representing screen coordinates: **x** represents the location of the cursor in the horizontal plane; **y** represents its location in the vertical plane. The top left corner of the Lisa screen is location 0,0; the lower right corner is location 79,31. If you supply a value of **x** or **y** beyond the limit for the coordinate, the limit value will be substituted.

If **lastx** and **lasty** represent the rightmost location and the downmost location respectively, the following example moves the cursor to the center of the screen:

```
$gotoxy lastx/2, lasty/2
```

9.2.6 Calling Another Exec Program

One exec program can call another either as a *user function*, which returns a string result to its caller, or as an *exec procedure*, which does not return a result. Although a single exec program can call any number of execs as procedures or functions, only one exec run file is generated. *Nested calls* are permitted; that is, a called exec may in turn contain exec procedure and function calls.

When an exec is called as a procedure by using the **SUBMIT** command, it must not return a function value. The exec procedure may end by executing its last line or by issuing a **RETURN** command with no argument.

When an exec is called as a user function, it must end with a **RETURN** command that returns a string result. See Section 9.2.6.3, Calling a User Function, for more information.

In the case of both exec procedures and user functions, the Exec Processor executes the exec lines in the called exec, processes its Workshop commands and places them in the exec run file. The exec run file contains the output from processing all of the input exec files.

The *invocation* of an exec procedure or user function specifies the pathname of the called exec, its parameter list, and Processor options where permitted. The Exec Processor treats the invocation as *text*--as if it were in a Workshop line. Within this text, imbedded built-in and user function calls are not permitted. The invocation must be on a single line. The length of the invocation after processing must not exceed 255 characters. The only processing performed on the invocation is as follows:

- Process tildes.
- Substitute the current values of variables. (Named variables must be enclosed in square brackets, as in Workshop lines.)
- Remove comments.

9.2.6.1 Calling an Exec Procedure with the SUBMIT Command

The **SUBMIT** command calls an exec procedure. The Exec Processor processes the called exec and puts its Workshop lines into the current exec run file.

The **SUBMIT** command must be on a single line. The format of the **SUBMIT** command is

```
SUBMIT exec-run-command
```

Exec-run-command is the invocation text and follows the rules described in the previous section. The format of **exec-run-command** is

```
filename (invocation-parameter-list) option-list
```

Filename is the pathname of the exec procedure.

Invocation-parameter-list is an optional list of initial values to be passed to the exec procedure; the values must be separated by commas. If the parameter list is empty and is followed by options or other significant text, its place must be indicated by parentheses.

Option-list is an optional list of Processor options; only the Imbed and Blanks options are valid on a **SUBMIT** command. See Section 9.3.2 for more information on Processor options.

Some examples of the **SUBMIT** command follow:

```
submit testexec
submit makeanyprog.text(oneprog)I
submit noparams()B
submit endweek ([ledger],[payroll])
```

9.2.6.2 The RETURN Command

The **RETURN** command tells the Exec Processor to resume processing the calling exec. In a user function, the **RETURN** command must be the last command executed. The format of the **RETURN** command is

```
RETURN function-value
```

Function-value is a string expression that contains the value returned by the called exec. If the called exec is a user function, **function-value** is required; if the called exec is a procedure, **function-value** is not permitted. Here are some examples of valid **RETURN** commands:

```
$return
$return "The data is %5."
$return 'done'
```

An exec procedure needs a **RETURN** command only if the exec procedure does not end by executing the **ENDEXEC** command. In the exec procedure shown below, the **RETURN** command terminates an endless loop:

```
EXEC
WHILE TRUE DO
   ...
   IF  <condition> THEN
      RETURN
   ENDIF
ENDWHILE
ENDEXEC
```

9.2.6.3 Calling a User Function

A user function is a user-written exec program that returns a string value using the **RETURN** command. You can call a user function from another exec wherever you would use a string expression. If the user function contains Workshop lines, including comment lines, they will be processed and placed in the exec run file.

The format of a user function invocation is

```
< filename (invocation-parameter-list)
```

where < tells the Exec Processor to process a user function. **Filename** is the pathname of the user function exec file. **Invocation-parameter-list** is optional and follows the rules for the **exec-run-command** (see Section 9.2.6.1). For instance,

```
$while <-taxes-quarter() > '0' do
```

Here is an example of a user function, **GETDATA**, that returns data to the calling exec each time the function is invoked; when no more data can be read, **GETDATA** returns the string value **'done'** in its function result. The function contains two **RETURN** commands; one or the other is executed as the last command. The **count** variable is set by the calling exec.

```
exec (count, data)     {GETDATA}
if count eq 1 then {open datafile}
  reset indata, data.text
endif
read (indata) data
if ioresult = '' then
  return data
else
  close indata
  return 'done'
endif
endexec
```

The routine below is from an exec program that calls the **GETDATA** function. (This routine does not use the same dollar-sign convention as **GETDATA**.)

```
$exec (counter,reply)
...
$set counter to '1'
$set reply to <getdata([counter])
$while reply <> 'done'
  $writeln '#',count,' = ',reply
  $set counter to eval(counter+1)
  $set reply to <getdata([counter])
$endwhile
```

9.2.7 Commands that Control the Exec Processor

There are four exec commands that affect the running of the Exec Processor:

- **HALT** tells the Exec Processor to stop processing and run the exec run file created thus far.
- **ABORT** tells the Exec Processor to stop processing without running the exec run file.
- **RUN** tells the EXEC Processor to run a program and then resume processing the exec file.
- **DOIT** tells the Exec Processor to run the current contents of the exec run file and then resume processing the exec file.

9.2.7.1 The HALT and ABORT Commands

The **HALT** command tells the Exec Processor to stop processing the exec source file and run the exec run file in its present state. The **ABORT** command terminates processing without running the exec run file. With either command you can display a message. The format of the commands is

```
HALT string-expression
ABORT string-expression
```

String-expression is optional; if present, the contents are displayed on the console. For instance,

```
$halt 'Processing stopped at program #3'
ABORT "Incorrect date [INDATE] in parameter list"
```

9.2.7.2 The Exec RUN and ENDRUN Commands

The exec **RUN** command is a command within an exec program--it is not the same as the Workshop Run command. The format of the exec **RUN** command is

```
RUN filename
```

Filename is a string expression resulting in the pathname of a program you want to run *during exec processing*. When the Exec Program finds the **RUN** command, it suspends processing of the exec source file and runs the program; then it resumes processing; for example,

```
$if day = '1' then
   $run '-monthly-firstday.obj'
$endif
```

The **ENDRUN** command is required only if you want to supply input data to the program named in the **RUN** command. In this case, you must also specify the **INPUT** keyword in the **RUN** command as follows:

```
RUN filename INPUT
   input lines
ENDRUN
```

The input lines between **RUN** and **ENDRUN** are placed in a temporary file and have no effect on the exec run file.

The Generate Processor option disables the exec **RUN** command.

9.2.7.3 The DOIT Command

In a simple exec program without **DOIT** commands, all of the exec commands are executed (by the Exec Processor) before any Workshop lines are executed. The **DOIT** command allows the execution of Workshop lines to be interleaved with the execution of exec commands. The format of the **DOIT** command is

```
DOIT
```

When the Exec Processor finds the **DOIT** command, the following actions are taken:

1. Processing of the exec source file is suspended.
2. The current contents of the exec run file are run by the Workshop.
3. The contents of the exec run file are erased, and a new exec run file is started.
4. Processing of the exec source file resumes at the point where it was suspended.

These actions occur even if you are stepping through the exec source file using the Step option. However, the Generate option disables the **DOIT** command. (For more information see Section 9.3.2, Processor Options.)

You can use the **DOIT** command to display run-time messages as Workshop lines are executed. If the **DOIT** command is omitted from the following example, the "Backup completed" message will be displayed before the backup actually takes place:

```
exec (fromVol, toVol)
   writeln 'Now starting backup...'
   $F{iler}B{ackup}[fromVol]-=,[toVol]-$
   $Q{uit the Filer}
   DOIT
   writeln 'Backup of ',fromVol,' to ',toVol,
             ' completed.'
endexec
```

9.3 Running an Exec Program

Exec programs are run under the main command line using the Workshop Run command. The Run command calls in the Exec Processor to read your exec source file, execute its exec commands, and create an exec run file containing only Workshop lines. The Workshop then runs the exec run file, which is automatically deleted at the end of the run unless you specified the Keep Processor option.

When a Workshop Run command is used to invoke an exec from within another exec, the result is *chaining*. The difference between submitting an exec procedure (see Section 9.2.6) and chaining is that an exec procedure is processed before any Workshop commands are executed; a chained exec is processed after all of the Workshop commands in the chaining exec have been executed. See Section 9.4.1, Exec File Chaining, for more information.

9.3.1 The Workshop Run Command

The format of the Workshop Run command that invokes the Exec Processor is

```
R<exec-run-command
```

or

```
REXEC/exec-run-command
```

The format of the **exec-run-command** invocation (also discussed in Section 9.2.6.1) is

```
filename (invocation-parameter-list) option-list
```

For example,

```
r<testexec
r<noparams()sb
rexec/-upper-compile(-lower-testprog)i
```

Filename is the pathname of the exec program you want to run. An extension of .TEXT is assumed unless you override the extension by adding a period at the end of **filename**. For example,

You type	The Workshop looks for
abc	**abc.text**
abc.xyz	**abc.xyz.text**
abc.	**abc**

Invocation-parameter-list is an optional list of initial values for parameters; if present, it is enclosed in parentheses. It can be empty, or it can include up to 20 parameter values separated by commas. Omitted parameters are specified by commas; for example **(10,,May)**. If a parameter is not specified, its value is an empty string ('').

Option-list refers to the options described below.

9.3.2 Processor Options

You can modify the Exec Processor's operation by specifying one or more single-letter Processor options following the invocation parameter list. Processor options allow you to tailor the processing of your exec

- By controlling the way spaces are handled--Blanks option.
- By proceeding even if errors are encountered while running--Error option.

- By processing the exec file without running it--Generate option.
- By imbedding your exec in a source file--Imbed option.
- By saving the exec run file that is normally deleted--Keep option.
- By stepping through an exec source file, selectively including Workshop lines for its exec run file--Step option.
- By running from a previously saved run file--Rerun option.

To request an option, type the option letter after the exec parameter list. Include an empty parameter list if you want to specify options but not parameters. For example,

```
R<firstexec(apples,,oranges)BE   {Blanks, Errors}
REXEC/anotherexec(9,17,7,23)S    {Step}
$submit lastexec()i              {Imbed}
```

The default condition for all options is that they are not in effect unless specified. Most Processor options are global--they apply to the exec on which they're specified and also to any execs it calls; they are therefore not permitted on a **SUBMIT** command invocation. The two exceptions are the Blanks and Imbed options, which are local and are permitted with **SUBMIT**. You may not specify Processor options when invoking a user function.

B *The Blanks option* tells the Exec Processor not to remove leading and trailing blanks from the Workshop lines it places in the exec run file. Leading blanks result from indenting lines to improve exec readability. (Leading and trailing blanks are not significant to Workshop programs, but they might be significant to other programs you run under the Exec Processor.)

E *The Errors option* tells the Exec Processor to continue processing even if errors are encountered that would normally stop exec file execution. This option is useful for forcing the completion of a test series.

G *The Generate option* tells the Exec Processor to generate an exec run file without running it. Syntax errors are flagged. The **DOIT** and **RUN** commands are disabled. By specifying the Keep option with the Generate option, you can retain the exec run file and examine or modify it using the Workshop Editor. If **K** is not specified, the exec run file is deleted.

I *The Imbed option* tells the Exec Processor to ignore the first line of the exec file because the exec is imbedded in a source program. For instance, the exec file can also be the source file for the Pascal Compiler. To use this technique, begin the first line of the exec file with the Pascal comment delimiter **(*** and follow the **ENDEXEC** command with the Pascal comment delimiter ***)**; then begin the source program.

Imbedding works with any language that allows you to extend a comment over more than one line, including exec language. Here is a Pascal source program in file **-Pascal.prog.text** that contains an imbedded exec program:

```
(* This Pascal program compiles itself!
$exec {Pascaltest}
P{ascal Compile}Pascaltest
Pascaltest.list
Pascaltest.obj
$endexec
*)
PROGRAM Pascaltest;
USES...
TYPE...
VAR...
BEGIN...
END.
```

To compile this program, simply type the following Workshop Run command:

```
R<Pascaltest()I
```

K *The Keep option* tells the Exec Processor not to delete the exec run file after the Workshop runs it. You may then rerun the file using the **R** option.

R *The Rerun option* tells the Exec Processor to run a previously processed exec run file that was saved using the **K** option. This option overrides all other options.

S *The Step option* puts the Exec Processor in Step mode so that it displays the exec run file one line at a time, prompting you for selective skipping of output lines and **SUBMIT** commands. Specify the Keep option also if you want to keep the exec run file. This option is further described below.

9.3.3 Using the Step Option

With the Step option, the Exec Processor processes the exec source file one line at a time and prompts you for a decision:

<= **Include** ? for a Workshop line.

<= **Submit** ? for a **SUBMIT** command line.

When you first enter Step mode, you can get an explanation of the possible responses by answering Y to the **More details** ? prompt. You can also get help by answering ? to the decision prompts. The responses are:

Y Include the Workshop line or submitted exec program in the exec run file.

N Omit the line or submitted exec program.

S Step through the submitted exec (with **Submit ?** only).

A Abort processing; the exec run file is not run.

K Keep the remaining lines of the exec source file as is (process exec lines, include Workshop lines without further prompting), and run the exec run file.

I Ignore the remainder of the exec source file, keep previously included Workshop lines, and run the exec run file.

You can use the Step option to skip over the first portion of an exec file--for instance, when debugging a series of programs. Step through the exec that runs the series, responding with **N** to eliminate the programs that ran successfully. Then when you get to the program that failed and has been corrected, respond with **K** to generate the exec run file with only the remaining programs in it.

You can also select separate sections or modules of a large application. In this case you can use Step mode most easily if you place each module in a separate exec file, as in the following high-level exec file called **RUNALL** which runs modules A, B, C, D, and E:

```
exec {RUNALL}
 submit Aexec
 submit Bexec
 submit Cexec
 submit Dexec
 submit Eexec
endexec
```

To select only modules B and D, invoke **RUNALL** in Step mode. If you want to keep the exec run file so that you can run it again without going through the selection process, specify the Keep option as well as the Step option:

```
r<runall()sk
```

Your dialog with the Exec Processor in Step mode as you select **Bexec** and **Dexec** for running is shown below, with your responses in italics:

```
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]
```

```
submit Aexec                      <= Submit ?   N
submit Bexec                      <= Submit ?   Y
submit Cexec                      <= Submit ?   N
submit Dexec                      <= Submit ?   Y
submit Eexec                      <= Submit ?   I
```

NOTE

If the exec you are stepping through contains a **DOIT** command, the contents of the exec run file are executed when the **DOIT** line is encountered (unless it's in the false part of a conditional statement); then you are returned to stepping.

9.3.4 The File Cache and the Input Buffer

The Exec Processor uses a file cache for improved performance. If you need to optimize the performance of an exec program that calls exec procedures and user functions, understanding the file cache can help you.

The *file cache* consists of five pages (a page is two blocks) that can contain five small files at a time in memory. A *small file* has a listed size of four blocks--according to the File Manager's List command--and contains one header page and one page of text. If an exec procedure or user function is called repeatedly--within the range of a **WHILE** statement, for example--it should be a small file so that it can be read from memory rather than from disk.

Small files that are accessed by a **SUBMIT** command or a function call are placed in the cache. Subsequent access to these files is made from the cache rather than from disk. The cache is maintained on a least-recently-used basis. That is, once the cache is full, the file least used recently is the one whose space is relinquished for a new small file.

If your exec modifies itself and then calls itself (and we don't recommend this), the modified version won't execute if the previous version is still in the cache. To avoid this problem, make the self-modifying exec larger than four blocks.

The *input buffer* is the area in memory where large exec files are read. If one large file is called repeatedly by a second large file, both files must be read from disk each time through the loop. To optimize performance, modularize the large files so that at least one file can be accessed from the cache.

9.4 Sample Exec Programs

The following sections contain a series of actual exec programs that demonstrate some useful techniques like chaining and recursive calls.

9.4.1 Exec File Chaining

Chaining takes place when the Workshop Run command is used from *within* an exec: the Workshop is executing the current exec run file--the *chaining* exec program--when it encounters a Workshop Run command; it then closes the current exec run file and invokes the Exec Processor to begin processing the new exec source file named in the Workshop Run command--the *chained* exec program. The chaining exec is *not* returned to for further processing; the Workshop Run command is effectively its last command. In the example illustrated by Figure 9-2 below, exec program A invokes exec program B by means of the Workshop Run command. Exec run file A is executed. Its last command is

```
R<ExecB.text
```

The Workshop then returns control to the Exec Processor, which processes exec source file B and gives its exec run file to the Workshop to run.

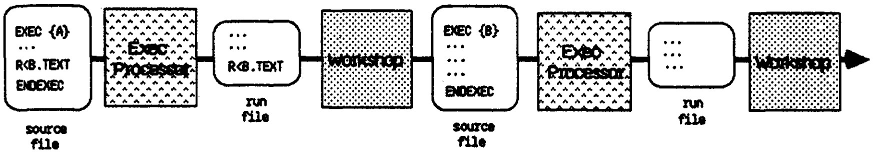


Figure 9-2. Chaining Exec Files

Here is a set of four exec files that demonstrates the use of exec file chaining, using Pascal compiles as an example.

- **COMP** performs a basic Pascal compile.
- **COMPIF** submits **COMP** only if the object file does not already exist or if the source file is newer than the object file.
- **LINKIF** links the three units if any of them was changed since the last link.
- **COMPLINKIF**, the calling exec, submits **COMPIF** for three separate Pascal units--conditionally compiling them--and then chains to **LINKIF**.

The **COMP** exec program follows:

```
EXEC (unit,objname) {*** COMP *** Pascal compile
        unit: source to compile
        objname: alternate name for object file }
DEFAULT objname TO unit {if no alternate name use source name}
$P{Pascal compile}[unit]
   ${no list file}
   $[objname]{object file}
ENDEXEC
```

The **COMPIF** exec program follows:

```
EXEC (unit,objname) {*** COMPIF *** conditional compile
       unit: source to compile
       objname: name of OBJ file
  DEFAULT objname TO unit {if no alternate name use source name}
  IF EXISTS ("[objname].obj") THEN
     IF NEWER ("[unit].text","[objname].obj")
            THEN {recompile if source newer than object}
        SUBMIT comp([unit],[objname])
     ENDIF
  ELSE  { object file does not exist, so generate it }
     SUBMIT comp([unit],[objname])
  ENDIF
ENDEXEC
```

The LINKIF exec program follows:

```
EXEC  {*** LINKIF *** Link the object modules into a
                        new executable program if any
                        of them was recompiled.}
   IF NEWER ('unit1.obj', 'program.obj')
   OR NEWER ('unit2.obj', 'program.obj')
   OR NEWER ('unit3.obj', 'program.obj') THEN
    $L{ink}unit1
    $unit2
    $unit3
    $iospaslib
    ${end of input}
    ${no list file}
    $program{executable output file}
   ENDIF
ENDEXEC
```

The COMPLINKIF exec program follows:

```
EXEC (unit1,unit2,unit3) {*** "COMPLINKIF" *** compile
                        if necessary, then chain to
                        link}
    SUBMIT compif([unit1])
    SUBMIT compif([unit2])
    SUBMIT compif([unit3])
    $R<LINKIF          { Chain to link exec file after compiles
                        have run so that LINKIF exec gets the
                        correct file dates.  Note the
                        difference between process time and run
                        time.}
 ENDEXEC
```

Here's what happens when **COMPLINKIF** is run:

1. **COMPIF** is invoked for **unit1**. If **unit1** needs to be compiled, **COMP** is submitted and the Workshop lines for the compile are placed in the exec run file.
2. In the same way, **COMPIF** is invoked for **unit2** and **unit3**, until the exec run file contains all of the commands necessary to compile any unit that requires it. The Workshop then runs the exec run file.
3. When the Workshop finds the command to Run the **LINKIF** exec, it calls on the Exec Processor to start a new exec run file. **LINKIF** now has available the dates of the most recent compiles. If **LINKIF** were submitted rather than chained to, the compiles would execute *after* **LINKIF** compared dates. (But you could accomplish the same effect as chaining by adding a **DOIT** command to force the compiles and then submitting **LINKIF**.)
4. The Workshop gives control back to the Exec Processor to process **LINKIF**, which creates a new exec run file containing commands for the Linker.

9.4.2 A Recursive Exec Program

The **RCOMP** exec performs up to ten Pascal compiles, using the **COMP** exec described in the previous section. **RCOMP** 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
```

9.4.3 A Recursive User Function

The **GETPROFLOC** exec is a function that 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. The function calls itself recursively until a valid device name is specified.

```
EXEC (pLoc) {***GETPROFLOC*** prompt user for ProFile location }
  REQUEST pLoc WITH
  'Where is the ProFile attached (paraport/slot2chan1/slot2chan2)'
  SET pLoc TO UPPERCASE (pLoc)
  IF (pLoc <> 'PARAPORT') AND (pLoc <> 'SLOT2CHAN1')
       AND (pLoc <> 'SLOT2CHAN2') THEN
```

```
      WRITELN 'That is not a valid device name.  Let''s try again.'
      RETURN <GetProfLoc  { recursive function call }
    ELSE
      RETURN pLoc
    ENDIF
  ENDEXEC
```

9.4.4 An Exec Application

The application listed below verifies the contents of a disk: **CHECK** lists missing files, and **CHECK2** lists extraneous files. The disk to be verified is compared against **GoodListFile**, a text file containing the list of valid files, one per line. The application consists of two main execs (**CHECK** and **CHECK2**), a user function (**DEVNAME**), and an exec procedure (**CHKIORESULT**). Both main exec programs call the **DEVNAME** function to format device names and the **CHKIORESULT** procedure to handle I/O errors.

The **DEVNAME** user function follows:

```
EXEC (DevName, DevDefault) { DEVNAME function returns device
            name with leading '-' }
DEFAULT DevName TO DevDefault
IF COPY (DevName, 1, 1) <> '-' THEN
   SET DevName TO CONCAT ('-', DevName)
ENDIF
RETURN DevName
ENDEXEC
```

The **CHKIORESULT** exec procedure follows:

```
EXEC (ErrorMsg, IORes)   { CHKIORESULT will abort if we get an
            IORESULT error; sounds bell and prints message }
IF IORESULT <> '' THEN
   SET IORes TO IORESULT { so WRITEs below will not change its
            value }
   WRITELN CHR(7), ErrorMsg
   WRITELN IORes
   ABORT 'Bye'
ENDIF
ENDEXEC
```

The CHECK exec program follows:

```
EXEC (GoodListFile, CheckVol, FileName)
{ CHECK looks for missing files on CheckVol; GoodListFile is a
            text file containing an alphabetical list of the
            files that should be on CheckVol, one file name per
            line. }
DEFAULT GoodListFile TO 'GoodFileList.Text'
SET CheckVol TO <devName([CheckVol],-MYVOL)

{ check for missing files }
RESET GoodFile, GoodListFile
SUBMIT chkIOResult(Could not open [GoodListFile])
WRITELN 'Check of ', CheckVol, ' against GoodListFile (',
            GoodListFile, ')'
WRITELN

REPEAT { get file name and see if file exists on CheckVol }
   READLN (GoodFile) FileName
   IF FileName <> 'EOF' THEN
      IF NOT (EXISTS ("[CheckVol]-[FileName]")) THEN
         WRITELN CHR(7), 'Missing file: ', FileName
      ENDIF
   ENDIF
UNTIL FileName = 'EOF'
CLOSE GoodFile
ENDEXEC
```

The CHECK2 exec program follows:

```
EXEC (GoodListFile, CheckVol, GoodName, FileName, LastGoodName)
{ CHECK2 looks for extraneous files on CheckVol; GoodListFile
            should be the name of a text file with an
            alphabetized list of the files that should be
            present, one file name per line. }
{ Note: this will not work if the volume being checked has
            sub-catalogs, since the Names command will not
            return the full pathnames for files within the
            catalogs. }

DEFAULT GoodListFile TO 'GoodFileList.Text'
SET CheckVol TO <devName([CheckVol],-MYVOL)
{ get the names of the files on CheckVol }
$F{ile-Mgr}N{ames}[CheckVol]-=,CHECK.TMP.TEXT
$Q{quit}
DOIT { execute File-Mgr commands to create list of files in
            CHECK.TMP.TEXT }
```

```
RESET NameFile, 'CHECK.TMP.TEXT'
SUBMIT ChkIOResult(Could not open CHECK.TMP.TEXT)
READLN (NameFile) FileName  {ignore 3 title lines from Names
        cmnd }
READLN (NameFile) FileName
READLN (NameFile) FileName
RESET GoodFile, GoodListFile
SUBMIT chkIOResult(Cound not open [GoodListFile])
SET LastGoodName TO 'A'    { alphabetically first }
READLN (GoodFile) GoodName { prime the pumps }
READLN (NameFile) FileName

REPEAT
   SET GoodName TO UPPERCASE (GoodName)
   SET FileName TO UPPERCASE (FileName)
   IF (GoodName < LastGoodName) AND (GoodName <> 'EOF') THEN
      WRITELN CHR(7), GoodName, ' is not alphabetical in ',
            GoodListFile
      ABORT 'Bye'
   ENDIF
   SET LastGoodName TO GoodName
   IF (GoodName = 'EOF') AND (FileName = 'EOF') THEN
      HALT 'Done'
   ELSEIF GoodName = 'EOF' THEN
      WHILE FileName <> 'EOF' DO
         WRITELN CHR(7), 'Extra file: ', FileName
         READLN (NameFile) FileName
      ENDWHILE
      HALT 'Done'
   ELSEIF FileName = 'EOF' THEN { missing files will be
            detected by other test }
      HALT 'Done'
   ELSEIF FileName = GoodName THEN
      READLN (GoodFile) GoodName
      READLN (NameFile) FileName
   ELSE { mismatch — list extra files & resynchronize }
      IF GoodName < FileName THEN {missing files}
         REPEAT
            READLN (GoodFile) GoodName
            SET GoodName to UPPERCASE(GoodName)
         UNTIL (GoodName >= FileName) OR (GoodName = 'EOF')
      ENDIF
      IF GoodName <> FileName THEN
         REPEAT
            WRITELN CHR(7), 'Extra file: ', FileName
            IF FileName < GoodName THEN
               READLN (NameFile) FileName
```

```
                    SET FileName TO UPPERCASE (FileName)
                 ENDIF
              UNTIL (FileName >= GoodName) OR (FileName = 'EOF')
           ENDIF
           IF FileName = GoodName THEN
              READLN (NameFile) FileName
           ENDIF
           READLN (GoodFile) GoodName
        ENDIF
     UNTIL FALSE
     ENDEXEC
```

9.5 Exec File Errors

The Exec Processor reports syntax errors, I/O errors, and other process-time errors; it also reports errors resulting from Operating System calls. The format in which the Exec Processor reports errors is:

ERROR in <error location>
<current line>
<error marker>
<error message>

where

<error location>	is either 'invocation line' or 'line #<n> of file <file>'.
<current line>	is the text of the exec line in which the error was detected.
<error marker>	is a question mark indicating the place in **<current line>** where the error was detected.
<error message>	is one of the messages listed below. The error message begins with an error number.

9.5.1 Syntax Errors

The line containing the syntax error does not conform to the rules of the exec language. Check to see that you have typed the line correctly; refer to Section 9.1.4, Syntax of Exec Lines and Workshop Lines, and to descriptions of the individual commands and options for more information.

1	More than 20 parameters on exec procedure/function call
2	No closing) found
3	End of Exec file before ENDEXEC
4	No Exec file specified
6	End of Exec file in comment
7	Invalid percent: not "%n" form
8	Garbage at end of command
9	File does not begin with EXEC
10	No argument to SUBMIT

11	ELSE, ELSEIF or ENDIF not in IF
12	ELSEIF after ELSE
13	Nothing following
14	EXEC command other than at start of file
16	More than 20 variables declared
19	ENDWHILE not in WHILE
20	Duplicate parameter/variable name
21	Bad number. Numeric constant expected
22	Number too large
23	ORD requires a string argument of at least one character
24	UNTIL not in REPEAT
25	Bad Number for first argument to numeric comparison
26	Number too large for first argument to numeric comparison
27	End of Exec file in RUN command input
28	Bad Number. String expression with numeric result expected
--	Invalid command. <token> expected.

<token> is one of the following:

String value
Numeric value
Number
String expression with numeric result
Boolean value
Parameter name
Parameter/variable
String compare operator
<>
Comma (list delimiter)
Command
Terminating string delimiter
Valid command keyword
(
)
"ENDIF"
"ENDWHILE"
"UNTIL"
Catalog specification
File Identifier
Clear command (Screen, EndScreen EndLine)
Cursor command (Home, Up, Down, Right, Left)
Program name

9.5.2 I/O Errors

The I/O error reported by the Exec Processor is followed by an additional line with the text of the corresponding Operating System error message.

201	Unable to open input file "<file>"
202	Unable to open exec run file "<file>"
203	Unable to access file "<file>"
204	Unable to rerun file "<file>"
205	Unable to reread file "<file>"
211	Unable to reopen input file "<file>"

9.5.3 Other Exec Errors

5	Line buffer overflow (> 255 chars)
15	Out of memory. Exec processing aborted
17	No value returned from file called as function
18	RETURN with value in file not called as function
28	Bad Number. String expression with numeric result expected
29	Number returned by string expression is too large
206	File variable "<id>" already in use
207	File variable "<id>" is undefined
208	File variable "<id>" is not open for input
209	File variable "<id>" is not open for output
210	Bad exec run file name generated: "<file>"

Chapter 10
The Transfer Program

The Transfer Program

10.1 Introduction

The Transfer program is a data communications utility that supports the transfer of text between your Lisa and another computer that we'll call the *remote computer.* The Transfer program can send text from a file to the remote computer. It can also act as a *terminal emulator:* Everything you type on the Lisa keyboard is transmitted to the remote computer. Text received by your Lisa can be stored in a standard text file that you can read using the Workshop Editor.

10.2 Hardware Connection and Software Configuration

Before you can use the Transfer program you must establish a physical connection between your Lisa and the remote computer. Then, in order to transfer data properly from one computer to another, you must set certain data communications characteristics on the Lisa to match the remote computer. Establishing this software connection is known as *configuration.*

If you want to connect the Lisa to the remote computer by telephone, attach a modem to your phone jack and to the Serial A or Serial B connector on the back of the Lisa.

If you want to connect the Lisa directly to the remote computer, connect a modem eliminator cable to an RS232 cable; attach the modem eliminator end of the cable to a serial port on the remote computer; attach the RS232 end of the cable to the Serial A or Serial B connector on the back of the Lisa.

To configure the Workshop software, let the Workshop know what's connected to the serial ports by selecting the Preferences tool from the System Manager command line and using the Device Connections menu to set either Serial A or Serial B to Remote Computer.

To configure the Transfer program software at the beginning of a transfer, choose a value from each of the following menus (described in detail in the next section):

- **Baud Rate** The speed at which data is transferred. Ten baud represents about one character per second; for example, 300 baud is equivalent to 30 cps.
- **Parity** The "insurance policy" that ensures the valid transmission of data.
- **Handshake** The hardware or software mechanism for synchronizing data transmission.

- **Duplex** The type of information flow between the Lisa and the remote computer.
- **Connector** The serial port you plan to use (A or B).

To control transmission while the transfer session is in progress, use the Control menu:

- **Control** Start or stop receiving or sending data; filter out control characters; increase transfer speed by suppressing text display; set line delay; exit from the Transfer program.

NOTE

When the Workshop shell is initialized, all serial ports are configured as follows: 9600 baud, DTR handshake, automatic linefeed insertion. When you leave the Transfer program, these defaults are automatically restored.

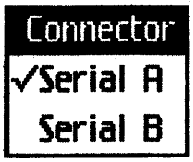
10.3 Setting Transfer Program Characteristics

In order to communicate with a remote computer, the Transfer program menus must be set so that the Lisa transmits and receives data in the same way as the remote computer. If you are dialing a service on a mainframe computer, use the settings specified in the mainframe computer manual. If you are connecting your Lisa to another Lisa, make sure that both Lisas are set to the same characteristics.

Each menu has a *default setting* that is in effect when you start the Workshop. To change the default, open the menu and click on the setting you want. When you exit from the Transfer program, the Workshop saves the last settings you used: In other words, you automatically create your own custom default settings that last until you change them.

10.3.1 The Connector Menu

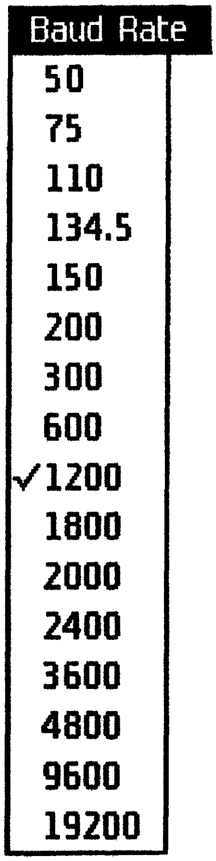
This menu allows you to specify which serial port you will use to connect your Lisa to the remote computer. (You can only use a connector if it is specified in the Preferences menu.) The default is Serial A. For more information on the serial connectors, see the Hardware chapter in the *Lisa 2 Owner's Guide*.



10.3.2 The Baud Rate Menu

Baud rate is the speed at which data is transmitted to or from the remote computer. The baud rate must be set to agree with the remote computer and modem you are using. The default is 1200 baud. Valid baud rate settings for the serial ports are shown in the "PortConfig" section of the Utilities chapter. Note that 3600, 7200, and 19200 baud are not available on Serial A.

On telephone-line connections, the faster the baud rate, the less reliability the data will have. If you are getting garbled transmission or missing data, you might need to use a lower baud rate (but remember to synchronize with the remote computer). Standard rates for transmission over telephone lines are 300 baud and 1200 baud.



10.3.3 The Parity Menu

Data transmission between computers can be unreliable because of pops, clicks, crosstalk, and noise on telephone lines; hard-wired lines are also subject to interference and weak signals. *Parity error detection* is the most common method of detecting data communications errors. This method does not *correct* errors; it merely points them out--the Transfer program displays characters with bad parity as highlighted question marks.

Parity error detection depends on the fact that the ASCII character set requires only seven bits of an eight-bit byte to encode the standard 128 characters. The eighth bit, known as the *parity bit,* can be set to make each character transmitted contain either an even number of 1-bits *(even parity)* or an odd number *(odd parity).* If a bit in a character is inadvertently changed during transmission, the number of 1-bits will not match parity and the byte will be highlighted as an error. (Note that this method can detect only an odd number of bit changes in a character. If two, four, or six bits change, parity checking will not detect an error. This means that parity checking works best with relatively reliable lines.)

Parity should be set to agree with the remote computer. The parity choices provided by the Transfer program are Even, Odd, or None. The default is None. If you are sending or receiving characters from the extended character set, choose None (see Section 10.4.2, Transmitting Special Characters, for more information).

10.3.4 The Handshake Menu

Handshaking is the exchange of predetermined signals between two computers in order to synchronize transmission. The Handshake menu allows you to select XON/XOFF (a software handshake), DTR (a hardware handshake), or None. The default is None.

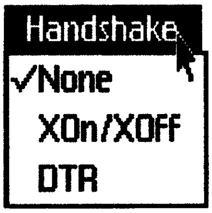
XON/XOFF is a software protocol for use with a modem or a modem eliminator. It allows the transfer of a continuous string of characters, pausing only when the receiving buffer is nearly full. Using this protocol, the Lisa can stop transmission from the remote computer by sending XOFF and start it again by sending XON; likewise, the remote computer can start and stop transmission from the Transfer program by sending XON and XOFF to the Lisa.

NOTE

If you use XON/XOFF and the information transmitted includes an XON or XOFF, the transmission will halt and the Lisa will time out. The XON and XOFF characters are the same as the ASCII Control-Q ($11) and Control-S ($13) characters.

DTR (Data Terminal Ready) is a hardware handshake for use with a modem eliminator cable or modem. The RS232 handshake lines associated with serial ports A and B are monitored for control signals that suspend or allow transmission of characters. This arrangement works well if you are connecting your Lisa to another Lisa.

If you get error message 647, the Transfer program failed to receive the appropriate handshake from the remote computer after a timeout. The session terminates. Before retrying, make sure that the characteristics settings for the Lisa are in agreement with those of the remote computer.

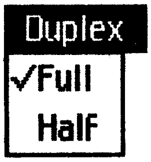


10.3.5 The Duplex Menu

This menu allows you to select Full or Half duplex. Most remote computer connections are made using full duplex mode. The default is Full duplex. Full duplex transmission allows information to flow in both directions at once; both the Lisa and the remote computer can send and receive information simultaneously. Half duplex transmission allows information to flow in only one direction at a time; when the Lisa is sending, the remote computer can only receive, and vice versa.

In full duplex mode, the characters you type are sent but not displayed on the Lisa screen. (Characters received from the remote computer are displayed.) Normally in full duplex mode the remote computer sends back the characters you type so that you can see them on the screen; this is known as *echoing*.

In half duplex mode, the characters you type are both sent and displayed. Normally the remote computer does not echo in half duplex mode. If it does, you'll see two characters for each one you type.



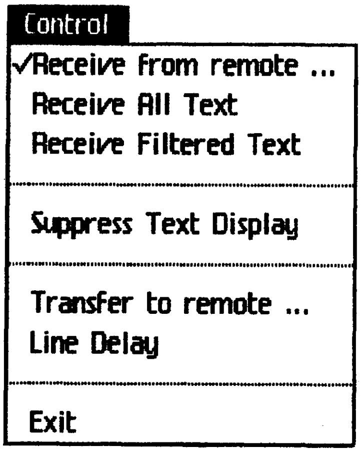
10.4 Using the Transfer Program

Start the Transfer program by typing *T* in response to the Workshop command line. Then use the characteristics menus described in Section 10.3 to configure the Lisa so that it matches the remote computer you want to communicate with. The Transfer program begins in *terminal emulation* mode: Whatever you type on one computer is received on the other computer. To send from a file or receive to a file, select the appropriate options from the Control menu.

10.4.1 The Control Menu

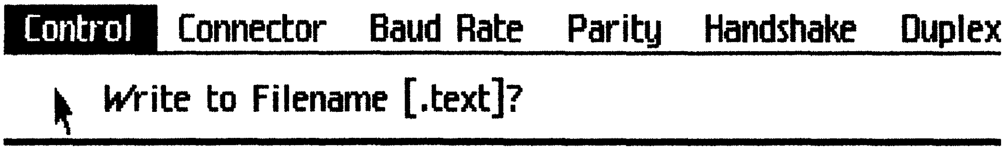
The Control Menu allows you to control transmission by receiving, sending, or exiting from the Transfer program. Some of the menu items in the Control menu are *toggles*: selecting a toggle item turns it on, selecting the same item again turns it off, and so forth. The toggle items are Receive From Remote, Suppress Text Display, and Transfer to Remote. The item is on if it has a checkmark next to it.

The Suppress Text Display option applies to either sending or receiving.



10.4.1.1 Receiving Text

When the Transfer program starts, you are able to receive text from the remote computer in terminal emulation mode. The first item on the Control menu, Receive From Remote, lets you specify a file in which to save the text sent by the remote computer. When you select Receive From Remote, the following message appears.



Write to Filename [.text]?

Type the name of the file you want to save the transmitted data in. It must be a text file.

Two options are associated with Receive From Remote. You must choose one of them; Receive Filtered Text is the default.

Receive All Text lets you store the transmitted data in the receiving file exactly as they are received, including control characters.

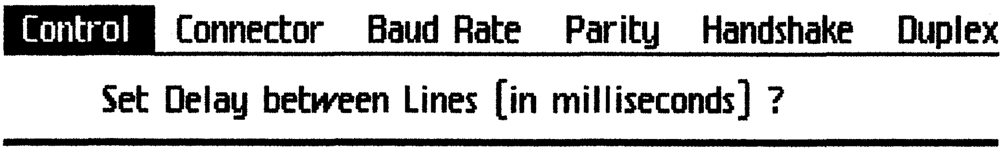
Receive Filtered Text does not save control characters in the receiving file. This option changes [RETURN] to [NEWLINE] and replaces [TAB] characters with the appropriate number of spaces. All other control characters are discarded.

To stop storing text in the file, toggle Receive From Remote to close the file. You can then read the file using the Workshop Editor or any program that reads standard text files. The Transfer program does not insert a [RETURN] at the end of the file, so if your file is a program file or other file that must end with [RETURN], use the Editor to insert one.

10.4.1.2 Sending Text

The Transfer to Remote menu item lets you send data directly from a text file rather than typing it at the keyboard. When you toggle Transfer to Remote to begin sending, you are prompted for the name of the text file.

The Line Delay option is associated with Transfer to Remote. When you select this menu item, you are prompted for the number of milliseconds the Transfer program will wait before sending each line of text. The default is zero. Specify a line delay when you are transmitting to a remote computer that is losing data because it cannot keep up with full speed transmission.



Set Delay between Lines [in milliseconds] ?

10.4.1.3 Suppressing Text Display

The Baud Rate menu lets you select a maximum transfer speed. However, the actual transfer rate may be slower because of the processing time required to display the text as it is sent or received. Suppress Text Display is a Control menu item that may be used with either Receive From Remote or Transfer to Remote. You can toggle it on or off at any time. When Suppress Text Display is selected, text that is received or sent is not displayed on the Lisa screen, and the data transmission speed is usually improved.

10.4.1.4 Exiting from the Transfer Program

When you have completed your communications session, select Exit from the Control menu. The current characteristics settings are saved and you are returned to the Workshop command line.

You must explicitly log off if the remote computer has a logoff procedure. If you choose Exit without logging off, neither the session nor the telephone connection is automatically terminated. When you return to the Transfer program, the session will still be active and you can proceed as if you'd never exited.

10.4.2 Transmitting Special Characters

By using special keys, you can type standard terminal control characters. To transmit a control character from the keyboard, hold down the Apple key and type a character, as shown in Table 10-1.

You can also send international, mathematical, and scientific symbols and other characters from the Lisa's extended character set (see Appendix B, Lisa Extended Character Set) if the remote computer is a Lisa. The extended character set uses the eighth (parity) bit as part of the character identity, so both Lisas must operate with parity set to None; if parity checking is on, the parity bit will be stripped and the character will be received as an ASCII character.

To transmit a character from the extended character set, hold down the Option key (or the Option key together with the Shift key) and type a character.

Table 10-1
Transmitting Special Characters from the Keyboard

Keyboard	*Transmits*
-backspace	DEL
clear	ESC
ENTER (alpha keyboard)	BREAK (233ms)
ENTER (numeric keypad)	RETURN
arrow keys	their symbols
-Q	XON
-S	XOFF
-character	other control characters
Option-character Option-Shift-character	Extended Character Set (see Appendix B)

Chapter 11
The Utilities

The Utilities

The Utilities are general-purpose programs that run in the Workshop environment. To run a utility program, use the Workshop Run command. For example, to run the Copy utility, type

RCOPY

from the Workshop command line. You can also run a utility program from an exec file.

The Utilities are arranged alphabetically in this chapter. Each utility program is documented as follows:

Synopsis	Tells briefly what the program does.
Dialog	Lists the program prompts and tells how to respond to them.
Description	Gives details on input, output, and processing.
Notes	Brings special information to your attention.

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, "File to Change", asks for the unlinked object file you want to change. To exit from the ChangeSeg utility at this point, type <CLEAR> <RETURN>.

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 CharCount

Synopsis

CharCount counts the number of characters in its input.

Dialog

```
Parameter(s) [? for help]:
```

The format for the parameters is: `<stdin >stdout`

Description

CharCount counts the number of characters in its input (StdIn), and writes the total to its output (StdOut). The defaults for both StdIn and StdOut are the console. If the input is from the console, use **-C** to indicate the end of file.

All characters are counted, including RETURN and DLE characters.

11.4 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 segment 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.1Unit 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

11.5 Compare

Synopsis

Compare compares the lines of two text files and prints out all their differences. Options let you compress blanks, delete trailing blanks, ignore case, set the number of lines that are allowed to mismatch and the number that must be equal to be considered a match, and control the format of the display.

Dialog

```
Parameter(s) [? for help]:
File #1: [.TEXT]
File #2: [.TEXT]
Options [? for help]:
Maximum stack depth:
Fixed/Minimum grouping factor:
Maximum display width:
```

Only the Parameters and Options prompts always appear; the other prompts don't appear if they are not needed or are specified as explicit parameters:

Parameter(s) [? for help]: file1 file2 [c1-c2] [depth] [g] [width] [>listing]

Where:

file1 and **file2**	are the input files
c1-c2	is a column range to compare (optional)
depth	is the stack depth for resynchronization (optional)
g	is the grouping factor (optional)
width	is the listing display width when the H option is used (optional)
>listing	specifies an alternate listing file (optional)

Typing **?** in response to a prompt displays information about the response needed.

Pressing CLEAR in response to a prompt terminates the program. After the prompts are processed, you can type ⌘-period to terminate.

Description

Compare reads in **file1** and **file2** in sequence, and compares them line for line. By default, entire lines are compared (up to a maximum of 132 characters), but you can specify that only the column range **c1-c2** be compared. (If **c1** is omitted, 1 is assumed; if **c2** omitted, 132 is assumed.) As soon as there is a mismatch, the mismatched lines are stored in two stacks, one stack for each file. Lines are then read alternately starting from the next input line in the second file until a match is found to put the files back in synchronization. The optional **depth** parameter specifies the maximum stack depth, that is, how far out of synchronization the files should get before it is no longer worth comparing them. Values allowed are 1 to 1000; the default is 1000.

A match is defined according to a grouping factor, G. G is the number of consecutive lines that must be the same to be considered matched. If the value of G is too small, the files may be put back into synchronization at the wrong place. The default value for G is dynamic, defined by the formula:

$$G = \mathrm{Trunc}(2.0 * \mathrm{LOG}_{10}(M) + 2.0)$$

where M is the number of lines saved in each stack so far. This means more lines must be the same after larger mismatches than after small mismatches before the two files resynchronize. Using the above formula, the following table shows the dynamic grouping factor as a function of the number of mismatched lines:

M			**G**
Number of	Mismatched Lines		Dynamic Grouping Factor
1	to	3	2
4	to	9	3
10	to	31	4
32	to	99	5
100	to	315	6
316	to	999	7
1000	to	3161	8
3162	to	9999	9

You can optionally set the lower limit on G with the **g** parameter, instead of using the values in this table, but it must be at least 2, because the formula is always applied. The default value for **g** is 2.

A second match option allows the grouping factor, G, to be fixed as a constant. If this S (static) option is used, the **g** parameter specifies a fixed grouping factor. Values allowed are 1 to 1000, but if G is too small, the files may be put into synchronization at undesirable points; try the dynamic grouping factor first.

There is a limit to how far out of synchronization the two files can get before it is no longer worth comparing them. For the dynamic G option, the limit on the number of mismatched lines is preset to 1000, but a lower limit may be chosen. For the static G option, the limit is always explicitly selected (the static value of G is also selected). Typical values for the static G are 1 to 5 and the number of mismatched lines about 10 to 50.

After a match has been found, the mismatched lines before the group of G matching lines are displayed. You can display the lines from the two files side by side, using the H (horizontal) option, specifying the width of the output listing display with the **width** parameter. By changing **width** you change the number of characters displayed in each portion. Values allowed are from 70 (which shows only 27 characters from each file; 15 characters are reserved for line number information) to 132 (which shows 58 characters).

Normally the output is displayed on the console. It may be redirected by entering **>listing** in the parameters line, where "listing" is a filename.

Any size files may be compared, as long as they don't get too far out of synchronization (line numbers are only displayed to four places, so the file size should be kept under 10000).

Options
The following options are available. Specify the options by listing them in response to the Options prompt.

B Delete trailing blanks and treat runs of blanks as one blank.

C Ignore case differences (convert all lines to lowercase).

D Dynamic grouping; prompt for the **depth** and the minimum **g** if they are not given as parameters. Dynamic grouping is done by default, you don't need this option unless you want to set the values of **depth** and the minimum **g** and didn't set them in the parameters line. The default values for **depth** and minimum **g** are 1000 and 2, respectively.

H Horizontal form of display. Only part of the mismatched lines from each file are displayed, side by side. How much of each line is displayed is controlled by the **width** parameter. (If H is not specified, entire lines are displayed, up to 132 characters, with the lines of the first file displayed before the lines of the second file for each group of mismatches.)

K Keep output file even if the input files are the same. The default is not to generate an output file when the files are equal.

S Static (fixed) grouping factor. If the **depth** and **g** values in the parameters line are missing or invalid, you will be prompted for them.

T Delete trailing blanks. (Does not compress runs of blanks; T is a subset of B.)

***** Ring the bell at completion of the execution.

Note: All comparison criteria that affect the individual lines *prior* to comparison (the column range, blank compression, trailing blanks, and case conversion) are applied to those lines before they are stacked. Thus when the lines are displayed, they are shown in their modified form.

Output
The following messages appear when mismatches are displayed. Lines are shown with their line numbers:

```
Non-matching lines
<both stacks are displayed>

Extra lines in 1st before <ln> in 2nd
<lines in file 1's stack are displayed>

Extra lines in 2nd before <ln> in 1st
<lines in file 1's stack are displayed>

Extra lines in 1st file
<lines in file 1's stack are displayed>

Extra lines in 2nd file
<lines in file 1's stack are displayed>
```

If, during resynchronization, an end-of-file condition occurs or the maximum stack depth is reached, then one of the above set of messages will appear followed by one of the following:

```
*** Nothing seems to match ***
*** Eof on both files ***
*** Eof on file 1 ***
*** Eof on file 2 ***
```

If both files are in synchronization, and both reach their end-of-files at the same time, then the message,

```
*** Eof on both files at the same time ***
```

will appear if *any* mismatches occurred previously.

If the files match, then the following message is displayed:

```
*** Files match ***
```

All of these termination ("*** ... ***") messages are shown on the console even if the output listing has been redirected to some other file.

11.6 Concat

Synopsis

Concat copies a list of files into one file.

Dialog

```
Parameter(s) [? for help]:
```

The format for the parameters is: `filename [, ...] >stdout`

Description

Concat copies the list of input files to the output file, StdOut. The default for StdOut is the console. Concat accepts a list of file name parameters separated by commas or spaces (.TEXT extensions will be added), and copies them to the output file in the order they were specified in the list. If there is only one input file, Concat behaves exactly like Copy.

11.7 Copy

Synopsis

Copy copies all or part of its input to its output.

Dialog

```
Parameter(s) [? for help]:
```

The format for the parameters is: `<stdin >stdout` [LineRange, ...]

Description

Copy is used to copy its input (StdIn) to its output (StdOut). The defaults for both StdIn and StdOut are the console. If the input is from the console, use **-C** to indicate the end of file.

Copy's optional parameter, LineRange, specifies what portion of the input you want copied. LineRange is a list of ranges separated by commas or spaces. Each range is a single line number or pair of line numbers in the form line1-line2. When specifying more than one line, if line1 is omitted, the first line of the file is assumed; if line2 is omitted, the end of file is assumed.

11.8 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

11.9 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.

11.10 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

11.11 ErrTool

Synopsis

ErrTool is used to create files of numbered messages.

Dialog

```
Error Input File [.TEXT]
Error Output File [<input file>][.ERR]
Error Listing File [.TEXT]
```

Description

ErrTool lets you create compacted message files in which each message is associated with a number. The Standard Unit (StdUnit) in SULib provides calls (SUErrTest and SUGetErrText) for retrieving the message associated with a given number. In spite of its name and the names of the SU calls, ErrTool is not limited to use for error messages (although that is how it's principally used by the Workshop tools). The output of ErrTool is a specially formatted file with a directory at the start indicating the offsets of the messages in the file.

ErrTool input consists of a text file with lines beginning with an integer (positive or negative), followed by a space and the message. ErrTool assumes that the message is everything between the space following the message number and the end of the line (no multi-line messages are allowed). The ErrTool input need not be ordered with respect to message number. Duplicate error numbers will be flagged as a fatal error. The listing file will contain a sorted list of the messages.

11.12 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.

11.13 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 while scanning lines that do no match.
+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.

11.14 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 files as input. When you have entered all the object files, press [RETURN] in response to the input file request.

GXRef accepts a maximum of 4095 procedure names.

11.15 IUManager

Synopsis

The IUManager utility is used to manage the directory of library files. You can add, delete, or change intrinsic units, segments, and files in the directory. To use the IUManager, you should be familiar with the way that units and segments are handled in Pascal on the Lisa.

Dialog

```
Input file [INTRINSIC.LIB]:
Output file [<input file>]:
```

INTRINSIC.LIB is the library directory that the system looks for at boot time. You can can edit INTRINSIC.LIB, or you can create and use your own library directories. (But be careful--don't change INTRINSIC.LIB unless you know what you're doing, or your system may not boot.)

Description

The IUManager has three modes, which do the following:

UNITS	Add, delete, or change intrinsic units. An intrinsic unit is a unit of Pascal code that can be accessed by different processes. There are two kind of intrinsic units--regular and shared. A regular intrinsic unit has a private global data area associated with it; shared intrinsic units share data as well as code.
SEGMENTS	Add, delete, or change segments. Units can be broken up into segments, so that interdependant parts of different units will be swapped in and out of memory at the same time. You can segment your code with either the $S Compiler option or the ChangeSeg utility.
FILES	Add, delete, or change library files. Units and segments are arranged in library files.

When you first enter the IUManager, you're in the FILES mode. To switch between modes, the following commands are available:

S(egments) Enter the SEGMENTS mode and display the segment table. Entries in the segment table have the following information:

SegName	The segment name
Seg#	The segment number
File#	The number of the file that the segment is in
FileLoc	The byte location of the segment in the file

Packed/ UnPacked	The number of packed or unpacked bytes in the segment
FileName	The name of the file that the segment is in

U(nits) Enter the UNITS mode and display the unit table. Entries in the unit table have the following information:

UnitName	The unit name
Unit#	The unit number
File#	The number of the file that the unit is in
Type	The type of unit: Intrinsic or Shared Intrinsic
DataSize	The number of bytes of global data (Shared Intrinsic units only)

F(iles) Enter the FILES mode and display the file table. Entries in the file table have the following information:

File	The file number
FileName	The file name

Other than the S(egments), U(nits), and F(iles) commands, the commands available in all three modes are the same:

C(hange) Change an entry in the currently selected table. You will be asked for the file, unit, or segment number, and prompted for changes in each field. If you enter an unused number, the Change command works just like the Add command.

A(dd) Add a new entry in the currently selected table. You will be asked for the file, unit, or segment number, and prompted for each field. If you enter a number already associated with an entry, the Add command works just like the Change command. The default entry number is the first unused number in the table. If you add a unit or segment and specify a file name that has not been used, a new file will be created with the next available file number.

D(elete) Delete an entry from the currently selected table. You are prompted for the file, unit, or segment name or number. If you try to delete a file that is used by the segment table or unit table, you will get a warning, and the file will not be removed. If you try to delete a segment that is used by the system table as a Public Interface segment, the segment will not be removed.

L(ist)	List the entries in the currently selected table.
Q(uit)	Quit the IUManager and rewrite the directory.
?	Typing **?** from the main command line displays the alternate command line, with the following commands:
I(nstall)	Install a library in the directory. This stores the segment and unit tables from the linked object file. The Install command puts you in the FILES mode if you're not already, displays the file table, and prompts you for the file name or number to install.
V(erify)	Verify that the information in the linked object file is consistent with the directory. You are prompted for the name of the file to verify.
P(rint)	Print all three tables. (You can send the tables to a .TEXT file instead of **-printer** if you want to look at them in the Editor.)

? Typing **?** from the alternate command line returns you to the main command line.

11.16 LineCount

Synopsis

LineCount counts the number of lines in its input.

Dialog

```
Parameter(s) [? for help]:
```

The format for the parameters is: `<stdin >stdout`

Description

LineCount counts the number of lines in its input (StdIn), and writes the total to its output (StdOut). The defaults for both StdIn and StdOut are the console. If the input is from the console, use **-C** to indicate the end of file.

11.17 LWCCount

Synopsis

LWCCount counts the number of lines, words, and characters in its input.

Dialog

```
Parameter(s) [? for help]:
```

The format for the parameters is: `<stdin >stdout`

Description

LWCCount counts the number of lines, words, and characters in its input (StdIn), and writes the totals as three lines to its output (StdOut):

1. Number of lines.
2. Number of words.
3. Number of characters.

The defaults for both StdIn and StdOut are the console. If the input is from the console, use ⌘-C to indicate the end of file.

A *word* is considered any sequence of characters not containing a blank or any control characters (e.g., RETURN or DLE). The character count includes RETURNs and DLEs.

11.18 MacCom

Synopsis

MacCom lets you move files back and forth between the Lisa and the Macintosh, using Macintosh-format diskettes. You can also perform other operations on Macintosh diskettes on your Lisa: initialize diskettes, delete files, set Finder information, and write boot blocks.

Dialog

The MacCom command line is:

```
{3.0} MacCom: Delete, Eject, Help, Init, Lisa->Mac, Mac->Lisa, Names, Quit, ?
```

Typing ? shows you the second half of the line:

```
BootBlocks, FinderInfo, Confirm, RemoveSlashes
```

To execute any command, type the first letter. The commands are described below. Other dialog, such as file name prompts, is self-explanatory.

Description

You can use the MacCom commands in whatever order you like, then Eject the diskette when you're finished. Each command is independent of the others; the Macintosh directory gets written at the end of any command that changes it. The Delete, Lisa->Mac, and Mac->Lisa commands support the ?, =, and **$** wildcard characters. You can escape from most prompts by pressing [CLEAR]. To abort the operation when you are being prompted for Yes/No answers using ?, type **⌘**-period followed by *N* for no. To abort when you're using =, just type **⌘**-period. When prompted for a file name, **<** followed by the name of a .TEXT file reads a list of names input from the file. This can be used recursively.

Note that you can use MacCom to back up Macintosh diskettes: first copy all the files on the diskette to the Lisa with the Mac->Lisa command, using the wildcard sequence **'=,temp/$'**; then initialize the second diskette (if you need to) using Init; then copy all the files from the Lisa onto the second diskette using Lisa->Mac with the wildcard sequence **'temp/=,='**.

Delete	Deletes files on the Macintosh diskette.
Eject	Ejects the diskette--this is safe at any time.
Help	Tells you what each command does.
Init	First checks the diskette and warns you if it already contains a Macintosh or Lisa OS format volume. Init formats the disk, then adds Macintosh boot blocks and a directory. The file 'Mac.Boot' must be on your Workshop boot volume or prefix volume to correctly initialize a diskette.

Mac->Lisa	Copies a file, or files, from the diskette to any Lisa volume. The Finder information for a Macintosh file is saved on the Workshop in a 1-block file using the Macintosh file name with a .MFEN (for Mac Finder ENtry) extension. A Macintosh resource file is saved with a .RSRC extension. If the Macintosh file has both a resource fork and a data fork, two separate files will be created on the Lisa volume; the data fork will have the Macintosh file name, and the resource fork will have the same name with a .RSRC extension. If a file has the same name as an existing file on the Lisa volume, you will be asked if you want the existing file overwritten. The dates on the Macintosh files are converted to the Lisa's date format.
Lisa->Mac	Copies a file, or files, from any Lisa volume to the Macintosh diskette. If a file you are sending to the Macintosh did not come from a Macintosh originally (if there isn't a .MFEN file for it), and you are not overwriting a file already on the Macintosh volume, default Finder information will be set for that file. The default values are '????' for Type and Creator, and the Bundle bit not set. Otherwise, the Finder information will be inherited from the .MFEN file or from the existing file. If you want to enter different Finder information for a file, use the FinderInfo command and you will be prompted for Finder information. Files with a .RSRC extension are assumed to be resource files; the extension is removed as the file is copied to the Macintosh, and the file is set up as a resource in the Macintosh directory. If a file has the same name as an existing file on the diskette, the file already on the diskette will be overwritten. If you want a chance to prevent such a loss of existing files, set Confirm (below) to True. The dates on the Lisa files are converted to the Macintosh's date format.
Names	List the names and directory information for all the files on a Macintosh diskette.
BootBlocks	Write the boot blocks on a Macintosh diskette. This includes writing the boot blocks from your own file.
FinderInfo	Set the Finder information for all files that you copy from a Lisa volume to the Macintosh diskette. You can set the Type (default '????'), the Creator (default '????'), and the Bundle bit (default not set).

Confirm	Ask for confirmation before overwriting old versions of files on the Macintosh diskette. The default is False; old files are automatically overwritten when you copy a new file with the same name as an old file.
RemoveSlashes	Remove Workshop prefixes (denoted by the / character) from file names as the files are moved from the Workshop to a Macintosh diskette. The default is False; prefixes are left on.

Notes
MacCom doesn't set the 12 tag bytes on each block while creating or accessing Macintosh diskettes. Because of this, *use the Macintosh to create a master disk for any product you are going to ship.* The tag bytes will be used in the future by a Scavenger to rebuild damaged Macintosh disks.

MacCom assumes that the diskette is in the internal Lisa diskette drive; no external drives are supported.

If you have been using a pre-3.0 version of MacCom, note that the default value for 'Type' in the Finder information settings has changed. Previous versions of MacCom had a default type of 'APPL'; the default is now '????'. You need to change your Examples/Exec command file to set the type to APPL when moving an application to the Macintosh. To do this, replace this line in Examples/Exec:

```
{set type to APPL}
```

(which accepted the default type) with the line:

```
APPL{set type to APPL}
```

Previous versions of MacCom automatically prompted you for the Finder information when copying a new file to a Macintosh volume. Now you are only prompted if you specify F for FinderInfo, otherwise the default values are used.

11.19 Pasmat

Synopsis

Pasmat reformats Pascal source code into a standard format that you can control.

Dialog

```
Parameter(s) [? for help]:
Input file: [.TEXT]
Output file: [<input file>] [.TEXT]
Correct /pattern/replacement/:
Options [? for help]:
Listing file: [-console] [.TEXT]
Rename file: [.TEXT]
Maximum line width: [w]
Indenting (tab) value: [t]
```

Only the Parameters and Options prompts always appear; the other prompts don't appear if they are not needed or are specified as explicit parameters:

Parameter(s) [? for help]: input output [rename] [width] [tab] [>listing]

Parameters can be separated by spaces or commas.

Typing **?** in response to a prompt displays information about the response needed.

Pressing [CLEAR] in response to a prompt terminates the program. After the prompts are processed, you can type ⌘-period to terminate; the output file will not be generated.

Description

Pasmat reformats Pascal source code into a standard format suitable for printouts or for compilation. Pasmat options let you:

- Convert a program to uniform case conventions.
- Indent a program to show its logical structure and adjust lines to fit into a specified line length.
- Change the comment delimiters (* *) to { }.
- Remove the break character (_) from identifiers, rename identifiers, or change their case.
- Remove all nonprinting characters from the source (except in strings).
- Format **include** files named in Pascal **include** directives.

Pasmat specifications are made through options or through special formatter directives, which resemble Compiler directives, and are inserted into the source file as Pascal comments.

Pasmat accepts full programs, external procedures, blocks, and groups of statements. A syntactically incorrect program usually causes it to abort. If this happens, the generated output will contain the formatted source up to the point of the error, unless the output file and input file are the same, in which case no output file is generated.

The input and output files are required parameters. The output file may specify pattern and replacement strings in the form **/pattern/replacement/** (single or double quotes may be used instead of slashes). This form causes the **!** option to appear when you are prompted for the options, implying that you want to process **include** Compiler directives and generate a *set* of formatted output files with the same **include** structure as the input. See the discussion of the ! option (below) for further details.

The rename, listing, width, and tab parameters are all optional. The rename parameter is also a filename, but it should be specified only if the M option is specified (see below). The width and tab parameters specify the initial values of the output line width and indenting tab value (i.e., the initial O and T directive values). Unless told otherwise, the default output width is 80 and the default indenting tab value is 3.

If you want to see of listing of the output, specify the >listing parameter. This implies the S option (see below). (If you specify the S option but didn't use the >listing parameter, you are prompted for a listing file.) The listing filename is preceded by a > character, which indicates to Pasmat that its standard output ("StdOut") is to be redirected.

Formatting Details

Comments: The following rules govern Pasmat's formatting of comments.

- A comment that stands alone on a single line is passed to the output unaltered. Its left end is set to the current indentation level, so that it's aligned with the statements before and/or after it. If it's too long to fit with this alignment, it is placed on the page as far right as it will go.
- A comment that begins as the first thing on a line and continues on another line is passed to the output unaltered, including its indentation. This type of comment is assumed to contain text formatted by the user.
- If a comment covered by one of the above rules doesn't fit within the defined output line length, the output line is extended as necessary to accommodate it, and a message is printed at the end of the formatting.
- A comment that is not the first thing on a line is formatted in with the rest of the code. Words within it are moved to the next line to make it fit, so nothing that has a fixed format should be used in such a comment. The comment is broken only at blanks, and if there is no

way to break a comment and still fit the output within the output line length, the line is extended as necessary, and a message is written at the end of the formatting. If no code follows a comment in the input line, then no code is placed after the comment in the output line. The J directive lets you force these comments to start in a specific column. This feature is useful for commenting declarations (see below).

- A comment that follows a statement on a line and begins with a specific character can be forced to start in a specific column. This feature is useful if you are making updates to a program and you want to show who made the update and when.

Statement Bunching: Statement bunching refers to the way Pasmat aligns a statement with respect to some component of another statement that precedes it. There are three cases:

- A statement following a **CASE** label.
- A statement following a **THEN** or **ELSE**.
- A statement following **FOR**, **WHILE**, or **WITH**.

Pasmat allows some control over how these statements are aligned.

Note: The following discussions describe how a statement can be aligned relative to its "lead-in" statement, whether it's indented after or on the same line as the lead-in. Therefore, *statement* in these cases refers to a simple statement. Compound statements are usually indented starting on a new line (except for their **BEGIN**'s as controlled by the C directive).

Bunching with CASE labels: The default formatting rule for a **CASE** statement is to place the selected statements on the same line as the case label(s). The A directive lets you specify that the statement appear on a separate line from the case label. The @ directive lets you control how far the statements following the case label are indented.

Bunching of IF statements: The default is to place the controlled statements on separate lines. The B directive tells Pasmat to place the controlled statements on the same line as the **THEN** or **ELSE**.

In the special case of an **ELSE IF**, the default is to put the **IF** on the same line as the **ELSE**. The Q directive lets you specify that the **IF** appear on the next line, indented after the **ELSE**.

Bunching with FOR, WHILE, and WITH: The default is to place the controlled statement on the same line if it fits. Otherwise, it is indented on the next line. The H directive lets you specify that the statement always appear on the next line.

Note: the H directive also affects the **IF** statement. With **IF**-bunching off (B- directive), and the H directive off (H-), the controlled statement would normally appear on a separate line. If there is no **ELSE**, then the H directive applies to the **IF** statement just like **FOR**, **WHILE**, and **WITH**; that is, the controlled statement is placed on the same line as the **IF** if it fits.

Tables: Many Pascal programs contain long lists of initialization statements, or of constant declarations that are logically a single action or declaration. You can fit these into as few lines as possible using the G (grouping) directive. If this is used (G=i form), tab stops are set up on the line, and successive statements or constant declarations are aligned to these tab stops instead of beginning on new lines.

Structured statements, which are normally formatted on more than one line, are not affected by the G directive. However, assignment and call statements may be grouped with the end of the structured statement (e.g., following an **END** statement). A special form of grouping directive is provided specifically for assignment and call statements.

Assignment and Call Statement Grouping: As described below, the grouping directive to format tables is G=i, where i is the maximum number of statements per line. This sets up tab stops to align i statements or constant declarations. However, for assignment and call statements, it is not always known how many statements will fit on a line. Even if it is, these statements aligned on tab stops may insert too much white space and produce an aesthetically unpleasing result. A special form of grouping can be specified using G+, which affects only assignment and call statements. They are grouped so that as many as possible fit on a line without exceeding the line length. They are never grouped on a line ending a structured statement, so the problem arising with the G=i form of grouping cannot happen.

You probably won't want to group all assignment and call statements together everywhere in your program. The preset option is G- to format assignment and call statements one per line. Bracket the sections of your code that you want grouped with G+ and G- directives.

If you are formatting a program that is already partially formatted and has sections of code grouped, you may not want it reformatted using G+ and G-. The "smart" grouping option (#+) lets you specify that if more than one assignment or call statement are on the same input line, and they don't exceed the output line width, they are kept grouped in the output.

Note: if G=i is in effect with i>1, it has precedence over the effect of G+ and #+. Thus G+ or #+ may be enabled and G=i still be used (except for G=1).

Declarations: If you want to align declarations so that the objects of the identifiers (constants or types) all start at a particular column, or align comments explaining the identifiers, use the J directive. It allows you to specify the number of columns to reserve for the identifiers and in which column the explaining comment is to begin.

Directives

Directives are specified by special comments included in the Pascal source code. These comments have the form:

```
{[directives] optional text}
```

The directives themselves are either switches, with the format

<character>+

or

<character>–

or are numeric directives with the format

<character>=<number>

or a character directive, which specifies a special character, "c", with the format

<character>c

For the J directive only, the numeric directive can also have the format

<character>=<number>c/<number>cc/<number>c

where the c's are characters and either or both of the first two entries can be omitted (but not the slashes separating them, e.g., //<number>c).

Multiple directives are separated by commas. Spaces within a directive are not allowed. For example:

{[b+, o=72, t=4, r–]}

sets the switch "b" on, "r" off, and sets the numeric directives "o" to 72 and "t" to 4. Case is ignored in directives.

The following directives are recognized:

A Place a statement following a **CASE** label on the same line if it fits.

Default A+

B Place a statement following **THEN** or **ELSE** on the same line if it fits.

Default B-

C Place **BEGIN** on same line as its introductory keyword. If C+ is specified, then K- (the default) should be used.

Default C-

D Replace the comment delimiters (* and *) with { and }.

Default D+

E Capitalize the first (or only) letter of identifiers and the first letter following a break or underscore character (_). Retain the underscore character. E overrides the L and W directives. See also the P (portability) option.

Default E-

F Turn formatting on or off. F goes into effect immediately following the comment in which it is placed. This is useful for saving hand-formatted portions of a program.

Default F+

G Group statements (i per line). G is specified either as a switch (G+ or G-) or as a numeric directive (G=i). For G=i, the space from the current indentation level to the end of the line is divided into i fieldc, and successive statements put on the boundaries of successive field. A statement may take more than one field, in which case the next statement again goes on the boundary of the next field. This is similar to using tabs on a typewriter. Any statement that requires more than one line may produce strange results on subsequent statements. The G=i form affects constant declarations and statements. By specifying the G+ form, only assignment and call statements are grouped together if they fit on a line. G+ only has affect if G=1 is set.

Default G-, G=1

H Bunch a single statement on the same line as **FOR**, **WHILE**, or **WITH** if it fits. Otherwise indent it on the next line. This also applies to **IF** (without an **ELSE**) if the B directive is off (B-).

Default H+

I Process **include {$I filename}** *Compiler* (not Pasmat) directives. Pasmat provides three different ways to process **include** files. The third way is recommended.

- Process all the **includes** in the input to produce a single output file. To do this, use the I+ Pasmat directive (or option). As each **include** Compiler directive is encountered, it is output on the line before the output of the included source. However, to avoid reprocessing of this directive by the Compiler (assuming the output is to be eventually compiled), the "I" in the directive is *not* output.
- Treat each **include** file separately. Each file is given individualy to Pasmat to format. By placing an I=n Pasmat directive at the start of each source input file, you can specify the initial indenting level for the file. Indenting for I=n starts at column n*t, that is, the specified level times the indenting tab value (see T directive). (To determine the indenting level for each **include** file, you can use the ProcNames utility, which displays procedure and function names and their level information.) Note that since individual **include** files need not

represent syntactically complete Pascal constructs (for example, an **include** file can contain a procedure with many nested inner procedures, but without the body of the outer procedure), Pasmat may report a syntax error. If this happens, check the output to see if the entire **include** file was processed.

- Process the entire source as in the first method above, but instead of generating a single source with the **include** directives removed, generate as many output files as there are input (**include**) files. The result is a set of formatted files with the same **include** structure as the input. All the **include** directives are output and edited to reflect the new filenames (which may be the original input and **include** filenames, yielding a facility that effectively reformats in place). This method of processing **includes** is indicated by specifying the **!** *option* when Pasmat is invoked. For further details, refer to the discussion of ! in the Options section.

Default I-, I=0 (**include** not processed)

J Special alignment of declarations and comments. The general format is J=<width>±/<col1>sd/<col2>c.

<width>± specifies that <width> columns are to be reserved for all following **CONST**, **TYPE** or **VAR** identifiers (you can also control the alignment of the colons in **VAR** declarations within the width by using the : option). The optional sign following the <width> indicates whether to apply the <width> to record field lists (if + is used or the sign is omitted) or to apply it to just the declared variables themselves (if – is specified).

<col1>sd specifies what column a comment following a statement on the same line is to start in. Note that <width> is a width specification, and <col1> is a column specification. <col1> allows you to align all comments in declarations. *All* comments following statements are aligned (when the comment is the last thing on the same line as the statement), unless you specify **s** or **d** following <col1>. (Case is ignored, and the letters may be in either order.) If **s** is specified, <col1> is applied only to statements and not to declarations. If **d** is specified, <col1> is only applied to declarations. Omitting both **s** and **d** is the same as specifying both; <col1> is applied to all comments following statements if the comment is the last thing on the line.

<col2>c specifies a starting column for comments, as <col1> does, but only affects comments that have the trigger character **c** as the first comment character.

If <width> is omitted, its previous value remains unchanged; the slash in front of <col1> is required. If <col1> is omitted, the previous value remains unchanged; the slash in front of it is optional unless <col2> is specified, in which case both slashes are required.

For constant declarations, the G=i (i>1) directive overrides <width>. Comments should then not be used for these statements. The <width> and <col1> values are ignored for a line if they cannot be used because an identifier or its declarative information are too wide. A value of 0 for <width>, <col1> or <col2> disables the corresponding alignment.

Default J=0/0/0

K Indent statements between **BEGIN/END** pairs. Normally the statements are indented to the same level as the **BEGIN/END** pair. The C directive determines the actual placement of the **BEGIN**. Normally the **BEGIN** appears on a separate line unless C+ is used. K- should be used if C+ is specified.

Default K-

L The case of reserved words and identifiers is to be a literal copy of the input. L overrides the W directive and is disabled by the P directive. The R directive overrides L for reserved words.

Default L-

N Group formal procedure parameters. This is similar to the G+ option, but only for formal parameters of procedure and function declarations. Normally these appear one per line.

Default N-

O This is a numeric directive (i.e., O=w) that specifies the output line width. The maximum value allowed is 132 characters. If a particular token will not fit in this width, that line is lengthened to fit it, and a message is displayed at the end of formatting.

Default O=80, or 3rd parameter, or 4th parameter with M option

P Sets portability mode formatting, which removes the underscore character (_) from identifiers. The first letter of each identifier and the first letter following each underscore character are capitalized while the remaining characters are in lower case. This overrides the L and W directives. The case of reserved words is set with the R directive.

Default P-

Q If an **IF** follows an **ELSE**, do not treat the **IF** specially. It is indented on the next line after the **ELSE**.

Default Q-

R Output all reserved words in upper case, otherwise (R-) output in lower case.

Default R-

T Specifies the amount of tab for each indentation level. This is a numeric directive (T=n). Statements that continue on successive lines are additionally indented by half this amount.

Default T=3, or 4th parameter, or 5th parameter with M option

U Case conventions for each identifer are based on its first occurrence in the source. The first occurrence of each identifier is left as is; all subsequent occurrences are made to look exactly like the first occurrence. U overrides the L and W options, but the E and P options can still be used.

Default U-

V Align an **IF** statement so that the **THEN** is indented on the next line after the line containing the **IF**. The **ELSE** is aligned with the **THEN**.

Default V-

W Convert identifiers to upper case, otherwise convert to lower case. W is overridden by the L, P, and E directives.

Default W-

X Suppress space around the arithmetic operators **+**, **-**, *****, and **/**, and the relational operators =, **<>**, **<**, <=, **>**, and **>=**. Normally, one space is placed on each side of these operators. X has no effect on the = used in **CONST** and **TYPE** declarations.

Default X-

Y Suppress space around the assignment operator ":=".

Default Y-

Z Suppress space after commas.

Default Z-

@ Controls **CASE** statement tags (labels). @ is specified either as a switch (@+ or @-), or as a numeric directive (@=i). In its @=i form, i indicates that the statements associated with the case tag are to start i columns after the start of the the case tag. (This is similar to the J=<width>/<col1>/<col2>c directive where <width> indicates how much space to reserve for an identifer being declared.) i indicates how much space to reserve for the case tag(s). If @=0 (the default), statements following a case tag are indented (using the current indenting tab value) on the line following the the tag. If @=1, the width of the first tag plus 2 (for the tag's colon and following space) is used to determine the space to reserve for all following tags in that case statement. This means you should put your longest case tag first. For @=i (i>1), i spaces are reserved for the case tags. If the tag is too wide for the specified width, then the statements that follow are placed on the following line, indented i spaces.

@+ and @- specify what to do with a *list* of tags that don't fit into the specified width. @+ indicates that a tag that is part of a list is to be put on the next line if it would exceed the i width. @- indicates that as many tags as possible are to be kept together on the same line. If the resulting list is longer than i, the statements are placed on the following line indented by i.

Default @-, @=0

: Positioning of colons in aligned **VAR** declarations. The reserved width for identifiers in declarations is controlled by the J directive's <width> parameter. In **VAR** declarations you have the choice of allowing the colons to immediately follow their identifiers by specifying :- or to align the colons at the right end of the reserved width by specifying :+.

Default :-

"Smart" grouping option. If #+ is specified, assignment and call statements that were grouped together on the same line in the input are grouped together on the same line in the output if they don't exceed the output line width.

Default **#**-

Options
Most of the options change the initial default settings of the directives described above. Options are specified by listing the letters (without the + or -) in response to the options prompt, or in a special options file (described at the end of this section).

A Set A- to disable **CASE** label bunching.

B Set B+ to enable **IF** bunching.

C Set C+ for placement of **BEGIN** on same line as previous word.

D Set D- to disable the replacement of (* *) with { } comment delimiters.

E Set E+ to capitalize identifiers.

F Set F- to disable formatting.

G Set G+ to group assignment and call statements.

H Set H- to disable **FOR**, **WHILE**, and **WITH** bunching.

I Set I+ to process Compiler **includes**.

K Set K+ to indent statements between **BEGIN/END** pairs.

L Set L+ for literal copy of reserved words and identifiers.

M Rename identifiers. This option requires that the third Pasmat parameter specify a file containing a list of identifiers and their corresponding new names. Each line in this file contains two identifiers of up to 32 characters each. The first is the identifier to be renamed in the input file. The second is the name that will replace all occurrences of the first identifier in the input when creating the output. There must be at least one space between the two identifiers. Leading and trailing spaces are optional. The case of the first identifier doesn't matter, but the second identifier must be specified exactly the way it is to appear in the output. The case of all identifiers not specified in the renaming file are subject to the other case options (E, L, U, and W). Reserved words cannot be renamed.

Instead of specifying the rename file as a parameter, if you have a file named **input.RENAME.TEXT** (where **input** is whatever the name of the input file is as specified on the Parameter(s) line), and the M option is not explicitly specified (along with its associated **rename** parameter), then the M option is implied and the implicit file is used.

N Set N+ to group formal parameters.

P Set P+ for portability mode.

Q Set Q+ not to treat **ELSE IF** sequence specially.

R Set R+ to show reserved words in upper case.

S Generate a display listing of the output. Unless you specified **>listing** (where **listing** is a filename) in the parameters line, you are prompted for the listing file. The listing file is ignored if either the output or the input file is specified as -CONSOLE.

U Rename all identifiers based on their first occurrence. The rename file has precedence over this option; if an identifier is specified in the rename file, the identifer's translation is based on the rename file rather than its first occurrence in the source.

V Set V+ to put **THEN** on a separate line.

W Set W+ to show identifiers in upper case.

X Set X+ to suppress space around operators.

Y Set Y+ to suppress space around :=.

Z Set Z+ to suppress space after commas.

: Set :+ to align colons in **VAR** declarations (only if a J Pasmat directive in the source specifies a <width>).

@ Set **@+** to force multiple **CASE** tags onto separate lines.

Set **#+** for "smart" grouping of assignment and call statements (grouped assignment and call statements on an input line appear grouped on output).

! Process **includes** and generate a set of output files with the same include structure as the input. The output file names are generated by editing the input (**include**) file names according to pattern and replacement strings. The **include** Compiler directives are also appropriately changed.

The pattern and replacement editing strings are specified by entering an **output** file name in the form /pattern/replacement/ (single or double quotes can be used in place of slashes). The pattern is a sequence of characters (ignoring case) that is to be looked for in the **input** pathname and each **include** pathname (the entire pathname is used). If the pattern is found, that sequence of characters is replaced by the replacement string. The result is a new pathname that becomes the name for an output file. Applying this editing operation to the input name and all **includes** produces a set of output files with the same structure as the input.

The following are examples editing operations and their associated effect:

"Prefix/'	Prefix each name with the sequence of characters "Prefix/".
/OldFile/NewFile/	Replace each name containing the string OldFile with the string NewFile.
///	Prefix each name with the null string--the output names are the same as the input names. The result is effectively an in-place formatting of the input.

If you specified an output file on the parameters line that looks like /pattern/replacement/ (where the slashes could be ' or " characters), Pasmat shows the ! on the options prompt. If you remove the ! from the options, Pasmat interprets the string as an output filename. Conversely, if you entered an invalid editing operation (e.g., you didn't use three slashes) but you intend to use the ! function, enter it on the options line. You are then prompted to correct the pattern and replacement. This prompt accepts as the delimiter whatever you use as the first character (e.g., #abc#def# specifies abc as a pattern and def as a replacement).

* Ring the bell at completion of the execution.

All options except M, S, !, and * have directive counterparts. If you use the embedded directives, you don't have to specify them as options each time you call Pasmat (though the Options prompt always appears).

In addition to explicity specifying options, you can create an options file called **PASMAT.OPTIONS.TEXT** that contains the options you want to use. Pasmat always looks for this file. Lines in the file contain a sequence of option characters grouped together on the same or separate lines. The lines may be commented using braces (**{ }**).

The options file may also specify the output line width (O=w), the indent tab value (T=n), and the CASE tag width (@=i). A typical options file might be:

```
n {group formal params on same line}
u {auto translation of id's based on 1st occurrence}
r {uppercase reserved words}
d {leave comment brackets alone}
# {smart grouping}
o=82 {output line width}
t=4 {indent tab value}
```

If Pasmat does find an options file, those options are shown on the options prompt line as if you typed them in. You can press [RETURN] to accept them, or change them by backspacing over them. If you specified the width and/or tab values, the specified values appear as the default values when the output width and tab prompts are given. If you specify the output width and tab values on the parameters line, those values take precedence and the associated prompts are not given.

Limitations and Errors

There are the following limitations on Pasmat.

- The maximum input line length is 132 characters.
- The maximum output length is 132 characters.
- Only syntactically correct programs, units, blocks, procedures, and statements are formatted. This must be taken into consideration when separate **include** files and *conditional Compiler directives* are to be formatted.
- The Pascal **include** directive should be the last thing on the input line if **includes** are to be processed. Pasmat does not act correctly if anything follows the **include** comment on the same line. **Includes** are processed to a maximum nesting depth of five. All **includes** not processed are summarized at the end of formatting. This assumes the I directive or option is in effect. Note that the "I" in the comment containing the directive is not output to avoid reprocessing when the output is eventually compiled.

The following errors are detected and noted:

- Any syntax error in the code causes the formatting to abort. An error message will give the input line number on which the error is detected. The output file will contain the output up to the point that the syntax error was detected. This output may help you determine what the error is. The error checking is not perfect; successful formatting is no guarantee that the program will compile.

- In general, premature end-of-file conditions in the input are not reported as errors, to accommodate formatting of individual **include** files that may only be program segments. There are cases, however, where the **include** file is a partial program that Pasmat interprets and reports as a syntax error. Check the output to see whether it really was a syntax error or just the premature end of file.
- There is a limit to the number of indentation levels that Pasmat can handle, and if this is exceeded, processing will abort. This probably will be rare.
- If a comment would require more than the maximum output length (132) to meet the rules given, processing will abort. This probably will be rare.
- If a token (identifier or string) is too long for the output line length, the length is extended for that line, and a summary is printed at the end of the formatting giving the places in the output where this occurred.
- If a comment line is extended according to rule 4 in the Comments section, a summary is printed at the end of the formatting giving the places in the output where this occurred.

11.20 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.

11.21 ProcNames

Synopsis

ProcNames lists all the procedure and function names in a Pascal program.

Dialog

```
Parameter(s) [? for help]:
Input file: [.TEXT]
Output file: [-console] [.TEXT]
Options [? for help]:
Intrinsic.Lib to use for this ProcNames:
```

The input and output prompts don't appear if they are specified as explicit parameters:

```
   Parameter(s) [? for help]:  input output
```

Typing **?** in response to a prompt displays information about the response needed.

Pressing [CLEAR] in response to a prompt terminates the program. After the prompts are processed, you can type ⌘-period to terminate.

Description

ProcNames takes a Pascal program as input and produces a listing of all its procedure and function names.

The input can be a *set* of files if you don't give the input file as a parameter, but let ProcNames prompt you for it; each file is processed separately. ProcNames continues prompting for input files until a null response is entered. The response can also be of the form **<filename**, where filename contains a list of file names. The default output file is the console. The output file can also be given by specifying **>filename** on the parameters line.

The names in the ProcNames listing are displayed indented to show their nesting level. The nesting level and line number information is also displayed.

ProcNames can be used in conjunction with the Pascal "pretty-printer" utility, Pasmat, when Pasmat is used to format separate **include** files. In this case, Pasmat requires that the initial indenting level be specified; this is the information provided by ProcNames.

The line number information displayed by ProcNames matches that produced by the Pascal cross-reference utility Xref (with *or* without USES being processed), so ProcNames can by used in conjunction with the Xref listing to show just the line numbers of every procedure or function header.

Options

The following options are available. You specify options by listing them in response to the options prompt.

C Do *not* process a used unit if the unit's name or its ($U) object filename (if a compiler $U- is in effect) is specified in the list of files to be processed. (This option has the same effect on the line numbering as does the C option in the Xref utility.)

M Macintosh mode. Ignore any $U± directives. ($U- is assumed.)

N Suppress all line number and level information in the output display. Only the procedure and function names are shown.

P Pasmat compatibility. The default is to list the procedure and function names as a function of their Compiler indenting level. However, for indenting purposes only, a special case is made of level 1 procedures in the IMPLEMENTATION section of a unit. Pasmat formats these procedures under the word IMPLEMENTATION, so they are indented as if they were level 2 procedures. If you intend to use the level information for Pasmat, specify the P option.

T Reset total line number count to 1 on each new file. The default is to number continuously through a list of files (agreeing with the listing produced by Xref).

U Process USES declarations. You need to process USES declarations if you want the line number information to agree with an Xref listing that also contains processed USES. The default is not to process the USES declarations, since they have no effect on the procedure name listing, only the associated line numbers. If you specify the N option to suppress line number information, the U option will be ignored.

$ Use a special intrinsic library directory; you will be prompted for the file name. The default is to use Intrinsic.Lib for intrinsic units. This option only has meaning if the U option is used.

***** Ring the bell at completion of the execution.

Example
The following shows the output produced by ProcNames (using source for ProcNames itself as the input).

```
Procedure/Function names for procnames/procnames.TEXT

   17   17    0   ProcNames          [ProcNames]
procnames/procnames.TEXT
  116  116    1     Stop
  131  131    1     NextChar
  173  173    1     ReadId
  202  202    1     Advance
  212  212    2       Opts
  218  218    3         GetSegName
  251  251    3         GetInclude
  288  288    3         GetUFname
```

```
 338  338    2        DoInclude
 396  396    1     Scan
 405  405    2        ScanId
 451  451    1     ProcDcl
 464  464    2        WriteProc
 524  524    2        ProcHdr
 562  562    2        ScanBody
 607  607    2        ScanINTERFACE
 650  650    2        ScanUSES
 661  661    3           Use
 673  673    4              OpenObjFile
 698  698    4              ProcessInterface
 719  719    4              FindUnit
 734  734    5                 NextByte
 756  756    5                 NextInt
 863  863    4              ReadInterface
 908  908    3           DupUse
 989  989    2        ScanMETHODS
1097 1097    1     Init
1108 1108    2        InitKeywords
1265 1265    1     Process1File

*** End ProcNames: 30 Procedures and Functions
```

The first two columnl are line number information; the third column is the level number. The first column shows the line number of a routine within the total source. The second column shows the line number within an **include** file (includes are always processed). As each **include** file changes, the name of the file from which input is being processed is shown along with the routine name on the first line after the change in source. Segment names (from Compiler $S directives) are similarly processed. These are shown enclosed in square brackets (a blank segment name is shown as "[<blank>]").

Limitations and Errors
Only syntactically correct programs are accepted by ProcNames. Conditional compilation Compiler directives are *not* processed.

11.22 RMaker

Synopsis

RMaker is used to create resource files for Macintosh applications.

Dialog

```
Input file [sysResDef][.TEXT]
```

Description

RMaker is the resource compiler, used to create resource files for Macintosh applications. It converts object files to a Macintosh executable form. The resource file created by RMaker lets the Macintosh Resource Manager know what resources (such as menus, icons, and fonts) your application uses.

The name of the RMaker output file must be specified on the first noncomment line of your RMaker input file.

Information on the format of RMaker's input file is currently in *Inside Macintosh,* Putting Together a Macintosh Application. The Macintosh Resource Manager is described in *Inside Macintosh,* The Resource Manager: A Programmer's Guide.

11.23 Search

Synopsis

Search copies all lines containing a specified pattern from its input to its output.

Dialog

```
Parameter(s) [? for help]
```

The format for the parameters is: `<stdin >stdout pattern`

Description

Search reads its input (StdIn) one line at a time, and writes to its output (StdOut) all lines that match the specified pattern. The defaults for both StdIn and StdOut are the console. If the input is from the console, use **⌘-C** to indicate the end of file.

The pattern is a concatenation of any of the following:

c	Literal character *c*.
?	Any character except [RETURN].
%	Beginning of line (only has meaning when first character of pattern).
$	End of line (only has meaning when last character of pattern).
[...]	Character class (any one of the bracketed characters).
[^...]	Negated character class (all but the bracketed characters).
*****	Closure (zero or more occurrences of the previous pattern) (has no meaning when first character of pattern).
~*c*	Literalized character (special symbol *c* taken as is, including ~~).
~n	[RETURN].

The special meanings for these symbols are lost when literalized with ~ or inside of brackets [...] (except ~).

A character class consists of zero or more of the following elements surrounded by brackets:

c	Literal character *c* (including [ ).
c1-c2	Range of characters (digits, uppercase or lowercase letters) (the dash has no meaning when at the beginning or end of a class).
^	Negated character class (only has meaning when first character in class).
~*c*	Literalized character.

For example, to copy all lines ending with a Pascal keyword or identifier:

```
   Parameter(s): <stdin [a-zA-Z][a-zA-Z0-9]*$ >stdout
```

To match anything between parentheses (not necessarily balanced):

```
   Parameter(s): <stdin (?*) >stdout
```

11.24 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.

11.25 ShowInterface

Synopsis

ShowInterface allows you to view the interface section of any unit.

Dialog

```
List file: [-console] [.TEXT]
Intrinsic.Lib: [-#11] [INTRINSIC.LIB]
$U filename:
Unit name:
```

Typing ? in response to a prompt displays information about the response needed.

Pressing [CLEAR] in response to a prompt terminates the program. After the prompts are processed, you can type -period to terminate.

Description

ShowInterface requires the same information as a Pascal USES statement: the unit's name, whether to process the unit in {$U+} or {$U-} mode, and if {$U-}, the object file (or library) containing the unit. Any number of units may be processed.

Library units, which are stored in a compressed format by the Linker, are formatted using a special version of the Pasmat utility. Noncompressed units are printed as is.

The default listing file is the console. The only way to change the list file is to rerun ShowInterface.

You can use a special intrinsic library, instead of the default INTRINSIC.LIB, for all units accessed by {$U+}. You must rerun ShowInterface to change the intrinsic library name. The specification of a special intrinsic library here corresponds to the **$W filename** Compiler invocation option, which allows you to use a particular intrinsic library for all the used {$U+} units of a particular compilation.

You can process a unit in {$U+} mode or {$U-} mode. The default is {$U+}. {$U-} is indicated by specifying an explicit library or object file. Functionally, this is similar to the following Compiler USES statements:

- No response to the "$U filename" prompt: **USES {$U+} unitname,...;**
- Library or object file name response: **USES {$U-} {$U filename} unitname,...;**

{$U+} indicates that the specified unit is to be searched for in the intrinsic library and, if not found, in the most recently specified $U filename. {$U-} means that the unit should be searched for only in the $U filename, never the intrinsic library. Also, in {$U-} mode, the specified filename is accessed as written. If that file can't be accessed, it is retried with a .OBJ extension.

The unit you specify is processed in the same manner as in the Compiler; before processing it, ShowInterface shows you the equivalent Compiler USES statement, and asks you if it's okay. If not, you are prompted again for a $U filename.

ShowInterface continues to prompt for $U filenames and unit names, and to process them, until you press [CLEAR]. You must exit and rerun the program to change the listing file or the intrinsic library used.

11.26 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 sensitivity. 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

11.27 Translit

Synopsis

Translit maps its input character by character and writes the translated version to its output.

Dialog

```
Parameter(s) [? for help]:
```

The format for the parameters is: <stdin [^]src [dest] >stdout

Description

Translit maps all the characters in its input file (StdIn) that match the characters in src into the corresponding characters in dest in the output file (StdOut). All characters not in src are simply copied from the input to the output. The defaults for both StdIn and StdOut are the console. If the input is from the console, use ⌘-C to indicate the end of file.

To replace all instances of "x" with "y":

Parameter(s): <stdin x y >stdout

Both the src and dest parameters may contain substrings of the form c1-c2, meaning all characters from c1 through c2, where c1 and c2 are both letters of the same case or both digits. To convert a file to all uppercase letters:

Parameter(s): <stdin a-z A-Z >stdout

If dest is omitted, then all characters specified in src are deleted. If dest is shorter than src, all characters in src that would map beyond the last character in dest are mapped to the last character of dest, and adjacent instances of such characters in the input are represented by a single instance of the last character in dest. To convert each string of digits in the input to the single digit 0:

Parameter(s): <stdin 0-9 0 >stdout

If src is preceded by a caret (^), then all characters *except* those in src are used as the source string--they are all deleted if dest is omitted, or they are collapsed to the last character in dest. To replace all nonalphabetic characters with asterisks:

Parameter(s): <stdin ^a-zA-Z * >stdout

The tilde (~) is a literalizing symbol in the src or dest parameters; it passes the following character as is. The special case "~n" represents a RETURN character. To replace all RETURN characters with spaces:

Parameter(s): <stdin ~n ~ >stdout

11.28 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

11.29 WordCount

Synopsis

WordCount counts the number of words in its input.

Dialog

```
Parameter(s) [? for help]:
```

The format for the parameters is: `<stdin >stdout`

Description

WordCount counts the number of words in its input (StdIn), and writes the total to its output (StdOut). The defaults for both StdIn and StdOut are the console. If the input is from the console, use **⌘-C** to indicate the end of file.

A *word* is considered any sequence of characters not containing a blank or any control characters (e.g., RETURN or DLE).

11.30 Xref

Synopsis

Xref is a cross-referencing utility that displays all variable references in a Pascal source program (or programs).

Dialog

```
Parameter(s) [? for help]:
Input file: [.TEXT]
Output file: [-console] [.TEXT]
Options [? for help]:
Maximum output line width:
Intrinsic.Lib to use for this Xref:
```

The input, output, and line-width prompts don't appear if they aren't needed or are specified as explicit parameters to the first prompt:

Parameter(s) [? for help]: input output width

Typing **?** in response to a prompt displays information about the response needed.

Pressing [CLEAR] in response to a prompt terminates the program. After the prompts have been processed, you can type ⌘-period to terminate.

Description

Xref lists each variable in the source program in alphabetical order, followed by the line numbers on which it appears.

The input can be a *set* of files if you don't give the input file as a parameter, but let Xref prompt you for it; each file is treated as an **include** file in the cross- reference display. Xref continues prompting for input files until a null response is entered. The response can also be of the form **<filename**, where filename contains a list of file names.

The width parameter is the maximum output width of the cross-reference listing (which determines how many line numbers are displayed on each line of the listing). The width can be a value from 40 to 132.

Line numbers in the cross-reference listing can refer to the entire source file, or can be relative to individual **include** files and units. Each variable reference indicates whether the variable is defined, assigned, or simply named (e.g., used in an expression).

Variables in Xref may be up to 16 characters. You can specify that the variables remain as they appear in the input, or they can be converted to all lowercase or all uppercase.

When **include** files are processed by Xref, each line number displayed is relative to the start of the **include** file; an additional key number indicates which **include** file is referred to. A list of each **include** file processed and its associated key number is displayed prior to the cross-reference listing.

USES declarations can also be processed by Xref (their associated $U filename, $U+ and $U- Compiler directives are processed as in the Compiler). These are treated exactly like **include** files, except that the line numbers refer to the lines of a unit's interface section as they are read from the library code file of a USEd unit, and, as in the Compiler, only the outermost USES declaration is processed (the USES declaration of a USEd unit is not processed). Also, as in the Compiler, a private Intrinsic.Lib may be used.

As an alternative to processing USES declarations, Xref accepts multiple source files. You can use this to get a cross reference of a set of main programs together with the units used by the programs. All the sources are treated like **include** files for display purposes. Xref checks to see if it has already processed a file (e.g., it appeared twice on the input list, or one of the files already USEd or included it), and if so, the file is skipped.

Options
The following options are available. Options are specified by listing them in response to the options prompt.

A Process all files, even duplicates of files already processed. The default is to process each file or unit only once.

B Suppress the lexical information on the source listing. See example for further details.

C Do *not* process a USEd unit if the unit's name or its ($U) object filename (if a Compiler $U- is in effect) is specified in the list of files to be processed.

D Delete all underscores in identifiers. The default is to retain the underscores and treat them as significant identifier characters (as in the Compiler).

I Do *not* process **include** files. The default is to process them.

L Force all letters in identifiers to lower case. The default is to leave them as they appear in the input. (If L and U are both specified, U is ignored.)

M Macintosh mode. Ignore any $U± directives. ($U- is assumed.)

N Do *not* process USES declarations. The default is to process them. If N is specified then the C option is ignored.

P Do *not* print the input source as it is being processed. The default is to list the input ($P Compiler directives generate a form feed to be generated).

S Suppress **include** and USES information in all displays. The cross-reference displays (the listing, if the P option is not used, and the cross-reference itself) will not contain any of the **include**/USES information. The T option is implied by specifying the S option.

T Cross-reference by total source line number instead of **include** file line number. The **include** information is still displayed (if S, I or N are not specified). This option is implied if the S option is specified.

U Force all letters in identifiers to uppercase. The default is to leave them as they appear in the input. (If L and U are both specified, U is ignored.)

$ Use a special intrinsic library directory; you will be prompted for the file name. The default is to use Intrinsic.Lib for intrinsic units.

* Ring the bell at completion of the execution.

Identifiers: Normally, Xref doesn't change the case of letters in identifiers or remove underscores, so you can see case differences in the cross-reference listing. If you use the L or U options, Xref ignores case (as the Compiler does). Up to 16 characters of each identifier are retained, so, unlike the Compiler, identifiers that differ in their spellings after the eighth character appear as different identifiers in the cross-reference listing.

Line numbers: You have the choice of which line numbers are displayed in the cross-reference listing: **include** file line number or total input line number. The default is **include** file line number. If you specify the S or T options, the listing shows total input line numbers. If the T option is used, **include** file information is still shown. The S option suppresses the **include** information.

Include/USES information: The I and N options control processing of **include** files and USES declarations, respectively. Normally, both of these are processed. You can suppress processing of **include** files by using the I option and suppress processing of USES with the N option.

If you don't specify N, Xref processes a USES declaration exactly like the Compiler. If you want to cross-reference an entire system, including all of the units of that system, processing the units through the USES declaration will only get you the INTERFACE section of each unit. To get both the INTERFACE and IMPLEMENTATION sections, specify a list of files to be processed that includes the sources to the units. In this case, you should specify the N option so none of the USES declarations are processed. If you don't have the sources to all the units (e.g., intrinsic units like SysCall), and want to process some on the USES declaration, while not processing the units whose sources are specified in the list, you need to use the C option.

With the C option, if the name of a USEd unit is the same as one of the filenames specified on the input list (ignoring any volume name and .TEXT extension), the unit will not be processed on the USES declaration, since its full source will be (or already has been) processed. If a Compiler $U- directive is in effect, then a {$U filename} Compiler directive specifies the name of the object code file to be used. This filename is also checked against the list of files. (The second check is required since a unit's name is not necessarily the same as its object code file name.)

To summarize, you have the choice of not processing the USES and specifying a list of all files you want to process (using the N option), or you can just process all the INTERFACEs through the USES declarations like the Compiler (by omitting the N option), or you can process some of the units through the USES and others as full sources (by specifying the C option). In all cases where a list of files is specified, no unit will ever be processed more than once, unless the A option is specified.

Limitations and Errors
Xref stores all its information on the Pascal heap. It gives a message if it runs out of space. If the console is not being used for the output listing, then Xref displays the amount of available space as it starts processing each file or unit. Three pieces of information are given:

- The total amount of heap space available.
- The maximum number of unique identifiers that can still be accepted.
- The minimum number of references that can be distributed across the identifiers.

Identifiers are accessed through a hashed symbol table that can hold a maximum of 5000 entries. The identifers themselves are not stored in the table, but are allocated dynamically on the heap. The identifier references are also dynamically allocated on the heap. Each identifier takes 16 bytes and each reference takes 8 bytes (10 if the T option is used). Both are competing for the heap space, so the information displayed shows the minimum number of references for the maximum number of available identifiers (i.e., refs = (2*MemAvail-16*id's) DIV 8) (or DIV 10). Fewer identifiers means more reference space.

Xref has a rather simple algorithm for determining whether a reference is defined, assigned, or just used. Although Xref will *never* miss a reference to a variable, the part of the algorithm that identifies a *definition* can be fooled into thinking a variable is defined when it actually isn't. One case in which this happens is in record structure variants. The record variant's case tag is flagged as a definition (even when there is no tag type), and the variant's case label constants (if they are identifiers) are sometimes incorrectly flagged. This only occurs in the declaration parts of the program.

Example
This example illustrates the output produced by Xref. The output at the end of this section is a small program listed by Xref together with its cross-reference listing. It has one included source.

Each line of source is preceded by five fields of information:

Field 1: The total line count.

Field 2: The **include** key assigned by Xref for an **include** (or USES) file (see below).

Field 3: The line number of each line within the **include** or main file.

Field 4: This field consists of two indicators (left and right) that reflect the static block nesting level. The left indicator is incremented (mod 10) and displayed whenever a BEGIN, REPEAT, or CASE is encountered. On termination of these structures with an END or UNTIL, the right indicator is displayed then decremented. It is thus easy to match BEGIN, REPEAT, and CASE statements with their matching terminations.

Field 5: A letter in the fifth field reflects the static level of procedures. The character is updated for each procedure nest level ("A" for level 1, "B" for level 2, and so on), and displayed on the line containing the heading, and on the BEGIN and END associated with the procedure body. Using this field you can easily find the procedure body for a procedure heading when there are nested procedures declared between the heading and its body.

Note that Xref does *not* process conditional compilation directives. Thus given the right combination of $IFC's and $ELSEC's, Xref's lexical information can be thrown off. If this happens, or if you don't want the lexical information, specify the B option.

The "(1)"s following the line numbers in the cross-reference listing are the **include** keys of the associated **include** files (shown in field 2 of the source listing). The **include** file names are shown preceding the listing (i.e., the "1. Factorial.TEXT"). Thus you can see what the line number is in which **include** file. The main file has no key and is shown as blank (if a *list* of files, even one, had been specified, the main file would be 1 and the include 2). An asterisk (*) following a line number indicates a definition of the variable. An equal sign (=) indicates an assignment. Nothing following a line number means a reference to the identifier.

The last line of the cross-reference listing summarizes the total number of identifiers and the number of references to those identifiers. The information in square brackets indicates how much space was still available at the end of the cross-reference. It shows the total number of bytes remaining, how many more identifiers could be accepted, and how many more references could be saved in the remaining bytes (see also Limitations and Errors, above). This information is also shown on the console (if it is not the output device) as each (**include**) file or unit is processed. In that case it reflects the state of memory at the time Xref starts processing the file or unit.

```
 1        1 —    PROGRAM XrefExample;
 2        2 —
 3        3 —       VAR
 4        4 —          Argument: LongInt;
 5        5 —
 6        6 —       {$i Factorial}
 7  1     1 — A     FUNCTION Factorial(Arg: LongInt): LongInt;
 8  1     2 —
 9  1     3 0- A       BEGIN {Factorial}
10  1     4 —             IF Arg<=1 THEN
11  1     5 —                Factorial := 1
12  1     6 —             ELSE
13  1     7 —                Factorial := Arg*Factorial(Arg-1);
14  1     8 -0 A       END; {Factorial}
15        7 —
16        8 0-      BEGIN {XrefExample}
17        9 1-         REPEAT
18       10 —             WriteLn;
19       11 —             Write('Enter argument: ');
20       12 —             Read(Argument);
21       13 —             IF (IOResult<=0) AND (Argument>=0) THEN
22       14 —                WriteLn('Factorial(', Argument: 1, ') =',
23       15 —                           Factorial(Argument): 1);
24       16 -1         UNTIL Argument<0;
25       17 -0      END. {XrefExample}
```

```
 1. Factorial.TEXT

-A-
  Arg                  1*( 1)    4 ( 1)    7 ( 1)    7 ( 1)
  Argument             4*       12        13        14        15        16

-F-
  Factorial            1*( 1)    5=( 1)    7=( 1)    7 ( 1)   15

-I-
  IOResult            13
```

```
-L-
  LongInt             4          1 ( 1)    1 ( 1)

-R-
  Read               12

-W-
  Write              11
  WriteLn            10         14

-X-
  XrefExample         1*

*** End Xref: 9 id's  24 references  [423312 bytes/4990 id's/42934 refs]
```

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
93 Only one .MAIN 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.

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

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

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. (This can be an error if a Unit in the link was trying to reference the library entry.)

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

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.

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

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.

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.

Data Initialization Segment Too Big:
The code segment used to copy the data into the initialized data segment is larger than 32K.

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.

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.

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

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.

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.

Segs 1-16 are Reserved:
The directory indicates that a segment name has been associated with one of the segments reserved for physical addresses.

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.

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.

A.2.3 Fatal Errors

A fatal error indicates a condition which prevents the link from continuing.

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.

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.

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.

Linker error -
Indicates an error in internal Linker logic, perhaps caused by an undetected disk or memory error.

No Starting location, linking Main Program:
The file containing the Pascal main program has been omitted from the input list, or is damaged.

Not Main or Intrinsic Link:
The Linker has not seen a valid input file to decide what type of link is desired.

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.

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.

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.

A.3 ObjIOLib Errors

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

Errors detected: No Output .LIB file written.
When the error count is nonzero, the directory file is not rewritten.

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.

A.3.2 Errors

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.

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.

A.3.3 Fatal Errors

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.

Errors during Installation:
Indicates errors during the installation of an object file library.

File Buffer less than 2 blocks:
Indicates an internal logic error in FileIO. Perhaps initialization was not called.

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.

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.

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
-1840	Packet ended in a resumable state (Archive)
-1293	Object is not password protected
-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
-876	File may be damaged due to I/O Error when flushing file buffer
-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 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 diskette drive
614	No disk present in drive
615	Wrong call version to diskette drive
616	Unpermitted diskette drive function
617	Checksum error on diskette diskette
618	Cannot format, or write protected, or error unclamping diskette
619	No more room in sysglobal for I/O request
623	Illegal device control parameters to diskette 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	RS232 driver called with wrong version number
641	RS232 read or write initiated with illegal parameter
642	Unimplemented or unsupported RS232 driver function
646	No memory available to initialize RS232
647	Unexpected RS232 timer interrupt

648	Unpermitted RS232 initialization, or disconnect detected
649	Illegal device control parameters to RS232
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 diskette 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
750	Position specified is out of range
751	No device exists at the requested position
752	Can't perform requested function while device is busy
753	Specified position is not a terminal node
754	Built-in devices cannot be configured
755	Isolated positions cannot be configured
756	The specified position is already occupied
757	Parallel Port doesn't exist on this type of machine
758	No room for more devices
790	Can't get buffer space to load configurable driver
791	Configurable driver code file is not executable
792	Can't get memory space for configurable driver
793	I/O error reading configurable driver file
794	Configurable driver code file not found
795	Configurable driver has more than one segment
796	Could not get temporary space while loading configurable driver
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
840	The allocation map of this file is truncated
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 not positive
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
1000	Unable to bring disk online (Priam)

1001	Error during disk formatting operation (Priam)
1002	Invalid Device_Control call for device (Priam)
1003	Unable to get sysglobal space for disk operation (Priam)
1005	Invalid request made to device driver (Priam)
1006	Error during disk write operation (Priam)
1007	Error during disk read operation (Priam)
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
1122	No space for catalog scan buffer (Reset_Catalog)
1124	No space for catalog scan buffer (Get_Next_Entry)
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
1281	Pathname is invalid because device or object is not present
1282	Pathname syntax is invalid
1283	Interior pathname component does not specify a directory object
1284	Directory cannot be deleted because it is not empty
1285	Operation is not allowed on a volume with a flat catalog
1286	Operation is not allowed on a directory object
1287	Cannot allocate SysLocal space for the directory scan stack
1288	Directory tree is inconsistent
1289	Operation not allowed against a volume or device (Quick_Lookup)
1290	The directory that contained the file has been deleted (Unkill_File)
1294	Object is password protected: no or incorrect password was supplied
1295	The allocation map of this file is damaged and cannot be read
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 diskette drive
1820	Write-protect error on diskette drive
1822	Unable to clamp diskette drive
1824	Diskette drive write error
1840	Unable to initialize disk drive (Priam)
1841	Error writing to disk (Priam) / Error reading from tape (Archive)
1842	Error reading from disk (Priam) / Error writing to tape (Archive)
1843	Error controlling tape (Archive)

1844	Packet ended in a nonresumable state (Archive)
1845	Packet command had an error (Archive)
1882	Bad response from ProFile
1885	ProFile timeout error
1998	Invalid parameter address
1999	Bad refnum

A.4.1 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)
10599	Disk I/O error while swapping in an OS code segment
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 diskette drive #2
10612	Spurious interrupt from diskette 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 diskette 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 diskette 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
10738	Can't find a required driver for the boot device
10739	Can't load a required driver for the boot device
10740	Boot device won't initialize

10741	Can't boot from a serial device
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

A.5 SULib Errors

A.5.1 IOPrimitives

32000	Attempt to use a private file control block
32001	File control block is already open
32002	Includes nested too deep
32003	Attempt to use a private buffer
32004	Not enough heap space for private file control block
32005	Not enough heap space for private buffer

A.5.2 ProgComm

32300	CommBufr open for read failed--bad key or not text
32301	CommBufr close failed--bad key
32302	CommBufr write failed--buffer not open or full
32303	CommBufr read failed--buffer not open

A.6 PasLib 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
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
6011	Bad text file format encountered
6050	Error trying to open **-printer** in QuickPort
6051	Error trying to write to **-printer** in QuickPort
6052	Error trying to close **-printer** in QuickPort

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

A.7 Exec File Errors

The Exec Processor reports syntax errors, I/O errors, and other process- time errors; it also reports errors resulting from Operating System calls. The format in which the Exec Processor reports errors is:

ERROR in <error location>
<current line>
<error marker>
<error message>

where

<error location>	is either 'invocation line' or 'line #<n> of file<file>'.
<current line>	is the text of the exec line in which the error was detected.
<error marker>	is a question mark indicating the place in **<current line>** where the error was detected.
<error message>	is one of the messages listed below. The error message begins with an error number.

A.7.1 Syntax Errors

The line containing the syntax error does not conform to the rules of the exec language. Check to see that you have typed the line correctly; refer to Section 9.1.4, Syntax of Exec Lines and Workshop Lines, and to descriptions of the individual commands and options for more information.

1 More than 20 parameters on exec procedure/function call
2 No closing) found
3 End of Exec file before ENDEXEC
4 No Exec file specified
6 End of Exec file in comment
7 Invalid percent: not "%n" form
8 Garbage at end of command
9 File does not begin with EXEC
10 No argument to SUBMIT
11 ELSE, ELSEIF or ENDIF not in IF
12 ELSEIF after ELSE
13 Nothing following

14 EXEC command other than at start of file
16 More than 20 variables declared
19 ENDWHILE not in WHILE
20 Duplicate parameter/variable name
21 Bad number. Numeric constant expected
22 Number too large
23 ORD requires a string argument of at least one character
24 UNTIL not in REPEAT
25 Bad Number for first argument to numeric comparison
26 Number too large for first argument to numeric comparison
27 End of Exec file in RUN command input
28 Bad Number. String expression with numeric result expected
-- Invalid command. <token> expected.
<token> is one of the following:

- String value
- Numeric value
- Number
- String expression with numeric result
- Boolean value
- Parameter name
- Parameter/variable
- String compare operator
- <>
- Comma (list delimiter)
- Command
- Terminating string delimiter
- Valid command keyword
- (
-)
- "ENDIF"
- "ENDWHILE"
- "UNTIL"
- Catalog specification
- File Identifier
- Clear command (Screen, EndScreen EndLine)
- Cursor command (Home, Up, Down, Right, Left)
- Program name

A.7.2 I/O Errors

The I/O error reported by the Exec Processor is followed by an additional line with the text of the corresponding Operating System error message.

201 Unable to open input file "<file>"
202 Unable to open exec run file "<file>"
203 Unable to access file "<file>"
204 Unable to rerun file "<file>"
205 Unable to reread file "<file>"
211 Unable to reopen input file "<file>"

A.7.3 Other Exec Errors

5	Line buffer overflow (> 255 chars)
15	Out of memory. Exec processing aborted
17	No value returned from file called as function
18	RETURN with value in file not called as function
28	Bad Number. String expression with numeric result expected
29	Number returned by string expression is too large
206	File variable "<id>" already in use
207	File variable "<id>" is undefined
208	File variable "<id>" is not open for input
209	File variable "<id>" is not open for output
210	Bad exec run file name generated: "<file>"

Appendix B
The Lisa Extended Character Set

Printing ASCII Characters

ASCII characters in the range hex 20 through hex 7E are supported for screen display, for printing on a dot matrix printer, and for printing on a daisy wheel printer with the following print wheels:

- Gothic, 15 pitch
- Prestige Elite, 12 pitch
- Courier, 10 pitch
- Boldface/Executive, PS.

Printing ASCII characters to a daisy wheel printer is not supported for the three print wheels with Modem type styles.

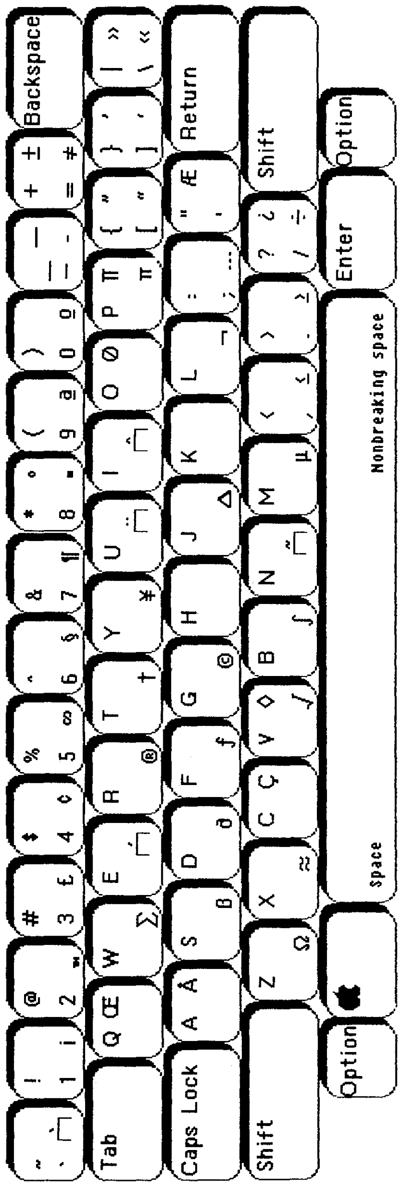
Lisa Extended Character Set

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	NUL	DLE	SP	0	@	P	`	p	Ä	ê	†	∞	¿	-	▨	▨
1	SOH	DC1	!	1	A	Q	a	q	Å	ë	°	±	¡	—	▨	▨
2	STX	DC2	"	2	B	R	b	r	Ç	í	¢	≤	¬	“	▨	▨
3	ETX	DC3	#	3	C	S	c	s	É	ì	£	≥	√	”	▨	▨
4	EOT	DC4	$	4	D	T	d	t	Ñ	î	§	¥	ƒ	‘	▨	▨
5	ENQ	NAK	%	5	E	U	e	u	Ö	ï	•	µ	≈	’	▨	▨
6	ACK	SYN	&	6	F	V	f	v	Ü	ñ	¶	∂	∆	÷	▨	▨
7	BEL	ETB	'	7	G	W	g	w	á	ó	ß	∑	«	◊	▨	▨
8	BS	CAN	(	8	H	X	h	x	à	ò	®	∏	»	ÿ	▨	▨
9	HT	EM	)	9	I	Y	i	y	â	ô	©	π	…	▨	▨	▨
A	LF	SUB	*	:	J	Z	j	z	ä	ö	™	∫	⎵	▨	▨	▨
B	VT	ESC	+	;	K	[	k	{	ã	õ	´	ª	À	▨	▨	▨
C	FF	FS	,	<	L	\	l	|	å	ú	¨	º	Ã	▨	▨	▨
D	CR	GS	-	=	M	]	m	}	ç	ù	≠	Ω	Õ	▨	▨	▨
E	SO	RS	.	>	N	^	n	~	é	û	Æ	æ	Œ	▨	▨	▨
F	SI	US	/	?	O	_	o	DEL	è	ü	Ø	ø	œ	▨	▨	▨

The first 32 characters and DEL are nonprinting control codes.

The shaded area is reserved for future use.

Characters Available with Option and Shift Keys



The Lisa Keyboard

Appendix C
Screen Control Characters

To perform standard screen control functions in Pascal, use the ScreenCtr procedure of PASLIBCALL, as described 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 below. Some actions take a single-character string, others require a two-character string.

In BASIC, use **PRINT** with the **CHR$** function, supplying the argument that corresponds to the desired action. For example, to erase the screen and position the cursor on the third line, enter the following BASIC statements:

```
10    print chr$(27); chr$(42); chr$(10); chr$(10)
20    end
            run
```

Table C-1
Screen Control Character Strings

Desired action	ASCII Char	HEX	Decimal	Coord
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	
position cursor at x,y*	ESC-=	1B-3D	27-61	y x

*To position the cursor at screen coordinates (x,y), use the two-character sequence [ESC]= followed by the coordinates: first the y-axis, then the x-axis. For example, to position the cursor at screen coordinates 0,1 in BASIC, enter either of the following statements:

```
10    print chr$(27); chr$(61); chr$(33); chr$(32)
```

or

```
10    print chr$(27); "="; "!"; " ";
```

The permissible ranges are shown in Table C-2 below. If you supply coordinates outside these ranges, a catastrophic system error may result. Refer to Appendix B for a complete chart of character equivalents.

Table C-2
Screen Coordinate Ranges

Axis	Limit	Screen Coord	Decimal	Keyboard Char
x	lower	0	32	[SPACE]
	upper	87	119	w
y	lower	0	32	[SPACE]
	upper	31	63	?

Appendix D
Common Problems

See also the Release 3.0 Notes for this appendix.

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.

Index

Please note that the topic references in this index are by *section number.*

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
STAM
HERE

apple computer
POS Publications Department
20525 Mariani Avenue
Cupertino, California 95014

TAPE OR STAPLE

Notes