HAYDEN *Macintosh Library* BOOKS

48409

MPW and Assembly Language Programming

For the Macintosh®

Scott Kronick

MPW® and Assembly Language Programming

For the Macintosh®

Scott Kronick

HOWARD W. SAMS & COMPANY
A Division of Macmillan, Inc.
4300 West 62nd Street
Indianapolis, Indiana 46268 USA

MPW® and Assembly Language Programming

For the Macintosh®

The Macintosh Programmer's Workshop for Assembly, Pascal, and C (with Emphasis on Assembly)

FIRST EDITION
FIRST PRINTING—1987

International Standard Book Number: 0-672-48409-9
Library of Congress Catalog Card Number: 87-61779

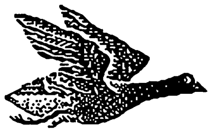
Acquisitions Editor: *Greg Michael*
Manuscript Editor: *Susan Pink Bussiere, Techright*
Designer: *T. R. Emrick*
Cover Art: *Celeste Design*
Compositor: *Shepard Poorman Communications, Indianapolis*

Printed in the United States of America

Trademark Acknowledgments

=	/	…	|	<	>	+
§	∂	≈	∞	¡	∆	«
»	÷	!	◊	~	*	&
ƒ	¬	#	^	$	\	≠

To order a disk containing all the example programs used in this book (plus additional fear and loathing sidetracks and Mac surprises) use the order form in the back of the book.



I wrote this book without any help from anyone, and if you believe that, my friends and I have some mosquito-free land in the Everglades we'd like to sell you.

Martha Lynch, Andy Hertzfeld, Michael Cermak, Sandi Pottberg, David Rand, Susan Barker, Doug Greene, Allen Cecil, Fred Huxham, Raines Cohen, Steve Costa, John Heckendorn, Kathy Ewing, and the BMUG crowd.

The book is dedicated to my mother, Ruth, and father, Mason.

Contents

Read Me First

Here you will find answers to the following questions:

- Exactly who is this book intended for?
- How is this book different from the manuals?
- Why is the emphasis on assembly language?
- Exactly what will I learn?

You have in your hands a starter's book using the Macintosh Programmer's Workshop (MPW).

That's right—starter's: Programmers who wish to begin using MPW and programmers who wish to begin using assembly language.

You might have heard that MPW is a sophisticated, professional programming environment intended for the *serious* developer. That's like saying educational TV is intended only for the serious thinker. Don't be snookered by pretentious programmers. The elegance of MPW benefits learners as well as developers.

The three sections within this text are designed to

- Provide show-me-how demonstrations of the extensive MPW command language for assembly, Pascal, and C programming. Programs in each of the languages are built from the ground up. Later chapters introduce additional MPW features along with explanations of resources and debugging.
- Provide a line-by-line, byte-by-byte instruction of Macintosh assembly

language fundamentals that assumes no previous experience in assembly programming. Included are ten fully-explained sample programs illustrating the Toolbox's mouse events, windows, Quickdraw, and menus.

- Provide MPW users with two complete dictionaries. The first contains summary entries for the entire MPW command language. The second contains the 68000 assembly language instruction set plus all the directives and Toolbox traps used in the example programs.

Unlike the manuals, the instruction here is not inundated with encyclopedic chapters listing every conceivable option for every obscure feature. The elements of MPW and assembly are introduced in the order in which a starter would use them. For example, you'll create a short, standalone application in the third chapter.

A large section of this book is devoted to giving a clear understanding of Macintosh assembly language. A knowledge of assembly principles offers considerable reward even for programmers who work primarily in Pascal or C. Assembly directly manipulates the computer's processor and memory, illustrating programming concepts that are difficult to visualize from within the higher-level languages like Pascal and C.

Assembly code, when well written, executes faster and uses less space than code from higher-level languages. MPW provides a convenient means of linking assembly code with Pascal and C code. Pascal and C programmers can improve a program's performance by coding in assembly those operations that demand speed.

Macintosh assembly is particularly accessible to all levels of programmers because so much of the groundwork coding is built into the Toolbox of the Macintosh ROM. In this text, MPW is illustrated with short assembly programs that help you write assembly programs of your own and increase your understanding of Pascal, C, and the hardware that supports all computer languages.

As long as you have the MPW disks and a Macintosh computer capable of running them, you can begin developing programs. You have three sections (none of which presume you are an experienced MPW user or programmer) to get you going.

Part 1. The Macintosh Programmer's Workshop

The purpose of part 1 is to get a starter up and running with program development in assembly, Pascal, and C. You can

- Explore the contents of the massive Workshop disk set, discovering which files a starter will need immediately and which can be set aside.

- Practice building applications in assembly, Pascal, and C. A sample development session in each language is presented in chapters 3 and 4.
- Experiment with the primary tools of the Workshop Shell, the central application that allows you to write, compile, and execute your programs from a single place.
- Unveil the potential of MPW, including structured commands, user-defined menus, automated development features, and resource and debugging tools.

Part 2. Up Bit Creek: The Assembly Tutorial

The purpose of part 2 is to get a starter writing and understanding assembly programs by examining individual lines of code in short example programs. (You'll also be entertained by a *fear and loathing* sidetrack in every chapter). Here, you can

- Review assembly's use of hexadecimal numbers.
- Witness snapshots of the processor and memory in action.
- Uncover, bit by bit, the low-level instructions that make a computer perform.
- Write short assembly programs using the Macintosh Toolbox, including mouse events, windows, Quickdraw, and menus.

Part 3. The MPW and Assembly Dictionaries

The purpose of part 3 is to keep the MPW user and the starting assembly programmer fully informed with a quick and complete reference for looking up new vocabulary. This section contains

- The MPW Shell command language
- The 68000 instruction set with directives and Toolbox traps

PART ONE

The Macintosh Programmer's Workshop

The first four chapters in part 1 set up an MPW environment for writing programs in assembly, Pascal, and C. Chapters 5 and 6 explain the files and command language that make programming with MPW versatile, powerful, and convenient. Chapters 7, 8, and 9 provide instruction on advanced MPW tools, including those that enable you to build resources and debug programs.

CHAPTER

1

I Bought the Disks, Now What?

What have I got and what do I need?

The Macintosh Programmer's Workshop (MPW) is a set of disks that provide the environment for developing programs in assembly, Pascal, and C. On these disks are numerous programs and data files that help you build your own independent, standalone application program.

The core of MPW includes a programming Shell, an assembler, tools, libraries, and examples. With this core, a programmer can write assembly language applications.

This book has been updated for use with MPW version 2.0. Users of earlier MPW versions will find that many MPW elements covered here are not implemented in their software. The programs used as examples in this book are not affected by the differences.

The following Macintosh screen illustrations, figures 1-1 to 1-5, show you the contents of the MPW (version 2.0B1) core disks.

The Pascal and C languages are supplements to MPW. These supplements, purchased separately, include compilers, tools, libraries, and examples. Another supplement, called MacApp, is a set of object-oriented libraries that provide an expandable application, that is, a base for a larger programming project.

Program development is easier and more efficient when you use a Macintosh with large internal memory and disk storage space. The actual memory and storage requirements vary with the MPW version and the scope of the project you are undertaking.

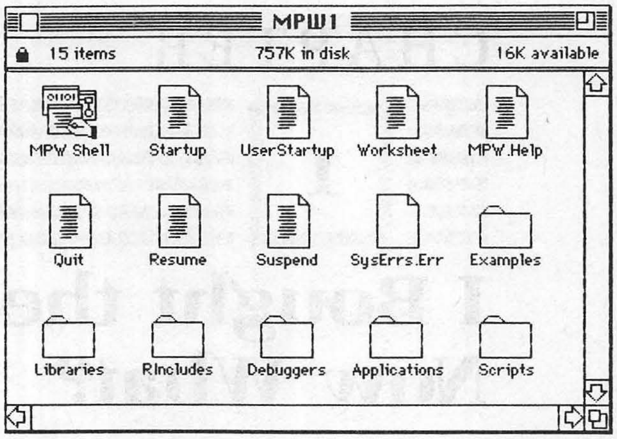


Figure 1-1

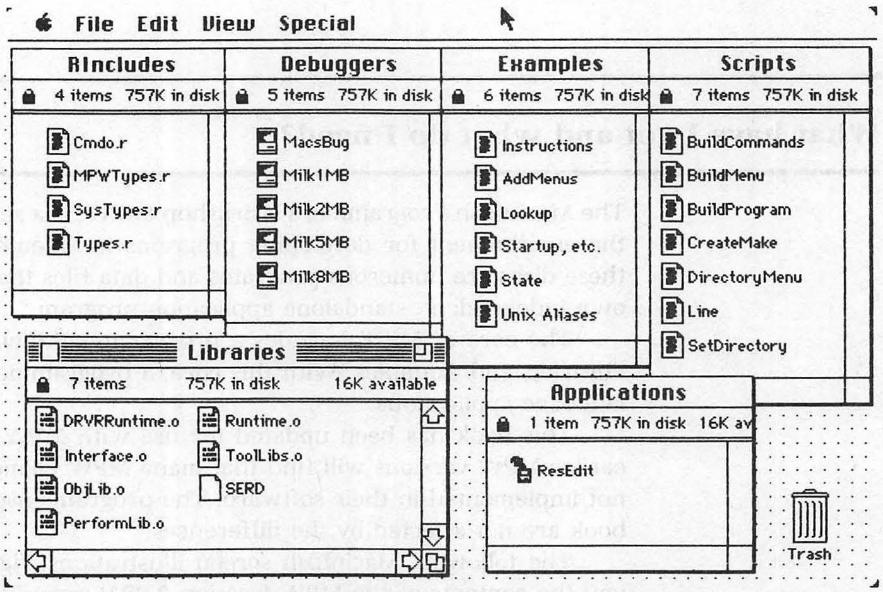


Figure 1-2

Consult the documentation for your particular MPW version to find its memory recommendations. The suggested minimum configuration for version 2.0 is 1M RAM, 128K ROM, a hard disk drive, and system 4.1 or newer. A RAM cache of 32K (set from the Control Panel) is suggested. The earlier 1.0 version of MPW can be used with floppy drives.

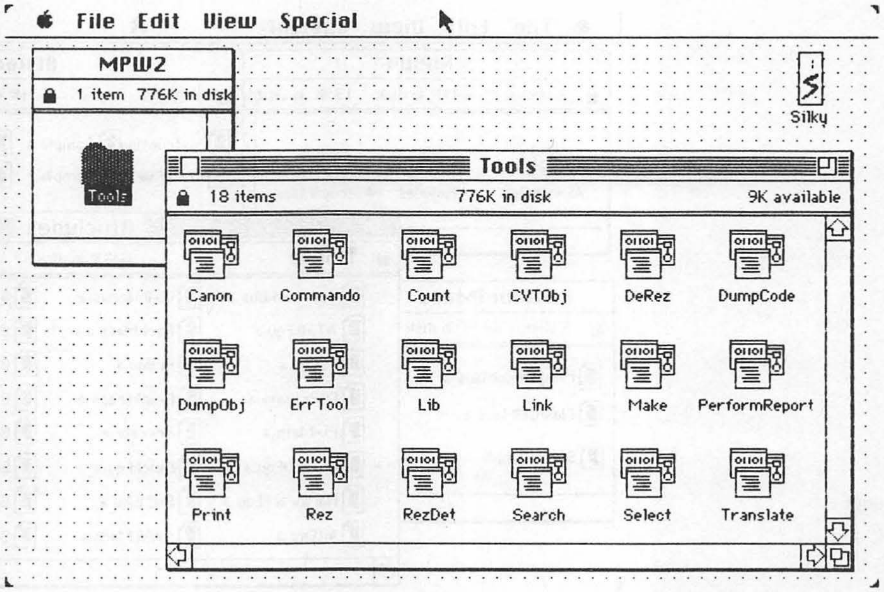


Figure 1-3

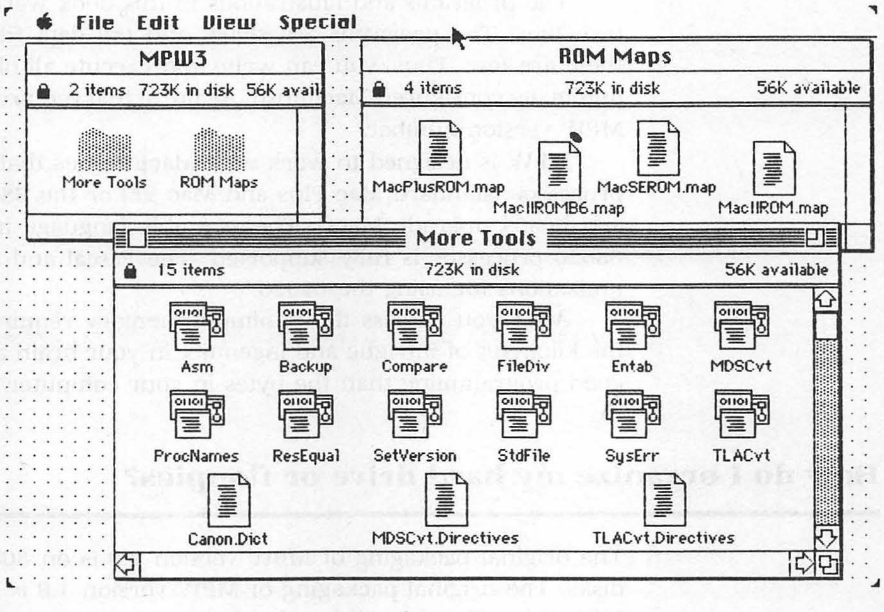


Figure 1-4

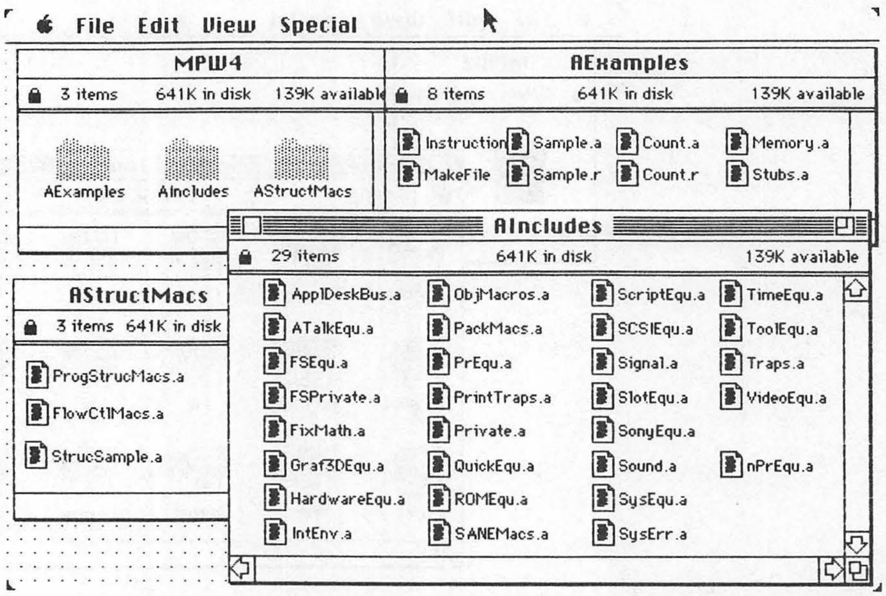

File Edit View Special
MPW4
3 items 641K in disk 139K available
AExamples AIncludes AStructMacs
AExamples
8 items 641K in disk 139K available
Instruction Sample.a Count.a Memory.a
MakeFile Sample.r Count.r Stubs.a
AIncludes
29 items 641K in disk 139K available
ApplDeskBus.a ObjMacros.a ScriptEqu.a TimeEqu.a
ATalkEqu.a PackMacs.a SCSIEqu.a ToolEqu.a
FSEqu.a PrEqu.a Signal.a Traps.a
FSPrivate.a PrintTraps.a SlotEqu.a VideoEqu.a
FixMath.a Private.a SonyEqu.a
Graf3DEqu.a QuickEqu.a Sound.a nPrEqu.a
HardwareEqu.a ROMEqu.a SysEqu.a
IntEnv.a SANEMacs.a SysErr.a
AStructMacs
3 items 641K in disk
ProgStrucMacs.a
FlowCtlMacs.a
StrucSample.a

Figure 1-5

The programs and illustrations in this book were created on a Macintosh Plus. The programs are short, and the data files required to create them are few. Thus, you can write and execute all of the programs with a minimally configured Macintosh. Again, actual requirements vary with your MPW version number.

MPW is designed to work with Macintoshes that use either the 68000 processor (standard Mac Plus and Mac SE) or the 68020 processor (Mac II and board upgrade Macs). The assembly language instruction set for the 68020 processor is fully supported. The Pascal and C compilers offer optimizations for using the 68020.

After you possess the minimum memory requirements, you will find the kilobytes of intrigue and ingenuity in your brain are more important to good programming than the bytes in your computer.

How do I organize my hard drive or floppies?

The original packaging of MPW version 2.0 is on 800K, double-sided, HFS disks. The original packaging of MPW version 1.0 is on 400K, single-sided, MFS disks. The core disks of either MPW version include everything you need to write programs in assembly language. The Pascal and C languages and the MacApp supplement are packaged on separate disks.

You should make copies of all MPW disks, then use the copies to make the work disks described in the rest of this section. That's right, three sets: originals, copies, and work disks. Do this now. The original MPW disks should be safely stored in a cupboard with the picture of your first boy-friend or girlfriend, and used only if your copies are lost.

If you are using a hard disk drive, you can access all information on the MPW disks by copying their contents onto the hard disk in appropriately named directories (folders). Unless your hard disk is filled with other material, all of MPW will fit with plenty of room left over for your own files. (As always, some file names and sizes could differ from what you see in the figures if you are using a different MPW version.)

Figure 1-6 shows a hard disk configuration where:

1. You create a new folder named MPW.
2. The contents of all MPW disks, including the Pascal and C disks if you have them, are copied to the hard disk and put in the new MPW folder.
3. The System Folder on your hard drive remains outside the MPW folder.
4. The contents of the folder named More Tools are put into the folder named Tools, then the More Tools folder is deleted.
5. If you have them, the Pascal compiler and tools (Pascal, PasMat, and PasRef) and the C compiler (C) are put into the Tools folder.
6. The program named MacsBug from the Debuggers folder is put inside the System folder.
7. The files MPW.Help, Quit, Resume, Startup, Suspend, UserStartup, and Worksheet remain in the same folder as the MPW Shell. Another Shell text file, MPW.Errs, is automatically created by the Shell and does not appear on your initial desktop.
8. For version 1.0 users only, the floppy disk and Lisa Startup files (named Startup.800K and Startup.XL) are deleted. (Macintosh XL/Lisa users should delete Startup, then rename Startup.XL to Startup.)

Most of the figures in this book are screen shots of a Macintosh running MPW version 2.0B1 with a hard disk drive. Floppy disk users with MPW version 1.0 can run all the programs in this book. Version 2.0, however, offers many new features and not all screen shots will look the same. Even hard disk users might want to look over these floppy disk instructions to see which files are essential MPW files for creating applications.

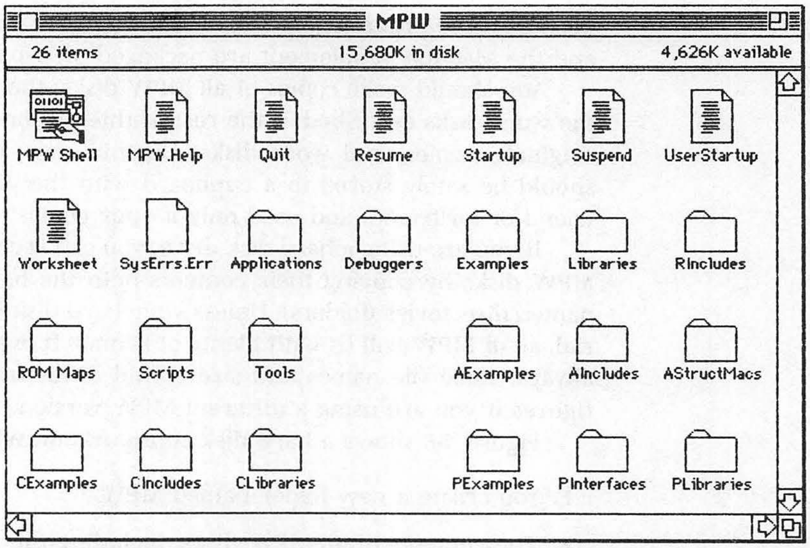


Figure 1-6

If you are using floppy disks and MPW version 1.0, access becomes trickier. The key to a workable configuration is to build a number of two-disk sets. Each pair of disks provides a standalone, single-language MPW environment.

The floppy disk configurations shown in the next set of figures contain the necessary files to follow along with the programs in this book. This minimum configuration fits on an 800K drive and a 400K drive (1200K total), with enough space remaining for your assembly programs.

Users with two 800K drives can have a complete set of data files (you can copy the entire AIncludes folder to your floppy). Additional disk drive space is particularly important for Pascal and C programming. Chapter 4 shows you how to create two-disk sets for Pascal and C programming.

The following three screens, figures 1-7 to 1-9, show a two-disk set for assembly programming where:

1. Select files are copied from the MPW disks onto one of the two floppy disks.
2. Select tools are combined into the folder named Tools.
3. The hard disk and Lisa Startup files (named Startup and Startup.XL) are deleted.
4. The floppy disk file named Startup.800K is renamed Startup.
5. The program named MacsBug, copied from the Debuggers folder, is placed inside the System folder.

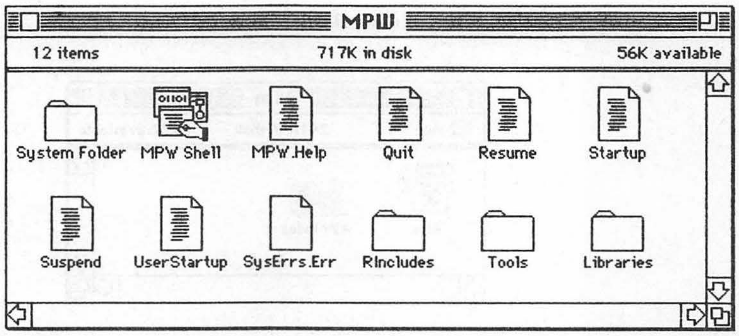


Figure 1-7

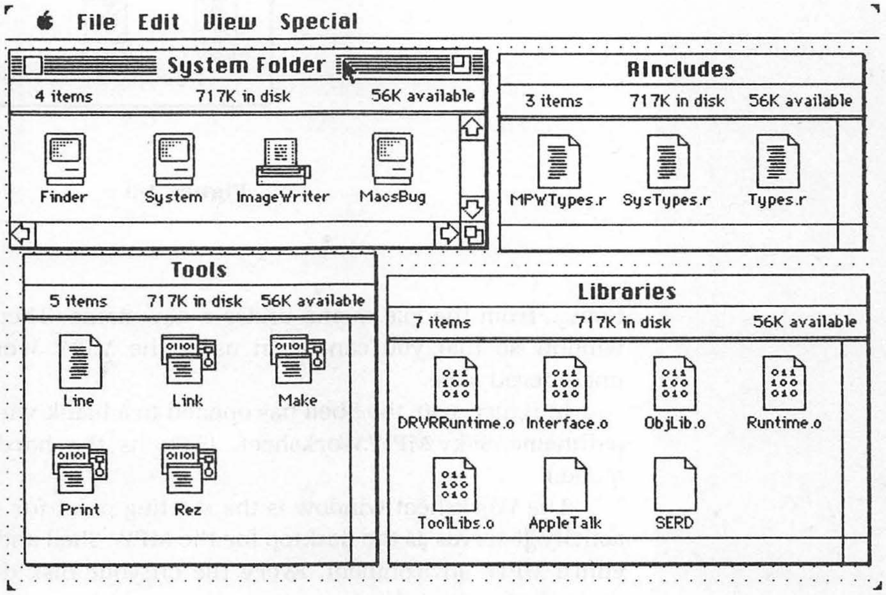


Figure 1-8

My drive is ready, how do I start programming?

To begin programming, double-click on the application named MPW Shell. A Worksheet window opens.

Your initial Worksheet window may be filled with a copyright notice, example instructions, and command information. After you glance through this information, you might want to print it or save a copy (choose Save a

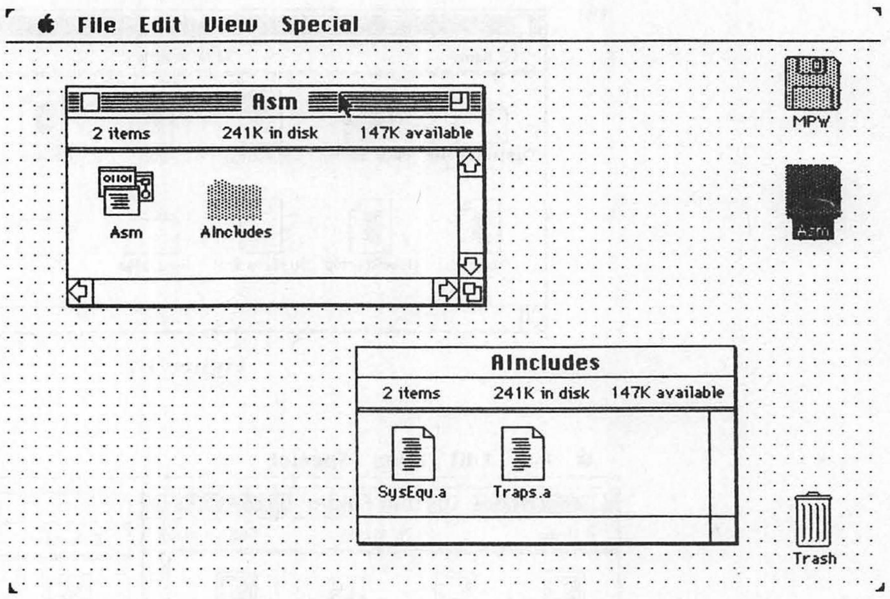


Figure 1-9

Copy... from the File menu) under a new name. Then clear the Worksheet window so that you can begin using the MPW Worksheet with a clean, uncluttered slate.

In figure 1-10, the Shell has opened to a blank window entitled with the pathname Silky:MPW:Worksheet. (Silky is the hard disk drive's volume name.)

The Worksheet window is the starting point for all your programming activity. It serves as the desktop for the MPW Shell and provides entry to the entire MPW environment. Every file on your disk drives can be accessed through the Worksheet.

Here are the two most salient features of the Shell's Worksheet:

- The user interface is primarily text oriented and uses an extensive command language. Although icons are not used, the Shell offers some menu options (you can create your own, too) and provides a method for producing informative dialog boxes that make learning and using the command language easier.
- The empty Worksheet window accepts text for nearly any purpose. Tasks such as text editing, file organization, program construction, and program execution can be performed directly in a Worksheet window using Shell commands.

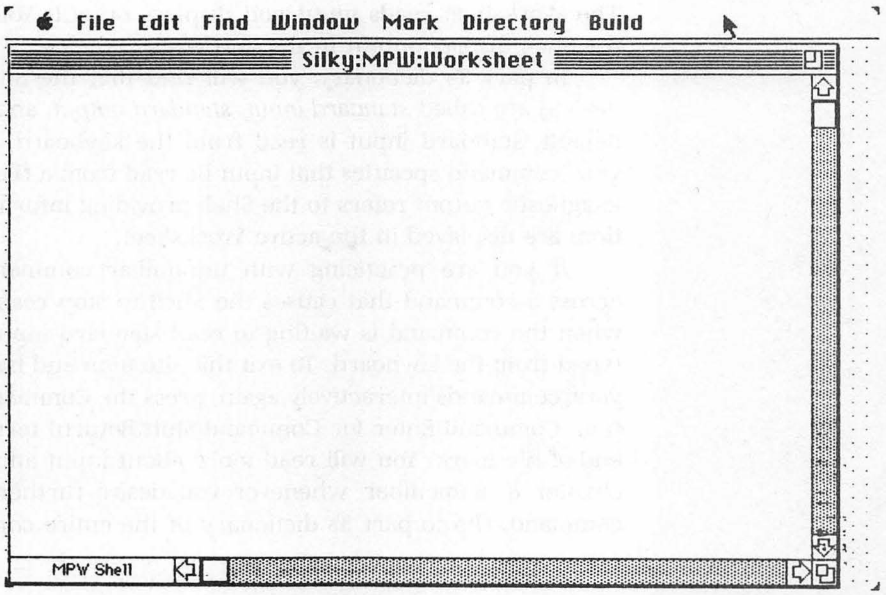


Figure 1-10

The Worksheet window accepts program code as well as the commands to compile, link, and execute the code. Without returning to the Finder or restarting the computer, a program can be written, tested, debugged, edited, and retested.

You can think of the Shell as a combination desktop Finder, word-processing editor, and programming language executor with a built-in command vocabulary. All language, data, and tool files on your MPW disks are integrated to work under Shell control.

Another special aspect of the Worksheet window is that it cannot be closed. When the MPW Shell is running, the Worksheet window is always open. Copies of the Worksheet can be saved under various file names. All text files created by MPW are, in essence, Worksheet copies with Worksheet capabilities.

This is important: The name *Worksheet* refers to any Shell window that is interacting with the Shell. The original Worksheet window always remains open, but any active window can be used to perform Shell commands and otherwise work as the original Worksheet.

In upcoming chapters, the command language is explored in more detail. You will see that the Shell can work both interactively (type a command, press Enter, and the command is executed) and as a file processor (type a program or series of commands, save the file, and execute the file).

The Worksheet reads input and displays output. Your first exercises with the Shell display output in the Worksheet window.

In part 3's dictionary, you will find that the Shell's input and output devices are called *standard input, standard output,* and *diagnostic output.* By default, standard input is read from the keyboard. More often, however, your command specifies that input be read from a file. Both types of output (diagnostic output refers to the Shell providing information about its operation) are displayed in the active Worksheet.

If you are practicing with unfamiliar commands, you might come across a command that causes the Shell to stop responding. This happens when the command is waiting to read standard input, that is, information typed from the keyboard. To exit this situation and have the Shell recognize your commands interactively again, press the Command/Enter key combination. Command/Enter (or Command/Shift/Return) terminates input with an end-of-file mark. You will read more about input and output at the end of chapter 8. Remember, whenever you desire further information about a command, flip to part 3's dictionary of the entire command language.

CHAPTER

2

What Are the Fundamental File Commands?

How do I get a handle on file handling?

Before you go about creating a file, you should explore a few MPW commands that handle files. These file handling commands make the MPW Shell a substitute for the Finder. You will practice with three commands, volumes, directory, and files, to see the contents of your hard or floppy disk.

All Worksheet windows can be used to enter Shell commands. Even if you have named a Worksheet with a file name, the window retains its interactive ability to take and respond to commands. If an error occurs, an error message is displayed on lines following the command.

Some commands are executed silently; others display standard output in the window. All Shell commands use the Status Panel, a small box in the bottom-left corner of the active window, to show the Shell's current activity.

When no command is being executed, the status panel displays *MPW Shell,* as shown in figure 2-1. Clicking inside the Status Panel is an alternative to pressing the Enter key.

Type *volumes* in the Worksheet window, then press Enter. The Enter key executes the Shell command on the same line as the insertion bar, so it is important that you do not press the Return key before pressing the Enter key.

You have to look quickly to see the name *volumes* in the bottom-left corner Status Panel as the command is executed. Figure 2-2 shows the output the command produces in the Worksheet window when using a hard disk drive that has the volume name Silky. If you are using a hard disk allocated as a single volume, your output will show the single volume name. The volumes command lists only mounted volumes.

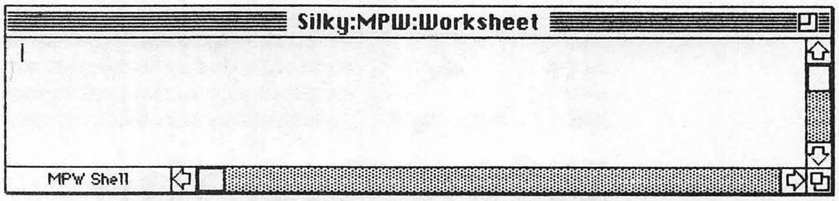
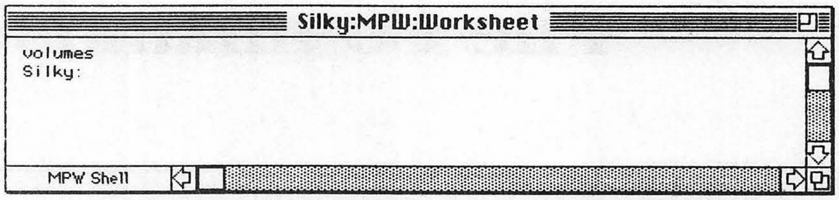


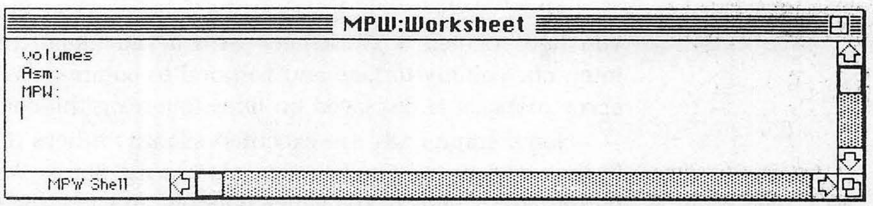
Figure 2-1



```
volumes
Silky:
```

Figure 2-2

Figure 2-3 is the output of the same command when used on a system with two floppy disks.



```
volumes
Asm:
MPW:
```

Figure 2-3

The output of the floppy disk system's volumes command is Asm: and MPW:. Each of these floppy disks is considered a volume. The disk name and the volume name are the same.

The colon that follows a volume name is important. Colons help distinguish volumes, directories, and files. When you specify a volume name, you must add the trailing colon, or MPW will consider it a file name.

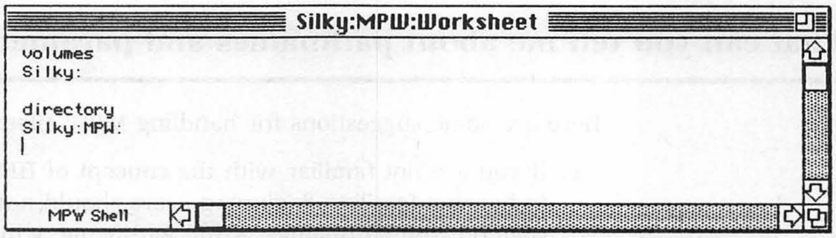
Press Return once (to help readability), then type *directory* in the Worksheet window and press the Enter key. Remember to press the Enter key while the insertion bar is on the same line as the command you want executed.

You can also execute a command in two steps by:

1. Selecting the command or command lines you want executed. A selection in the active window appears in inverse, white letters on a black background.
2. Pressing the Enter key to execute the selection. (Clicking in the Status Panel or pressing the Command/Return key combination is equivalent to pressing the Enter key.) If the selection contains more than one command line, each command is executed sequentially.

Note: MPW version 2.0 has expanded the methods of directory control through the use of menus and dialog boxes. After reading the next few paragraphs that explain the directory command, you might want to flip to dictionary definitions of the commands directoryMenu and setDirectory. These commands provide a versatile menu alternative to the window-based command directory.

Figure 2-4 shows the output the directory command produces in the Worksheet window.



```
volumes
Silky:

directory
Silky:MPW:
|
```

Figure 2-4

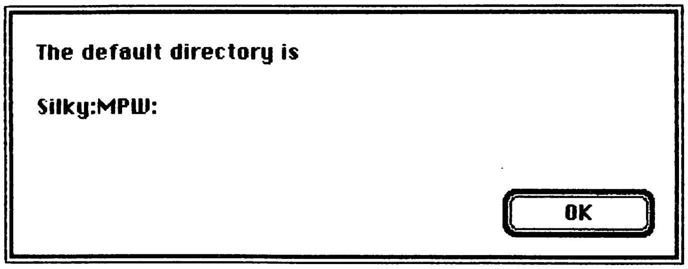
Caution: MPW uses the term *directory* in two related, yet distinctly different, ways. In one use, directories are analogous to folders, representing groups of files accessed through pathnames. However, the command directory refers to a scope of activity, a kind of primary folder that always represents a single volume (or floppy disk).

When working with MPW, you must always be concerned with your scope of activity. MPW won't find files outside its current scope unless their pathnames are specified. The default directory represents MPW's immediate file handling scope.

The directory command, with no additional parameters, writes the name of the MPW default directory to the Worksheet (standard output) window. The directory command output shows that the file handling scope is currently set to be the directory whose pathname is Silky:MPW:.

Parameters, by the way, are strings (often file names) that follow command words to give the command a special meaning. Each command's parameters are explained in the dictionary definitions in part 3.

As noted, you can also show the name of the default directory by selecting Show Directory from the Directory menu. Figure 2-5 is an example dialog box that displays directory information.


The default directory is

Silky:MPW:

Figure 2-5

What can you tell me about pathnames and parameters?

Here are some suggestions for handling MPW files:

1. If you are not familiar with the concept of HFS pathnames, get ready to become familiar. Beginning users should not use shortcuts. Instead, spell out full pathnames. A full pathname, which may include zero, one, two, or more directory names, has the format:

   volumeName:directoryName1:directoryName2:fileName

2. Uppercase and lowercase letters are treated the same (but they are not when you use the C compiler). Each file or directory name must be less than 32 characters. Names that contain spaces must be enclosed in quotation marks so that they are considered a single name. Colons cannot be part of a name.
3. For advanced users: The characters shown in figure 2-6 can be used to help specify pathnames.
4. Don't forget your colons. Volumes and directories need colons.

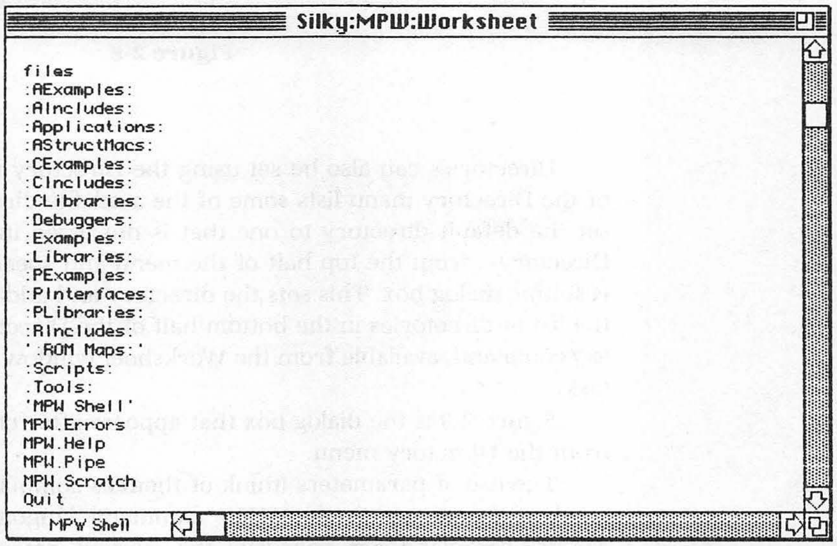
?	Wildcard character. Match any single character (except colon or Return).
≈	(≈ = Option/X) Wildcard character. Match any string of characters (except colon or Return).
§	(§ = Option/6) Selection character. Match currently selected text in the default window or in the window given by *name.§*.

Figure 2-6

Type *files* in the Worksheet window, then press the Enter key. Once again, remember to press the Enter key while the insertion bar is on the same line as the command you want executed. Alternately, you can select the command or commands (white letters on a black background) to execute the entire selection.

The output of the files command is shown in figure 2-7. (The list of files is longer than a single windowful can show. The scroll bar lets you see the entire list.)

The files command, with no parameters, writes the names of the files in the MPW default directory to the Worksheet window. Notice the use of

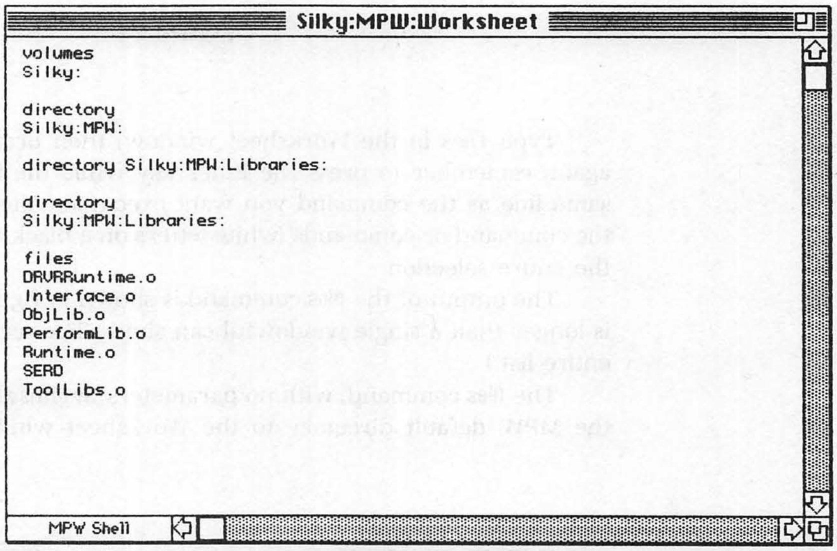


```
files
:AExamples:
:AIncludes:
:Applications:
:AStructMacs:
:CExamples:
:CIncludes:
:CLibraries:
:Debuggers:
:Examples:
:Libraries:
:PExamples:
:PInterfaces:
:PLibraries:
:RIncludes:
':ROM Maps:'
:Scripts:
:Tools:
'MPW Shell'
MPW.Errors
MPW.Help
MPW.Pipe
MPW.Scratch
Quit
```

Figure 2-7

colons to designate directories. Directories use both preceding and trailing colons in their names.

You can change the default directory by once again using the directory command, with parameters. In figure 2-8, the current directory is changed to Silky:MPW:Libraries:, then the files command outputs only the library files in its current file handling scope.



```
volumes
Silky:

directory
Silky:MPW:

directory Silky:MPW:Libraries:

directory
Silky:MPW:Libraries:

files
DRVRRuntime.o
Interface.o
ObjLib.o
PerformLib.o
Runtime.o
SERD
ToolLibs.o
```

Figure 2-8

Directories can also be set using the Directory menu. The bottom half of the Directory menu lists some of the available directories. If you want to set the default directory to one that is not listed in the menu, choose Set Directory... from the top half of the menu and specify the directory in the resulting dialog box. This sets the directory and adds the directory name to the list of directories in the bottom half of the Directory menu. The setDirectory command, available from the Worksheet window, performs the identical task.

Figure 2-9 is the dialog box that appears after choosing Set Directory... from the Directory menu.

The use of parameters (think of them as command specifications) gives much more power to the MPW command language. In figure 2-8, the directory command not only lists the current default directory but, with parameters, also changes it.

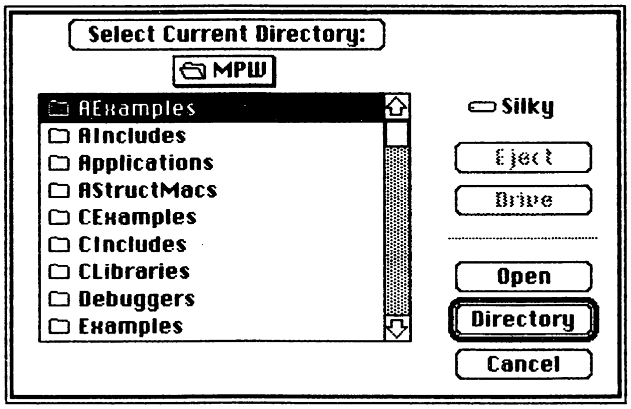


Figure 2-9

Take a moment to glance through the MPW dictionary in part 3. You will find that most commands have a variety of optional and required parameters as well as numerous minus sign (–) options that add even more versatility.

The files command can also accept parameters. By adding a volume or directory name as a parameter, you can specify which files you would like listed, without changing the default directory.

Try using the command

```
files Silky:MPW:Tools:
```

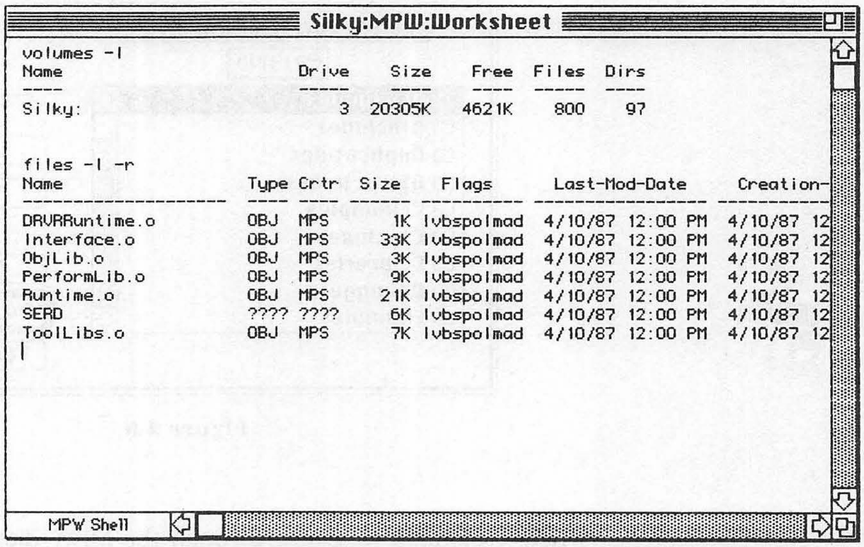
while the default directory remains Silky:MPW:Libraries:.

When your Worksheet gets cluttered with information you no longer need, select the unwanted text and press the Backspace key. To completely clear a Worksheet, you can choose Select All from the Edit menu, then press the Backspace key.

Take a few moments now to experiment with the three commands you have used in this chapter. Look up volumes, directory, and files in the dictionary to see what options are available. Figure 2-10 shows the volumes and files commands with minus sign options.

Where can I find file help when I need it?

MPW's help command assists you in using the variety of commands, expressions, characters, patterns, selections, and shortcuts. By typing the following commands, MPW writes summary information about each to the Worksheet.



```
volumes -l
Name                      Drive   Size    Free  Files  Dirs
----                      -----   ----    ----  -----  ----
Silky:                        3  20305K   4621K   800    97


files -l -r
Name                  Type Crtr  Size  Flags       Last-Mod-Date      Creation-
--------------------- ---- ---- ----- ----------- -----------------  ----------
DRVRRuntime.o         OBJ  MPS    1K  lvbspolmad  4/10/87 12:00 PM   4/10/87 12
Interface.o           OBJ  MPS   33K  lvbspolmad  4/10/87 12:00 PM   4/10/87 12
ObjLib.o              OBJ  MPS    3K  lvbspolmad  4/10/87 12:00 PM   4/10/87 12
PerformLib.o          OBJ  MPS    9K  lvbspolmad  4/10/87 12:00 PM   4/10/87 12
Runtime.o             OBJ  MPS   21K  lvbspolmad  4/10/87 12:00 PM   4/10/87 12
SERD                  ???? ????   6K  lvbspolmad  4/10/87 12:00 PM   4/10/87 12
ToolLibs.o            OBJ  MPS    7K  lvbspolmad  4/10/87 12:00 PM   4/10/87 12
|
```

Figure 2-10

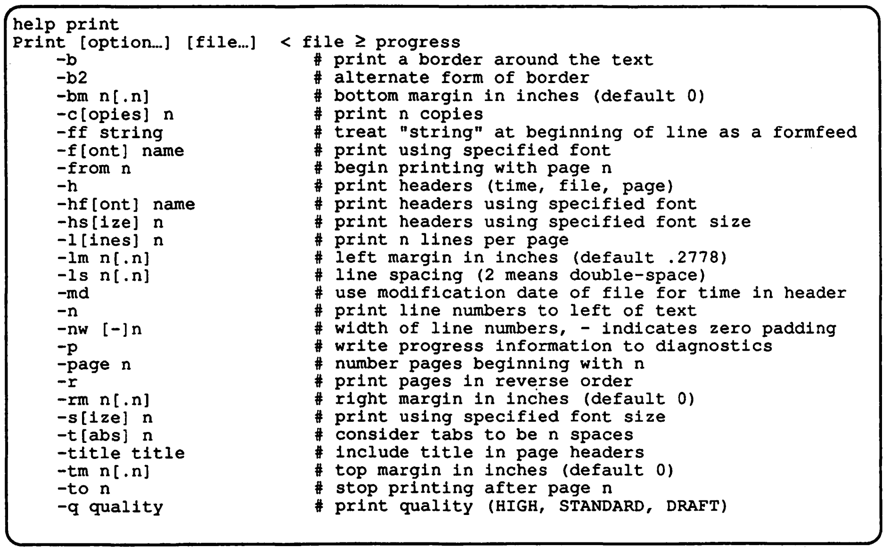
```
help Commands
help Expressions
help Characters
help Patterns
help Selections
help Shortcuts
```

Using help Commands provides a list of all available minus sign options and other parameters. In addition, the help command can be used with any particular command as its parameter. The output is a list of only that command's possible parameters. Figure 2-11 is an example of this.

Another tool to facilitate using the commands of the MPW Shell is called commando. The commando command produces dialog boxes that help you compose the line-oriented text of the Worksheet. In these dialog boxes, you can choose parameter pathnames and minus sign options, read help text, and either execute immediately or write to output the command itself.

You can use the commando interface in two ways:

1. An ellipsis character (...) allows immediate Do It execution of a commando-created command line. (An ellipsis character is produced by pressing simultaneously the semicolon key and the Option key. Do not enter three periods.) When you insert the ellipsis character,



```
help print
Print [option…] [file…]  < file ≥ progress
    -b                      # print a border around the text
    -b2                     # alternate form of border
    -bm n[.n]               # bottom margin in inches (default 0)
    -c[opies] n             # print n copies
    -ff string              # treat "string" at beginning of line as a formfeed
    -f[ont] name            # print using specified font
    -from n                 # begin printing with page n
    -h                      # print headers (time, file, page)
    -hf[ont] name           # print headers using specified font
    -hs[ize] n              # print headers using specified font size
    -l[ines] n              # print n lines per page
    -lm n[.n]               # left margin in inches (default .2778)
    -ls n[.n]               # line spacing (2 means double-space)
    -md                     # use modification date of file for time in header
    -n                      # print line numbers to left of text
    -nw [-]n                # width of line numbers, - indicates zero padding
    -p                      # write progress information to diagnostics
    -page n                 # number pages beginning with n
    -r                      # print pages in reverse order
    -rm n[.n]               # right margin in inches (default 0)
    -s[ize] n               # print using specified font size
    -t[abs] n               # consider tabs to be n spaces
    -title title            # include title in page headers
    -tm n[.n]               # top margin in inches (default 0)
    -to n                   # stop printing after page n
    -q quality              # print quality (HIGH, STANDARD, DRAFT)
```

Figure 2-11

anywhere on the same line as the command's name, execution causes the commando's dialog box to appear. After you have composed the command line by answering the dialog box, clicking in the Do It button executes the command itself.

2. The word *commando* permits delayed Worksheet execution of a commando-created command line. By preceding a command with *commando* (instead of using the ellipsis), execution causes the commando's dialog box to appear. After you have composed the command line by answering the dialog box, clicking in the Do It button writes the command line to the Worksheet for execution at a later time.

In the case of complex commands, commando dialog boxes offer a variety of menus and controls. These include text fields, radio buttons, check boxes, pop-up menus, multiple input files and directories, repeatable options, and nested dialog boxes. Each command has its own commando interface. Experimenting with various commando commands helps to illustrate graphically a tool's functionality. MPW also permits users to create a commando interface for their self-built tools.

Figure 2-12 is a sample commando interface for the Print tool. (You'll see more of commando in chapter 8.)

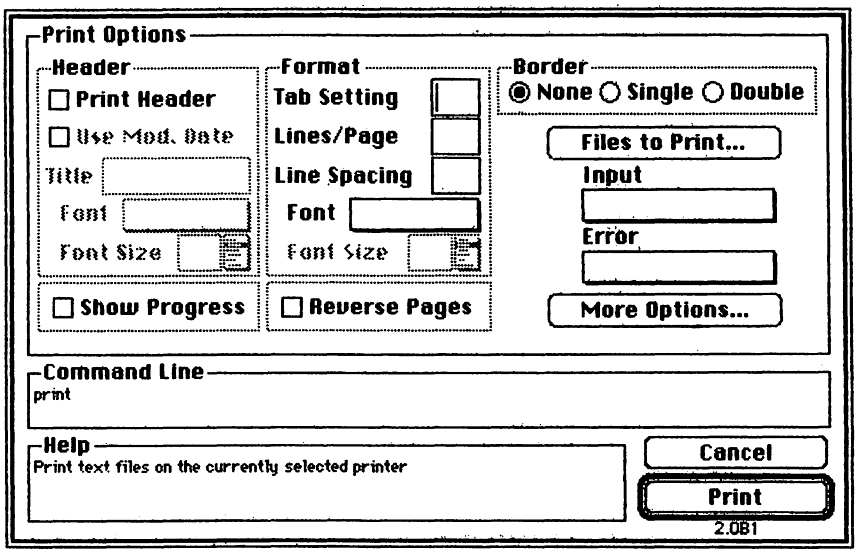

Print Options
Header
☐ Print Header
☐ Use Mod. Date
Title
Font
Font Size
Format
Tab Setting
Lines/Page
Line Spacing
Font
Font Size
Border
[✓ None] Single Double
Files to Print...
Input
Error
☐ Show Progress
☐ Reverse Pages
More Options...
Command Line
print
Help
Print text files on the currently selected printer

Figure 2-12

As more of MPW's commands are introduced in this book, try using their commando dialog boxes. Using dialog boxes allows you to spend less time referencing tool definitions in the dictionary and cuts down on Worksheet typing errors.

CHAPTER

3

Can You Show Me a Program in Assembly?

Can't I just start programming now?

In this section you will type, assemble, link, and run an assembly program. You do not need to know assembly to create this program; just type in what you see. Part 2, the assembly tutorial, explains the mechanics of the program you use here. Creating an assembly program requires four steps:

1. Knowing the directory location of all required files.
2. Entering the program code in a new Worksheet window.
3. Entering a sequence of compile and link commands in the original Worksheet window.
4. Executing the command sequence.

In the later chapters of part 1, you learn about tools that automate the process of compiling and linking programs. In particular, the Build menu offers a commando interface to create and, if desired, execute a command sequence. The example program in this chapter avoids these tools in the hope that you will more easily understand the underlying process by manually building your program.

First, clear your current Worksheet or create a new, clean Worksheet. The MPW commands clear and new perform these functions. Alternately, you can choose New from the File menu to create a new window; a dialog box asks you to name the new window. A third alternative is to select the entire contents of the Worksheet and press the Backspace key. This removes the contents of the Worksheet, but who cares? You can always repeat the file handling commands that you used in the last section. (Shortcut note: You can select text larger than the screen by dragging, by choosing Select All from the Edit menu, or by the shift key/click method that selects all text between the cursor and the insertion bar at a shift key/button press.)

Fastidious types always like to keep a clean Worksheet and save only those Worksheets that contain command sequences likely to be repeated. Because your command sequences are short in the first few chapters, there is no need to save the Worksheet. Saving Worksheets can be done either through the File menu or the Shell's save command.

Now that you have a clean Worksheet, type in the program in listing 3-1, CorneredCoin. To make the typing easier, you can omit the program's comments. Comments, which help a programmer understand the code but are ignored by the assembly process, include any text that follows a semicolon. You can see in the program that all comments are in the rightmost column.

The indentation and spacing of the program code is important. At the end of each line of code, press the Return key to get to the next line. Make sure that the lines of code at the left margin are aligned at your Worksheet's left margin. That is, do not insert any preceding spaces or tabs. Use the Tab key to line up the other columns. Uppercase letters serve only for readability.

Listing 3-1

```
                                        ;Program CorneredCoin

            INCLUDE 'Traps.a'           ;define trap names

            MAIN
            PEA     -4(A5)              ;push pointer to Quickdraw globals
            _InitGraf                   ;initialize Quickdraw
            _InitFonts                  ;initialize font manager
            _InitWindows                ;initialize window manager
            _InitCursor                 ;initialize cursor to arrow

            SUBQ    #4,SP               ;make room for pointer result
            CLR.L   -(SP)               ;allocate on heap
            PEA     WindowSize          ;push pointer to rectangle
            PEA     WindowName          ;push pointer to name
            ST      -(SP)               ;yes, window is visible
            CLR.W   -(SP)               ;use document window
            MOVE.L  #-1,-(SP)           ;put window on top
            SF      -(SP)               ;no, window has no goAway box
            CLR.L   -(SP)               ;NIL window refCon
            _NewWindow                  ;make the window
            _SetPort                    ;make window current port

            PEA     CoinSize            ;push pointer to rectangle
            _FrameRect                  ;draw rectangle frame

            MOVE.L  #$006E007A,-(SP)        ;specify integer coordinates
            _MoveTo                     ;place Quickdraw pen at point
            PEA     CoinLetters         ;push pointer to string
            _DrawString                 ;draw string at pen location

FlipCoin
            _SystemTask                 ;give screen time to resynch
            SUBQ    #4,SP               ;make room for point result
            MOVE.L  SP,-(SP)            ;push pointer to result space
            _GetMouse                   ;get cursor coordinate point
            MOVE.L  (SP)+,D3            ;store point in register
```

Listing 3-1 *cont.*

```
            SUBQ     #2,SP              ;make room for boolean result
            MOVE.L   D3,-(SP)           ;retrieve cursor point
            PEA      CoinSize           ;push pointer to rectangle
            _PtInRect                   ;see if point is in rectangle

            TST.B    (SP)+              ;set Z flag accordingly
            BEQ.S    TryButton          ;branch if Z is set (not in rect)

            PEA      CoinSize           ;push pointer to rectangle
            _InverRect                  ;invert rectangle

TryButton
            SUBQ     #2,SP              ;make room for boolean result
            _Button                     ;see if button is pressed
            TST.B    (SP)+              ;set Z flag accordingly
            BEQ.S    FlipCoin           ;branch if Z is set (no press)

            _ExitToShell                ;return to Desktop

WindowSize  DC.W     80,60,290,450      ;window bounds (Top,Lft,Bot,Rgt)
WindowName  DC.B     'Cornered Coin'    ;window title

CoinSize    DC.W     80,100,130,290     ;rectangle bounds
CoinLetters DC.B     'BlackHeads/WhiteTails' ;string in rectangle

            END                         ;code end directive
```

Any errors you may have typed are not noticed by the Editor. Only in the later stages of assembling, linking, or running a program will errors be noted.

After you type in the code, save the code in the MPW folder under the name Coin.a. Floppy disk users should save the code onto the Asm: disk.

The simplest way to save Coin.a is to choose Save a Copy... from the File menu, then answer the dialog box by typing the file name and pressing Return. The code in the Worksheet is saved in a new window and file named Coin.a. Beneath the Coin.a window is the ever-present Worksheet window.

The Save As... item is not available from the Worksheet window because the Worksheet cannot be closed or replaced. Save As... replaces a window with a new file name, whereas Save a Copy... creates a second window with a new file name. The Save item saves to disk without a file name change.

At this point, it is easier if you do *not* put Coin.a into any subfolder. This will make it easier to find the Coin.a file (one less directory pathname to specify) when you are ready to compile the source code.

Take note of the .a suffix you appended to the file name Coin. Suffixes play an important role in identifying the files used in creating an assembly language application. Certain MPW programs create files with suffixes already attached. Adding other suffixes is the programmer's responsibility.

You are almost ready to assemble and link your code. First, bring the Worksheet window back on top (click on it).

Hard disk users should make their current directory Silky:MPW: where Silky is the volume name of the hard disk drive. If necessary, use the command

```
directory Silky:MPW:
```

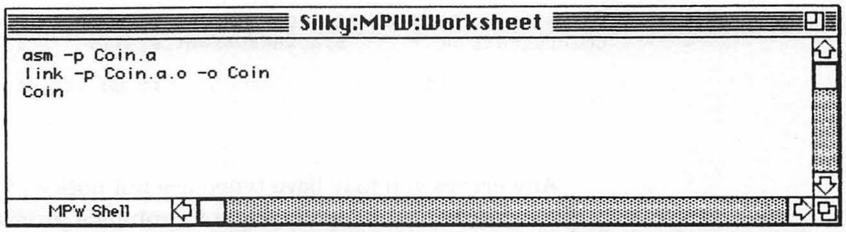
to set the directory, or choose Set Directory... from the Directory menu and answer the dialog box.

Floppy disk users should specify the current directory as Asm:. If you did not do so earlier in the chapter, use the command

```
directory Asm:
```

to change the directory to Asm:. Executing the directory command without parameters displays the current directory.

Now type the three command lines exactly as you see them in figure 3-1. Use the Return key to start a new line, but don't press the Enter key (the execution starter) yet.



```
asm -p Coin.a
link -p Coin.a.o -o Coin
Coin
```

Figure 3-1

Is it time to create the program from the source code?

You are going to execute each of these command lines individually. To do this, place the insertion bar on the same line as the command you want executed, then press the Enter key. If you select all three lines, then press Enter, all three lines are executed. However, this would make the assembly process more difficult to follow.

If you want to know more about the asm and link commands, turn to the dictionary in part 3. The dictionary explains the necessary parameters and the available minus sign (–) options. Also, part 2, the assembly language tutorial, explains assembly programming in much more detail.

Any errors found while assembling or linking cause messages to be displayed in the Worksheet. If an error message specifies a line number, click back on the Coin.a window and make sure the code appears exactly as shown in figure 3-1. If an error message states *file not found,* make sure you have your disk and folders set up as shown in chapter 1. Also, make sure your current default directory is set such that the Coin.a file can be found.

Now execute the first command line in the Worksheet: asm -p Coin.a. The

-p on the command line is an instruction that outputs a progress report of the assembly (also called the *compilation*) to the Worksheet. At the end of this chapter is a listing of the entire, error-free, assembly and link progress report.

In addition to the progress report, an error message report is output to the Worksheet if any errors are found. The error message report tells you in its last line how many errors were encountered.

Correct any assembly errors before attempting to execute the link command line. Oftentimes, a single error in one line of code produces a string of error messages that scroll down your screen and specify every line of code as an error. If this happens, just correct the first error reported in the source code, go back to the Worksheet, and try executing again.

Now execute the second command line: link -p Coin.a.o -o Coin. A progress report of the link is output to the Worksheet. If all goes well, this command produces your completed standalone application named Coin.

You might be wondering what is being linked to what, because the idea of a link normally involves more than one part. For now, assume that your assembled program is linked to internal files, which results in a standalone application. Although the Link tool can link more than one assembled program into a single application, here you are linking a single file.

Correct any linking errors before attempting to execute the third command line. The progress report tells you about any errors that are encountered. Usually linking errors are the result of the linking program being unable to find the designated files. Proper use of pathnames, either explicitly or set by a directory command, solves this problem.

Finally, execute the third command line: Coin. This command executes the standalone application just as if you had double-clicked on the application from the Finder. The MPW Shell closes all its windows and gives complete control of the Macintosh to your application.

Before MPW closes any file that has unsaved changes, the Save As... dialog box will appear. You can circumvent this dialog box by choosing Save from the File menu after you make a change.

The CorneredCoin program simulates a coin toss. Put the cursor over the cornered coin and it will flip (invert). Move the cursor away and, unless your reflexes are superhuman, it's a fifty-fifty chance whether the coin will be BlackHeads or WhiteTails.

To exit the application, press the mouse button. The CorneredCoin program exits to the Shell, returning you to the Worksheet just as you left it. You can use the files command to see your new application listed among the contents of your disk.

If you choose Quit from the Shell's File menu, you return to the Finder. On the desktop, you will find the CorneredCoin application, Coin. In addition to the application icon, the assembly process creates an intermediary file, Coin.a.o. When you executed asm, Coin.a.o—a binary compilation of Coin.a—was created. When you executed link, it was actually Coin.a.o that was linked into an application.

After an application is created, the source code file and the intermediary file are not needed for the application to run. You can drag your application icon to any disk, and the program will work the same. That is what is meant by a standalone application. However, you'll always want to keep your source files in case you want to alter the program.

Figure 3-2 is the progress report of the compilation and link of the Coin program.

```
asm -p Coin.a

MC68020 Assembler - Ver 2.34 (4/1/87)                    12:29:50 09-Jul-87
Copyright Apple Computer, Inc. 1984-1987
All rights reserved.

...reading Coin.a
...including Silky:MPW:AIncludes:Traps.a
...continuing with Coin.a
#0001

Elapsed time: 5.35 seconds.

Assembly complete - no errors found.  1012 lines.


link -p Coin.a.o -o Coin
MC68000 Linker - v. 2.0B1 Release March 16, 1987          Start: 12:30:53 PM 7/9/8

 Copyright Apple Computer, Inc. 1985, 1986, 1987
 All rights reserved.

Reading files:
  1 "Coin.a.o"

Doing active analysis.
  Max. depth of search: 1
Size of global data area: 0

Input summary:
  Read   Max             Bytes
     6       Strings
     1       Str Blks      2054
     4 16383 Symbols        200
     1       ID-Sym Blks    128
     1       Files
     2       Segments
     1       Modules
     0 32766 Ref. Lists
   Total bytes:     2382

1 active and 1 visible entries of 1 read.
  2 segments, 1 Jump Table entries.
  No data initialization.
  1   154 Main

link completed normally

There were 0 errors.

Execution required 6 seconds.

Coin
```

Figure 3-2

At this point, you might want to print a copy of Coin.a. The following command, executed from the Worksheet window, prints Coin.a in standard ImageWriter quality:

```
print   -q standard   Coin.a
```

CHAPTER

4

Now How About Pascal and C?

Which files do I need for Pascal and C development?

The following screen illustrations show you the contents of the version 2.0B1 disks for MPW Pascal (figure 4-1) and MPW C (figure 4-2). The contents of the CIncludes folder is shown separately in figure 4-3 because it is too large to show in the same screen shot as the other folders.

Hard disk users should have the contents of the Pascal, or C, or both Pascal and C floppy disks on their drives in the MPW folder. If you have not done so already, put the programs Pascal, PasMat, PasRef, and C in the Tools folder. The folders from the Pascal and C disks should be left intact in the MPW folder.

MPW Pascal stores its ROM/Toolbox routines in Interface and Library files; MPW C stores its comparable routines in Include files. Although these files have different names, all serve essentially the same purpose of storing names and addresses of prewritten pieces of code (or, in the case of Libraries, the code itself) that your programs will use.

The remainder of this section is for floppy disk users using MPW version 1.0 software, though even hard disk users might want to take note of the floppy disk setup to become familiar with minimum configurations needed to construct the sample programs.

Floppy disk users need to build dedicated Pascal or C disks in the same way that the dedicated assembly language disk was built in chapter 1. The 800K disk named MPW: does not change. Only the second disk of the two-disk set is new.

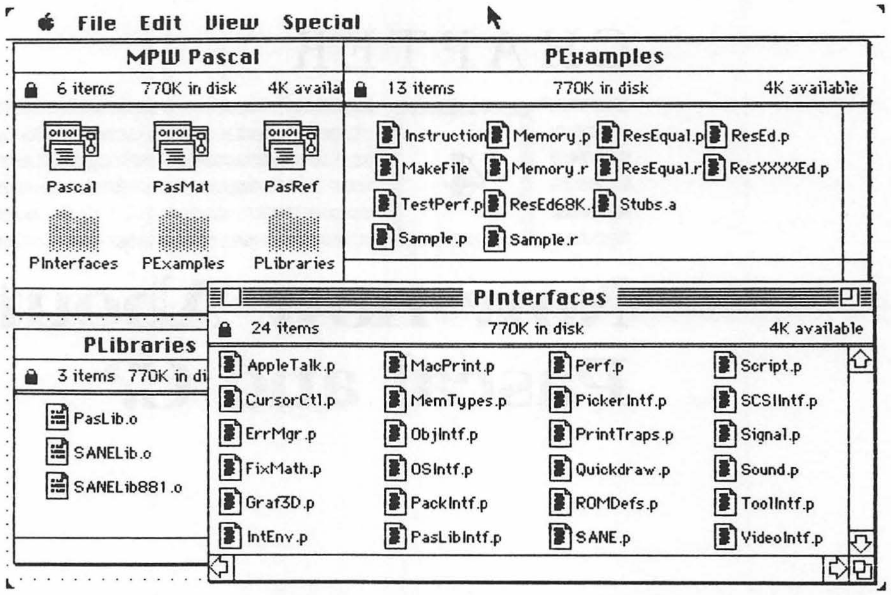


Figure 4-1

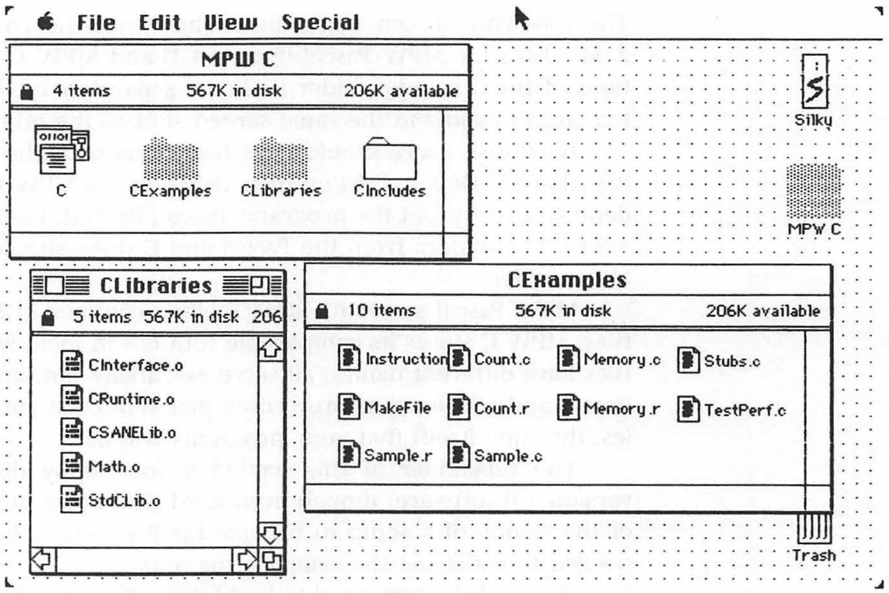


Figure 4-2

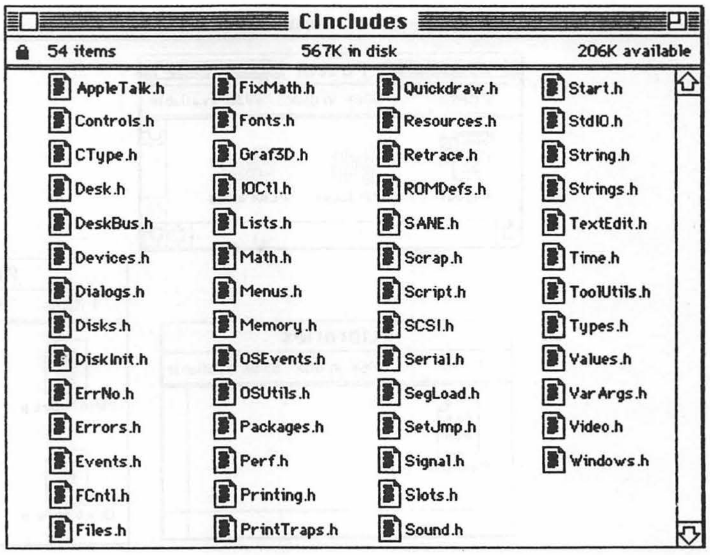

```
CIncludes
54 items              567K in disk              206K available

AppleTalk.h    FixMath.h      Quickdraw.h    Start.h
Controls.h     Fonts.h        Resources.h    StdIO.h
CType.h        Graf3D.h       Retrace.h      String.h
Desk.h         IOCtl.h        ROMDefs.h      Strings.h
DeskBus.h      Lists.h        SANE.h         TextEdit.h
Devices.h      Math.h         Scrap.h        Time.h
Dialogs.h      Menus.h        Script.h       ToolUtils.h
Disks.h        Memory.h       SCSI.h         Types.h
DiskInit.h     OSEvents.h     Serial.h       Values.h
ErrNo.h        OSUtils.h      SegLoad.h      VarArgs.h
Errors.h       Packages.h     SetJmp.h       Video.h
Events.h       Perf.h         Signal.h       Windows.h
FCntl.h        Printing.h     Slots.h
Files.h        PrintTraps.h   Sound.h
```

Figure 4-3

The floppy disk configurations in the next set of figures contain the necessary files to create the programs in this book. Users with two 800K drives can have a complete set of data files (you can copy the Library, Interface, and Include files to your floppy). 800K users might want to provide increased disk space for programs by removing data files that will not be used.

Note: For Pascal programmers, the minimum configuration will fit on an 800K and a 400K drive, but you may not have enough room left on the disk to create the sample Pascal application shown in this chapter. C programmers will have plenty of room left for their sample program.

Figure 4-4 shows the core files that a floppy disk user should put on a dedicated Pascal disk. Figure 4-5 shows the core files that a floppy disk user should put on a dedicated C disk.

To verify the contents of your disks, type *files -r* in the Worksheet window, then press the Enter key. The -r option displays the contents of directories.

Remember, the Enter key executes the Shell command on the same line as the insertion bar. If you press the Return key before the Enter key, the insertion bar will be on a blank line and nothing will be executed.

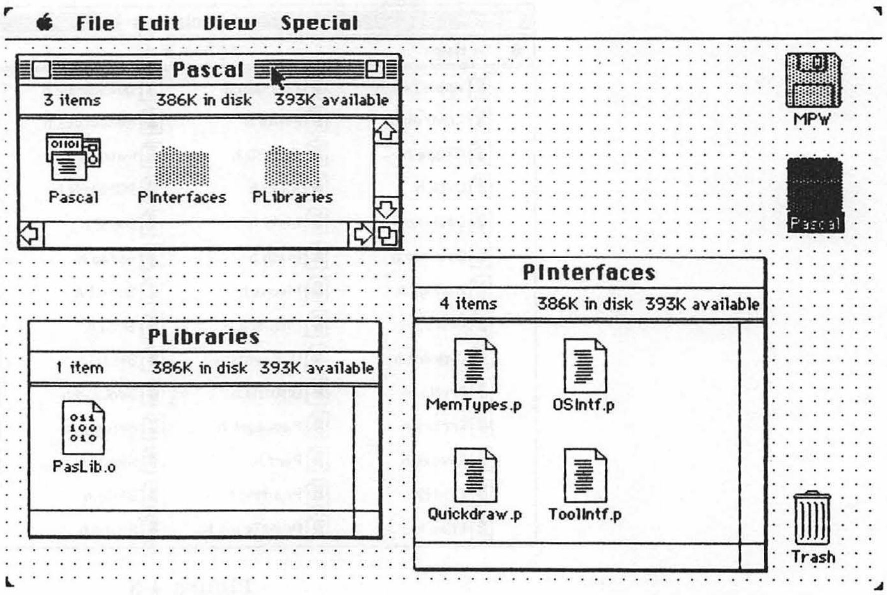


Figure 4-4

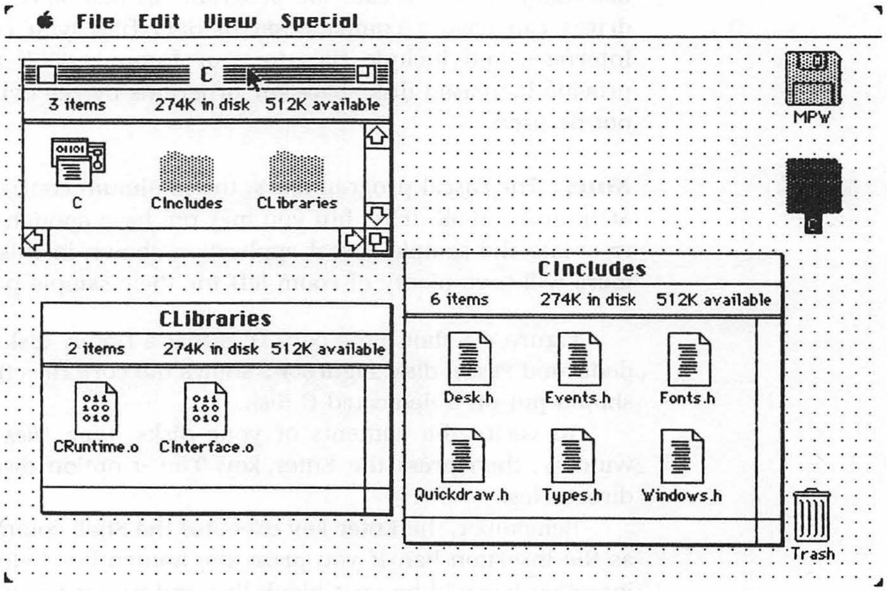


Figure 4-5

Floppy disk users should make the Pascal or C volume the default directory by executing the directory Pascal: or directory C: command. An error message stating that a file or directory cannot be found often occurs because the Shell is searching in a different directory than the one you want.

Can I write a Pascal or C program now?

At this point, the instructions for creating a Pascal or C program are virtually the same as those for creating an assembly program. The single difference is the suffix appended to the text file that holds the source code of your sample program. This suffix is important for identifying your source code. By convention, assembly source uses the .a suffix, Pascal source uses the .p suffix, and C source uses the .c suffix.

To avoid repetition and possible confusion, the rest of this chapter shows instructions using the .p suffix (for Pascal programmers). C programmers should follow the same instructions, but substitute .c for the .p suffix.

Start with a fresh Worksheet. If you don't want to disturb the contents of your previous Worksheet, execute the command

```
new Coin.p
```

to create a blank window titled Coin.p in which to enter your program. Or easier yet, select New from the File menu and type the name *Coin.p* into the resulting dialog box.

Figure 4-6 shows the blank Worksheet window for your Pascal program. Figure 4-7 is the Worksheet window for your C program.

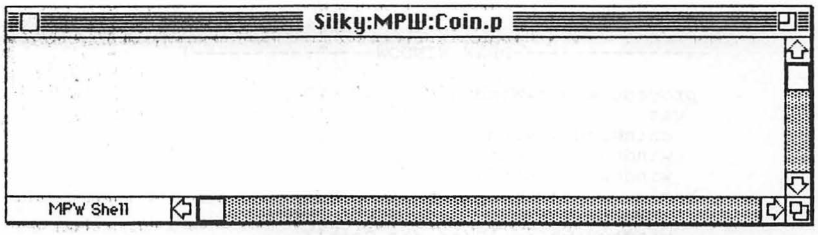


Figure 4-6

In this section you will type, compile, link, and run a Pascal (or C) program that simulates a coin toss. Creating a Pascal or C program requires the same four steps as creating an assembly program:

1. Knowing the directory location of all required files.
2. Entering the program code in a new Worksheet window.
3. Entering a sequence of compile and link commands in the original Worksheet window.
4. Executing the command sequence.

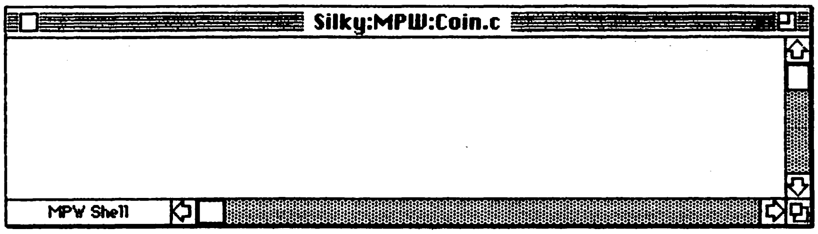


Figure 4-7

Pascal programmers should type the program in listing 4-1. C programmers should type the program in listing 4-2. An important difference between MPW Pascal and MPW C is the use of uppercase and lowercase letters. Programs in Pascal are not case sensitive; use uppercase letters wherever you think it makes code easier to read. Programs in C are case sensitive; the sample program should be typed exactly as shown to avoid errors. Although the indentation and spacing of Pascal and C program code does not necessarily affect execution, it is important for program clarity. Use the Tab key to make the code line up in columns.

Listing 4-1

```
Program CorneredCoin;
  uses
    Memtypes, Quickdraw, OSIntf, ToolIntf;
  var
    r:rect;

{----------------DRAW WINDOW----------------}

  procedure drawWindow;
   var
    coinWindow:windowPtr;
    windowSize:rect;
    windowName:str255;
  begin
    setRect(windowSize,60,80,450,290);
    windowName := 'Cornered Coin';
    coinWindow := newWindow(nil,windowSize,windowName,true,0,pointer(-1),false,0);
    setPort(coinWindow);
  end;

{----------------DRAW COIN------------------}

  procedure drawCoin;
  begin
    setRect(r,100,80,290,130);
    frameRect(r);
    moveTo(122,111);
    drawString('BlackHeads/WhiteTails')
  end;
```

Listing 4-1 *cont.*

```
{----------------FLASH COIN----------------}

  procedure flash;
    var
      pt:point;
  begin
    systemTask;
    getMouse(pt);
    if ptInRect(pt,r) then
      invertRect(r)
  end;

{-----------------MAIN--------------------}

begin
  initGraf(@thePort);
  initFonts;
  initWindows;
  initCursor;
  drawWindow;
  drawCoin;
  repeat
    flash
  until button
end.
```

Listing 4-2

```
/*  Cornered Coin  */

#include <types.h>
#include <quickdraw.h>
#include <fonts.h>
#include <events.h>
#include <windows.h>
#include <desk.h>

Rect    r;


/*--------------------- DRAW WINDOW ---------------------*/

void    drawWindow() {
    Rect        windowSize;
    WindowPtr   coinWindow;
    char        *windowName;

    SetRect(&windowSize,60,80,450,290);
    windowName = "Cornered Coin";
    coinWindow = NewWindow(nil,&windowSize,windowName,true,0,(WindowPtr)-1,true,0)
    SetPort(coinWindow);
}

/*--------------------- DRAW COIN ---------------------*/

void    drawCoin() {
    SetRect(&r,100,80,290,130);
    FrameRect(&r);
    MoveTo(122,111);
    DrawString("BlackHeads/WhiteTails");
}

/*--------------------- FLASH COIN ---------------------*/

void    flash() {
    Point       pt;
```

Listing 4-2 *cont.*

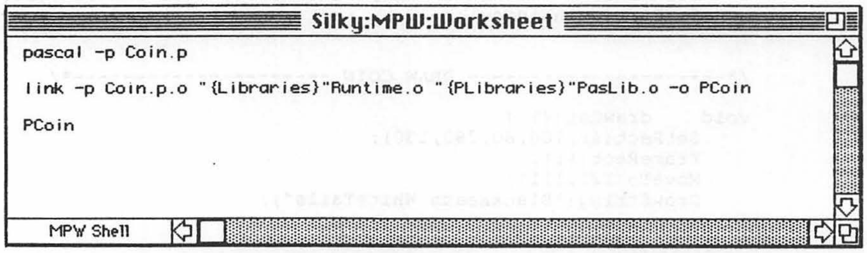
```
        SystemTask();
        GetMouse(&pt);
        if (PtInRect(&pt,&r)) {
            InvertRect(&r);
        }
}

/*--------------------- MAIN ---------------------*/

main()
{
        InitGraf(&qd.thePort);
        InitFonts();
        InitWindows();
        InitCursor();
        drawWindow();
        drawCoin();
        while(!Button()) flash();
}
```

As you saw with your assembly language source code, text typed in a Worksheet is treated only as text, and thus no error checking is performed. A program is checked for errors only in the later stages of compiling, linking, and running.

Save the program code onto disk under the name Coin.p (or Coin.c for C code). The File menu offers the simplest way to save your program code, though you could return to the original Worksheet and execute the command save Coin.p.

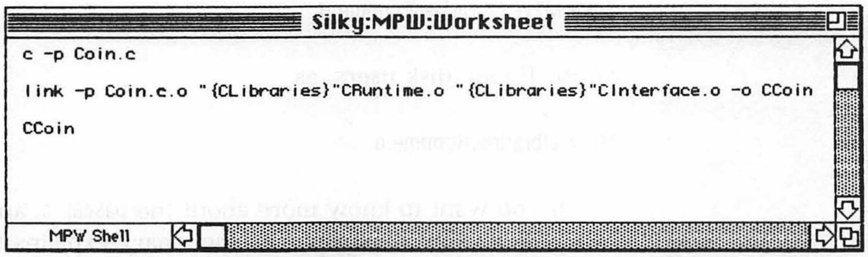
The instructions in the remainder of this chapter are nearly identical to those you used in creating your assembly program. The MPW environment changes very little among assembly, Pascal, and C. The only significant differences are the source code, the longer link command line, and, for floppy disk users, the file contents of your second work disk.

Now you are ready to assemble and link your code. First, bring the Worksheet window back on top (click on it). Now type the three command lines exactly as you see them in the following Pascal illustration (figure 4-8) or C illustration (figure 4-9). Don't press the Enter key (the execution starter) just yet.



```
pascal -p Coin.p
link -p Coin.p.o "{Libraries}"Runtime.o "{PLibraries}"PasLib.o -o PCoin
PCoin
```

Figure 4-8



```
c -p Coin.c

link -p Coin.c.o "{CLibraries}"CRuntime.o "{CLibraries}"CInterface.o -o CCoin

CCoin
```

Figure 4-9

You will execute each of these command lines individually. To do this, place the insertion bar on the same line as the command you want executed, then press the Enter key. If you select all three lines and then press Enter, all three lines would be executed. However, this would make the compilation process more difficult to follow.

You might be wondering why the `link` command line has so many words, braces, and quotation marks. Here are some notes that should help you understand.

The compiling process (performed by the `pascal`, `c`, and `asm` commands) translates English-like source code into an intermediary file of binary object code. The generated object code file name is the source code file name with an .o suffix.

The linking process for Pascal and C involves libraries of precompiled code (object files). These libraries hold general-purpose code that helps support the Macintosh environment and other tasks involved in creating a standalone application. Relevant pieces of these library files are joined (linked) with your intermediary object code.

The library files Runtime.o, PasLib.o, CRuntime.o, and CInterface.o are included with your MPW core and language disks, and bear the distinctive binary icons filled with 0s and 1s. MPW contains other library files whose object code in not used by your sample programs. Thus, these files are not included in the link process.

The peculiar notation of quotation marks, braces, and the words *Libraries, PLibraries,* and *CLibraries* is MPW shorthand for directory pathnames. This shorthand notation is set in the Startup files, a topic discussed in chapter 5. For example, the shorthand pathname

```
"{Libraries}"Runtime.o
```

used in the `link` command line of the Pascal Worksheet could be written for hard disk users as

```
Silky:MPW:Libraries:Runtime.o
```

or, for floppy disk users, as

```
MPW:Libraries:Runtime.o
```

If you want to know more about the pascal, c, and link commands, turn to the dictionary in part 3. The dictionary explains the necessary parameters and the minus sign (–) options.

Any errors found while compiling or linking will cause messages to be output to the Worksheet. If an error is detected, click back on the Coin.p window and make sure the code appears exactly as you see it in listing 4-1 or 4-2.

Pascal programmers should now execute the first command line in their Worksheet: pascal -p Coin.p. C programmers should now execute the first command line in their Worksheet: c -p Coin.c.

Don't confuse the -p option with the .p Pascal suffix. The -p option is available to both Pascal and C programmers. It is an instruction to the compiler (called a *directive*) that outputs a progress report of the compilation to the Worksheet.

Correct any compilation errors before you attempt to execute the link command line. The error report, in its last line, tells you how many errors (if any) were encountered.

Pascal programmers should now execute the second command line: link -p Coin.p.o "{Libraries}"Runtime.o "{PLibraries}"PasLib.o -o PCoin. C programmers should now execute their second command line: link -p Coin.c.o "{CLibraries}"CRuntime.o "{CLibraries}"CInterface.o -o Coin. Again, the -p option outputs a progress report of the link to the Worksheet. If all goes well, this command produces your completed standalone application, named PCoin for a Pascal program and CCoin for a C program.

Correct any linking errors before attempting to execute the third command line. If any errors are encountered, the linking process halts and error messages are output onto the Worksheet. Usually, linking errors result when the linking program is unable to find the designated files. Proper use of pathnames and making sure the designated files are on your disk solve this problem.

Finally, Pascal programmers should execute the third command line: PCoin. C programmers should execute their third command line: CCoin. These commands execute the standalone application just as if you had double-clicked on the application icon. The MPW Shell closes all of its windows and gives complete control of the Macintosh to your application.

The Pascal and C Coin programs function identically to the assembly version. Put the cursor over the cornered coin and it flips (inverts). Move the cursor away and it's a fifty-fifty chance whether the coin will be BlackHeads or WhiteTails.

To exit the application, press the mouse button. The Coin program exits to the Shell, returning you to the Worksheet just as you left it. You can use the files command to see your new application listed among the contents of your disk.

If you choose Quit from the Shell's File menu, you are returned to the desktop. On the desktop, you will find the Coin application icon. In addition to the application icon, the Pascal compilation process has created the intermediary file, Coin.p.o.

Once again, when you executed the pascal command line, Coin.p.o—a binary compilation of Coin.p—was created. When you executed link, it was actually Coin.p.o that was linked into an application.

The same thing occurs in the C compilation process. After choosing Quit from the Shell's File menu, you return to the Finder. In addition to the Coin application icon, the intermediary file, Coin.c.o, is displayed. When you executed the c command line, Coin.c.o—a binary compilation of Coin.c—was created. When you executed link, it was actually Coin.c.o that was linked into an application.

After an application is created, the source code file and the intermediary file are not needed to run the application. You can drag your application icon to any disk, and the program will work the same. That is what is meant by a standalone application. However, you'll always want to keep your source files in case you want to alter the program.

CHAPTER

5

What's the Story with Startup and Files in General?

What does a programmer need to know about files?

The hardest part of using files is finding them. After you know the location of a file, it is easy to open, close, change, delete, duplicate, move, rename, and save it. File handling commands allow you to perform all these tasks from a Worksheet window. But first you must find the file.

Certain file commands offer help in locating files. You used the command `files` to list the contents of a particular disk and `volumes` to list the volumes (disk drives) that are on-line. With this information, you should be able to construct the correct pathnames that represent every MPW file.

HFS uses pathnames to access a file. Remember, simple pathnames have the following format:

volumeName:fileName

If the file is within a directory (folder), the pathname format is

volumeName:directoryName:fileName

If the directory is within another directory, the pathname format is

volumeName:outerDirectory:innerDirectory:fileName

If you have more than two directories, they must be listed in the order that they are nested.

As you might guess, even with a single directory, pathnames can get unwieldy. MPW offers many shortcuts for dealing with pathnames. Here are some hints for dealing successfully with pathnames:

- Full pathnames always begin with a volume (disk) name that ends with a colon. A colon does not precede a volume name because there is no preceding layer to a volume. A volume is the file system's starting point.
- Partial pathnames can begin with a directory name, a file name, or colons. Because volume names must end with a colon, a name that does not end with a colon must be a directory or file name.
- HFS searches for files by using partial pathnames appended to the current default directory.
- In the MPW command language, directory (or Show Directory from the Directory menu) displays the current default directory; volumes displays the volumes currently on-line; files displays the files in the default volume.
- You can change the default directory with the directory or setDirectory command. Beginners, however, may find it easier to supply full pathnames rather than worry about current defaults.
- MPW allows you to define Shell variables. The set command equates a variable name with a string value. MPW uses system-wide variables (set in the Startup file is explained later in this chapter) to search for its files. These variables work like partial pathnames. Users can set system-wide variables by defining them in the UserStartup file.

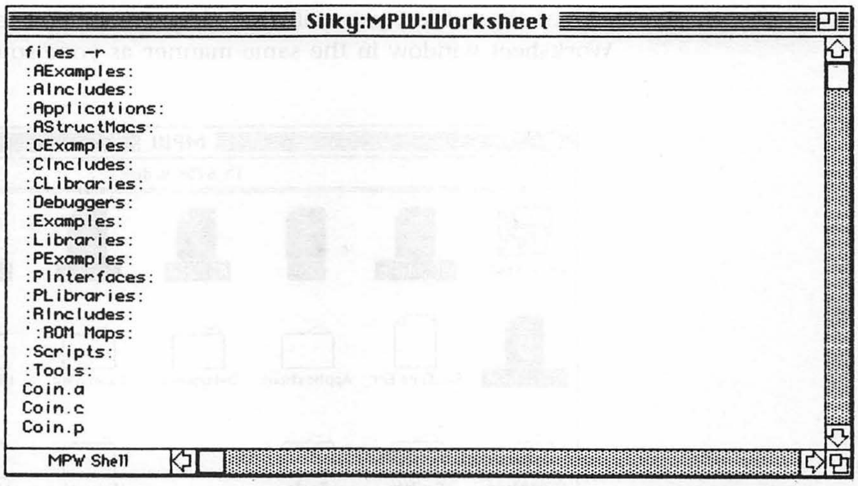
Experiment with the file handling commands. Use part 3's dictionary to learn more about a command's options and parameters. For example, look at figure 5-1 to see the file command's -r option in use, then look up file in the dictionary to see what the option is doing.

How do you deal with data files and tools?

By now you have seen that a large percentage of MPW's files are data files that allow the assembly, Pascal, and C languages to access the Macintosh ROM and the System file. These data files are in directories that have *Include, Interface,* or *Library* in their names.

A programmer will rarely, if ever, have to enter and change one of these files. The programmer's responsibility is to make sure the appropriate data file is on-line and available when it is needed.

For example, if an assembly program uses a trap call from the Toolbox, the file Traps.a must be on a disk and included (using the assembly directive



```
files -r
 :AExamples:
 :AIncludes:
 :Applications:
 :AStructMacs:
 :CExamples:
 :CIncludes:
 :CLibraries:
 :Debuggers:
 :Examples:
 :Libraries:
 :PExamples:
 :PInterfaces:
 :PLibraries:
 :RIncludes:
 ':ROM Maps:
 :Scripts:
 :Tools:
Coin.a
Coin.c
Coin.p
```

Figure 5-1

Include) in the program code. Likewise, if the program uses a system name such as ScrnBase, the data file that defines the name (SysEqu.a) must be on-line and included with a code directive.

The other MPW files serve a variety of purposes. In addition to the System Folder, there are languages, example files, debuggers, map files, command tools, general command files, Pascal tools, ResEdit, the MPW Shell, and a small selection of Shell command files.

In the Tools folders, you will find command tools that work in the same way as the built-in commands of the MPW Shell. From the user's standpoint, the single requirement for using external tools is that the file exists in the Tools folder. This presents no problem for hard disk users, who should have plenty of disk space, but floppy disk users have to pick and choose among tools they want to access.

If you try to perform a Shell command that requires a file that cannot be found in the Tools folder, you will get an error message to that effect. All the files that belong in the Tools folder are shown in chapter 1's screen illustrations, and are explained in the dictionary. (The Line file is not a tool, but a command file that selects the line of an error.)

What's a command file and why does Startup get top billing?

Look at figure 5-2 to see which command files are given a special position in the same folder as the MPW Shell. You might recognize the icon for each of these files as that of the standard Shell Worksheet file. If you double-click on any of these files, you will find they are indeed text files displayed in a Worksheet window in the same manner as your source code files.

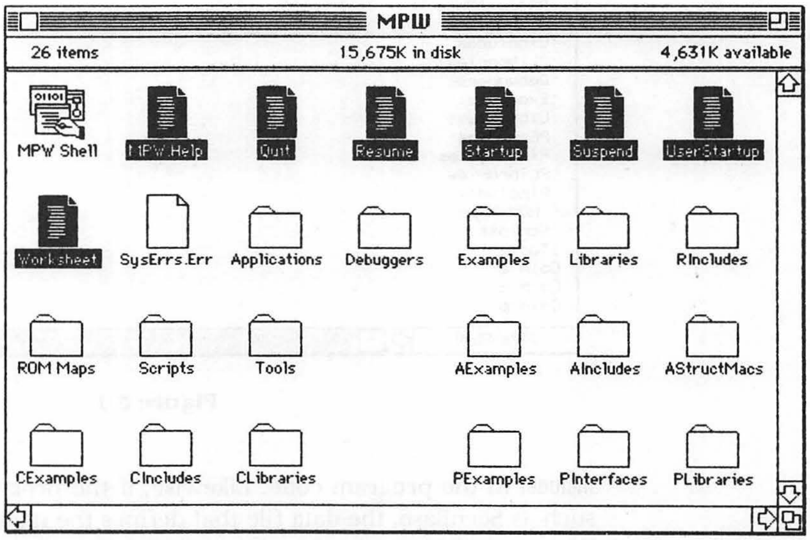


Figure 5-2

As you saw in the first three chapters, Worksheet files can be saved under any file name. When a file is opened, the Worksheet window is titled with the file name. It's still an ordinary text file with most of the characteristics of the original Worksheet. All text files are actually Worksheet clones. Because each file opens to its own window, the concept of a window and a file are, for the most part, synonymous.

If you enter program code into a Worksheet window, you might call the file Program Source Code. If you enter the recipe for Mrs. Fields's chocolate chip cookies into a Worksheet window, you might call the file Recipe Text. If you enter a list of commands from the MPW command language, you might call the file Command File.

The contents of a Worksheet window have no particular significance to MPW until you try to execute the contents. When you supply program source code as input to a command that says "execute language," the text is evaluated as you would expect. If you supply Mrs. Fields's cookie recipe to a language command, you will get a long list of syntax errors.

To execute a command file without error, it must follow the syntax expected by the MPW command language. With this requirement filled, MPW executes a command file just as it executes a single command typed into the Worksheet. The ability to put a series of commands into a text file, then execute all the commands at once, gives MPW programmers a convenient means to perform repetitive tasks.

Command files (also called *scripts*) can be executed in a number of ways. The most direct method is to enter the command file name in a Worksheet window, then press the Enter key. This executes all the commands within the file just as if the command file name was part of the command language.

Did you catch that? *Command files are executed in the same way as individual commands—the name of the file is used to execute every command contained within the file.*

The Worksheet command files titled Startup, UserStartup, Quit, Suspend, and Resume are given prominent position with MPW because they serve a special purpose to the functioning of the MPW Shell. There is nothing intrinsically special about the composition of these files—they are simple text made up of commands. However, the Shell uses these command files to create the following aspects of your programming environment:

- The Startup and UserStartup files are automatically executed when you run the Shell application. UserStartup allows the user to insert Startup commands without altering the original Startup file. The last section of this chapter goes into this in more detail.
- The Quit file is automatically executed when you exit from the Shell and return to the Finder.
- The Suspend and Resume files are automatically executed when you temporarily exit from the Shell to run an application.

As long as these files retain their given names and remain in the same directory as the MPW Shell, they execute automatically. At this point, they can be forgotten. You do not need to open, change, or otherwise investigate any of these files to use MPW.

Tell me what Startup does or do you want your wagon fixed?

Open the file Startup. You can do this by double-clicking on the Startup icon from the Finder or by typing *Open Startup* in the Worksheet window and pressing Enter.

Figure 5-3 is a printout of the Startup file. (Startup version 1.0 for floppy disk users is slightly different.)

The listing looks long and complicated. But if you remove all the comment lines (they begin with a number sign, #), you will find only four different types of commands: set, export, alias, and execute. Figure 5-4 is a printout of Startup without comments.

```
#   Startup - MPW Shell Startup File
#
#   Copyright Apple Computer, Inc. 1985-1987
#   All Rights Reserved.


#   {Boot} - The boot disk.  (Predefined.)
            Export Boot

#   {SystemFolder} - The directory that contains System & Finder.  (Predefined.)
            Export SystemFolder

#   {ShellDirectory} - The directory that contains MPW Shell.  (Predefined.)
            Export ShellDirectory

#   {Active} - The active (topmost) window.  (Predefined.)
            Export Active

#   {Target} - The target (previously active) window.  (Predefined.)
            Export Target

#   {MPW} - The volume or folder containing the Macintosh Programmer's Workshop.
            Set MPW "{Boot}MPW:"
            Export MPW

#   {Commands} - Directories to search for commands.
            Set Commands ":,{MPW}Tools:,{MPW}Scripts:,{MPW}Applications:"
            Export Commands

#   {AIncludes} - Directories to search for assembly language include files.
            Set AIncludes "{MPW}AIncludes:"
            Export AIncludes

#   {Libraries} - Directory that contains shared libraries.
            Set Libraries "{MPW}Libraries:"
            Export Libraries

#   {CIncludes} - Directories to search for C include files.
            Set CIncludes "{MPW}CIncludes:"
            Export CIncludes

#   {CLibraries} - Directory that contains C libraries.
            Set CLibraries "{MPW}CLibraries:"
            Export CLibraries

#   {PInterfaces} - Directories to search for Pascal interface files.
            Set PInterfaces "{MPW}PInterfaces:"
            Export PInterfaces

#   {PLibraries} - Directory that contains Pascal libraries.
            Set PLibraries "{MPW}PLibraries:"
            Export PLibraries

#   {RIncludes} - Directory that contains Rez include files.
            Set RIncludes "{MPW}RIncludes:"
            Export RIncludes

#   {CaseSensitive} - If non-zero, pattern matching is case sensitive.
            Set CaseSensitive 0
            Export CaseSensitive

#   {Tab} - Default tab setting for new windows.
            Set Tab 4
            Export Tab

#   {WordSet} - Character set that defines words for searches and double-clicks.
            Set WordSet 'a-zA-Z_0-9'
            Export WordSet

#   {PrintOptions} - Options used by the Print Window and Print Selection menus.
            Set PrintOptions '-h'

#   {Exit} - If non-zero, command files terminate after the first error.
            Set Exit 1
            Export Exit
```

Figure 5-3

```
#    {Echo} - If non-zero, commands are echoed before execution.
             Set Echo 0
             Export Echo

#    {Test} - If non-zero, tools and applications are not executed.
             Set Test 0
             Export Test

#    Commando Support
             Export Windows
             Export Aliases
             Set Commando Commando
             Export Commando

#    Aliases
             Alias File Target


#    The file UserStartup can be used to override definitions made in Startup,
#    or to define additional variables, exports, and aliases.  UserStartup may
#    also be used to define menu items, open windows, etc.  The file should be
#    located in the directory containing the MPW Shell.

             Execute "{ShellDirectory}UserStartup"
```

Figure 5-3 *(continued)*

Here are the general duties performed by the Startup file:

1. Certain volume, directory, and file variable names predefined in the Shell application are *exported* so that they are recognized as entities available to the command language.
2. Certain variable names are *set* to be equivalent to specified string values.
3. A command name is given an *alias* that serves as an alternate name for a word or a list of words.
4. A command file is *executed* by specifying its pathname as a parameter. (The execute command provides another way of executing a command file.)

You can find out more about export, set, alias, and execute by looking up their definitions in the command dictionary in part 3. Pay particular attention to the set commands because they illustrate the use of pathnames (and represent the only difference between the hard disk and floppy disk versions of Startup). Many of Startup's set commands provide shortcut pathnames.

For example, the hard disk version contains the following command lines:

```
Set Libraries "{MPW}Libraries"
Export Libraries
```

The floppy disk version performs the same task with these lines:

```
Set Libraries "MPW:Libraries"
Export Libraries
Export Boot

Export SystemFolder

Export ShellDirectory

Export Active

Export Target

Set MPW "{Boot}MPW:"
Export MPW

Set Commands ":,{MPW}Tools:,{MPW}Scripts:,{MPW}Applications:"
Export Commands

Set AIncludes "{MPW}AIncludes:"
Export AIncludes

Set Libraries "{MPW}Libraries:"
Export Libraries

Set CIncludes "{MPW}CIncludes:"
Export CIncludes

Set CLibraries "{MPW}CLibraries:"
Export CLibraries

Set PInterfaces "{MPW}PInterfaces:"
Export PInterfaces

Set PLibraries "{MPW}PLibraries:"
Export PLibraries

Set RIncludes "{MPW}RIncludes:"
Export RIncludes

Set CaseSensitive 0
Export CaseSensitive

Set Tab 4
Export Tab

Set WordSet 'a-zA-Z_0-9'
Export WordSet

Set PrintOptions '-h'

Set Exit 1
Export Exit

Set Echo 0
Export Echo

Set Test 0
Export Test

Export Windows
Export Aliases
Set Commando Commando
Export Commando

Alias File Target

Execute "{ShellDirectory}UserStartup"
```

Figure 5-4

Both set command lines equate the variable name Libraries with the string value in quotation marks. The hard disk version uses braces to specify the location (partial pathname) of the file. The floppy disk version uses the colon separator. The second lines export the name Libraries so that its value is recognized in all command files.

Braces indicate a variable substitution. Quotation marks, in this usage, delimit (form the boundaries of) the variable substitution.

To go one step further, MPW is itself a variable name defined in a preceding set command line. If you look at the first set command in the Startup printouts, you will see that MPW is equated with

```
"{Boot}MPW:"
```

designating the starting volume. If you want to make any changes to the way the Startup command file creates your working environment, you can add your own command instructions in the file called UserStartup. UserStartup works under the same premise as Startup. It provides the MPW Shell with starting instructions that initialize and tailor the MPW command environment. It allows the user to provide a custom Startup command script without altering the original Startup file.

The last command in the Startup file runs UserStartup. As a result, the commands entered in UserStartup override those used in Startup.

In the version 2.0B1 release of MPW, the UserStartup file contains two commands, DirectoryMenu and BuildMenu, and lots of comments prefaced with the # command designator. These two commands execute command files (located in the Scripts folder) that add two menus to the Shell menu bar. You will read more about these commands in the next chapter and can look them up in part 3's dictionary.

Figure 5-5 is a printout of UserStartup in its initial state.

```
#   UserStartup - MPW Shell UserStartup File
#
#   Copyright Apple Computer, Inc. 1985-1987
#   All Rights Reserved.

#   This file (UserStartup) is executed from the Startup file, and can be used
#   to override definitions made in Startup, or to define additional variables,
#   exports, and aliases.  UserStartup may also be used to define menu items,
#   open windows, etc.  The file should be located in the directory containing
#   the MPW Shell.


#   Create the Directory menu

#   The parameters to DirectoryMenu become the initial list of directories
#   in the Directory menu.  The parameters below specify each of the
#   Examples directories, and the current directory.  Replace them with
#   your favorite directories.

    DirectoryMenu `(Files -d -f "{MPW}"≈Examples≈) ≥ Dev:Null` `Directory`


#   Create the Build Menu

    BuildMenu
```

Figure 5-5

CHAPTER

6

Can You Give Me a Perspective on the Entire Command Language?

Can you summarize what I've learned about files thus far?

MPW is a programming environment composed of a large number of program files and data files. As all Macintosh users know, every file has its own icon in the Finder desktop.

To write computer programs in the MPW environment, you must learn how to manipulate these files. The Finder, itself, allows you to do certain file manipulations such as listing, copying, renaming, and launching. It does not, however, perform other important file tasks such as creating and editing computer programs.

MPW uses a core program, called the Shell, that manipulates files with much more freedom (and much less friendliness) than the Finder. By launching the Shell, you enter a new MPW desktop consisting of a blank Worksheet window. The Worksheet accepts text of all sorts, including file commands that make it unnecessary for an MPW user to return to the Finder.

Starting with a blank Worksheet, the MPW user begins the process of creating computer programs. The process can be as simple as:

1. Typing in the source code of a computer program in assembly, Pascal, or C.
2. Saving the source code using a file name that ends with the suffix .a, .p, or .c (depending on the language used). The saved code is given its own window on top of a blank Worksheet.

3. Clicking within the Worksheet, then typing a two-line sequence of MPW commands that compile and link the source code, and create the standalone application.

The MPW command language also offers complexity for programmers who want to tailor their programming environment. If you are a beginner trying to get started with MPW, you don't want to learn everything. You want to recognize what you can safely avoid learning until later.

A beginner should not expect to know all MPW features before attempting to program. After all, the purpose of a programming environment is to enhance programming, and any time spent learning the environment is time away from programming. A compromise—where the MPW environment interferes the least with your programming effort—is necessary.

Don't forget to take advantage of the help command and the commando dialog box interface. They can save you the time of referencing books when you want to know particulars about the command language. Use help Commands to get a complete listing of available commands, or use a particular command name as the help parameter to get summary information about a single command. Use the commando command when you need dialog box help to select the proper minus sign options and parameters to command lines.

The Finder desktop offers one of the best perspectives on the MPW environment as a whole. Better yet, figure 6-1 is a screen shot of Andy

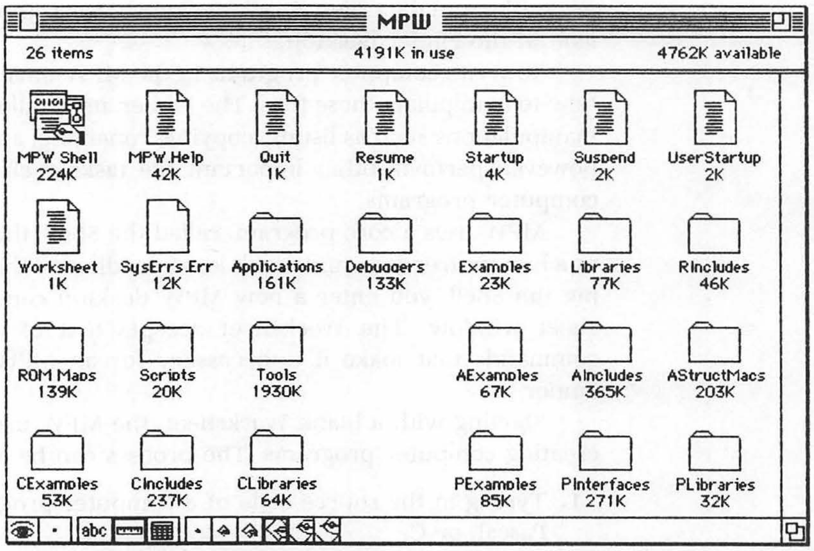


Figure 6-1

Hertzfeld's Servant program (a Finder replacement) that shows sizes below the names of all MPW folders and files.

Your goal as a programmer is to use MPW to create one or more of your own files. Your programming result might be an application, a tool, a desk accessory, or anything else a computer can perform. By getting a handle on the various types of files in MPW, you are well on your way to grasping the capabilities of the command language.

There are four general categories of commands available in MPW. Any of the following commands can be executed from a Worksheet window by typing in the command (or selecting an existing command name) and pressing Enter. A series of commands can be executed as a unit by selecting the names of all the commands and pressing Enter.

Built-In commands

The MPW Shell performs file handling, informational output, editing, and structured sequences through single-word commands. These built-in commands, and their parameters and options, are defined in the command dictionary in part 3.

Tools

Tools are programs that can be executed from the MPW Shell in the same manner as built-in commands. Tools are separate files represented by icons on your disk. As such, tools can be removed, and new tools can be devised and added. The tools that come on the MPW disks are defined in the command dictionary in part 3 and can be accessed through commando dialog boxes.

Command files

As described in chapter 4, command files (also called scripts) are text files composed of commands that can be executed as a unit. The Shell uses command files (for example, Startup and Suspend) to help automate the editing environment. MPW offers other command files (in the Scripts folder) to make certain tasks easier for users. You can also create your own command files.

Applications

The MPW Shell can run standalone programs in the same way as the Finder. The Shell suspends operation while an application is running and resumes control when the application is exited. Applications run outside the MPW environment.

How about a few words on command format and parameter options?

The following dictionary definitions have been taken from part 3 to illustrate typical built-in commands.

Date Display the clock's date and time

```
Date (-a | -s) (-d | -t)
```

Writes the date and time from the Macintosh clock to standard output.

Options

-a Shorten the date notation by using three-character abbreviations for the month and the day of the week.

-d Write only date output.

-s Shorten the date notation by using mm/dd/yy notation and not providing the day of the week.

-t Write only time output.

Rename Rename disk files and directories

```
Rename (-c | -n | -y) name newname
```

Changes the name of a file or directory from *name* to *newname.* If a file or directory using *newname* already exists, a dialog box asks confirmation to overwrite same-name objects.

Options

-c A same-name object conflict halts the command, circumventing a confirmation dialog box.

-n Do not overwrite same-name objects, circumventing a confirmation dialog box.

-y Overwrite same-name objects directly, circumventing a confirmation dialog box.

The following rules will help you use the command language:

- A single word, always stated first, identifies the command. It is the command name, tool name, command file name, or application name. Uppercase and lowercase letters are treated the same.
- Parameters are most often one of two types: options or files. Options are identified by a preceding minus sign (–). Files are given by file names. Some commands use other kinds of parameters such as directories, numbers, text selections, or special strings.
- Commands must be terminated. A Return character usually ends a command, though MPW offers alternative terminators for more complex command operations.
- Text that is preceded by a number sign (#) is treated as a comment. Comments are not executed and end at the next Return character.
- At least one space must separate command names, options, and file names. When a parameter uses a string that contains a space, the string must be within quotation marks.
- Parameters listed in parentheses are optional; others are required. MPW's help command lists optional parameters in brackets.

More on windows, and how come you never mention menus?

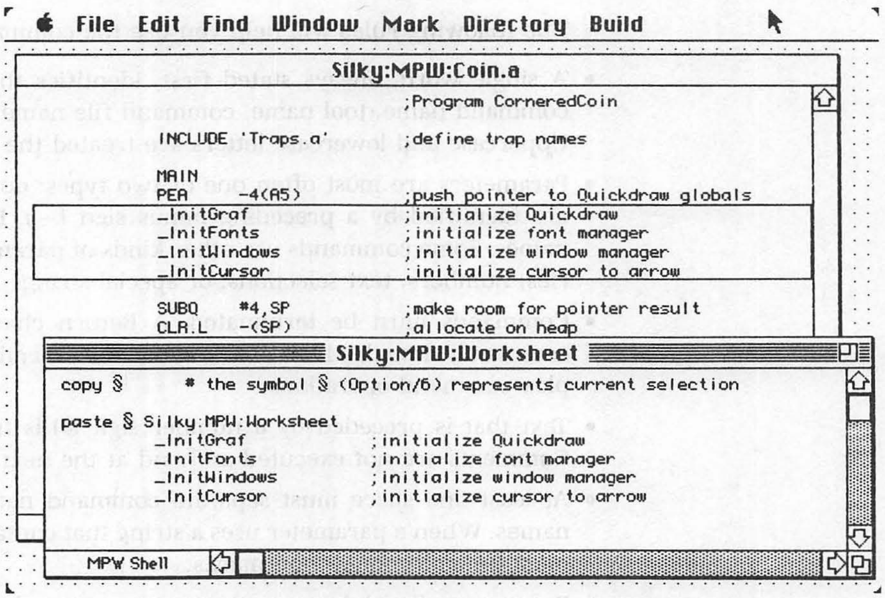
Menu options offer only a small subset of command language capabilities. Almost all of MPW's Shell menu options can also be performed by using the command language. Although the number of options is limited, menus lessen the need for memorization and allow functions to be performed directly on the active window.

Menus act upon the *active* (topmost) window. This contrasts with similarly functioning Shell commands that act upon the *target* window. By default, the target window is the second window from the top.

To illustrate this difference, consider the copy function. When this command is executed from the Edit menu, it copies the selection from the active window. When it is executed from a Worksheet command line, it copies the selection from the window that is layered one below the active window.

Mac users will quickly recognize selections in the active window—they are shown in inverse, white letters on a black background. When such a selection is in a nonactive window, such as the second-from-top target window, the selection is highlighted (boxed) by a rectangular outline. Figure 6-2 shows an example of selections in both the active and target windows.

You can create an example of text highlighting in two windows by opening any two windows (use New from the File menu or new from the Work-



```
                                ;Program CorneredCoin
        INCLUDE 'Traps.a'       ;define trap names

        MAIN
        PEA     -4(A5)          ;push pointer to Quickdraw globals
        _InitGraf               ;initialize Quickdraw
        _InitFonts              ;initialize font manager
        _InitWindows            ;initialize window manager
        _InitCursor             ;initialize cursor to arrow

        SUBQ    #4,SP           ;make room for pointer result
        CLR.L   -(SP)           ;allocate on heap
```

```
copy §          # the symbol § (Option/6) represents current selection

paste § Silky:MPW:Worksheet
        _InitGraf               ;initialize Quickdraw
        _InitFonts              ;initialize font manager
        _InitWindows            ;initialize window manager
        _InitCursor             ;initialize cursor to arrow
```

Figure 6-2

sheet) and arranging them so they do not completely overlap. When you type in text, select the text, then click in the other window, the selected text of the background window becomes highlighted in the boxed fashion.

Again, the command language equivalents to the menu commands operate, by default, on the target window. For example, executing `copy` from the Worksheet copies the boxed selection in the target window onto the Clipboard. Executing copy from the Edit menu copies the inverse selection from the active window onto the Clipboard.

You can override the default use of the target window by providing a window parameter to a Worksheet command. Any other window can be specified as the target window by providing the window's name as a parameter. For example, `copy Coin.a` copies the selection from window Coin.a onto the Clipboard.

Here is a review of the MPW menu commands that may be unfamiliar.

File menu

The File menu's New... command produces a dialog box in which to specify the file name and directory location of the new file you want to create.

The File menu offers three commands to save files. The Save command saves the contents of a file to disk without any change of file name. The Save as... command saves the contents of a file to disk by producing a dialog box in which to specify a new file (and window) name. The Save a copy... command works the same as Save as... except the name of the active window does not change to the new file name. It truly saves a copy without changing your current work file in any way.

The File menu offers an Open Selection item whenever a file name is selected within a window. This is a shortcut that bypasses the Open... command's dialog box.

The Print Window/Print Selection command prints the contents of a window or a selection within a window. The default menu item name is Print Window. If a selection has been made, however, the menu item appears as Print Selection.

The Print commands substitute a global shell variable called PrintOptions, defined in the Startup file, for the Print dialog box. These options include number of copies to print, pages to print, print quality, font and font size, headers, titles, borders, and order of printing. You can change these printing options (the Startup file specifies headers) by setting PrintOptions in the UserStartup command file with the minus sign options described under Print in the dictionary.

Edit menu

The Edit menu's Format... command produces a dialog box that allows you to change tab size, automatically indent (after a Return, text lines up with the previous line), and show the invisible characters in figure 6-3. The system's fonts are also shown.

¬	Return
◊	Space
Δ	Tab
¿	All other control characters

Figure 6-3

The Align command makes currently selected text line up vertically with the top line of the selection.

The Shift Left and Shift Right commands move blocks of currently selected text according to tab boundaries, leaving alignment within the block intact.

The Edit menu's Execute command operates the same as pressing the Enter key (but is different than execute in the text command language).

Find menu

The Find menu offers numerous options to find and replace text within the active (topmost) window. The Find Same and Replace Same items are shortcuts for repeating operations without displaying dialog boxes. Switches allow operations for Search Backward, Entire Word, Case Sensitive, and Selection Expression. Using the Selection Expression switch, the wildcard characters in figure 6-4 can be used to find text patterns.

`?`	`Match any single character (except Return).`
`≈`	`(≈ = Option/X) Match any string of characters (except Return).`
`[`*`charList`*`]`	`Match any character in the list.`
`[¬`*`charList`*`]`	`(¬ = Option/L) Match any character not in the list.`

Figure 6-4

Window menu

The top half of the Window menu offers Tile Windows and Stack Windows, two methods of displaying multiple windows on the screen. The bottom half of the menu offers the names of all open windows—a check marks the active window, a bullet marks the target window, and an underline marks a window that has been changed since the window was last saved. When you select a window name, the window appears topmost as the active window.

Mark menu

The top half of the Mark menu offers two commands for identifying sections of text by name. The Mark... command produces a dialog box in which to assign a name (called a *marker*) to a previously selected section of text. The Unmark... command produces a dialog box that allows you to delete the association of the name and marked text.

The bottom half of the Mark menu offers the names of current markers. When you select a marker name, the Shell jumps to the marked text in the same manner as the Find command.

Directory menu

The top half of the Directory menu offers two commands. The Show Directory command displays the current default directory in an alert box. The Set Directory... command produces a dialog box in which you assign the default directory and add its name to the bottom of the Directory menu.

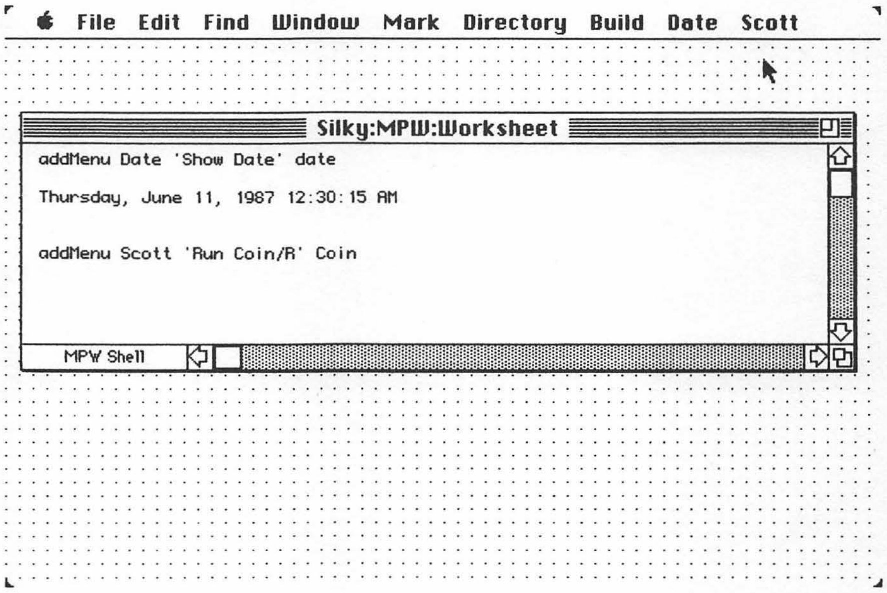
The bottom half of the Directory menu offers the names of currently available directories. Available directories include those assigned by the Set Directory menu item and the setDirectory text command, and those specified in the UserStartup command file (initially set for Example folder files and the default directory). When you select a directory name, the directory is set as the current default directory.

Build menu

The Build menu offers commands to select a program for building and to perform the build. The Create Build Commands... item produces a commando dialog box in which you enter the name and select the source files of the program you want to build. A makefile is created (with the suffix .make appended) using the MPW make tool. The makefile contains the simple commands necessary to build the program.

The Build... and Full Build... items execute the program build commands. Full Build... creates a complete set of files, whereas Build... creates only the files modified since the last build.

The Show Build Commands... and Show Full Build Commands... items write the program build commands to the Worksheet without executing them. Show Full Build Commands... displays a complete set of files, whereas Show Build Commands... displays only the files modified since the last build.



```
addMenu Date 'Show Date' date

Thursday, June 11, 1987 12:30:15 AM


addMenu Scott 'Run Coin/R' Coin
```

Figure 6-5

Custom menus

The command language allows you to install and delete your own menus to perform operations the same as if you executed the text from a Worksheet window. The addMenu and deleteMenu commands, described in the dictionary, create and dispose of user-defined menus. Figure 6-5 is an example of the addMenu command in which the menu titles *Date* and *Scott* are added to the menu bar.

CHAPTER

7

How Do You Do Resources?

Note: The next three chapters, beginning with this one, contain more advanced MPW material. If you are a beginning assembly language programmer, skip these chapters until you are well into part 2. In part 2, you will be advised when it might be a good idea to come back and study these advanced topics. If you are already familiar with Macintosh resources, the Shell's fancy features, and the MacsBug debugger, you might want to skim the material in the next three chapters to see what is of interest. An example of using resources in an assembly language program is shown in chapter 23.

Certainly MPW hasn't changed the way resources work, right?

Sorry, icon-face. MPW supplies a new resource compiler called Rez and a resource decompiler called DeRez. Resources, the Macintosh mini-language that works as an adjunct to assembly, Pascal, C, and other language code, increase an application's adaptability to the needs of the user.

Some of the more common program features that result from resource programming are windows, menus, dialog boxes, icons, and strings. Although these program features could be produced without using the resource mini-language, any feature modification is drastically complicated compared to how resource-based features are modified.

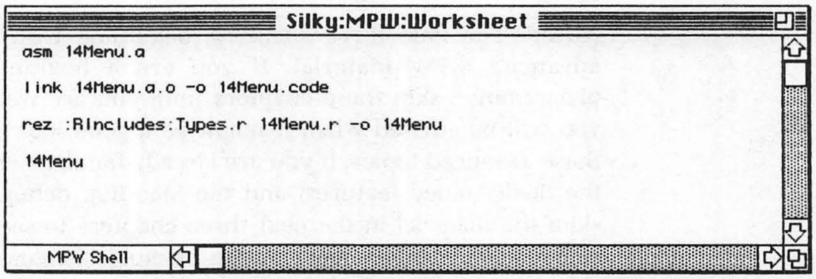
MPW resources are manipulated using two tools and an application:

Rez	Creates a linkable resource file by compiling a text file called a *resource description file.*
DeRez	Translates a resource file back into the text of the resource description file.
ResEdit	Edits an existing or new resource file graphically.

The best way to learn about resources is to look at examples. The resource mini-language has strict syntax requirements and bears little resemblance to English or computer languages. The DeRez tools and the ResEdit application are much better teaching tools than a manual's written explanation.

How do resources fit in the scheme of MPW programming?

Take a look at figure 7-1, a Worksheet window that contains rez as part of the command sequence. (Note: The example program used in chapters 7, 8, and 23 is named 14Menu.)



```
asm 14Menu.a
link 14Menu.a.o -o 14Menu.code
rez :RIncludes:Types.r 14Menu.r -o 14Menu
14Menu
```

Figure 7-1

Your programming experiences in previous chapters have shown you the purpose of asm and link, the first words on the top two lines of the command sequence. The last line starts with rez, the command name of MPW's resource compiler.

When executed, rez performs an action that is similar to asm (the Pascal and C compile commands, too) and link combined. Here are the similarities and one difference. Like asm, rez requires source code (a Worksheet text file) as an input parameter, translating it into object code output. Like link, rez joins code files (resource code files) to help produce a standalone application. But unlike asm and link, rez compiles and links only resources. The contents allowed in resources are described in the next section.

The Rez compiler, and resources in general, offers a great deal more potential than is illustrated by the example in this book (here and in chapter 23). You have many ways to investigate this potential. In addition to using DeRez and ResEdit, you can print out the .r data files on your MPW disk that contain templates for a large number of predefined resources.

Remember, resources are a Macintosh mini-language that works as an adjunct to MPW assembly, Pascal, and C. Writing source code for resources presents the same difficulties as writing source code for other languages. You have to use a special vocabulary and a special syntax. To gain familiarity with the vocabulary and syntax, examine the use of resources in other programs. Memorizing rules won't work.

Can you show me how to add a resource to a simple program?

Here you will look at a resource description file that, when compiled by Rez, creates a menu.

The resource description file can contain five different kinds of statements. Only two of these statements are necessary to create a menu (resource and include), but here is a list of all five types:

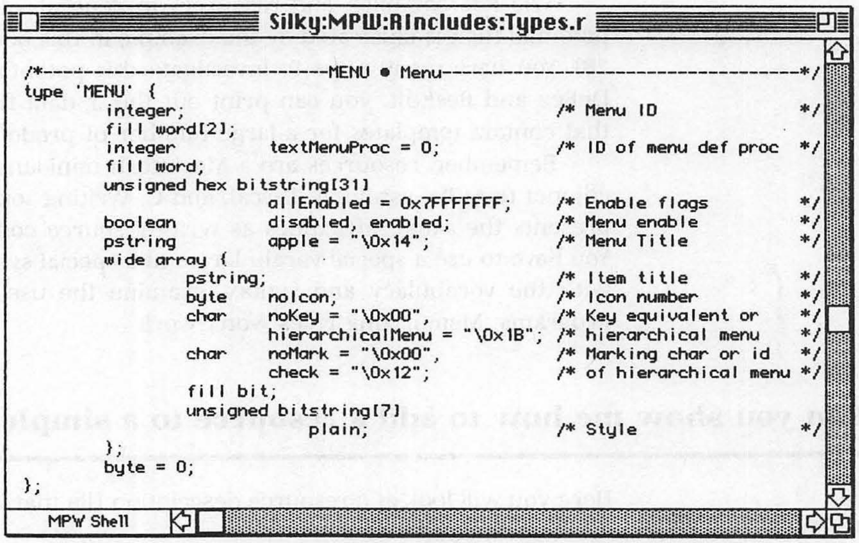
type	Provides a resource type declaration where the pattern of resource data is established as a template.
resource	Provides resource data that fills the pattern set by a previous type declaration.
include	Includes resources that are part of a separate file.
read	Includes resources that are read from the data fork of a file.
data	Provides resource data that is unpatterned.

The resource description file can contain comments as long as they are enclosed with the delimiters /* and */. Comments are ignored by the Rez compiler.

In the example in this section, you will see two kinds of statements: resource and include. Every resource definition requires a type declaration, but you do not need to define the type explicitly. Instead, you instruct Rez (by specifying a parameter) to include the necessary type declaration from a file of such declarations. The Types.r file, located in the RIncludes folder, contains the template for the 'MENU' resource declaration.

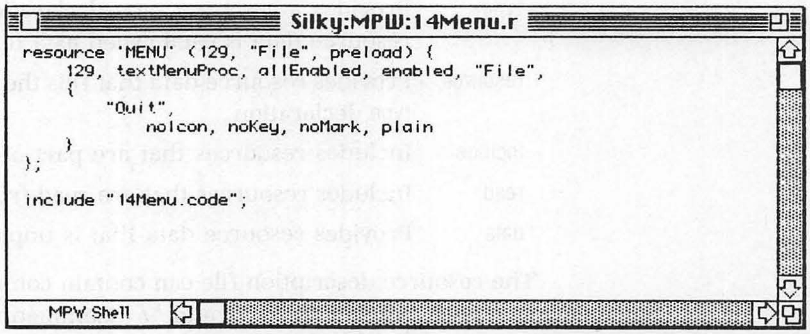
The type declaration in Types.r provides the resource template. The resource definition fills the template with data for a particular resource. First, figure 7-2 is the type 'MENU' declaration in Types.r.

Figure 7-3 is the resource definition that is typed in and saved under the name 14Menu.r.



```
/*----------------------------MENU • Menu--------------------------------------*/
type 'MENU' {
        integer;                                        /* Menu ID              */
        fill word[2];
        integer         textMenuProc = 0;               /* ID of menu def proc  */
        fill word;
        unsigned hex bitstring[31]
                        allEnabled = 0x7FFFFFFF;        /* Enable flags         */
        boolean         disabled, enabled;              /* Menu enable          */
        pstring         apple = "\0x14";                /* Menu Title           */
        wide array {
                pstring;                                /* Item title           */
                byte    noIcon;                         /* Icon number          */
                char    noKey = "\0x00",                /* Key equivalent or    */
                        hierarchicalMenu = "\0x1B";     /* hierarchical menu    */
                char    noMark = "\0x00",               /* Marking char or id   */
                        check = "\0x12";                /* of hierarchical menu */
                fill bit;
                unsigned bitstring[7]
                                plain;                  /* Style                */
        };
        byte = 0;
};
```

Figure 7-2



```
resource 'MENU' (129, "File", preload) {
    129, textMenuProc, allEnabled, enabled, "File",
    {
        "Quit",
            noIcon, noKey, noMark, plain
    }
};

include "14Menu.code";
```

Figure 7-3

Here are some observations about this example resource description file.

1. You have typed in a resource definition according to the syntax rules of the MPW resource description mini-language.
2. The resource definition is made up of resource data that fills the type 'MENU' template. The template, found in the Types.r file, is necessary as the resource type declaration.

3. The grammar rules for all resource statements are stringent. Punctuation and syntax notation characters must be written as shown. The use of uppercase for words that are part of the description mini-language is optional.
4. The template and the data that fills it specify the menu ID, the ID of the menu definition procedure (def proc), the item enable flags, the menu enable flag, the menu title, the item title, the icon number, the key equivalent, the marking character, and the style.

The RIncludes folder contains three resource template files for standard resource types. The resource types commonly used in applications are defined in the file Types.r. The resource types used for system and tools programming are defined in the files SysTypes.r and MPWTypes.r.

In addition to the standard resource types, MPW allows you to create custom resources and offers a command that operates in reverse of the rez command. The deRez command creates a resource description file (the original .r text file) from an existing resource.

The output of the deRez command is written to standard output. Figure 7-4 is an example of the deRez command decompiling the menu resource in the application 14Menu.

Is this how all resources are compiled?

The simple resource example shown in the previous section (figure 7-1) uses the rez command after the asm and link commands. You can also compile resources before assembling and linking. In this section, you will see an alternative method of using rez. Again, the resource file 14Menu.r is compiled independently of the source code 14Menu.a. This alternative method is used by the commando-based Build menu described in the next chapter.

Here, in figure 7-5, is an alternative command sequence for chapter 23's sample program (produced by the Show Full Build Commands... option of the Build menu).

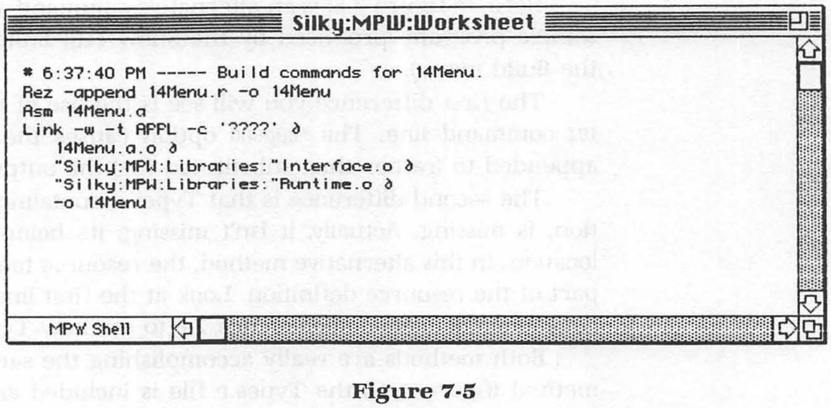
The first difference you will see is the use of the -append option on the rez command line. The -append option causes the compiled output to be appended to (rather than substituted for) the output file.

The second difference is that Types.r, containing the menu type declaration, is missing. Actually, it isn't missing; it's being specified in a different location. In this alternative method, the resource templates file is included as part of the resource definition. Look at the first line of the revised resource definition file 14Menu.r in figure 7-6 to see how Types.r is specified.

Both methods are really accomplishing the same objective. In the first method (figure 7-1), the Types.r file is included as a parameter of the rez

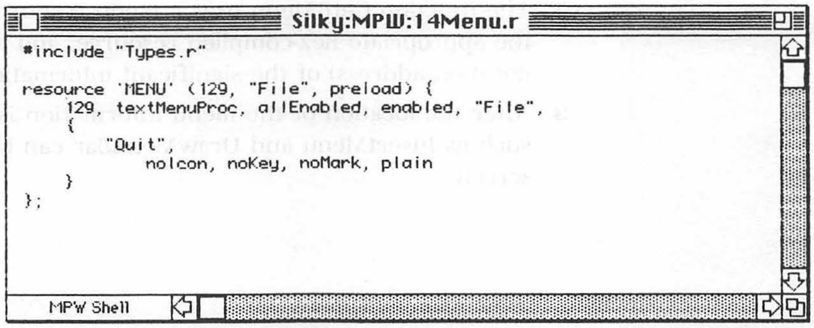
```
# Executing the …deRez command line produced a commando dialog box from
# which the input files 14Menu and Types.r were selected.

…deRez
resource 'MENU' (129, "File", preload) {
    129,
    textMenuProc,
    allEnabled,
    enabled,
    "File",
    {   /* array: 1 elements */
        /* [1] */
        "Quit", noIcon, "", "", plain
    }
};

data 'CODE' (0, purgeable) {
    $"0000 0028 0000 0200 0000 0008 0000 0020"            /* ...(..........  */
    $"0000 3F3C 0001 A9F0"                                 /* ..?<..©. */
};

data 'CODE' (1, "Main", locked, preload) {
    $"0000 0001 486D FFFC A86E A8FE A850 A912"            /* ....Hm..®n®.®P©. */
    $"A930 203C 0000 FFFF A032 594F 42A7 487A"            /* ©0 <....†2YOBßHz */
    $"00C0 487A 00C4 50E7 4267 2F3C FFFF FFFF"            /* .¿Hz.ƒP.Bg/<.... */
    $"51E7 42A7 A913 A873 3F3C 000A A89C 594F"            /* Q.Bß©.®s?<..®úYO */
    $"3F3C 0081 A9BF 4267 A935 A937 A9B4 554F"            /* ?<.Å©øBg©5©7©¥UO */
    $"3F3C FFFF 486D 0108 A970 4A1F 67EE 6108"            /* ?<..Hm..©pJ.g.a. */
    $"4A2D 0118 67E6 A9F4 302D 0108 5340 6706"            /* J-..g.©.0-..S@g. */
    $"422D 0118 4E75 554F 2F2D 0112 486D 011A"            /* B-..NuUO/-..Hm.. */
    $"A92C 301F 0C40 0003 671C 5340 66E2 594F"            /* ©,0..@..g.S@f.YO */
    $"2F2D 0112 A93D 321F 301F 4A41 67D2 1B7C"            /* /-..©=2.0.JAg".| */
    $"0001 0118 4E75 486D 0112 A871 2A2D 0112"            /* ....NuHm..®q*-.. */
    $"2805 554F A973 4A1F 67B6 594F 2F0F A972"            /* (.UO©sJ.g∂YO/.©r */
    $"261F B644 67EC 45ED 0100 2485 2544 0004"            /* &.∂Dg.E...$Ö%D.. */
    $"2F0A A8B7 2543 0004 2F0A A8B7 2803 60D2"            /* /.®∑%C../.®∑(.`" */
    $"0050 003C 0122 01C2 1B4D 656E 7573 3A20"            /* .P.<.".¬.Menus: */
    $"4C69 7465 7261 7475 7265 206F 6620 4769"            /* Literature of Gi */
    $"616E 7473 0000 0000 0000 0000 0000 0000"            /* ants............ */
    $"0000 0000 0000 0000 0000 0000 0000 0000"            /* ................ */
    $"0000"                                                /* .. */
};
```

Figure 7-4



```
# 6:37:40 PM ----- Build commands for 14Menu.
Rez -append 14Menu.r -o 14Menu
Asm 14Menu.a
Link -w -t APPL -c '????' ∂
    14Menu.a.o ∂
    "Silky:MPW:Libraries:"Interface.o ∂
    "Silky:MPW:Libraries:"Runtime.o ∂
    -o 14Menu
```

Figure 7-5



```
#include "Types.r"

resource 'MENU' (129, "File", preload) {
    129, textMenuProc, allEnabled, enabled, "File",
    {
        "Quit",
            noIcon, noKey, noMark, plain
    }
};
```

Figure 7-6

command. In the second method (figure 7-6), the Types.r file is included by the resource directive #include (the # indicates a text file).

The link command is also presented differently. In the first method, the output file of the link command (specified with the -o option) is an intermediary application file called 14Menu.code. The original resource definition file (figure 7-3) uses the directive include "14Menu.code", the last line of code, to link to the intermediary file and produce the final application.

In the second method, this intermediary file is not used. (The directive include "14Menu.code" is omitted from the resource definition file.) The rez command appends its compiled resource to the 14Menu file (which is in an intermediary state), then the asm and link commands complete the application so that the output file of the link is the final application.

You should also notice that the link command has a few additional options that specify file characteristics. (You can look these up in part 3's dictionary.) Note in figure 7-5 that the long command line has been broken up into five Worksheet lines. This is made possible by the character at the end of the line (produced by pressing Option/D). This character, at a line's end, indicates that the command continues on the next line.

How does your program code know about resources and how are resources linked?

The Macintosh ROM uses a set of Toolbox calls that provide access to resources. The program illustrated in chapter 23 of part 2, the assembly tutorial, demonstrates this access for a menu resource. Briefly, here is the process:

1. The Menu manager (or any other manager applicable to your resource task) is initialized by the procedure InitMenus.

2. The function GetRMenu uses a menu resource ID parameter to read the appropriate Rez-compiled resource, and returns a handle (location address) of the significant information.
3. After the location of the menu information is available, Toolbox calls such as InsertMenu and DrawMenuBar can create the menu on the screen.

CHAPTER 8

Who Uses Make and the Structured Commands?

Note: As was stated at the beginning of the last chapter, the last three chapters in part 1 contain more advanced MPW material. If you are a beginning assembly language programmer, skip these chapters until you have made progress with part 2. If you are already familiar with advanced Shell features, you might want to skim the material here.

When should I start using MPW's fancy features?

Foremost, the purpose of a programming environment's fancy features is to speed development. Fancy features make a better final product only to the extent that the saved time allows the programmer to work more on code writing and less on code mechanics. When you feel you are wasting a lot of time waiting for the computer to compile, link, and run your code, you might start investigating how an MPW feature might speed the process.

Don't forget the price to pay: You have to learn how the fancy features work. Before you jump in writing makefiles and structured command files to process your twenty-line sample programs, be sure you understand exactly what the fancy feature is doing. Otherwise, you could introduce new problems into your program that cannot be identified quickly.

The fancy feature explanations covered in the remainder of this chapter are intended to inform you of the options available for further study. Only through practice will the use of these features become easy enough that development time will be reduced. Programmers working with long and complex projects will benefit the most.

First, a few comments on commands and their structure. A command is made up of one or more words, separated by spaces or tabs, with the last word terminated by a special character, usually Return. The first word is the command name. Subsequent words are parameters to the command. Parameters are most often options (specified with minus sign or letter notation) or file names. The dictionary shows the required and optional parameters for all Shell commands. Optional parameters are surrounded by parentheses.

When you execute a command, the command name appears in the Status Panel in the bottom-left corner of the active window, and a status value is returned in the {Status} variable predefined by the Shell. A command that returns a zero status value has been executed successfully; a nonzero value usually signifies a type of error.

Commands and parameters can be referenced numerically. The command name is considered parameter 0 stored in variable {0}, and each parameter can be referenced in the sequential variables {1}, {2}, and so on.

In addition to the Return character, commands can be terminated by a semicolon, a pipe symbol, or branching operators as follows:

commandOne ; commandTwo	The commands are executed sequentially with multiple commands allowed on a single line.
commandOne | commandTwo	The output of *commandOne* is piped through as the input to *commandTwo.*
commandOne && commandTwo	*commandTwo* is performed only if *commandOne* succeeds as shown by a zero status value.
commandOne || commandTwo	*commandTwo* is performed only if *commandOne* fails as shown by a nonzero status value.

You can include comments anywhere within a file or window without affecting operation by preceding the text of the comment with a number symbol (#). Each line that has a comment must show the number symbol because comments are always terminated by the Return character.

What does Make make?

In developing a program, a programmer creates many files. For example, to produce the program Coin, you created three files: Coin.a, Coin.a.o, and Coin, itself. By adding a single resource as shown in the last chapter, you would add two more files: Coin.r. and Coin.code. Changes to any one of these files would require that Coin be rebuilt through the compilation, link, and Rez (resource) process.

The make tool automates the programmer's building process. (Later in this chapter you will see the automation taken one step further with the createMake and build commands from the Build menu.) To use the make tool, a programmer must create a set of instructions that establish the building process. These instructions are saved in a file referred to as a makefile. In the example that follows, the makefile is named 14Menu.make.

The make tool performs these services for the programmer:

- It keeps track of all the files and commands that are necessary to build a program.
- It establishes the dependencies of the component files. That is, it establishes which files depend on which other files for their makeup (for example, object file Coin.a.o depends on source code file Coin.a for its makeup).
- When one component is updated, it allows you to selectively update only the components that depend on the updated component. Unaffected components are left intact without undergoing unneeded interaction.
- The execution of a makefile produces an automated, streamlined command file that, itself, is available for execution.

A simple makefile uses the format shown in figure 8-1. The Option/F character translates to "is a function of." Thus, a dependency is established: *targetFile* is a function of *prerequisiteFile.* If *prerequisiteFile* is missing or is newer than *targetFile,* then *targetFile* needs to be rebuilt.

```
targetFile  ƒ    prerequisiteFile
   commandList

(ƒ = Option/F)
```

Figure 8-1

The rebuilding process is established by the *commandList...* line. Command lines are always indented by a space or tab. When the dependency on the top line indicates that rebuilding is needed, the command lines that follow indicate the appropriate action that needs to be taken.

After a makefile is created and saved using the previous format, the make command, called from the Worksheet window, uses the makefile as its parameter. The make command has the following format:

make *(optionList) (targetFileList)*

The -f *makefile* option (-f 14Menu.make is used in the example to follow) causes the command to read information from the parameter *makefile.* Other options are explained in part 3's dictionary. (If the -f *makefile* option was omitted, the command would try to read information from a default file named MakeFile.)

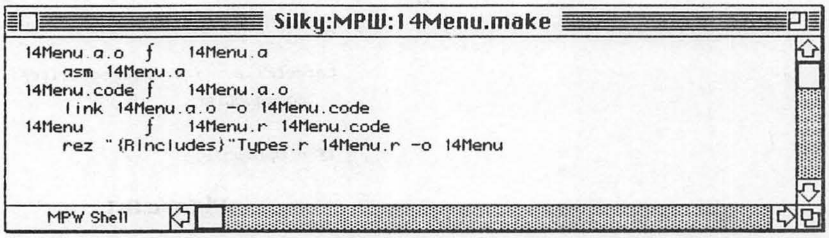
The *targetFileList* parameter specifies the end product that is being built. In the case of a single dependency, the *targetFile* is the first (leftmost) file in the makefile. When the makefile consists of three dependencies, however, such as in the following example, it is necessary to establish the ultimate target of the make process (the application 14Menu).

Note: Executing the make command does not cause the commands in *CommandList* to be executed. Executing make causes the needed rebuilding commands to be displayed in standard output.

If you want the make output to be redirected to a command file instead of standard output, you can add the redirection parameter, > *fileName,* to the make command line. This technique is shown in a later figure.

The following three screens show the process of creating and executing a makefile that will build the Coin program. Here is a step-by-step summary of the make process:

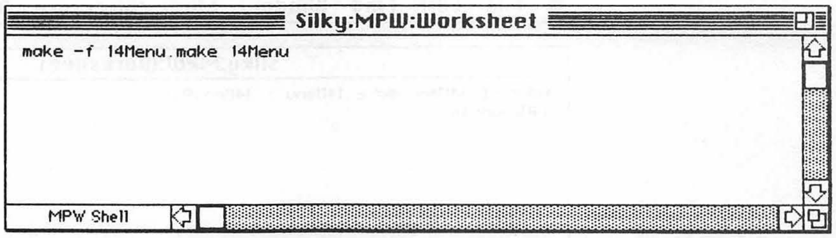
1. Create and save a makefile such as 14Menu.make. See figure 8-2. A makefile consists of two kinds of lines: dependency lines and command lines. Command lines are indented by a space or tab.



```
14Menu.a.o    ƒ    14Menu.a
    asm 14Menu.a
14Menu.code   ƒ    14Menu.a.o
    link 14Menu.a.o -o 14Menu.code
14Menu        ƒ    14Menu.r 14Menu.code
    rez "{RIncludes}"Types.r 14Menu.r -o 14Menu
```

Figure 8-2

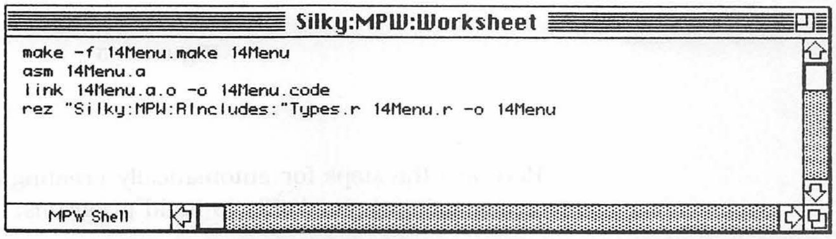
2. Execute the make command from a Worksheet window. You will want to supply the make command with the -f *makefile* option, in which *makefile* is the name of the dependency information file you saved in step 1. Also, you will want to specify the *targetFile* parameter. See figure 8-3.



```
make -f 14Menu.make 14Menu
```

Figure 8-3

3. The output of the make command is displayed in standard output and consists of the command lines necessary to rebuild the target, as shown in figure 8-4. You can execute these commands by selecting them and pressing the Enter key.



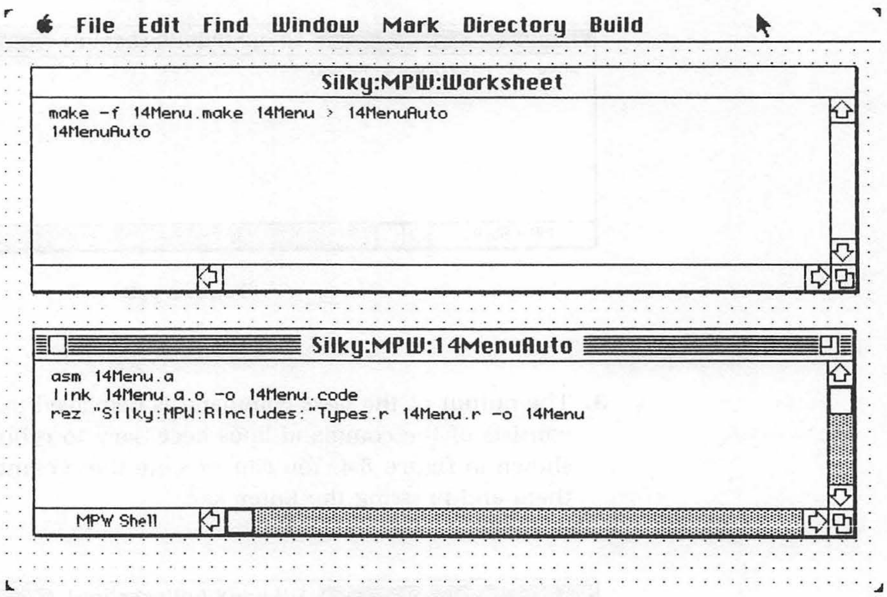
```
make -f 14Menu.make 14Menu
asm 14Menu.a
link 14Menu.a.o -o 14Menu.code
rez "Silky:MPW:RIncludes:"Types.r 14Menu.r -o 14Menu
```

Figure 8-4

Figure 8-5 shows the make command using a redirection parameter. The > 14MenuAuto addition to the command line causes output to be directed to (in this case, create) a command file rather than standard output. By executing the 14MenuAuto command file (the second line of the Worksheet), you can complete the automated build process.

Why doesn't the computer know how to build programs?

For simple programs that do not use programmer-designed dependencies, the make process, itself, can be automated. The commands to do so are available from the command language (createMake, buildProgram, and buildCommands). It is simpler, however, to use the commando interface of the Build menu items.



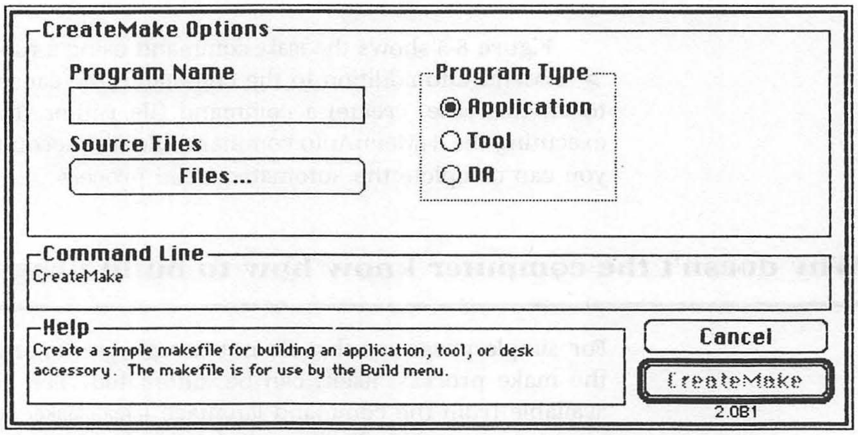
```
make -f 14Menu.make 14Menu > 14MenuAuto
14MenuAuto
```

```
asm 14Menu.a
link 14Menu.a.o -o 14Menu.code
rez "Silky:MPW:RIncludes:"Types.r 14Menu.r -o 14Menu
```

Figure 8-5

Here are the steps for automatically creating a makefile and the four options for using the makefile to build programs:

1. Choose Create Build Commands... from the Build menu. The commando dialog box in figure 8-6 appears.



Program Type: [✓ Application] Tool DA

Command Line
CreateMake

Help
Create a simple makefile for building an application, tool, or desk accessory. The makefile is for use by the Build menu.

Figure 8-6

2. Type in the name you want to give your application in the Program Name box. The sample program from chapter 23 is named 14Menu.
3. Press the Source Files button to produce a file dialog box.
4. One by one, select the source files from the upper list and press the Add button to include the file in the Source Files list. Figure 8-7 shows the dialog box with the two source files for 14Menu added.

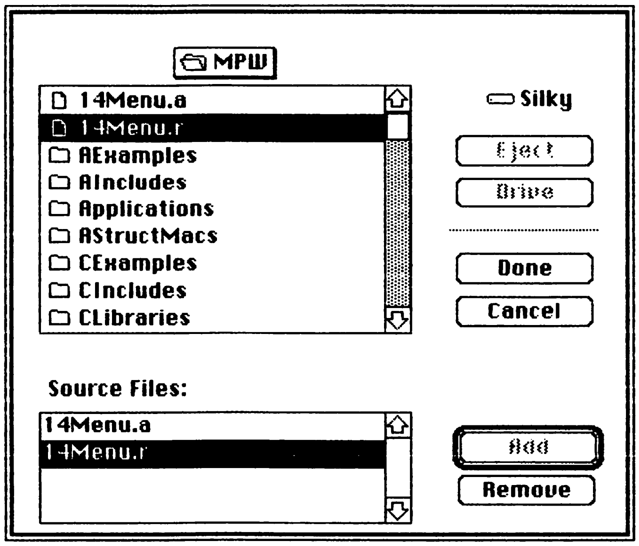
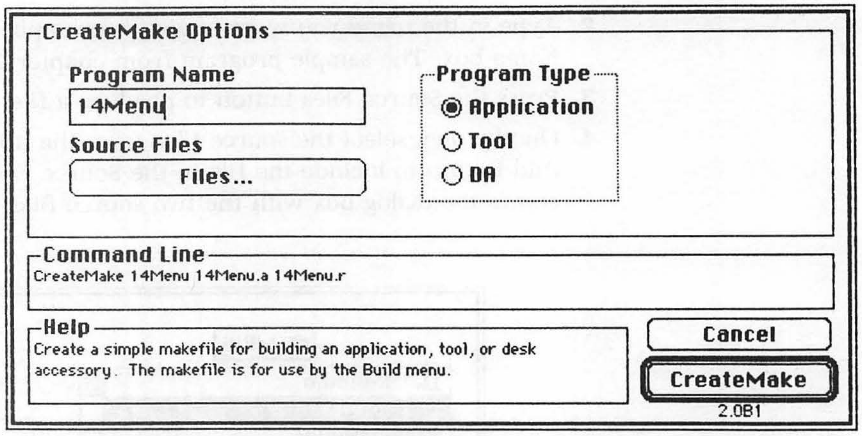


Figure 8-7

5. Press the Done button to return to the CreateMake dialog box. Refer to figure 8-8. You can see, in the Command Line box, the text of the command that will be executed by pressing the CreateMake button.
6. Press the CreateMake button to create a makefile with the name program.make. Remember, the make and createMake commands do not execute their makefile contents. They simply create the makefile.
7. (This step is optional. It is not part of the build, yet it lets you see the simple makefile dependencies and command lines.) Open the newly created makefile by typing *open 14Menu.make* in the Worksheet. The screen shown in figure 8-9 appears.
8. Choose Show Full Build Commands... from the Build menu. A dialog box asks you to name or confirm the application you want to build by clicking in the Okay box.



Program Name: 14Menu

Program Type: [✓ Application] Tool DA

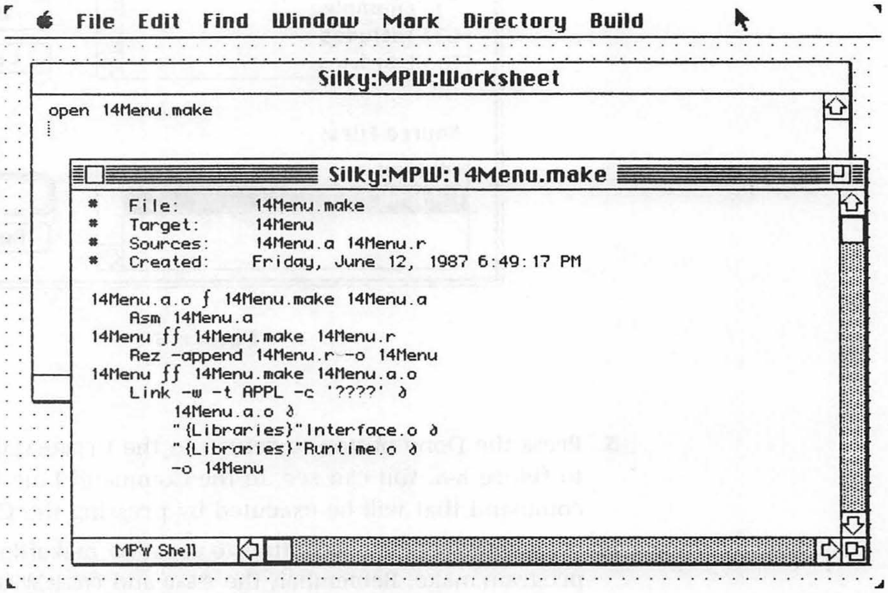
Command Line

```
CreateMake 14Menu 14Menu.a 14Menu.r
```

Help

Create a simple makefile for building an application, tool, or desk accessory. The makefile is for use by the Build menu.

Figure 8-8



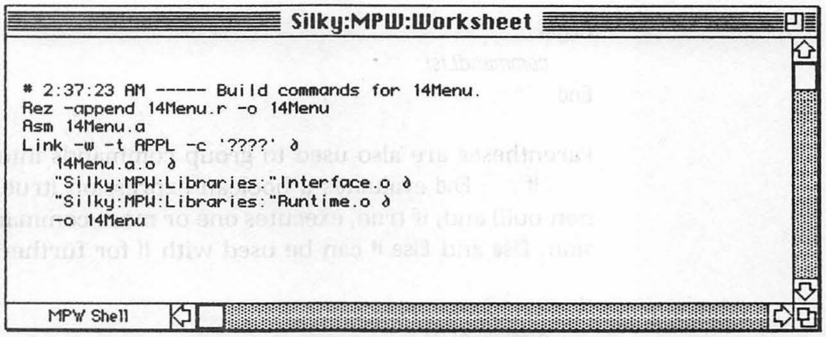
```
open 14Menu.make
```

```
#   File:       14Menu.make
#   Target:     14Menu
#   Sources:    14Menu.a 14Menu.r
#   Created:    Friday, June 12, 1987 6:49:17 PM

14Menu.a.o ƒ 14Menu.make 14Menu.a
    Asm 14Menu.a
14Menu ƒƒ 14Menu.make 14Menu.r
    Rez -append 14Menu.r -o 14Menu
14Menu ƒƒ 14Menu.make 14Menu.a.o
    Link -w -t APPL -c '????' ∂
        14Menu.a.o ∂
        "{Libraries}"Interface.o ∂
        "{Libraries}"Runtime.o ∂
        -o 14Menu
```

Figure 8-9

9. Click in the Okay box. The screen in figure 8-10 appears. The command lines neccesary to build the application are displayed in the standard output Worksheet and are available for execution.

In addition to the Show Full Build Commands... option, the Build menu



```
# 2:37:23 AM ----- Build commands for 14Menu.
Rez -append 14Menu.r -o 14Menu
Asm 14Menu.a
Link -w -t APPL -c '????' ∂
   14Menu.a.o ∂
   "Silky:MPW:Libraries:"Interface.o ∂
   "Silky:MPW:Libraries:"Runtime.o ∂
-o 14Menu
```

Figure 8-10

offers three other options that produce similar results. The Show Build Commands... option displays only the files modified since the last build. Like the Show Full Build Commands... option, it causes the program build commands to be written to the Worksheet without being immediately executed.

The Build... and Full Build... items cause the program build commands to be executed instead of written to the standard output Worksheet. A progress report of the execution is written to output. Full Build... creates a complete set of files, whereas Build... creates only the files modified since the last build.

When do the structured commands come in handy?

MPW has adopted a characteristic of programming languages called *structured commands.* Structured commands allow a set of simple commands to be treated as a single block and executed according to rules that apply only to that block.

A structured command differs from a series of simple commands because the structure dictates the order of execution. Simple commands are executed in sequential order. MPW allows structured commands to be used interactively and as part of a command file. Execution begins after the entire structured command is read.

Pascal and C programmers should recognize the structured commands from their Pascal and C equivalents. The format of the structured commands requires Return characters (or semicolons) as line delimiters. The following is a description of MPW's structured commands and their format.

Begin . . . End brackets a group of commands so that they are treated as a single unit.

```
Begin
    commandList
End
```

Parentheses are also used to group commands into a single unit.

If . . . End evaluates a boolean expression (true values are nonzero and non-null) and, if true, executes one or more commands following the expression. Else and Else If can be used with If for further conditional execution.

```
If expression
    commandList
(Else If expression
    commandList)
(Else
    commandList)
End
```

For . . . In . . . End executes a set of commands once for each item in a word list. A given *name* represents the current value of the word in *wordList.*

```
For name In wordList
    commandList
End
```

Loop . . . End executes a set of commands repeatedly until a Break command exits the loop.

```
Loop
    commandList
End
```

Break ends execution of the innermost For or Loop command, and exits the structure.

```
Break (If expression)
```

Continue causes the innermost For or Loop command to end its current iteration and continue with its next iteration.

```
Continue (If expression)
```

Exit ends execution of its host command file. The optional parameter *number* returns the status value of the command file. Exit also ends command execution interactively.

```
Exit (number) (If expression)
```

How about some input on output and vice versa?

Your practice with entering commands and programs into a Worksheet has shown you that the Shell can work both interactively (type a command, press Enter, and the command is executed) and as a file processor (type a program or series of commands, save the file, and execute the file). In each case, the Shell takes input and gives output. Input generally has come from a parameter file, and output generally has gone to a Worksheet window.

The Shell's input and output devices are actually internal files. The Shell uses three open files called *standard input, standard output,* and *diagnostic output.* By default, standard input is read from the keyboard, and both standard output and diagnostic output are displayed in the active Worksheet.

Certain commands read standard input (for example, catenate without parameters) and, consequently, must be terminated explicitly by holding down the Command/Enter key combination (or Command/Shift/Return). While these commands are running, their name appears in the Status Panel in the bottom-left corner of a Worksheet window. Only after Command/Enter is pressed will control return to the Shell.

The two types of output are often interleaved in the Worksheet window. You can distinguish between them by thinking of standard output as the requested result and diagnostic output as by-product information that evaluates the result. Diagnostic output contains status values (error codes) corresponding to the performance of the previously executed command.

The assembling and compiling commands asm, pascal, and c, when used with -p (the progress information option), produce a large amount of diagnostic output describing the performance of the assembly or compilation. With these commands, the standard output of the assembly or compilation is sent to an .o object file, and the diagnostic output is sent to a Worksheet window.

The Shell's input and output can be redirected from their standard files to chosen files. The following symbols redirect input and output:

< *file*	Reads standard input from the named file.
> *file*	Standard output creates the named file, replacing the contents of any previous file of that name.
>> *file*	Appends standard output to the named file, or creates the file if it does not already exist.
≥ *file*	The contents of the named file are replaced with diagnostic output.
≥≥ *file*	Appends diagnostic output to the named file.

What other features are there to further complicate matters?

Here's a list of additional Shell features that allow a more specialized and intricate programming environment. The MPW manual provides reference material on these features.

User-Defined variables

Creating variables or modifying the Shell's predefined variables makes MPW's parameters, default settings, status code, and user data easier to remember and manipulate. The commands set, unset, and export (explained in their dictionary entries) define, modify, and transmit variables. Also, the Startup file provides an example of the use of predefined and command file variables.

Command substitution

A command typed within single back quotation marks (*'command'*) is replaced by the command's output when the command is executed. For example, the command Print 'Files' prints the output of the Files command.

Special quoting

Because the MPW Shell uses special characters to perform Shell functions (for example, ? is a wildcard character), the use of different types of quotation marks enables you to disable the effect of special characters or insert invisible characters in text. The symbols in figure 8-11 can be used as quotation marks.

∂*char*	Quote only the single character that follows ∂, the Option/D character.
'*string*'	Use the enclosed *string* literally, with no substitutions.
"*string*"	Use the enclosed *string* literally, but allow ∂*char*, variable substitutions, and command substitutions.
/*string*/ or \string\\	Use the entire *string* and its quotation marks literally, allowing substitutions.

Figure 8-11

File name generation

Special characters within words are used to represent a file name pattern. (The Find command uses the same wildcard characters to find text patterns.) A list of file names that match the pattern replace the word. The characters in figure 8-12 possess these wildcard characteristics.

`?`	Match any single character (except colon or Return).
`≈`	(≈ = Option/X) Match any string of characters (except colon or Return).
`[`*charList*`]`	Match any character in the list.
`[¬`*charList*`]`	(¬ = Option/L) Match any character not in the list.

Figure 8-12

CHAPTER

9

Am I Debugging Yet?

Note: This chapter is largely a reference to MacsBug, a debugging utility. If you are a beginning assembly language programmer, skip this chapter until you have made progress with part 2 and want to know more about debugging. If you are already familiar with debuggers, you might want to skim the material here.

Where does the debugging process start?

One of the files that comes with MPW is named MacsBug. MacsBug is not a standalone application; it installs itself when the Macintosh is turned on or reset. The installation has two prerequisites:

1. The MacsBug file retains its original name (i.e., don't change MacsBug to McBug or anything else).
2. The MacsBug file is in the current System Folder of a startup disk.

When MacsBug is installed in this manner, every subsequent startup of the Macintosh automatically installs MacsBug. To verify the installation, the startup screen that reads "Welcome to Macintosh" now includes the message "MacsBug Installed."

After MacsBug is installed, the program sits dormant in RAM (memory). The Macintosh is free to run other programs. The debugger is transparent and relatively unobtrusive. MacsBug takes control of the Macintosh only when you activate the debugger (the easiest method is to press the Interrupt button of the Programmer's Switch). If you followed the disk configuration instructions in chapter 1 and since then have restarted the Macintosh, MacsBug will be installed and available for use. You are ready to examine the debugger.

What can the debugger do for me?

Programmers use debuggers in the way that biologists use microscopes. Debuggers show program activity performed by the hardware's components: snapshots of processors and memory. These stop-action glimpses can help a programmer discover a bridge between a specific line of written code and the action that line is producing.

Before a biologist becomes a biologist, the microscope is an important learning tool. The same is true for debuggers. MacsBug illustrates how a computer works in the nitty-gritty world of assembly language. Output from MacsBug is used right from the beginning of part 2, the assembly language tutorial.

All programmers should understand what a debugger offers them. For assembly programmers, the benefits are obvious: the bridge from code to code action is direct. Pascal and C programmers can derive the same benefit from using MacsBug, though the task requires an intimate knowledge of how Pascal and C relate to assembly language.

Can you show me how the basics work?

You can experiment with MacsBug while running any program, including the MPW Shell. Press the Interrupt (rear) button on the Programmer's Switch, and MacsBug takes control over the screen, producing output similar to the following:

```
401F52:                        SUBQ.L    #$2,A7
PC = 00401F52   SR = 00002004   TM = 0000084A
D0 = 00000000   D1 = 00000002   D2 = 00000000   D3 = 00000000
D4 = 00000000   D5 = 00000000   D6 = 00000000   D7 = 00000000
A0 = 00F80000   A1 = 000EF9F2   A2 = 00000000   A3 = 000EFC1A
A4 = 0000FF5E   A5 = 000EFBF8   A6 = 000EE5A0   A7 = 000EF9EC
>
```

You can switch between the original application screen and the MacsBug output by pressing the tilde/opening quote key (~') in the keyboard's upper-left corner. A press of any character key displays the MacsBug output.

Because the debugger works as a bridge between code and code results, it is important for the programmer to determine exactly where the debugger freezes the program and takes control of the screen. The Interrupt button does not provide a way of stopping execution at a specific line of code. Therefore, a programmer often wants to put in the program code a debugger instruction that automatically activates the debugger when that line of code is executed.

Activating MacsBug from within the program code differs depending on the language used. Here are example calls to enter Macsbug from each language:

Assembly	`_Debugger`
Pascal	`Debugger`
C	`Debugger()`

Exactly what does the MacsBug output show?

The MacsBug output in the previous section shows the disassembled current instruction in the upper-right corner, the address of the current instruction in the upper-left corner (and in the PC register), and the contents of each of the registers in the rows below. The greater than symbol (>) acts as a prompt awaiting a user command. The MacsBug command language offers numerous one-character and two-character commands to manipulate debugger output. Many commands use parameters to specify the desired action.

There are six groups of MacsBug commands: general, memory, breaks, A-Traps, heap zone, and disassembly. The following list, by group, ought to help a beginner discover the debugger's potential. Parenthesized parameters are optional. The more advanced commands are listed without explanation.

General

General commands provide MacsBug help information or provide general control operations for exiting MacsBug.

`?`
Show a list of available commands and their parameters.

`DV`
Display version information of MacsBug.

`RB`
Reboot the system as if the reset button had been pressed.

`ES`
Exit to the current Shell, usually the MPW Shell or Finder.

EA
Exit to the current application.

Memory

Memory commands display or manipulate specific slots of memory.

CV *expr*
Convert the expression (*expr*) and show it in the format of unsigned hex, signed hex, signed decimal, text, and binary.

DM *(address (number))*
Display memory starting at the given *address. Number,* rounded to the nearest 16 bytes, specifies the number of bytes to display. Omitting the parameters on subsequent calls causes the next 16 bytes to be displayed.

SM *address exprList*
Set memory starting at *address* with the given expressions.

DB *(address)*
The byte at *address* is displayed.

SB *address (expr)*
Set byte at *address* to expression or, if no expression is given, to 0.

D*n (expr)*
Data register *n* is displayed or, if an expression is provided, set to the expression.

A*n (expr)*
Address register *n* is displayed or, if an expression is provided, set to the expression.

PC *(expr)*
The program counter is displayed or, if an expression is provided, set to the expression.

SR *(expr)*
The status register is displayed or, if an expression is provided, set to the expression.

TD
Perform a total display of the registers and the PC, as well as a disassembly of the next instruction to be executed.

F*n (expr)*
Floating data register.

FI *(expr)*
Floating instruction address register.

FC *(expr)*
Floating control register.

FS *(expr)*
Floating status register.

CS *(address1 (address2))*
Checksum.

Break

Break commands perform a variety of stop-and-go functions that allow the debugger to pinpoint particular sections of code as a program executes.

BR *(address (count))*
A breakpoint is set at the given *address,* stopping execution at its first occurrence. The *count* parameter allows program execution to bypass the breakpoint for a given number of repetitions. Omitting both parameters displays all breakpoints—eight are allowed.

CL *(address)*
Clear the breakpoint at the given *address.* Omitting the parameter clears all breakpoints.

G *(address)*
Execution goes (continues), beginning at the next instruction or, if given, the instruction of the parameter *address.*

GT *address*
Execution goes (continues) till the given *address.* The given *address* functions as a breakpoint for one break only. The program counter provides the first instruction to be executed.

T
Trace through code by executing a single instruction. The instructions within a trap call are considered a single instruction.

S *(number)*
Step through code by executing the given *number* of instructions. Omitting the parameter executes one instruction. The instructions within a trap call are considered individually rather than as a single instruction.

SS *(address1 (address2))*
Step spy.

ST *address*
Step through code till the given *address.* The instructions within a trap call are stepped through individually rather than as a single instruction.

MR *(offset)*
Magic return.

DX
Debugger exchange.

A-Trap

A-Trap commands follow the action of the trap calls in the Macintosh ROM. Six parameters can be used to specify the conditions under which specific traps are to be followed. Traps can be identified by a number, a range of numbers, a range of memory addresses, and a range of values in data register 0. The command G (go) used after an A-Trap command will begin the A-Trap displays.

BA *(trap1 (trap2 (address1 (address2 (D1 (D2))))))*
Break in application.

AA *(trap1 (trap2 (address1 (address2 (D1 (D2))))))*
Application A-Trap trace.

AB *(trap1 (trap2 (address1 (address2 (D1 (D2))))))*
A-Trap break.

AT *(trap1 (trap2 (address1 (address2 (D1 (D2))))))*
A-Trap trace.

AH *(trap1 (trap2 (address1 (address2 (D1 (D2))))))*
A-Trap heap zone check.

AR *(trap1 (trap2 (address1 (address2 (D1 (D2))))))*
A-Trap record.

AS *address1 (address2)*
A-Trap spy.

AX
A-Trap clear.

Heap zone

Heap zone commands manipulate the current heap zone, an area of memory whose dimensions are initially set for use by the application.

HX *(address)*
Heap exchange.

HC
Heap check.

HS *(trap1 trap2)*
Heap scramble.

HD *(mask)*
Heap dump.

HT *(mask)*
Heap total.

SC
Stack crawl.

Disassembler

Disassembler commands help to reverse the assembly process—object code is interpreted by the symbols, subjects, and instructions of the source code.

SX
Symbol exchange.

SD *(address)*
Symbol dump.

DH *number*
Disassembles a hex byte, word, or long word, providing as output the instruction that corresponds to the *number* value.

ID *(address)*
Instructions are disassembled at the given *address.* Omitting *address* causes the next logical location to be used by default.

IL *(address (number))*
Instruction list.

F *address count data (mask)*
Find.

WH *expr*
Where.

More tips

Unlike program code, all numbers represented in the debugger are hex unless otherwise indicated. The dollar sign ($) can proceed hex numbers if desired. The number sign (#) must precede a decimal number. The plus and minus signs (+ and −) are used for signed hex representation (study signed versus unsigned hex for math use). The less than symbol (<) sign-extends a hex word to a long word.

In addition to numbers, MacsBug uses strings, symbols, and expressions. A string of one to four characters is stored as a hex long word. Two-character symbols represent address registers, data registers, the program counter, and the current Quickdraw port. The dot symbol (.) indicates the last referenced address. The following operations are used in expressions:

+	Addition or assertion
−	Subtraction or negation
@ or *	Indirection prefix
&	Address prefix
<	Sign extended addition or sign extension prefix

Here is some additional information that will give you more MacsBug capabilities after you become familiar with its basic use.

- MacsBug can also be entered by FKEYs and INIT resources.
- Strings can be added to traps.
- MacsBug can be taken out of RAM by renaming the MacsBug file, then restarting the Macintosh. Also, you can stop the MacsBug file from being installed by holding down the mouse button when restarting the Macintosh.
- MacsBug takes up 43K of RAM and could interfere with some applications, especially on a 512K Macintosh.

Practice using the debugger. Always have backup copies of your disks because debugger experimentation often causes unrecoverable failures. With luck, you'll be able to restart your application, return to the Shell, and reboot, or turn off the Macintosh and have the disk reboot. At worst (boot blocks damaged), you will have to reinitialize your disk.

PART TWO

Up Bit Creek: The Assembly Tutorial

Each chapter in part 2 contains three sections. The first two sections present technical instructions on writing MPW assembly language programs. The third section offers a *fear and loathing* diversion—sometimes light-hearted, always forthright—to enliven the task of learning assembly.

CHAPTER 10

Slots: All the Bits That Fit

Slot is my name, hex is my game

Get ready. You are going to see a picture that represents everything a computer knows. By the time you understand each element of this picture, you'll understand how a computer works. Upcoming chapters describe the picture's elements in detail. For now, you are asked only to recognize the primary element: hexadecimal (hex) numbers.

Assembly language programmers use hex numbers constantly. The material in this book does not require a lot of hex arithmetic. Proficiency in assembly programming, however, does require a thorough understanding of it.

Hexadecimal is a counting system based on 16 different digits. The digits look like this:

```
0 1 2 3 4 5 6 7 8 9 A B C D E F
```

The first 10 digits are the same as those used in the decimal numbering system. The numeric quantities 10, 11, 12, 13, 14, and 15—which require 2 digits to be expressed in decimal form—are represented by A, B, C, D, E, and F in hex form.

In program code, hex numbers are preceded by a dollar sign ($). For example, $A is equivalent to the decimal number 10. $F is equivalent to 15. $4 is equivalent to 4. Later, you will see how to express hex numbers larger than $F (decimal 15).

Different slots for those delicate spots

The following is a snapshot of the Macintosh as it manipulates hex numbers. (All numbers are in hex. Only program code requires the dollar sign.) This snapshot, and the others that follow, can be reproduced using the MacsBug debugger, as described in chapter 9.

```
401F52:                          SUBQ.L    #$2,A7
PC = 00401F52   SR = 00002004    TM = 0000084A
D0 = 00000000   D1 = 00000002    D2 = 00000000   D3 = 00000000
D4 = 00000000   D5 = 00000000    D6 = 00000000   D7 = 00000000
A0 = 00F80000   A1 = 000EF9F2    A2 = 00000000   A3 = 000EFC1A
A4 = 0000FF5E   A5 = 000EFBF8    A6 = 000EE5A0   A7 = 000EF9EC
>
```

This snapshot displays a very small part of the Macintosh's memory. Yet this small part is particularly important because of the 6-digit number on the left side of the top line. This number is the current value of the *program counter* (PC). Immediately below the 6-digit number, this information is repeated:

```
PC = 00401F52
```

A Macintosh functions by manipulating single *slots* of information. The 68000 processor, the heart of the machine, can keep track of only one slot at any one moment. The program counter identifies the slot that will be acted on by the processor.

Everything a computer knows is stored in slots. The program counter itself is a slot. It so happens that the PC slot contains information about the slot being acted on by the processor.

The two most important revelations about slots are:

- All the information a computer knows is contained in slots.
- When slots refer to other slots, they use addresses or, in a few special cases, names.

The 68000 processor uses eighteen special slots, called *registers,* that are identified by two-character names. Ordinary slots have no names; they are referenced by hex numbers representing *addresses.* Register slots are convenient, speedily accessed storage. Ordinary slots are the primary components of the Macintosh's memory. The names of the registers are shown in figure 10-1.

Refer back to the snapshot, and you will see that the contents of these special slots are hex numbers. The snapshot is representative of all com-

Register Names

PC = Program Counter
SR = Status Register
A0, A1, A2, A3, A4, A5, A6, A7 = Address Registers
D0, D1, D2, D3, D4, D5, D6, D7 = Data Registers
SP = Stack Pointer (same as A7)

Figure 10-1

puter memory: information is stored numerically, and the most common numeric form used in assembly language programming is hexadecimal.

The following *fear and loathing* section introduces the word *bit,* a common programming term that designates the smallest unit of computer information. Slots are filled with bits of information. That's all the technical gruel for now.

Wet feet on bit creek

Machines are very intelligent. Spaniels are fairly intelligent. Woodchucks have sharp teeth, but routinely err when solving binary arithmetic problems. Where do people fit in?

Before people were intelligent, they could count only to two. Starting with zero, they would say, "Zero, one, uh, let's go hunting for a bit to eat." Ever since, a bit has come to represent the value 0 or 1.

Counting fingers and toes had not been discovered yet. People had two hands, two feet, two eyes, two ears. Men had two of something that women did not and women had two of something that men did not. Nature provided men and women with an instinct for one important kind of togetherness, but only after their intelligence evolved could they put two and two together.

Machines are very intelligent, but nature did not bless them with the fun things given to men and women. As a result, machines lead boring lives. They can put two and two together till hell serves Häagen Dazs. Yet they always sleep alone at night. Would you like to be a machine?

Study the following illustration (which ought to be familiar from the last section), titled *Nude Ascending a Staircase*:

```
401F52:                               SUBQ.L    #$2,A7
PC = 00401F52   SR = 00002004   TM = 0000084A
D0 = 00000000   D1 = 00000002   D2 = 00000000   D3 = 00000000
D4 = 00000000   D5 = 00000000   D6 = 00000000   D7 = 00000000
A0 = 00F80000   A1 = 000EF9F2   A2 = 00000000   A3 = 000EFC1A
A4 = 0000FF5E   A5 = 000EFBF8   A6 = 000EE5A0   A7 = 000EF9EC
>
```

Machines appreciate the aesthetics of abstract art. People like to look at nudes. Are you contemplating the subtle nuances of hexadecimal notation as the artist intended, or are you trying to find the person on the staircase?

In the March 1987 issue of *High Tekkie Times,* the journal of semiconductor paraphernalia for those under the influence of recreational substances, a critique of *Nude Ascending a Staircase* appeared. The review was written by Hunter S. Kafka, a media correspondent whose previous work delved into the deevolutionary work of his great grandfather, Franz. The younger Kafka had provided credible evidence that through sustained psychopharmacological technique, one could indeed turn into a giant lizard.

Writes Kafka: "In *Nude Ascending a Staircase,* we have the cubist's classic lines confronting the neo-impressionism of a postmodern romanticist. Of course, on an entirely different level, we see a snapshot of a two-bit computer program, splayed open, its disassembled guts assaulting the visual senses.

"Of one thing we can be certain: the artist has intense contempt for humanity. The work reflects a vulgar rendition of anatomical features without the slightest hint of warmth and compassion. The digits line up like wanton fingers. The lewd abbreviations suggest acts that exceed binary decency. The portrayal of characters locked into garish equations offends nature's symmetry, demeaning both subject and medium under the flimsy guise of abstraction.

"That voyeurism has penetrated the microprocessor arena should come as no surprise. Every since the microscope began revealing single cells, science has made a point of unearthing disgusting art forms. Imagination has been picked bare by algorithmic vultures. No tourniquet can arrest this flow of numerics. If the injury must be fought—and I dare say it must—we shall attack with an artist's fervor, with gossip, and with innuendo to make electronic canvasses such as *Nude Ascending a Staircase* the laughing stock of every chic gallery, boutique, and eatery in the land."

The next month, *High Tekkie Times* published this letter in response:

Editor,

Kafka missed the point in his critique of *Nude Ascending a Staircase*. The artist never intended the work to be viewed as an end product. Rather, it's a pursuit of pleasure from the heart of a young machine. Like a rainbow, the work paints shared memories with resplendent hues of code.

For instance, look at the item titled PC. (Kafka calls them equations, which ought to tell you something about the man whose prior writings in the field of pharmacology chronicled his Tuesday magazine stand visits, where he inhaled the ink vapors off fresh copies of *Modern Bride*.) Here, memory is displayed starting at the current value of the program counter (PC).

Although the memory of a machine might seem as convoluted as the table talk of twenty-two trial lawyers, certain tools make memory observation easier. Notice the 6-digit number on the top line. This portrays the "throb of the heart" captured in time. With each throb, a new memory image springs forth. To you, the characters below may be gibberish. But to a machine, they might carry the memory of its first day out of the carton, admired, caressed, gently stripped of packing materials, and plugged into a grounded outlet.

Some say PC stands for *program counter*. Others believe it is short for *passion consummated*. In either case, the artist has drawn a picture that illustrates, across a single line, the current throb of a machine's heart and the specific memory associated with that throb. This is just an example of the artful magnificence that Kafka failed to recognize in his wordy morass. Works such as these show the heart pouring hexadecimal lifeblood to a young machine. If, for only a moment, Kafka would lift his drug-addled head, he would discover a refreshing perspective above the rim of his lowly consciousness.

Mr. Moss

CHAPTER

11

See Dick and Jane Grimace: First Lines of Assembly Code

A sneak peek at a slot mover

Later in this section you will see a single line of assembly language code, but first, a little background on computer architecture.

Some people think computers have brains. That's dumb. Computers don't have brains. They only have hearts. They don't think; they pump. The heart of a computer is called its processor. The 68000 is one type of processor, just like rock is one type of music.

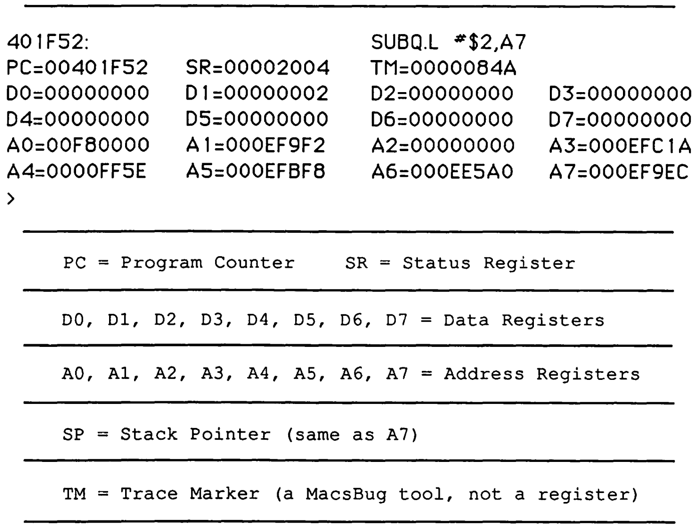
People's hearts are made of muscle. A computer's heart is made of a silicon and metal processor chip. People's hearts are filled with feelings and emotion. A computer's heart is filled with mailbox slots.

Envision a post office. Envision the wall filled with post office box slots. Well, those slots are what fills a computer's heart. Luckily, a computer's processor uses electricity to push mail into and out of slots instead of relying on postal employees. If a processor relied on postal employees, you would have to wait until the computer finished reading the current issue of *People* magazine before it paid any attention to you.

The 68000 heart consists of not much more than eighteen special mailbox slots that push information into and out of a large number of ordinary mailbox slots. These special slots are called *registers.* Think of them as the heart's primary veins and arteries.

Numbers such as 128K, 512K, 1M, and 4M refer to the thousands (K) or millions (M) of mailbox slots in which a machine can collect and deliver information. The number of ordinary slots is also known as the amount of memory that a computer possesses. Ordinary slots do not reside in the 68000 heart. Instead, they sit on other silicon and metal chips and act like the subsidiary veins and arteries throughout your body. Instead of blood, slots transmit numbers.

The names of the eighteen special registers within the 68000 heart are shown to the left of the equals signs in *Nude Ascending a Staircase.* See figure 11-1. Registers DO through D7 are called data registers. Registers A0 through A7 are called address registers. (The A7 register is also called the stack pointer (SP), which you'll use later.) Register PC is called the program counter. Register SR is called the status register. In later chapters, you will discover the purpose of these registers as you use them in your programming.



```
401F52:                          SUBQ.L  #$2,A7
PC=00401F52   SR=00002004   TM=0000084A
D0=00000000   D1=00000002   D2=00000000   D3=00000000
D4=00000000   D5=00000000   D6=00000000   D7=00000000
A0=00F80000   A1=000EF9F2   A2=00000000   A3=000EFC1A
A4=0000FF5E   A5=000EFBF8   A6=000EE5A0   A7=000EF9EC
>
```

```
PC = Program Counter    SR = Status Register

D0, D1, D2, D3, D4, D5, D6, D7 = Data Registers

A0, A1, A2, A3, A4, A5, A6, A7 = Address Registers

SP = Stack Pointer (same as A7)

TM = Trace Marker (a MacsBug tool, not a register)
```

Figure 11-1

The numbers and letters to the right of the register names and the equals signs are the current contents of these special mailbox slots. Note that each slot has eight character positions. The contents of the slots are numbers expressed in hexadecimal, or base 16, notation.

You saw in a previous section that hex numbers range from 0 to 15, and are represented by 16 characters:

```
0 1 2 3 4 5 6 7 8 9 A B C D E F
```

The hex characters 0 through 9 are easy to remember because they are the same as decimal numbers. The trickier numbers are A through F, which correspond to the decimal numbers 10 through 15, respectively.

For now, that's all you need to know about registers. They are special mailbox slots that have names. Most hold 8-digit hex numbers as their contents. The purpose of nearly all assembly instructions in a computer program is to shovel numbers into and out of one or more of these registers. Remember, the 68000 processor is the heart, the registers are the primary veins and arteries, the ordinary slots are the subsidiary veins and arteries, and the numbers (often in hex format) are the blood.

This is an example of a line of code a programmer types on the keyboard to create an assembly language program:

```
MOVE.L  D1,(A1)+
```

D1 and A1 are the names of special register slots. MOVE.L is one instruction from a set of instructions that manipulate the 68000 processor and its memory slots. The example line of code also uses a space, a comma, parentheses, and a plus sign. Your task is to understand how these components work together to manipulate information in slots.

Because everything that a computer knows is contained in slots, programs are written to manipulate slots. Pascal, C, and other programming languages manipulate slots implicitly. Assembly language, from which other languages are created, manipulates slots explicitly.

If you are guessing that this example of assembly language code was chosen as an introduction because it is among the simpler, you are wrong. It is among the more difficult. You are not expected at this point to understand what the code does. The purpose of this chapter is to reaffirm two facets of programming that you already know.

- Slots (register slots in the processor and hex address slots in memory) hold all the information a computer knows.
- Assembly language moves, rearranges, tests, adds, subtracts, multiplies, divides, inverts, shifts, branches to, compares, and otherwise manipulates slots.

Never enough Fourplay

Get ready. You are going to see a complete assembly language program. Its name is Fourplay. If computer languages give you hives, make believe Fourplay is the best paragraph from a French novel titled *Pleasures Among the Alps.* The translation to English might disappoint you. But you will see that programs, like dirty books, can be written without months of monastic discipline, deprivation, dispossession, or worse yet, attending school.

The code (called the *source code*) of Fourplay is shown in listing 11-1.

Listing 11-1

```
                                        ;Program FourPlay

            INCLUDE 'Traps.a'           ;define Toolbox traps
            INCLUDE 'SysEqu.a'          ;define ScrnBase

            MAIN
            MOVE.L  ScrnBase,A1         ;load screen base address
            MOVE.W  #5471,D0            ;screen size in long words
            MOVE.L  #$44444444,D1       ;screen pattern 01000100

FourLoop                                ;begin screen fill
            MOVE.L  D1,(A1)+            ;put pattern on screen
            DBRA    D0,FourLoop         ;loop until size exhausted

Wait                                    ;begin button wait
            SUBQ    #2,SP               ;make room on stack
            _Button                     ;call button trap
            TST.B   (SP)+               ;set Z flag accordingly
            BEQ.S   Wait                ;loop if Z is set (no press)

            _ExitToShell                ;return to Desktop/Shell
            END                         ;code end directive
```

When you run Fourplay, the Macintosh screen fills with thin black lines on a white background. When you press the mouse button, the program ends and the screen returns to the Shell format. Figure 11-2 is a picture of the Macintosh screen after you run Fourplay.

The contents of the Fourplay program are aligned in four columns. When you type Fourplay onto the screen of the computer, use the Tab key to keep the columns orderly.

Take a moment to examine some of the abbreviated instructions in the second column of Fourplay:

```
MOVE.L
MOVE.W
DBRA
SUBQ
TST.B
BEQ.S
```

These are members of the primary instruction set (for all computers using the 68000 microprocessor) that controls the register and memory slots (remember the mailbox analogy) of a computer. All primary instructions, plus many more instructions specific to the Macintosh computer, are described in the dictionary in part 3. Feel free to flip to part 3 whenever you get curious about a particular instruction. For now, you should recognize that assembly instructions are aligned in a single column.

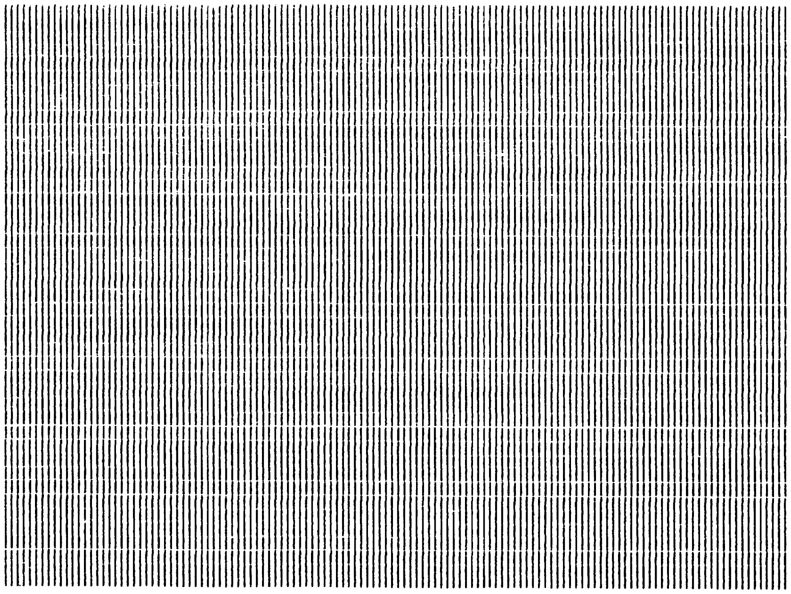


Figure 11-2

Here are two more instructions in the second column:

```
_Button
_ExitToShell
```

These are members of an additional instruction set, particular to the Macintosh, that enhances the writing of assembly programs. This type of instruction accesses the Macintosh Toolbox, a powerful, built-in resource that Macintosh programs use often. You can recognize a Toolbox instruction by the underscore character, which always precedes the instruction's name.

Here are the three remaining instructions in the second column:

```
MAIN
INCLUDE
END
```

These are directives to the MPW assembler. Directives give information about a program's structure and contents to help an assembler efficiently build a standalone application.

Because 68000, Toolbox, and directive instructions belong in the second column of an assembly program, you might be wondering what goes in the first, third, and fourth columns. A quick glance at Fourplay shows you that the first column has words, the third column has a mix of numbers, names, and symbols, and the fourth column has short phrases.

The four columns of an assembly program could be titled:

```
Label   Instruction     Subject             Comment
```

A parallel in English grammar is:

```
Martha says:  Come to  house, Mr Moss.  (begin the fun)
```

Each column has a distinct purpose:

First column	Labels who is speaking.
Second column	Instructs which action to perform.
Third column	Provides the subject to be acted upon.
Fourth column	Makes a comment about the purpose of the other columns.

The same sentence, closer to assembly's grammar, is:

```
MarthaSays:   MOVE.W   MrMoss,myHouse      ;begin the fun
```

Or, substituting the number one for Mr. Moss and address register A1 for Martha's house:

```
MarthaSays:   MOVE.W   #1,A1   ;Moss to Martha's for fun
```

Another name commonly used for instructions is *operation.* Another name commonly used for subject is *operands.* If you feel comfortable using a prissy word like operand, then use it. But sculptors use tools to bend solids, musicians use instruments to tweak sound, and programmers use instructions to manipulate subjects.

To bore a cabbage to coleslaw

"Yes, our computer science textbook would bore a cabbage to coleslaw. Life is full of boring, tedious things. The better to prepare our students. The purpose of schools is to educate. We like Mr. Moss's *Fear and Loathing with 68000 Assembly Language,* but his entertaining style sets a bad precedent. We need a serious book, one that never strays from the subject.

"We want a book that concentrates on structured programming. We want a book that is clear and comprehensive. We want a book that enables students to understand the fundamental concepts of programming, and provides examples of good technique. We want a book that explains the instruction set and addressing modes along pedagogic principles. That's pedagogic, the adjective of pedagogy, the science of teaching.

"Here at Uppity University we view teaching as a science. As scientists, we get to make mistakes and act pompous under the auspices of empirical pursuit. That's why we chose to use Hunter S. Kafka's *Programming the 68000 in Purgatory* as our textbook. We owe it to our students to present the facts and nothing but the facts, as best we choose. There is so much to learn, so we simply cannot waste time reading about colorful customs, personalities, and romantic adventures."

Look at the following two paragraphs—the first by Kafka, the second by Moss—to see which imparts the more lasting knowledge.

`MOVE.L D1,(A1)+` is an assembly language instruction that places the contents of data register D1 into the memory address specified by `(A1)+`. The instruction uses the indirect postincrement addressing mode. The long word contents of D1 are placed into the address specified by the contents of address register A1, then A1's contents are incremented by a long word.

A shy boy wants to tell a girl that he thinks she is hot stuff. He rolls up a long word note and puts it in a donut named D1. He walks over to her house on A1 street, but she's not home. She's visiting her friend down the block. So the boy walks down the block to her friend's house (to deliver donut D1 to the address pointed at from address A1). He sets down donut D1, rings the bell, then jumps over the hedge to the house next door. The assembly instruction `MOVE.L D1,(A1)+` performs the boy's actions by delivering the contents of slot D1 to the slot address pointed to by `(A1)`. After the delivery is made, the slot address is incremented (+) by a long word.

CHAPTER

12

The ABC's of Blocks of Code

A review of terms for the memory blocked

There is more to Fourplay than can be covered in this small section, but you should be able to follow the path of program execution, as well as identify the five functional blocks of code. Lines have been added to Fourplay to mark the five blocks. But first, we'll review some common programming terms.

What's a block?

A block is a series of lines of program code that accomplishes a task in the same way that a paragraph is a series of sentences that conveys a message. A programmer separates code into blocks for clarity. MPW allows more formal blocks called *modules* and *segments* for enhancing large programs.

What's an assembler?

An assembler is a program that reads assembly language code, then organizes the code into a ready-to-run program. A programmer types code into the computer using an editor such as the MPW Shell, then directs the assembler to construct the program. An assembler reads assembly code in the same way that a Pascal compiler reads Pascal code or a C compiler reads C code.

What is program execution?

A computer executes program code by performing the tasks, line by line, that the programmer has typed.

What is the path of program execution?

Each line of a program's code tells the computer, either implicitly or explicitly, which line of code to execute next. The order in which lines of code are performed is the path of execution. Unless otherwise directed, execution begins with the topmost line of code.

Listing 12-1 shows Fourplay separated into blocks. An upcoming chapter contains a complete, line-by-line explanation of how the commands of Fourplay fill the screen with thin black lines on a white background.

Listing 12-1

```
;-----------------------------------BLOCK 1-------------------------

                                   ;Program FourPlay

            INCLUDE 'Traps.a'      ;define Toolbox traps
            INCLUDE 'SysEqu.a'     ;define ScrnBase

;-----------------------------------BLOCK 2-------------------------

            MAIN
            MOVE.L  ScrnBase,A1    ;load screen base address
            MOVE.W  #5471,D0       ;screen size in long words
            MOVE.L  #$44444444,D1  ;screen pattern 01000100

;-----------------------------------BLOCK 3-------------------------

FourLoop                           ;begin screen fill
            MOVE.L  D1,(A1)+       ;put pattern on screen
            DBRA    D0,FourLoop    ;loop until size exhausted

;-----------------------------------BLOCK 4-------------------------

Wait                               ;begin button wait
            SUBQ    #2,SP          ;make room on stack
            _Button                ;call button trap
            TST.B   (SP)+          ;set Z flag accordingly
            BEQ.S   Wait           ;loop if Z is set (no press)

;-----------------------------------BLOCK 5--------------------------

            _ExitToShell           ;return to Desktop/Shell
            END                    ;code end directive
```

The five blocks of Fourplay, from top to bottom, do the following:

- Block 1 equates memory slot addresses with English names. (Otherwise, the addresses would have to be referred to by hex numbers.)
- Block 2 puts numeric data representing the screen location, the screen size, and the black-and-white dot pattern into register slots.
- Block 3 moves the selected dot pattern into the first screen address slot, then repeats the process for each successive screen address slot.
- Block 4 tests to see if the user has pressed the Macintosh mouse button, and continually repeats the test until the button has been pressed.
- Block 5 returns the screen to the Shell format and marks the end of the assembly code.

More explicit blocks of Fourplay

This section reviews Fourplay, a short computer program that consists of seventeen lines of code that a programmer has typed on a computer keyboard. The programmer has grouped the lines into five blocks to help illustrate the tasks the program performs. Lines preceded by a semicolon (the semicolon does not affect the execution of the program in any way) help the programmer easily distinguish the five blocks.

The overall task of Fourplay is to fill the computer screen with a pattern of thin vertical lines, then return the screen to normal when the user presses the mouse button. In the previous chapter, you saw an illustration of the thin lines in the screen display after Fourplay was executed.

The following section is a more detailed explanation of the task that each block of code performs. It shows how five smaller tasks accomplish Fourplay's overall task. Unless you already know assembly language, you should not expect to understand the individual lines of code.

Block 1

The first block contains two INCLUDE statement directives. INCLUDE directs the computer to search small dictionaries so that certain terms used in the code are equated with appropriate hex slot addresses. The small dictionaries used in Fourplay are 'Traps.a' and 'SysEqu.a', and are represented as files on an MPW disk.

Block 2

The second block of code contains a MAIN directive and three MOVE statements. MAIN gives the assembler a starting point for 68000 and Toolbox code. The instructions MOVE.W and MOVE.L place the subject matter of the source (notated before the comma) into the destination (notated after the comma). MOVE commands fill register slots with values. The values in these slots help determine screen location, size, and content.

Block 3

The third block contains three lines. In the reference column, FourLoop marks the beginning of a loop that clears the screen except for thin black lines. This block is called a loop because the three lines are executed repeatedly to make the pattern fill the entire screen. Because the DBRA instruction has an automatic decrement feature, this repetition takes place without the programmer having to retype commands over and over.

Block 4

The fourth block begins with the reference marker Wait. The Wait loop waits for the user to press the mouse button. The commands in this loop are repeated until the button is pressed.

Block 5

The last block contains two statements. The Toolbox instruction _ExitToShell executes the current Shell program. The last line, containing the directive END, signals to the assembler that the end of the code has been reached.

The Moss Man revealed

Scientists can carry pretensions that would gag a maggot from across a lecture hall. Some of the most hideous verbiage since the Huns invented bathroom graffiti has started with the words, "Scientific studies have shown. . . ." If untrue words began to sweat and palpitate, you'd need thigh-high boots to wade through the journals of muck that had shaken off library shelves.

As luck would have it, no one reads what scientists write until it's condensed for newspapers and magazines. Editorial policy dictates that no scientific exposé be any longer than what the average person can read in a single visit to the can. Poetic justice lives.

After the necessities of work, food, sleep, and prime time television have been sated, a spark of inquisitiveness prompts humans to reach out and acquire new experiences. Some reach for a book; some seek a teacher. Some reach for a bottle; some smoke the flowers of plants. Some reach for mountains and lakes; some visit Club Med. Some reach for their mate; some reach within themselves.

These inquisitive moments are what science is about. The traditional sciences that send rockets to Uranus and wage battle with microorganisms benefit from an obtuse vernaculuar comprised of words worse than *obtuse* and *vernacular.* But not all science needs such pomp. The science of experience knocks the socks off the textbook variety. That's because the finest contribution science has to offer—after the essentials of food, shelter, and well-being are fulfilled—is cheap thrills.

Shed of pretensions and relieved of the tedium of human survival, a science explorer can go about the business of concocting a cheap thrill, something to take the bite out of inescapable mortality. For instance, some people find a cheap thrill with pomology (from the Latin *pomum* meaning *apple*), the science of fruit culture. (Pomology was chosen as the example because it precedes *pomp* in the dictionary, a word that had to be looked up for a previous paragraph.)

More people find a cheap thrill with synchronicity (from the Greek *syn* meaning *together* and *chronos* meaning *time*), the science of coincidence. Discovering a fine-sounding science like pomology while looking up pomp is an example of synchronicity.

If fruit is not your thing and you fear that flaky stuff like synchronicity will irritate your friends, you might consider computer science. It's a fairly new science, one that shouldn't embarrass you, and it's discussed at length in this book. You might not get rich. You might not get aroused. But as far as cheap thrills go, knowing how a computer thinks will let you grow old a little less daunted by the technology of your generation. Getting daunted ought to be done alone with your mate with the door of your room closed.

From a question and answer session with P. Moss, February 14, 1987.

What do you need to program a computer?

A computer. You can buy one at a store. You'll also need a language. Languages come on disks that get inserted inside computers. They are often sold by the same stores that sell computers. After you have a computer and a language . . .

I don't understand. What's a language?

If you think of a computer as your car's cassette deck, then a computer program is a cassette tape. You can buy prerecorded tapes that contain word processors, games, spreadsheets, and so on. You can also buy blank tapes on which to record your own programs. Computer languages are a special kind of blank tape. They don't do anything useful on their own, yet they are specially designed to understand programs you type in.

Just to refresh my memory—what's a program?

A program is information that computers play, just like music is information that cassette decks play. People create music by twanging strings, blowing into hollow instruments, or striking resonant objects. People create programs by typing peculiar looking commands on a computer keyboard, then making the computer organize those commands into a ready-to-run program.

So after I've got a computer and a language that's like a blank cassette tape, I type in peculiar looking commands to create a program?

Simple as rhubarb pie. First, you enter the commands, also called the program code. Then, if you are using assembly language, you make the computer organize the commands by a process called assembling and linking. Then you or anyone else is ready to run the program.

And after I know how to program, I can make zillions of dollars and drive a BMW and take vacations where only attractive people go?

I don't see why not.

It sounds too good to be true. What's the catch?

You don't want to know.

Yes, I do. This could be a career decision. Computers are hot. I've seen the future of the good life and its name is computers. We'll be able to feed the hungry, shelter the homeless, heal the sick. Computers are wonderful. So what's the catch?

I hate to be the one to tell you this, but scientific studies have shown unequivocally that computer programming destroys brain cells and causes chromosome damage. A corporate whitewash has kept the public ignorant, but this will change as casualties mount. With the advent of personal computers, the programming base has grown large enough to substantiate what programming pioneers feared all along: People who program become the walking dead. Mutant time bombs. Zombies.

I don't believe you. It can't be. I know programmers and they're not . . . well, they don't seem . . . I mean, still, what's this about chromosome damage?

Impaired reproductive capacities. Some of these people are so bad off they can't even meet a mate. They sit home every night in front of a cathode ray tube. Turn off their computers, and they go for Ted Koppel's *Nightline* show and *Late Night with David Letterman*. Very sad.

But what about you? You program. You've written books about programming. You're saying you're brain damaged?

I was fortunate. I met a kind, intelligent woman who discovered me wasting away. I was given an experimental rehabilitation treatment for silicon abuse. She spent entire nights by my side.

You mean your treatment was . . . that.

Not all medicine is disagreeable.

Are you recovered now? Was any of the damage permanent? Do you still program? How can you teach programming to people while knowing what will happen to them? What happens to those people who never find a mate? Who is this woman?

CHAPTER

13

Back to the Slot That Got You Here

Give me memory or give me . . . I forget

So far, computer memory and the hex numbers that address this memory have been referred to as mailbox or memory slots. This name is as good as any, but 68000 assembly programmers conventionally use names such as bits, nibbles, bytes, words, and long words. Each of these words refers to a distinct size of memory slot, as shown in figure 13-1.

The numbers 128K, 512K, 1M, and 4M refer to bytes. You might think of a byte as a standard-sized memory slot. A conversion chart for other slot sizes would tell you: 8 bits to a byte, 2 nibbles to a byte, 2 bytes to a word, and 4 bytes to a long word. But, at this point, just think of a byte as a memory unit addressed by each hex number. Thus:

$0	1st byte
$1	2d byte
$2	3d byte
. . .	
$A	11th byte
$B	12th byte
. . .	
$1FFFE	127,999th byte
$1FFFF	128,000th byte

. . .

$7FFFF	512,000th byte
. . .	
$FFFFF	1,024,000th byte
. . .	
$3FFFFF	4,096,000th byte

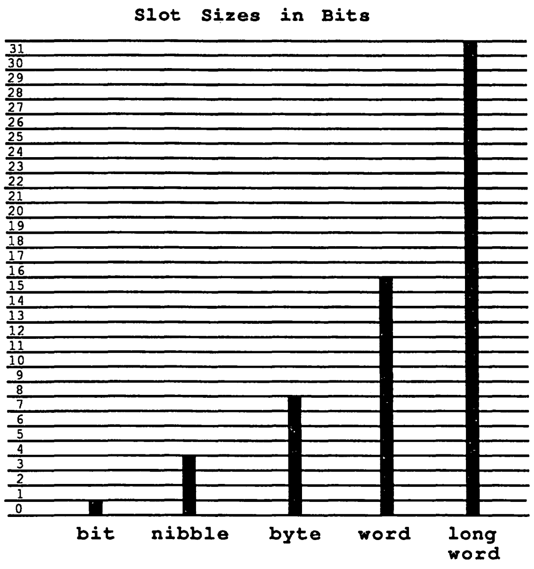


Figure 13-1

Hex numbers are used both to address slots and to represent the contents of slots. Any one of the byte-sized slots in memory can represent a decimal number from 0 to 255. In hex notation, a byte-sized slot can represent a number from #$0 to #$FF.

By custom, a single byte is represented by 2 hex digits (a leading 0 is added if necessary). Often, however, you will see hex digits displayed in groups of either 4 or 8 digits. These longer numbers represent slot sizes larger than the byte-sized slot. Four hex digits (2 bytes) represent a single word. Eight hex digits (4 bytes) represent a single long word. Figures 13-2 and 13-3 show you the slot capacities of ordinary memory and the special registers.

The following notes should help you keep the idea of bytes from putting the bite on you.

- Slots have addresses and contents. Both addresses and contents are often expressed with hex numbers. When you are working with slots, you must know whether you are dealing with the slot's address or its contents.

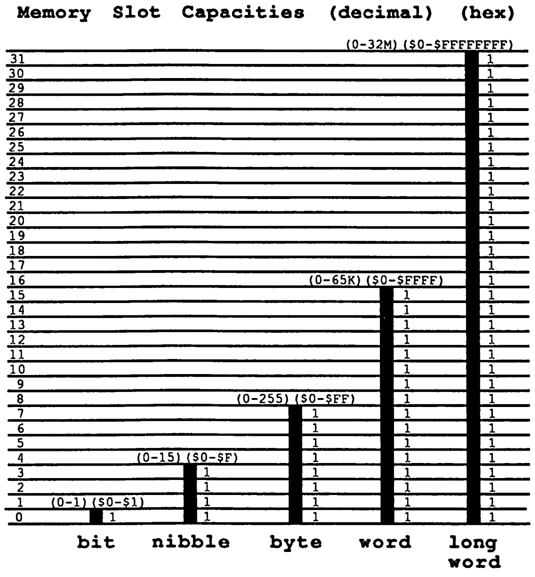


Figure 13-2

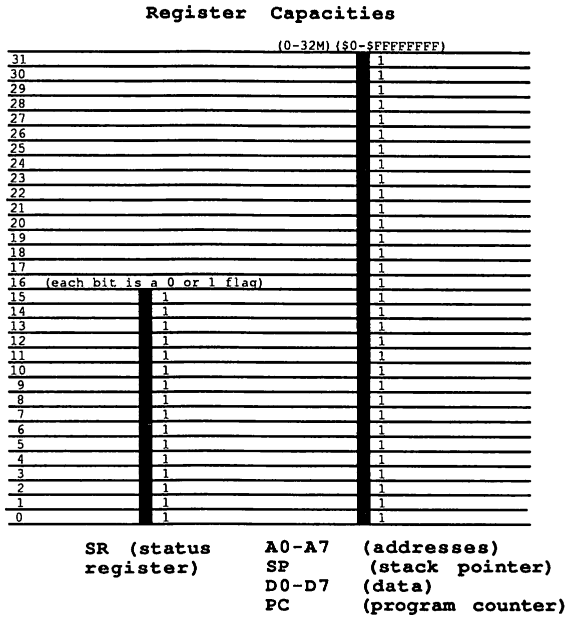


Figure 13-3

- A byte is a slot size that can hold 2 hex digits.
- Each byte-sized slot has a unique address.
- Because a byte-sized slot can hold only 2 hex digits and a hex address usually requires more than 2 hex digits, a byte-sized slot often is too small to contain a hex address.
- A long word is a slot size that can hold 8 hex digits. This is the slot size most often used to hold hex addresses.
- Assembly instructions use the suffixes .B, .W, and .L to refer to byte, word, and long word slots, respectively.

```
.B    $FF          byte
.W    $FFFF        word
.L    $FFFFFFFF    long word
```

Examine the slot from inside and out

Take a look at a slightly expanding, familiar work of art.

```
401F52:                         SUBQ.L    #$2,A7
PC=00401F52    SR=00002004     TM=0000084A
D0=00000000    D1=00000002     D2=00000000    D3=00000000
D4=00000000    D5=00000000     D6=00000000    D7=00000000
A0=00F80000    A1=000EF9F2     A2=00000000    A3=000EFC1A
A4=0000FF5E    A5=000EFBF8     A6=000EE5A0    A7=000EF9EC
>DM  CC60
00CC60  4A1F 67F8 A9F4 0000 0020 0000 011C 0000
```

The bottom line of *Nude Ascending a Staircase* gives a straightforward glimpse of computer architecture. The line is chock-full of hex numbers.

The data on this line represents a very small portion of the Macintosh's large memory. Or to maintain the mailbox metaphor, this line displays a byte slot address followed by the contents of a handful of ordinary mailbox slots.

Here is the example line:

```
00CC60  4A1F 67F8 A9F4 0000 0020 0000 011C 0000
```

The leftmost number is the slot address of the first byte of a series of consecutive memory slots. A computer's memory contains so many memory slots that numbered addresses are used to identify most of the slots. Only the special register slots have two-character names such as D0, D1, D2, A0, A1, PC, and SR.

The question that might be going through your mind is: How many slots of which size are represented by the example line? The answer is: You can interpret the slot size in any of three ways—byte, word, or long word.

Here is the same line interpreted according to slot size:

16 bytes:

```
4A 1F 67 F8 A9 F4 00 00 00 20 00 00 01 1C 00 00
```

8 words:

```
4A1F  67F8  A9F4  0000  0020  0000  011C  0000
```

4 long words:

```
4A1F67F8    A9F40000    00200000    011C0000
```

When interpreted as 4 long word slots, the contents of a memory address take the same form as the contents of the special register slots. You can see the similarity (8-digit hex numbers) by looking at any of the register slots in *Nude Ascending a Staircase.*

At this time, it is important for you to be able to correlate the byte slot address (00CC60) with the memory contents shown beside it. The byte slot address is the address of only a single byte-sized slot. Therefore, the slot contents that 00CC60 addresses is 4A, the first byte-sized slot displayed.

The second byte-sized slot displayed, 1F, has the address 00CC61. To show every single address of every single byte, however, would take up too much room on the screen. Therefore, *Nude Ascending a Staircase* displays slots in 16-byte increments, with each new slot address (always in the leftmost column) 16 larger than the prior address.

If you prefer to interpret the displayed memory as word slots, the addresses perform the same function. The first word-sized slot is addressed by 00CC60, the second word-sized slot is addressed by 00CC62, the third word-sized slot is addressed by 00CC64, and so on in 2-byte (1-word) increments.

Likewise, interpreting the size of the slots as long word, the first slot is addressed by 00CC60, the second slot is addressed by 00CC64, the third slot is addressed by 00CC68, and so on in 4-byte (long word) increments.

Now is a good time to reflect on what you have read:

- Computers are made up of slots.
- Special slots are called registers. They have two-character names, such as D0, D4, A1, and A3.
- Ordinary slots are called memory. They are identified by hex numbers representing addresses.
- The content of both special and ordinary slots is information that can be expressed as hex numbers.

- Sometimes the content of a slot is the address of another slot.
- Slots can have different sizes. Byte, word, and long word are size names for slots.

Slot addresses allow a programmer to locate specific information anywhere in the computer's memory. Everything a computer knows can be tracked down by locating the information's address.

You can see an example of this in the following five lines taken from an alternate version of *Nude Ascending a Staircase.* Here you can see a portion of Fourplay's program code stored in memory.

```
00CC5C: SUBQ.W    #$2,A7
00CC5E: TOOLBOX   $A974     ;Button
00CC60: TST.B     (A7)+
00CC62: BEQ.S     *-00006   ;0000 CC56
```

When MPW assembles Fourplay, it stores a modified form of Fourplay's code in memory slots. Although this code is not identical to the source code you entered (the assembler modifies the code for internal efficiency), you should be able to recognize the modified code as Fourplay's fourth block.

The memory slot addresses of each line of code are in the left column. For example, the Toolbox instruction _Button is at hex address $00CC5E. Notice that here, the hex addresses of each line of code increment by 2 bytes. That's because each of these lines of source code needs two byte-sized slots to be stored in memory.

You might be wondering how a particular memory slot can show lines of program code in one instance, but show hex numbers in another instance. Code and hex numbers represent the same information expressed in different formats. Translating assembly language code into hex code (and ultimately into machine-readable binary code) is one part of what an assembler does.

For example, in the program code TST.B (A7)+ the hex code translation (stored at address $00CC60) is 4A 1F. Depending on how you wish to view *Nude Ascending a Staircase,* you can see memory displayed as code or hex. (Look back to the chapter on debugging, chapter 9, to see how to manipulate pictures such as *Nude Ascending a Staircase.*)

Letters and the Oakland bus stop

Dear Mr. Moss,

I was greatly disturbed by your flippant remarks on brain damage. I don't know which you displayed more—your insensitivity or your ignorance. The people, and the families of those people, who suffer from the many forms of brain damage surely would not find this topic a source of humor or lightheartedness. Perhaps if you visited a rehabilitation center for the mentally disabled or donated some of your time to working with these people, your perspective would change.

Brain damage is a horrible disease that can cripple and kill, and oftentimes sentence its victims and their families to a life of pain, anguish, and devastating hardship. It is no more funny than cancer, heart disease, or severe injury. I think you owe your readers an apology.

Sincerely,
Hunter S. Kafka

Dear Mr. Kafka,

Your letter made me sad. I want to answer you with a child's favorite response: I didn't mean to do it. But I'm afraid you wouldn't understand. You might erroneously think that I'm apologizing for my insensitivity and ignorance. I'm not. What makes me sad is that the words I wrote and the words you read were not the same.

They might look the same. They might sound the same. But inside our heads, they are as different as chicken soup and chicken feathers. I wish that the words *brain damage* that I wrote and that you read were more alike. Then the computer programmers I wrote about would not be confused with the unfortunate people you read about.

I want to call you a dundering lunkhead for even thinking I might be attempting humor at the expense of the disabled. But you wrote a tempered, considerate letter with intentions I agree with fully. Perhaps you would grant me the same benefit if you reread my words.

You might be interested to know how it occurred to me that the phrase *brain damage* applied to my work. There was a year-long period in my early twenties when I did not speak to anyone. I did not speak because I was unhappy and had nothing I wanted to say. I watched TV in my tiny studio apartment, took short walks to the library, and wrote a few angry short stories.

During that year I noticed I had a hole in one of my teeth, a bicuspid. In the hole was decay, and it hurt when I poked inside it with an opened paper clip. So I decided to see the county welfare dentist. I had very little money.

You have to get to Highland Hospital at 7 A.M. and wait about four hours to see the county dentist. So at 6:30 A.M. I was standing at the bus stop on Oakland's MacArthur Boulevard waiting for the 57 Southbound. The only other person at the bus stop was a tall, slightly plump woman with olive skin, close to my age. She was nicely dressed in a business suit, a white blouse with a big bow, and a black velvet hat that had plastic flowers in it. She smelled strongly of perfume, and she was smiling, not at me but as if some thought was pleasing her. When our eyes met, her smile opened slightly, and she said to me, "It's a lovely morning, isn't it?"

I said yes. My voice cracked with its first word in a long time. I gave my best attempt at a smile. It was not half as buoyant as hers.

Her voice was loud and full, though I detected an impediment as she continued to speak. "I've ghat a doctor's appointment to go to thizz morning. Ordinarily, I'd be going to work, but I had to mizz thizz morning."

I told her I was going to a doctor's appointment also. She went on to tell me about her parents, her house, her job at the center, and how much she enjoyed being outdoors on nice days. She asked me questions about my family and what I liked to do. The impediment in her speech seemed to go away the more she spoke. When I noticed she had some trouble with her leg that made her shift her weight awkwardly, I assumed that was the reason for her doctor's visit.

She was unusually cheerful and talkative for a woman standing beside a strange man at an Oakland bus stop at 6:30 A.M., but I knew that a person with my lifestyle should hardly be the judge of unusual behavior. Her warmth and smile charmed me. I had spoken only a few words—she had done most of the talking—but the last thing I said to her before the bus came and took us away has stuck in my mind. I said it without thinking. I *couldn't* think because I had been stunned by what she had told me as an afterthought to her story about how she happened to be going to the doctor today.

This sweet, bright woman said to me, "I have brain damage." And I said back, "We all do."

Mr. Moss

CHAPTER

14

Fourplay: From Head to Toe with Lots of Time for the Middle

Play it again, Peat

For the work of figuring out seventeen lousy lines of programming gibberish, you can tell your family and friends that you basically understand how a computer understands language. That's right, seventeen lines of code. If through your childhood you have taken seventeen bites of creamed spinach, boiled rutabaga, or unbuttered brussel sprouts, you certainly can survive seventeen lines of 68000 assembly.

This is the last you will see of the original Fourplay program. The explanation in this chapter is expanded to help you understand the consequences of each instruction. Remember to use the dictionary in part 3 when you need more information about an instruction.

Before any number in assembly code, the following rules apply.

- The number sign (#) indicates that the number is a quantitative value (as opposed to a memory slot address).
- The number sign alone indicates that the number is a quantitative value expressed in decimal.
- The dollar sign ($) indicates that the number is a hex value (as opposed to a decimal value).
- The dollar sign alone indicates that the number is a hex address.
- Together, the number and dollar signs (#$) indicate that the number represents a quantitative value expressed in hexadecimal.

Listing 14-1 is the source code for the Fourplay program. Figure 14-1 is the screen display produced by running the program.

Listing 14-1

```
                                            ;Program FourPlay

            INCLUDE 'Traps.a'               ;define Toolbox traps
            INCLUDE 'SysEqu.a'              ;define ScrnBase

            MAIN
            MOVE.L  ScrnBase,A1             ;load screen base address
            MOVE.W  #5471,D0                ;screen size in long words
            MOVE.L  #$44444444,D1           ;screen pattern 01000100

FourLoop                                    ;begin screen fill
            MOVE.L  D1,(A1)+                ;put pattern on screen
            DBRA    D0,FourLoop             ;loop until size exhausted

Wait                                        ;begin button wait
            SUBQ    #2,SP                   ;make room on stack
            _Button                         ;call button trap
            TST.B   (SP)+                   ;set Z flag accordingly
            BEQ.S   Wait                    ;loop if Z is set (no press)

            _ExitToShell                    ;return to Desktop/Shell
            END                             ;code end directive
```

Line by line, every block on its body

Block 1

```
INCLUDE  'Traps.a'
INCLUDE  'SysEqu.a'
```

INCLUDE is a special kind of instruction called a *directive*. Directives are special because they operate on the assembly process rather than on memory slots. MPW, like all assemblers, defines its own directives. This contrasts with the 68000 instruction set, which remains the same among all 68000 assemblers.

INCLUDE directs the assembler to look for definitions kept in a separate file. In this case, the file names are 'Traps.a' and 'SysEqu.a'. These files should be on your assembly language disk already. Be sure to include the single quotation marks in your code.

The files 'Traps.a' and 'SysEqu.a' are definition files that allow assembly programmers to use English-like words such as ScrnBase, _Button, and _ExitToShell instead of hard-to-remember and easy-to-mistype hex slot addresses. In the definition files, each English-like word is equated with a slot address. The INCLUDE directive allows the assembler to automatically substitute the proper slot address.

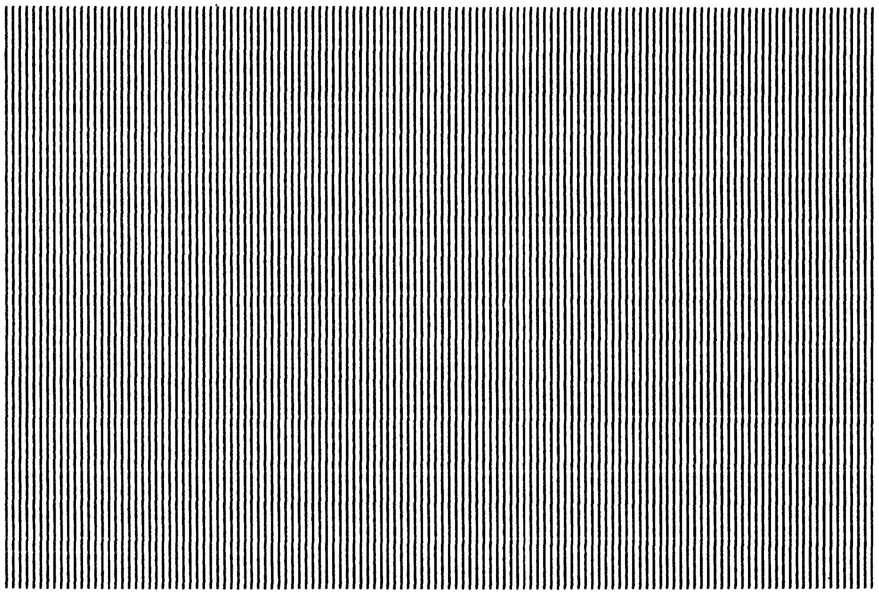


Figure 14-1

Block 2

```
MAIN
MOVE.L   ScrnBase,A1
MOVE.W   #5471,D0
MOVE.L   #$44444444,D1
```

MAIN is the assembler directive that designates the start of the main code module. Code modules are more significant in larger programs because they allow programmers to split programming tasks into smaller, discrete parts.

MOVE.L ScrnBase,A1 places the hex address represented by the system term ScrnBase (defined by INCLUDE 'SysEqu.a') into address register A1. MOVE.W #5417,D0 places the decimal number 5417 into data register D0. MOVE.L $44444444,D1 places the hex number $44444444 into data register D1.

The MOVE instruction moves a subject (notated before the comma) into a destination (notated after the comma). The .L and .W suffixes of the MOVE command indicate the slot size to be affected. The .L suffix indicates a long-word-sized move (4 bytes, the entire size of an address or data register). The .W suffix indicates a word-sized move (2 bytes, half the size of a data register).

In upcoming chapters, you will see the significance of the numbers

#5471 and #$44444444. These numbers are put into data register slots to calculate the size of the Macintosh screen and determine the pattern of black and white dots to fill the screen.

Block 3

```
FourLoop
         MOVE.L  D1,(A1)+
         DBRA    D0,FourLoop
```

In the reference column, FourLoop marks the position of the code to which program execution will loop back after the two instructions are performed. MOVE.L D1,(A1)+ places the contents of data register D1 into a memory address specified by (A1)+. (The particular addresss specified by (A1)+ is discussed in an upcoming chapter.) DBRA D0,FourLoop decrements the contents of data register D0 by 1. It then checks to see if D0 now equals −1 and, if not, branches (redirects program execution) to the reference marker FourLoop.

Remember, the initial values of A1, D0, and D1 are set by the MOVE instructions in the previous block of code. Use the dictionary in part 3 if you need more help understanding instructions such as MOVE and DBRA.

This block of code is called a loop because program flow loops back to a previously recognized reference marker. As a result, the MOVE.L D1,(A1)+ and DBRA D0,FourLoop statements are performed repeatedly until the DBRA instruction decrements the value of D0 to −1, whereby program flow continues at the next consecutive statement—no longer branching to the reference marker FourLoop (dropping out of the loop).

Block 4

```
Wait
          SUBQ     #2,SP
          _Button
          TST.B    (SP)+
          BEQ.S    Wait
```

The fourth block begins with the reference marker Wait. This loop waits for the user to press the mouse button to end the program. (A Macintosh uses a mouse; another computer might use a keyboard key.) The code to accomplish this task involves four statements.

Some of the details in this block involve advanced topics, so don't get hung up trying to understand what has not been explained. The important parts are recognizing the logical flow of statements and becoming familiar with the format.

SUB.Q #2,SP performs a subtraction on a special register called the *stack pointer* (SP). The decimal value 2 is subtracted from the value represented by SP, and the result is stored as the new value of SP. This subtraction sets a place in memory—called the *stack*—that the following instruction uses to store its subject matter.

Next, the Toolbox instruction _Button is performed. The Toolbox, particular to Macintosh computers, supplements the 68000 instruction set with a large number of useful commands. There is no subject matter necessary for Toolbox instructions because the source and destination of their subjects are found in the stack pointer and other registers.

TST.B tests the subject matter (providing a yes or no response) left on the stack by the _Button instruction, and records the result in yet another register called the *status register*. The parentheses and the plus sign beside the stack pointer symbol, (SP)+, are discussed in an upcoming chapter. You might have noticed that the status register is not referenced in the subject matter. Many instructions, including TST, affect the status register implicitly.

BEQ.S evaluates the status register, using the yes or no result to either branch to the marker Wait or drop out of the loop to the last block.

Block 5

```
_ExitToShell
END
```

The last block contains two statements. The Toolbox instruction _ExitToShell returns the screen to the Shell from which the program began. The assembly directive END signals to the assembler that the end of the code is reached.

You have used a sequence of assembly statements to alter the Macintosh screen and, at the user's control, return the screen to the Shell. (Although the statements might look odd, the path of programming logic is straightforward.) You accomplished this by directly manipulating the registers and memory slots that make up the Macintosh hardware. You "spoke" with a language much closer to what a machine understands than PASCAL, BASIC, FORTH, LOGO, LISP, LUST, or others made up by hobnosed university professors.

The depths and dips of science

Computer science is a cruddy term. No one refers to biology science, or physics science, or television and telephone science. Computers don't need the word *science* attached to them.

Science is for kids under thirteen. Science explains heaven and earth, the body, mind and spirit. It explains why magnets pick up staples and why gases rise in the bathtub. Science explains everything to a child under thirteen except why mom and dad's bedroom door is sometimes locked at night. The bedroom door mystery can be explained satisfactorily only by becoming thirteen, fourteen, and fifteen.

At thirteen, science begins to dissolve into innumerable fields of ignorance. A child knows that science must have the answer, whereas an adult knows that science's answer is less fun than food, sleep, sex, fresh air, and playing with the dog. The remnants of science are left for a group of experts who have narrowed their talents to the point that they can't walk and chew gum at the same time.

Some people believe that personal computers will help make science more important to adults. No, the innumerable fields of ignorance will never again solidify to a child's concept of science. But the quotient of pleasure that an adult derives from dabbling in one or more aspects of science could be increased dramatically.

Personal computers might allow the branches of science to be approached without the expense, boredom, pretentiousness, and pipe smoke of college professors, and the gloomy libraries that hold their recorded knowledge. In turn, the treasured few instructors with wit and warmth could allot more time to fine-tuning their comic delivery and urbane nuances. Schools and colleges would retain their primary objective of bringing together single people in search of a mate.

Computers are handy machines. But they are not a relevant subdivision of science. If science must be subdivided, it ought to be done under the categories established by the bumper sticker, "You live, you get sick, then you die."

You live.

This is science at its best. It includes trees, birds, flowers, and ice cream. It also includes 68000 assembly programming and the budding romance of a boy and a girl at the rose garden. Endowed with adventure and pleasure, intricacy and intimacy, an explorer might be lucky enough to fill up seventy years worth of scientific *knowing*. (And almost as many years of the biblical variety.)

You get sick.

This is how most people think of science. It's through viruses, bacteria, genetic mutation, toxic chemicals, environmental pollution, nuclear weapons, radiation, lasers, and the dreaded unknown origin. (The next major scientific discovery will be that viruses are visitors from outer space who enter our bodies for observation and experimentation.)

Then you die.

Euphemistically known as "kicking the bucket," "croaking," "meeting one's maker," "passing away," and "buying the farm," this is the one component of science whose study no one escapes. From the viewpoint of the one to whom it matters the most, it's where science has the least satisfactory answers. (PS: Don't count on computers to help you here.)

CHAPTER

15

The Addresses of Screen Stars

Suffix to say

This chapter, and each one hereafter in part 2, begins with the source code of a complete assembly program. The LightsOut program (in listing 15-1) is nearly the same as Fourplay. Only two changes have been made. First, the screen pattern of #$7F7F7F7F7F is substituted for $#44444444 so that a new effect is created when the program is run. See figure 15-1. Second, new reference names are used with no effect on the program's operation. The similarity of the two programs will give you a chance to examine in greater depth the characteristics of data and addresses.

Listing 15-1

```
                                    ;Program LightsOut

            INCLUDE 'Traps.a'       ;define Toolbox traps
            INCLUDE 'SysEqu.a'      ;define ScrnBase

            MAIN
            MOVE.L  ScrnBase,A1     ;load screen base address
            MOVE.W  #5471,D0        ;screen size in long words
            MOVE.L  #$7F7F7F7F,D1   ;screen pattern 01111111

DoLoop                              ;begin screen fill
            MOVE.L  D1,(A1)+        ;put pattern on screen
            DBRA    D0,DoLoop       ;loop until size exhausted

TryButton                           ;begin button wait
            SUBQ    #2,SP           ;make room on stack
            _Button                 ;call button trap
            TST.B   (SP)+           ;set Z flag accordingly
            BEQ.S   TryButton       ;loop if Z is set (no press)
```

Listing 15-1 *cont.*

```
        _ExitToShell            ;return to Desktop/Shell
        END                     ;code end directive
```

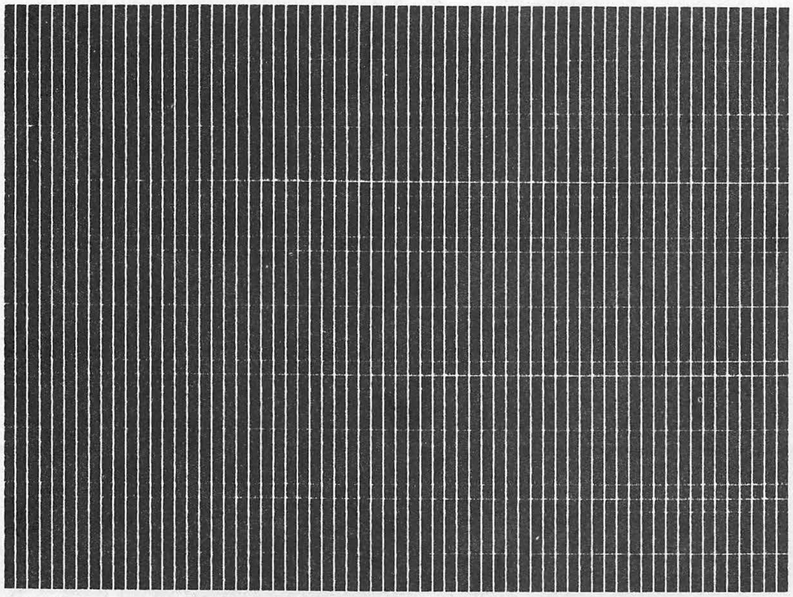


Figure 15-1

Between the primary 68000 commands and the Macintosh Toolbox calls, there are hundreds of instructions in assembly. Each instruction has distinct data and address needs. Faced with a problem this immense, a beginning programmer must adopt the proper attitude, first propagated by the venerable A. E. Neuman: What? Me Worry?

Also helpful is the reminder that assembly manipulates register and memory slots. After you understand these devices, and the numbers that occupy them, your programming task becomes surmountable.

Take a look at these two numbers:

```
#$44444444
#$7F7F7F7F
```

Here are the characteristics that should be going through your mind.

- The number sign (#) means the number is a numeric value, not a memory address.

- The dollar sign ($) means the number is a hex number. You do not need to translate every hex number you see into decimal.
- There are 8 digits in each of these numbers.
- The sixteen data and address registers (slots D0-D7 and A0-A7) can each hold up to 8 hex digits.
- The name for this length of data is a long word.

Find the following line in the code of LightsOut:

```
MOVE.L    #$7F7F7F7F,D1
```

The MOVE instruction places the subject matter of the source (notated before the comma) into the subject matter of the destination (notated after the comma). The long word data #$7F7F7F7F is moved into the register slot D1.

The .L suffix of the MOVE instruction indicates the extent, or length, of the transfer. The .L suffix is short for *long word.* The .W suffix is short for *word.* The .B suffix is short for *byte.* These suffixes are used with many instructions to indicate the extent in size to which the subject matter is affected.

There is also the .S suffix, short for *short,* though it serves only in the context of short branches rather than a measurement of subject matter.

Here are some more observations about MOVE.L #$7F7F7F7F,D1. Because #$7F7F7F7F is a long word, it makes sense that there is an .L suffix on MOVE.L. Trying to move a long word value with MOVE.W or MOVE.B would undoubtedly cause problems.

D1 is a data register capable of holding a long word value, so it should have no trouble accommodating #$7F7F7F7F. If D1 holds a value prior to the MOVE.L command, the old value of D1 is wiped out.

#$7F7F7F7F is a number. D1 is a register (a slot). The statement MOVE.L D1,#$7F7F7F7F would not make sense. You cannot transfer a register (or anything) into a number. The destination of a MOVE instruction must be some sort of slot.

The statement MOVE.L D1,D2 would make sense. You can transfer the long word contents of register D1 into register D2. After the execution of MOVE.L D1,D2, the original contents of D1 are not erased or changed. The MOVE command simply places a copy of the source into the destination.

You will have lots of opportunities to practice with bytes, words, and long words. Long words might be the easiest to understand because they fill up all the space in the data or address register slot. Bytes and words only fill part of the space of a register slot, so you have to figure out which part of the register is occupied by the byte or word.

The problem of how to keep track of bytes, words, and long words becomes more acute when dealing with memory slots. Horror of horrors, it turns out that assembly programmers must sometimes count bytes. For example, if you put a long word at address $4, then the next highest available address is $8. Remember, memory is counted in bytes, and a long word occupies 4 bytes.

Bits of the postman's leg

In the previous section you examined long words, the length capacity of your data and address registers. At some point in your programming, you'll remember all the size relationships, such as 32 bits to a long word, 16 bits to a word, 8 bits to a byte, and so on. More importantly, you should recognize that a bit can take one of two values: 0 or 1.

Here is a row of 4 bits equivalent to $7:

```
0111
```

Here is a row of 8 bits equivalent to $7F:

```
01111111
```

Here is a row of 32 bits (a long word) equivalent to $7F7F7F7F:

```
01111111 01111111 01111111 01111111
```

If all the bits that a 1M Macintosh could store were printed here, this book would be heavier than the mood at an IBM sales conference.

The small-screened Macintosh reserves over 160,000 bits (20K bytes) of its memory to display black or white dots on its screen. These dots create all the graphics, including text, that you see on the screen. A single bit corresponds to a single dot. If the screen bit equals 0, the dot is white, and if the screen bit equals 1, the dot is black.

If you assigned all the screen memory bits to 1, the Macintosh screen turns black. Likewise, if all screen memory is assigned values of 0, the screen turns white. In LightsOut, the pattern set by $7F (01111111) turns the screen black with thin white lines.

Examine the following code in LightsOut:

```
MOVE.L   ScrnBase,A1
MOVE.W   #5471,D0
MOVE.    #$7F7F7F7F,D1
```

The system term ScrnBase (defined in the 'SysEqu.a' file) references the lowest address of the portion of memory devoted to creating the Macintosh screen. Above this base address are 5,471 long words of memory, each containing 32 bits assigned values of 0 or 1, that map the black and white dots of the screen. By assigning the bit pattern of #$7F7F7F7F to all 5,471 long words of screen memory, the entire screen is painted.

Try changing the screen pattern of LightsOut. You already saw the pattern #$44444444 used in Fourplay. You might want to try to make the screen entirely black. If you can accurately predict how the screen will appear by the hex number you choose, you'll have gone a long way toward understanding the bit, byte, word, and long word mathematics of assembly.

Figure 15-2 is the screen map of Fourplay, where each long word has the value 01000100. Remember, 5,472 long words multiplied by 4 bytes per long word equals 21,888 bytes.

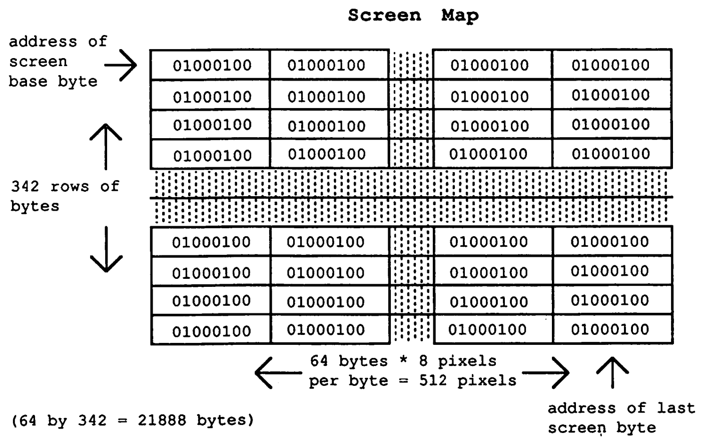

Screen Map
address of screen base byte
342 rows of bytes
01000100
64 bytes * 8 pixels per byte = 512 pixels
address of last screen byte
(64 by 342 = 21888 bytes)

Figure 15-2

To rehash a few old topics: Assembly programmers manipulate registers and memory slots. Registers have names like A4, D2, and SP. Memory slots are referenced by addresses. Because memory slots come in many sizes (bits through long words), an exact knowledge of which memory slot to manipulate becomes crucial.

Consequently, the method of addressing memory slots represents a large part of any programming task. Take a look at the memory map in figure 15-3. Near the top of the curves, find the portion of memory that holds the screen information. In the preceding section, you saw that this section of memory was addressed using ScrnBase, a system equate term that represents the bottom address of screen memory. From this base address, you were able to address the next 5,471 long words of memory above the base address that map the rest of the screen.

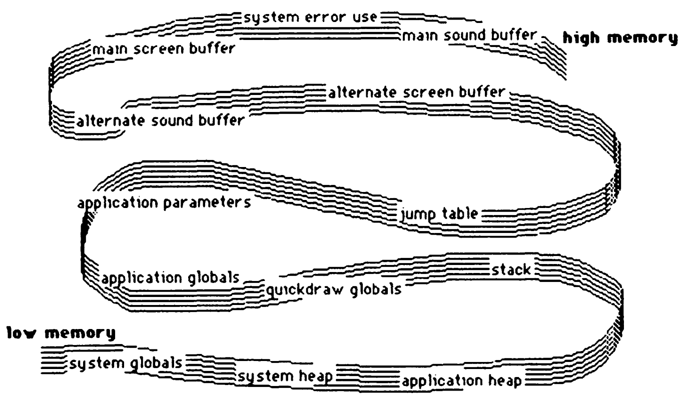


Figure 15-3

Figure 15-4 shows screen memory displayed in a more slot-like format.

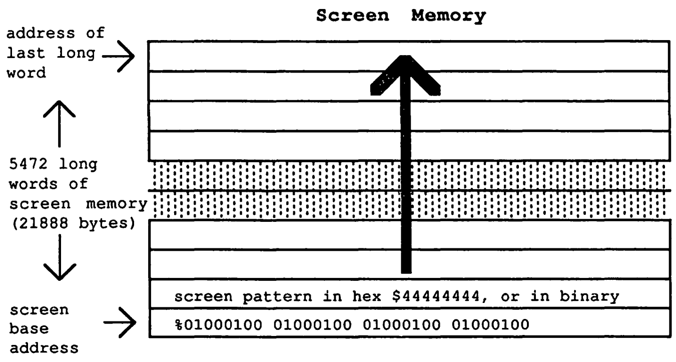

Screen Memory
address of last long word
5472 long words of screen memory (21888 bytes)
screen base address
screen pattern in hex $44444444, or in binary
%01000100 01000100 01000100 01000100

Figure 15-4

The statement that helps you address all the long words of memory above the ScrnBase address is:

```
MOVE.L  D1,(A1)+
```

This form of addressing is one of many different ways of addressing memory. How many different ways are there? Who the heck cares? This isn't a book of statistics. If you tried to read about all methods of addressing at once, you'd forget about the first long before you got to the last. (When Mr. Moss starts getting surly you know a *fear and loathing* sidetrack is near.)

The command MOVE.L D1,(A1)+ places the long word contents of register D1 into the location specified by register A1, and then increments the address of that location by a long word. This is called postincrement indirect addressing. This addressing mode results in the following. The contents of D1 are not placed into register A1 because the parentheses around A1 indicate an indirection. The destination specified by (A1) is not the register itself, but the location addressed by the contents of A1. For example, if D1 contained #7 and A1 contained $300, the statement's indirection would move the value 7 to memory location $300.

The plus sign after (A1) indicates a postincrement of the contents of A1. The address contained in A1 is incremented by one unit after the MOVE takes place. For example, if D1 contained #7 and A1 contained $300, the statement's postincrement indirection would move the value 7 to memory location $300, then increment the contents of A1 by a long word. D1 still contains #7, memory location $300 also contains #7, and A1 contains $304. The unit of increment for the statement is 4 bytes (a long word).

At this point, you may be able to see the convenience of this kind of addressing for the task of filling all 20K of screen memory. With the ScrnBase address (the original contents of A1) automatically incremented by the (A1)+ addressing mode, you can repeatedly move the pattern (stored in D1) until the entire screen is filled.

The statement DBRA D0,DoLoop keeps a count of the exact number of repetitions necessary to fill the screen. The original value of D0 is assigned to #5471, the number of long words (less the one already filled) in screen memory. The DBRA instruction (*test condition, decrement, and branch*) subtracts 1 from D0 before each branch to DoLoop. It continues to do so 5,471 times, until the test condition shows D0 equal to −1, which results in no branch.

The Macintosh 2 and Macintoshes using full page displays have a different screen size than #5472 long words. If you run Fourplay, LightsOut, or the next chapter's program on a machine with a different screen size, the effect is different. But current, and likely future, Macintoshes use the system term ScrnBase as the name for the bottom address of screen memory. Also, the system equate term ScreenBits.bounds can be accessed to find the exact boundary coordinates of the current screen in use.

This illustrates an important lesson in Macintosh programming: Protect your programs from system obsolescence. Mr. Moss has chosen to use a creaky device only because it allows you to write extremely short sample programs without creating windows. All programs after chapter 16 are done in Macintosh windows, as should yours.

Conversations on computers

Question of the day
Who would you most like to replace your keyboard with?
Asked at the Berkeley Macintosh Users Group

Tom Bellins, age 18, English major, junior
Daryl Hannah. If Daryl Hannah was my keyboard, I would be willing to type in the *World Book Encyclopedia.* In each language. With annual supplements. Make that daily.

Sue Weatherly, age 24, biochemistry, graduate student
How about Kris Kristofferson or Willie Nelson? It would be nice to have a keyboard that sings. I'd use Willie during the days and Kris at night.

Walter McIllheney, age 54, civil engineer
If you are asking who I would like to have my hands on, the answer would be my wife. There is no one else I would even consider. It's good to work with someone you know.

Patricia Hanes, age over 30, teacher
On a Macintosh you don't need the keyboard much. You've got a mouse to roll around. Now if I could pick a man to point and click with, I'd want someone real smooth. Someone who would fit in my hand comfortably. Someone kind of cute, like the guy who runs these meetings.

Michael Feinstein, age 28, accountant
That TV newsperson Connie Chung would be nice. She's smart, articulate. She seems like someone you could count on. I don't think her keys would ever get stuck.

Mr. Moss, age 17 and holding, gardener
My brain. Let the computer take input directly. Confuse the heck out of it. Then the computer could choose who it wanted for a keyboard. It would probably fall in love with my girlfriend.

CHAPTER

16

Check the Status of the Stack, Captain

A Stack at the International House of Carbohydrates

The NotOverYet program in listing 16-1 makes a final revision to Fourplay. Instead of moving a long word pattern into screen memory, you use the NOT instruction to invert all the dots on the screen. Figure 16-1 is the screen display produced by running NotOverYet. Much of the code from Fourplay and Lights-Out remains intact, giving you a familiar setting to examine the all-important stack and the ever-present status register. (If you think variations on Fourplay have gone on too long, Mr. Moss and his girlfriend outvote you two to one.)

Listing 16-1

```
                                        ;Program NotOverYet

            INCLUDE 'Traps.a'           ;define Toolbox traps
            INCLUDE 'SysEqu.a'          ;define ScrnBase

            MAIN
            MOVE.L  ScrnBase,A1         ;load screen base address
            MOVE.W  #5471,D0            ;screen size in long words

DoInvert                                ;begin screen inversion
            NOT.L   (A1)+               ;invert one long word
            DBRA    D0,DoInvert         ;loop until size exhausted

TryButton                               ;begin button wait
            SUBQ    #2,SP               ;make room on stack
            _Button                     ;call button trap
            TST.B   (SP)+               ;set Z flag accordingly
            BEQ.S   TryButton           ;loop if Z is set (no press)

            _ExitToShell                ;return to Desktop/Shell
            END                         ;code end directive
```

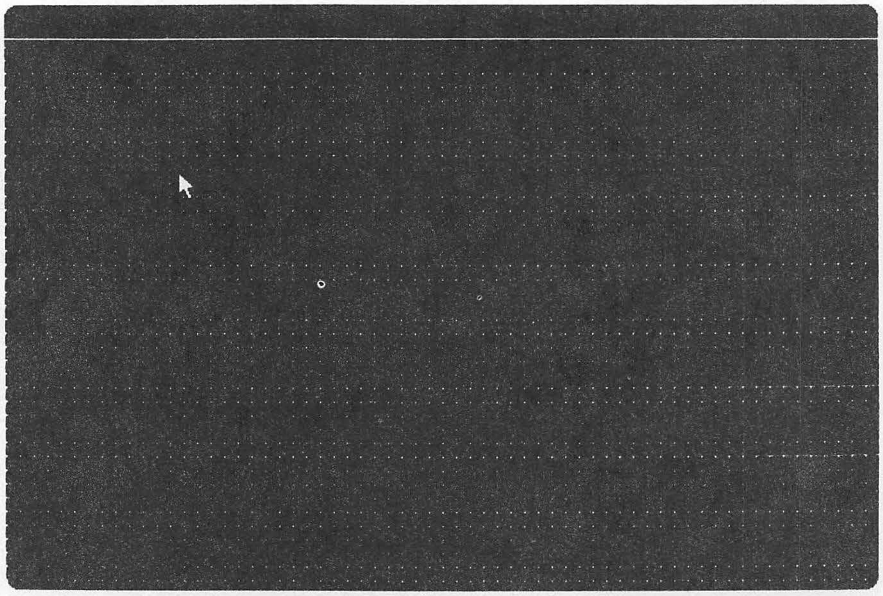


Figure 16-1

NotOverYet uses one less instruction than its forerunners. The MOVE.L instruction that places the screen pattern into register D1 is omitted. Instead of inserting a long word pattern into screen memory, this program changes the value of each screen bit, effectively changing all black dots to white, and white dots to black.

Without a screen pattern, there is no need to use D1. Consequently, the statement NOT.L (A1)+ requires just a single subject. You might remember that the previous two programs use a MOVE.L D1,(A1)+ command to place a screen pattern into the screen's memory locations. The NOT.L instruction uses the contents of its single long word subject, and inverts all 32 bits.

The addressing mode of NOT.L (A1)+ remains the same as MOVE.L D1,(A1)+. The parentheses around A1, followed by a plus sign, indicate postincrement indirect addressing. This is a convenient, automatic means of inverting the bits of all 5,471 long words of screen memory above the ScrnBase address.

Chapter 15 explained how the subject of indirect addressing was not the address in register A1, but the contents of the memory location addressed by A1. The postincrement feature adds a single unit (in this case, a long word) to A1's address, so that the next time the instruction is performed, the subject is the contents of the next higher memory location.

The only other change in NotOverYet is the reference name that marks the screen changing loop. You can choose any name to mark a section of code. You'll reward yourself, however, if you select a name that clearly reflects the activity of the code. Also, if you decide to change a reference name, make sure to update the subject matter of all instructions that branch to that reference.

If you run NotOverYet from the MPW Shell, you'll get a different result than if you first return to the Finder desktop and run the program. Try moving windows and icons on the desktop, and run NotOverYet again. It's strange to run a program and have the screen be inverted rather than redrawn, but you really get the sense of directly manipulating screen memory.

You have already read about Toolbox instructions. Here are some observations to refresh your memory.

- Toolbox instructions are always preceded with the underscore character.
- The subject matter of a Toolbox instruction, unlike a 68000 instruction, is not listed in the adjacent column. Instead, separate 68000 instructions listed before the Toolbox instruction place subject matter on the stack or in a register. Thus, when the Toolbox is called, instructions automatically find their subject matter.
- For most Toolbox instructions, you must prepare the stack or registers with the appropriate subject matter. You can use the dictionary in part 3 to look up the subjects needed by the Toolbox calls in this book.

The only subject matter necessary for the _Button instruction is a 2-byte space left open on the stack to store the boolean (yes/no) result. The statement SUBQ #2,SP provides this space by subtracting #2 from the stack pointer.

The stack is a group of consecutive memory locations. These memory locations are ordinary in every way except they use the stack pointer (SP), a special register that provides quick and economical access to the contents of these locations. The stack pointer holds a single address at any one time. And, within limits, it can be moved up or down to point to any of the stack's memory locations. Figure 16-2 shows the stack as a downward pointing arrow. The stack pointer register contains the address of the slot at the arrow's tip.

Understanding how to manipulate the stack and the stack pointer gives you access to nearly all Toolbox commands. Here are some notes on their use.

The stack pointer helps define the size of the stack because it always points to the extensible boundary of the stack. The other boundary of the stack is a base address, that is, it cannot extend or compact the stack.

Because the stack pointer always points to the movable boundary, moving the stack pointer away from the other boundary makes the stack grow, and moving the stack pointer toward the other boundary makes the stack shrink.

You can move the stack pointer yourself, or the computer can do it automatically. When you push subjects onto the stack, the stack grows. When you remove subjects from the stack, the stack shrinks.

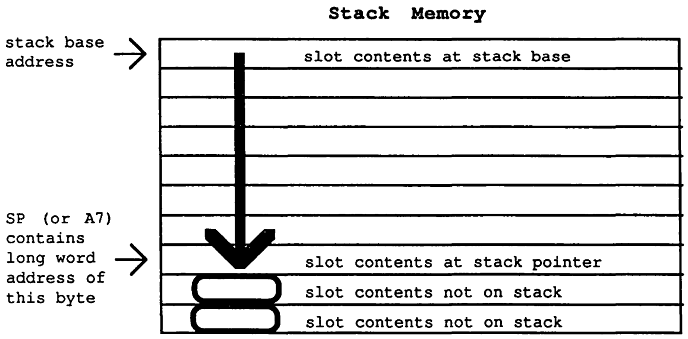

Stack Memory
stack base address
SP (or A7) contains long word address of this byte
slot contents at stack base
slot contents at stack pointer
slot contents not on stack
slot contents not on stack

Figure 16-2

On the Macintosh, the stack's base address boundary has a higher memory address than the movable boundary represented by the stack pointer. Like the drill of an oilwell, the stack grows downward in memory and shrinks upward toward the base. The stack pointer represents the tip of the drill.

This last note helps explain the statement SUBQ #2,SP preceding the _Button instruction. Because the _Button call needs stack space to store the result of its button press test, SUBQ #2,SP complies by extending the stack by 2 bytes. SUBQ (*subtract quick*) subtracts #2 from SP (the stack pointer) and stores the result in SP. See figure 16-3.

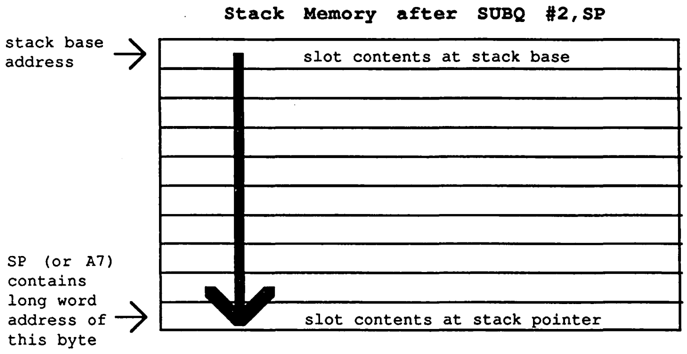

Stack Memory after SUBQ #2,SP
stack base address
SP (or A7) contains long word address of this byte
slot contents at stack base
slot contents at stack pointer

Figure 16-3

The effect is that the tip of the oilwell drill has sunk 2 bytes deeper into memory, extending the stack. The contents of those 2 bytes of memory are not affected. In other words, the tip of the drill has been sunk a distance of 2 bytes, but the dirt, rock, oil, water, IBM dress code, or other fossil contents of those 2 bytes of earth have yet to be investigated.

Figure 16-4, an illustration of all RAM memory, shows the stack contents growing downwards and the source code growing upwards.

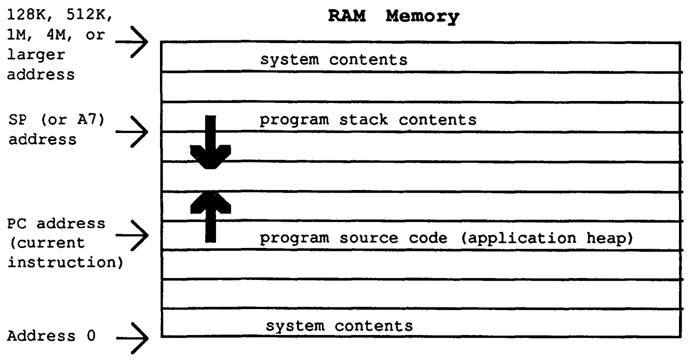

RAM Memory
128K, 512K, 1M, 4M, or larger address
system contents
SP (or A7) address
program stack contents
PC address (current instruction)
program source code (application heap)
Address 0
system contents

Figure 16-4

Status symbols for the weird

The _Button call examines the status of the mouse's button, and puts a word's worth of information on the stack for the programmer to retrieve and act upon. If the button is being pressed when the _Button call is executed, the word put on the stack is nonzero.

Otherwise, the word put on the stack equals zero, that is, all 16 bits of the word are set to:

```
#0000000000000000
```

The _Button instruction does not require all 16 bits of a word to communciate whether or not the button has been pressed. Actually, a single bit should suffice because both the button result and a bit's value are boolean in nature (true or false, 1 or 0). But the memory of the Macintosh is most easily accessed along word boundaries, that is, even byte addresses. So, though you will examine bits and bytes individually, you will pick and choose bits and bytes from the word and long word addresses in which they are contained.

The statement TST.B (SP)+ tests a byte-length value of its subject matter (in this case, the contents of the location pointed to by the stack pointer), then increments the stack pointer by 2 bytes. You should recognize the addressing mode of (SP)+ as postincrement indirect, the same mode that helped you increment the screen addresses.

The indirection indicated by the parentheses means you are testing not the address in the stack pointer, but the contents of the location addressed by the stack pointer. The plus sign outside the parentheses means you are incrementing the address in the stack pointer.

Remember, adding to the stack pointer is like lifting the oilwell drill out of the ground—the stack shrinks, and all memory contents deeper than the drill tip (SP) are lost.

You might think the plus sign would increment the stack pointer by only 1 byte because the instruction TST.B indicates a byte-length instruction. Good thinking. The assembler, however, is smart enough to know that addresses must be accessed on word boundaries (even, as opposed to odd, byte addresses), so an additional byte is automatically added.

At this point, you have lost the _Button results that were put on, then popped off, the stack. Of course, the TST.B instruction is worthless if its test results aren't stored somewhere. This storage place is called the status register.

The status register has the following special qualities:

- The register is only 16 bits (1 word) long. All other registers are 32 bits in length.
- The register is manipulated implicitly by 68000 and Toolbox instructions. That is, the status register cannot be the subject matter of an instruction.
- Nearly every bit of the register's 16 bits performs a special task. These tasks are discussed as they are needed and used by your programs.
- Whereas the bits of other registers are usually taken as a whole to represent a number or an address, the bits of the status register are individual boolean tests of the microprocessor's current status.

The statement TST.B (SP)+ pops the _Button results off the stack, but it records the boolean result in a single bit of the status register. This bit is called the Z, or zero, bit. If the byte popped off the stack equals 0, the Z bit is set to 1. If the byte popped off the stack equals anything else but 0, the value of the Z bit is 0.

TST.B tests only a single byte, yet 2 bytes were put on, and popped off, the stack. You might be wondering which of the 2 bytes is tested, and how to tell the bytes apart. The names given to the 2 bytes of a word are high-order byte and low-order byte. The high-order byte has the higher memory address. When representing part of a word length number, the high-order byte contains the higher value digits. Here are examples of the same value represented three ways:

full number	high order	low order
%1101011100101010	%11010111	%00101010
$D72A	$D7	$2A
55082 (decimal value)	55040	42

A long word can also be divided according to its high-order and low-order words. For example, for the hex long word $4B38D72A, the high-order word is $4B38 and the low order word is $D72A. The high-order byte of the high-order word is $4B. The low-order byte of the high-order word is $38, and so on.

On paper, numbers are written left to right, high order to low order. In the Macintosh's memory, numbers are stored bottom to top, high order below low order. Once again, if you picture the oilwell drill as the stack, you'll see that the drill tip—the stack pointer—points to the deeper byte address, which is the high-order byte.

The TST.B instruction uses as its subject matter the first byte pointed to by the stack pointer. As such, the high-order byte is tested for a zero value. This is just what you want, because the _Button call returns its boolean result in the high-order byte of the returned word.

If the button has not been pressed, the high-order byte is 0, and the TST.B instruction sets the Z bit of the status register to 1. Remember, the Z bit says, "If my value is 1, then yes, I reflect a zero value. If my value is 0, then no, I do not reflect a zero value." Only when the button has been pressed, and the high-order byte is nonzero, will TST.B leave the Z bit unset (0).

The last instruction of the TryButton loop, BEQ.S (*branch equal zero, short*), examines the Z bit to determine whether or not to branch to its subject matter. The branch to the reference marker TryButton occurs only if the Z flag is set (1). Otherwise, no branch occurs, and program execution drops out of the loop to the instruction _ExitToShell.

As you might expect, the loop repeats over and over until the button has been pressed and the following take place.

1. _Button returns a nonzero high-order byte of its word result.
2. The Z bit is unset (0). TST.B has tested the high-order byte and found it to be nonzero.
3. The branch to TryButton does not occur because the condition for BEQ.S to branch (Z equals 0) has not been met.

This concludes experiments with Fourplay and its offshoots. If talk of the stack and the status register has not sent you back to playing harmonica in the hills of Kentucky, you've got clear sailing ahead. But, come to think of it, Mr. Moss doesn't know how to play the harmonica and doesn't own a sailboat, so he might as well start work on his new novel.

Heck, maybe he'll just expand upon the stories in the upcoming chapters.

The publishing connection

The following is the gist of a June 14, 1987 telephone conversation between Mr. Moss and his editor, Ignatius Kafka (Hunter's brother), concerning the partially completed manuscript of *Fear and Loathing with 68000 Assembly Language.*

Kafka: Hello, Peat?

Moss: Yeah. Hi Iggy. Did you get my little package?

Kafka: Yes, I did. I've just been reading it over. There are a couple of things I'd like to talk about with you.

Moss: Uh huh.

Kafka: It's not exactly what I expected.

Moss: Uh huh.

Kafka: I was wondering: Where are you expecting to go with this? I was thinking about what kind of reader you were trying to reach.

Moss: Geez. You know. Someone who likes to read, I guess. Maybe someone who wants to learn something about programming.

Kafka: There seems to be two different paths here: one about these people and their stories, and another that's filled with technical information. I'm not sure it works together.

Moss: Me neither. Did you read all the chapters I sent?

Kafka: Yes, I looked them all over.

Moss: And you know about the way boys and girls met in Estonia around the turn of the century, and about the way the Great Plains Indians could tell if a couple would be compatible, and the strange behavior of the Los Alamos Sluggers.

Kafka: Don't get me wrong. I enjoyed the stories, but I'm unsure of how they help the technical programming aspect. They don't seem to add to the understanding of assembly's difficult concepts.

Moss: But you read the chapters, right?

Kafka: Yes, I read what you sent.

Moss: So you know something about hexadecimal numbers and the mailbox memory slots that hold them? And you saw the little program called Fourplay that pushed numbers into and out of the slots to make the computer screen a pattern of thin black lines?

Kafka: Yes. On their own, the programming sections seemed fine.

Moss: It sounds like you understand a lot. Do you really think that if every section of every chapter was about 68000 assembly programming, you would have been able to remember more than you do now?

Kafka: Well, maybe not me. But someone who had purchased the book for the purpose of learning assembly language might.

Moss: C'mon, do you think I'd take the time to write a whole book just for some geeks who want to learn assembly language? You already read that it causes brain damage. This way, you've got a book that lets you understand an aspect of head trauma without running into a wall.

Kafka: We've got to look at the market. Who's going to buy this?

Moss: Anyone who wants an offbeat science book. More people would read about science if it wasn't pretentiously bequeathed by all-knowing weenies. Here you get a chance to see how a computer works without being bored silly. Heck, I'm sure I'll be able to get everyone in the Berkeley Macintosh Users Group to buy a copy. That's four thousand books right there.

Kafka: Are you sure?

Moss: Sure, I'm sure.

CHAPTER

17

Macintosh Programmers Do It in Windows

Initialize the universe, make way on the stack

The code in WindowMaker, listing 17-1, draws a bona fide Macintosh window, then waits for a button press to end the program. See figure 17-1. The programs you write will use windows for nearly all onscreen activity. In building a window, you'll find that the Toolbox does the hard work, while your job is to take care of the stack.

Listing 17-1

```
                                ;Program WindowMaker

INCLUDE 'Traps.a'               ;define trap names

MAIN
PEA       -4(A5)                ;push pointer to Quickdraw globals
_InitGraf                       ;initialize Quickdraw
_InitFonts                      ;initialize font manager
_InitWindows                    ;initialize window manager
_InitCursor                     ;initialize cursor to arrow

SUBQ      #4,SP                 ;make room for pointer result
CLR.L     -(SP)                 ;allocate on heap
PEA       SizeWindow            ;push pointer to rectangle
PEA       NameWindow            ;push pointer to name
ST        -(SP)                 ;yes, window is visible
CLR.W     -(SP)                 ;use standard document window
MOVE.L    #-1,-(SP)             ;put window on top
SF        -(SP)                 ;no, window has no goAway box
CLR.L     -(SP)                 ;NIL window refCon
_NewWindow                      ;make the window
_SetPort                        ;make window current port
```

Listing 17-1 *cont.*

```
TryButton    SUBQ     #2,SP              ;make room for boolean result
             _Button                     ;see if button is pressed
             TST.B    (SP)+              ;set Z flag accordingly
             BEQ.S    TryButton          ;branch if Z is set (no press)

             _ExitToShell                ;return to Desktop/Shell

SizeWindow   DC.W     80,60,290,450      ;window bounds (Top,Lft,Bot,Rgt)
NameWindow   DC.B     'Empty Window'     ;window title

             END                         ;code end directive
```

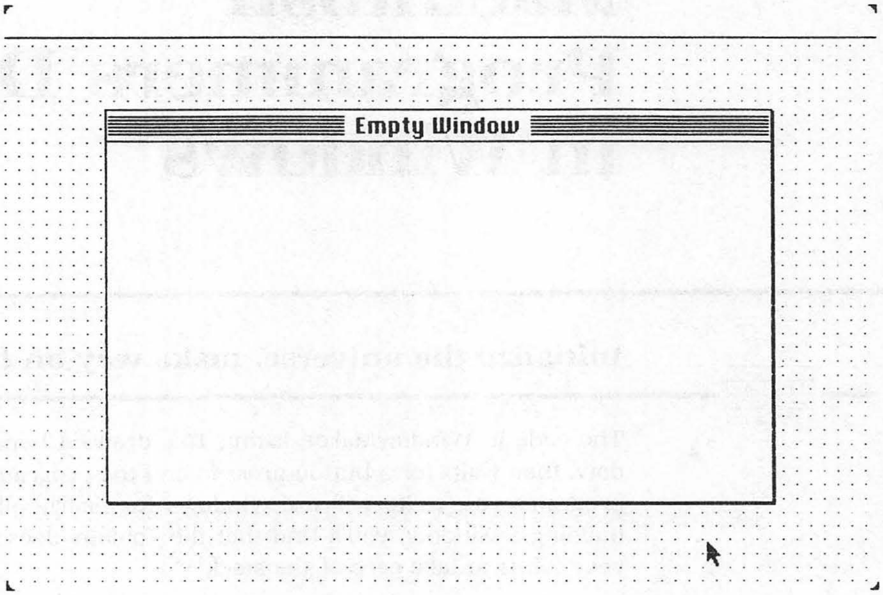


Figure 17-1

In the beginning (Before Documentation), Macintosh programmers saw a strange PEA -4,(A5) at the start of every program that escaped Cupertino. Few knew why, but the consensus was "better keep it there." Certain faiths believe that when God created the Earth, the skies, and their inhabitants, She first pushed the effective address 4 bytes before the A5 pointer, then initialized the Toolbox managers.

In a few chapters, you will know more about the special use of A5. Now, you just need to know about some of the Toolbox's _Init calls. In short, each _Init instruction runs a short program that prepares Quickdraw, Font, Window, and Cursor instructions for your use, unscrambling any prior values.

From previous chapters, you are familiar with the INCLUDE 'Traps.a' directive, which associates your Toolbox instruction names with addresses in the

Toolbox code. The name derives from the fact that Macintosh incorporates its Toolbox instructions into 68000 code through a trapping mechanism. The instructions that use the Toolbox are called trap instructions, or traps. Do you remember from the last three chapters that you made space on the stack before the _Button trap for its boolean result? The statement SUBQ #2,SP extended the stack for _Button by moving the stack pointer down 2 bytes.

Now you will do a similar preparation for _NewWindow, a Toolbox trap that creates a new window. But the preparation of the stack is much more extensive when creating a window than it is when testing the mouse button. The _NewWindow trap needs subject matter to provide eight characteristics that determine how the window looks and acts. In addition to characteristics, _NewWindow needs a result space on the stack to store a pointer to the window it will create.

A pointer is a long word address telling you where some other information is stored. You might think of a pointer as a reference marker that has no name.

Look at the third block of code, which begins with SUBQ #4,SP and ends with _SetPort. The eleven statements in this block create and draw a new window by:

1. Extending the stack by 4 bytes to make room for the long word result of _NewWindow (a pointer to the new window).
2. Extending the stack with all the subjects (eight characteristics, eight statements) that determine how the new window looks and acts.
3. Calling the _NewWindow trap, which automatically retrieves (and removes) the eight subjects from the stack, stores the data somewhere in memory, returns a long word pointer to the stack in the space provided by the first statement, and draws the window. The pointer represents an address where the new window's characteristics are stored.
4. Calling the _SetPort trap, which automatically retrieves (and removes) its single subject from the stack (the long word pointer left by _NewWindow), uses that address to find the window's characteristics, and sets the environment for all activity to be done inside the window.

Again, you can see that every Toolbox instruction has individual requirements. Some need stack space to return a result (for example, _Button and _NewWindow); some do not (for example, _SetPort and _ExitToShell). A complex call such as _NewWindow requires a result space and eight subjects to be pushed onto the stack to provide characteristics for its window. Furthermore, the subjects must be of a particular type, and must be pushed onto the stack in a particular order.

The dictionary in part 3 is invaluable when you want to refresh your memory about the requirements for a Toolbox trap used in this book. A listing of the required subject matter for all Toolbox traps can be found in Apple's reference volume *Inside Macintosh.*

Read the program comments beside each statement (comments always begin with a semicolon) to find out which window characteristic is being pushed onto the stack. Briefly, here are the subjects (in correct order) that _NewWindow requires, and the means by which the subjects are placed on the stack:

1. A space for the pointer result. Subtract 4 from the stack pointer for a long word result.
2. A pointer to where to store the window data. Clear a long word for a NIL pointer.
3. A rectangle to give the window's size and location. Push the effective address of a type rect constant.
4. A string to give the window's name. Push the effective address of a type string constant.
5. A byte to indicate if the window should be drawn or left invisible. Set the byte to the boolean value true so the window is visible.
6. An integer to indicate the definition ID for the window's type. Clear a word because ID #0 defines a standard document window.
7. A pointer to indicate the window's plane. Move the value # −1 to place the new window in front of all other windows.
8. A byte to indicate if the window should have a go-away box. Set the byte to the boolean value false because a go-away box is not drawn.
9. A long integer to give a reference value for use by the application. Clear a long word for a zero value.

That may seem like a lot of characteristics to draw a window on the screen. But consider how much work would be required to address screen memory directly—drawing and keeping track of window frames, go-away boxes, size boxes, window planes, and more. The single Toolbox trap _NewWindow performs many instructions on the basis of the eight characteristics you push onto the stack.

It is your responsibility to know the size, type, and sequence of subjects to push on the stack. Many subject types are predefined by the Toolbox, so you are asked to know not only assembly language, but also Macintosh software design. For the little extra work, you get a lot of extra power.

For example, you have to look up the window definition ID constant that corresponds to one of the six predefined window structures: standard document window, alert box, plain box, plain box with shadow, document window without size box, and rounded-corner window. But after you have a table of those predefined constants in front of you (see _NewWindow in part

3's dictionary), you can choose a window by selecting a single number (or using the system constant name).

After all the subjects have been pushed onto the stack, the statement _NewWindow does the rest of your work. The trap call draws the window, empties the stack of all subjects, and leaves a result pointer to the new window record in the space allocated by SUBQ #4,SP. The stack pointer, which always sits at the tip of the stack drill, points to the window record, the only remaining contents of the stack.

The statement _SetPort takes a long word subject from the stack (the window record pointer left by _NewWindow), and sets a grafPort based on that window record. This grafPort is a standard window environment in which the graphics system of the Toolbox (Quickdraw) operates. The _SetPort statement cleans the stack of its subject, leaving the stack empty (the same as before you started using the stack).

New instructions and pseudos

WindowMaker uses a handful of new instructions and one new addressing mode. You could flip to part 3 to find out what each new instruction does. But, because this is still an early chapter, another few paragraphs on putting subject matter into and out of memory and registers seems worthwhile. Here are the new instructions:

```
CLR.L
PEA
ST
SF
DC.W
DC.B
```

Take a look at the bottom instructions first. DC, short for *define constant,* is not a primary 68000 instruction, but a directive to the assembler. DC.W sets aside word-sized memory space for its subject—four integers that define a rectangle's coordinates. DC.B sets aside byte-sized memory space for the ten characters of its subject.

SizeWindow references the type rect constant, whose global coordinates (top, left, bottom, and right) are 80, 60, 290, and 450. NameWindow references the string constant, which contains the characters of the window's title, Empty Window, between the single quotation marks. The memory space occupied by these constants can be accessed through their names, SizeWindow and NameWindow. These names represent the effective address of the location of their contents.

As a result, when the statement PEA SizeWindow is executed, the effective address of the defined rectangle is pushed onto the stack. Don't be confused by the fact that SizeWindow is defined at the bottom of the program code, but is referenced near the top. The assembler is smart enough to seek out definitions, and install them in memory, before it attempts to execute primary instructions.

Likewise, when you need to put the string title subject onto the stack, you only need to push the effective address of NameWindow, the constant defined with the DC.B directive. As you might have guessed by now, PEA stands for *push effective address.*

The three remaining undiscussed instructions invoke no surprises. CLR.L -(SP) adds to the stack a long word whose bits are all 0. CLR is short for *clear.* ST -(SP) adds to the stack a byte whose bits are all 1. ST, short for *set true,* is one of many *set according to condition* instructions you will find in part 3 under Scc.

SF -(SP) adds on the stack a byte whose bits are all 0. SF, short for *set false,* is also one of the Scc instructions.

Both CLR and SF add a value whose bits are all 0. If you are wondering if there is more than one way to accomplish the same result, the answer is yes. But that is not to say that all ways are equally efficient. For example, the SF instruction takes less time to execute than CLR.B. But SF requires a byte-sized subject, so CLR.L works better for clearing a long word space.

Speed and space are constraints that wary programmers always remember when building code. Of course, programming artists tend to get a bit heady about their ability to keep code short and fast. If Moses had come down from Mt. Sinai with the Ten Commandments and handed them to a programming artist, the artist would say: "That's pretty good, but I could do it with fewer instructions."

You probably had no problems figuring out the addressing mode -(SP) that the instructions CLR.L, ST, and SF use to push values on the stack. This mode, called *indirect with decrement,* first subtracts a unit value from the stack pointer, then decrements not the stack pointer itself, but the contents of the memory location addressed by the stack pointer.

The indirection, indicated by the parentheses, means the register's contents give the effective location. The minus sign before the parentheses indicates the decrement takes place before the instruction's execution. And no doubt you remember that the stack acts like an oilwell drill. Thus, its stack pointer (drillbit) must be decremented in memory to extend the stack's size.

The block of code that waits for the button press is identical to the code in previous chapters. You can guess what would happen if it wasn't included. The window would be drawn, and immediately thereafter—before you could recognize the window—you would be returned to the Shell.

For the girls to see

In the Estonian towns of Stakentz, Glosen, and Werbbe, now a part of the Soviet Union, a custom was observed on the Sunday afternoons of May and September. On these afternoons, unmarried males sat in front of their homes. Or, if their homes were in the outskirts, they sat by the corner of the nearest thoroughfare connecting the towns. They sat on whatever was convenient—tree stump, box, blanket, or step—set their hands on their knees, and perched themselves like grounded owls.

Most of the boys and the few older men chose to sit alone. If there were two brothers from the same home, more than likely they would sit at a distance. Elders might have suggested that the boys use this time to deliberate on their goodness and productivity. Yet sitting was an act of individualism, and whatever suggestions a boy might hear would be reason enough to seek a different train of thought.

As a 14-year-old, Charles Moss remembered sitting in front of his home the year before he and his family emigrated to America. The year might have been 1904. By 1915, most people in these towns had fled the country rather than give their young men to the csar's army for lifetime conscription. The Sunday afternoon sit in May and September had no more participants.

For fifty-five years, until he died in 1980, Charles Moss hammered auto bodies in his repair shop in Schenectady, New York. Among his satisfied customers were Rod Serling, Kurt Vonnegut, and Robert Hoffenwald. Serling and Vonnegut became best known for their writing. Hoffenwald was the engineer at the General Electric company who devised the modern electronic tube, which would become the primary component in the first television and the first computer.

Just before Moss died, he could not remember knowing anyone named Serling or Vonnegut or Hoffenwald. He could not recognize his own grandchild, Peat, though a feeling slightly more powerful than memory made their togetherness comfortable. An old man whose memory span averages the length of a single sentence makes conversation challenging. Peat knew better than to ask his grandfather anything about the last quarter century. Instead, he asked his grandfather what he remembered about Russia.

"I sat on the step by our home. I was young to be sitting."

"You were sitting in Russia. Why were you young to be sitting?" Peat asked.

"For Sunday sitting. For the girls to see."

"You were sitting in Russia on Sunday for girls. What did the girls want to see?"

"Me." His grandfather laughed. "I was a boy. This is how girls chose."

With a battery of reiteration and questions, Peat pried a lucid story of the custom that disappeared along with the inhabitants of the three Estonian towns. In the way a peacock struts his colors, the solitary, meditative sitting displayed a boy's ability to reflect. While the boys showed themselves entrenched in thought, the girls walked freely, alone or together, able to tease, taunt, entice, or otherwise distract the boys without retort. Girls had the opportunity to see who was available among the three towns. By knowing where a boy lived, a girl could arrange with the town busybody for a "coincidental" meeting.

"You were sitting in Russia, Grandpa. For a Sunday sit. Then the girls came by. What did they say?"

"Why, the girls made fun of us. Tried to make us smile. We would barely give them a look. They'd say, 'Hey silly boy, whatcha thinking about so hard?' And we'd ignore them. Make believe we had important matters to consider. But we'd remember who talked to us. How their voices sounded. In a couple days, when we went for a haircut or to the store, guess who just happened to be there?"

Charles Moss gave a big smile. In a few moments he forgot why he was smiling. The nervousness came back. While he had talked, four other members of the family had stopped what they were doing to come near. Later, they told Peat it was the longest thought Grandpa had spoken in years. Now Grandpa saw himself at the center of attention and he didn't know why. He was embarrassed because he forgot who the young man was. With the uneasy smile of one who does not understand, he looked to his wife for help.

* * *

There is a little white lie in the previous story. Robert Hoffenwald was not the primary designer of the modern electronic tube. He was the parking attendant at the main lot of General Electric's Van Buren Street entrance. The primary designer of the modern electronic tube might have been Rudolph the Red-Nosed Reindeer, which is not to diminish the importance of Hoffenwald's contribution.

Without his expertise, the real designer might have lost so much time getting to and from his automobile that the electronic tube might have ended up as the primary component of the 12-speed blender.

The problem with white lies is not so much their false information as their detriment to credibility. After you know that Hoffenwald parked cars, who is to say where the deception will end? The machine does not yet exist that can read text and determine for certain when the pages are fibbing. Now, if there was a way to get words to sweat and palpitate, perhaps a polygraph could tell when the pages of a book were lying.

People sometimes lie. Books sometimes lie. The universe is swirling whims of perception. It shimmies between the ethereal heights of the mind and the proverbial pinch on the bottom. Within the pages of this book you are seeing how a young machine works. If you are looking for truth, try the late night preacher selling carpets on UHF. Because machines, like people and books, sometimes tell little white lies.

CHAPTER

18

Quickdraw Goes Inside the Window

WindowMaker recapsulated

The code of InsideStuff, listing 18-1, draws the same window as the program in the last chapter. But it also draws shapes and text within the window, as shown in figure 18-1. You will see how screen locations are plotted within a window using a local (as opposed to a global) coordinate map. The Toolbox's graphic traps (Quickdraw) do most of the work, requiring only that you push the appropriate subject matter onto the stack.

Listing 18-1

```
                         ;Program InsideStuff

INCLUDE 'Traps.a'        ;define trap names

MAIN
PEA      -4(A5)          ;push pointer to Quickdraw globals
_InitGraf                ;initialize Quickdraw
_InitFonts               ;initialize font manager
_InitWindows             ;initialize window manager
_InitCursor              ;initialize cursor to arrow

SUBQ     #4,SP           ;make room for pointer result
CLR.L    -(SP)           ;allocate on heap
PEA      WindowSize      ;push pointer to rectangle
PEA      WindowName      ;push pointer to name
ST       -(SP)           ;yes, window is visible
CLR.W    -(SP)           ;use document window w/o size box
MOVE.L   #-1,-(SP)       ;put window on top
SF       -(SP)           ;no, window has no goAway box
CLR.L    -(SP)           ;NIL window refCon
_NewWindow               ;make the window
_SetPort                 ;make window current port
```

Listing 18-1 *cont.*

```
                PEA      CoinSize              ;push pointer to rectangle
                _FrameRect                     ;draw rectangle frame

                MOVE.L   #$006E007A,-(SP)      ;specify integer coordinates
                _MoveTo                        ;place Quickdraw pen at point
                PEA      CoinLetters           ;push pointer to string
                _DrawString                    ;draw string at pen location

TryButton
                SUBQ     #2,SP                 ;make room for boolean result
                _Button                        ;see if button is pressed
                TST.B    (SP)+                 ;set Z flag accordingly
                BEQ.S    TryButton             ;branch if Z is set (no press)

                _ExitToShell                   ;return to Desktop/Shell

WindowSize      DC.W     80,60,290,450         ;window bounds (Top,Lft,Bot,Rgt)
WindowName      DC.B     'Inside Coin'         ;window title

CoinSize        DC.W     80,100,130,290        ;coin bounds (Top,Lft,Bot,Rgt)
CoinLetters     DC.B     'BlackHeads/WhiteTails' ;string in rectangle

                END                            ;code end directive
```

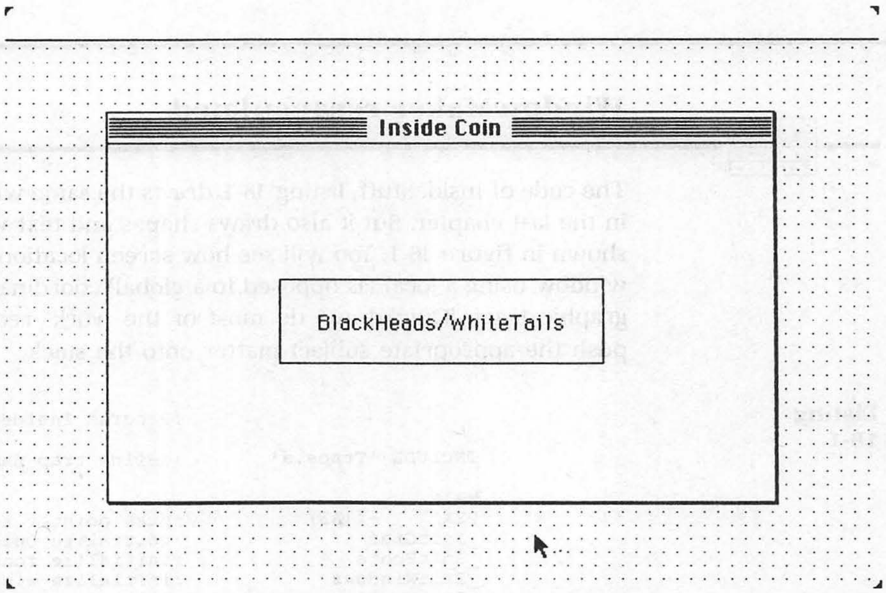


Figure 18-1

All the code of last chapter's WindowMaker is contained in InsideStuff. Once again, the INCLUDE and _Init commands start the ball rolling. The list of _Init commands grows larger as you take advantage of more of the System and Toolbox's predefined traps.

When you edit text, manipulate menus, keyboard events, and dialog boxes, and more, you add their respective _Init commands to the front of your code. InsideStuff uses the same Quickdraw, Font, Window, and Cursor managers. The new traps are all under the category of Quickdraw traps, the largest of all managers. The _NewWindow and _SetPort traps create the window and set up the current grafPort. The _NewWindow command received the most attention in the last chapter because it required so much subject matter. The _SetPort command simply took the result of _NewWindow (a pointer to the new window), and set up the environment for all activity in the window.

Of course, WindowMaker never performed any activity in the window. Maybe you are wondering why WindowMaker used a _SetPort trap if there was no activity. Or, more relevantly, what is a grafPort and why do you need to set one up?

In the last chapter, there was no specific need for a grafPort. However, the _SetPort trap performed an important task when it cleared the stack. _SetPort removed from the stack the long word pointer left by _NewWindow. Alternatively, you could have cleared or subtracted that long word address from the stack (and in less computing time than it took _SetPort). But the purpose of creating windows is to do things inside them, so it makes more sense to use _SetPort, and ready yourself for action.

A grafPort is a Macintosh construct that corresponds to a working window-like environment. (Pascal calls it a record; assembly uses a clump of consecutive memory slots.) Every window has a unique grafPort that defines the graphic activity within the window's boundaries. Whereas the window manager keeps window structures distinct and operational, the grafPort information of each window prescribes the graphic activity within.

You do not need to stipulate every characteristic of a grafPort to use one. The default settings maintain order until you choose to override them. Your _SetPort call augments only a fraction of the information that a grafPort holds. Specifically, the _SetPort trap establishes _NewWindow's grafPort (the pointer to the window was left on the stack by _NewWindow) as the current port.

The first instance of new code in InsideStuff is a two-statement block that draws a rectangle. The process should be familiar because creating a rectangle is a simplified case of creating a window. Unlike windows, which require numerous subjects including a size dimension, a rectangle requires only a size dimension. This single subject is pushed onto the stack with the statement PEA CoinSize.

In the same manner that the four integers of the constant SizeWindow (the name has been changed to WindowSize in this chapter) define a rectangle to serve as the window's boundaries, a second rectangle called CoinSize is defined. Toward the bottom of the program code you will find the assembler directive that places the reference CoinSize and its four integers in memory:

```
CoinSize        DC.W        80,100,130,290
```

Rectangles can be defined by four integers, as mapped on a coordinate plane, representing the rectangle's top, left, bottom, and right sides. Another way of looking at the same information allows a rectangle to be defined as two points, the opposing corners of the rectangle. In the case of CoinSize, the top-left point (80,100) and the bottom-right point (130,290) define the same rectangle as the four integers.

The rectangle is a primary Macintosh structure (called a type rect record from Pascal) used as a parameter in many Toolbox traps. Although assembly, unlike Pascal, does not require types to be defined, the parameter subjects of the traps (type rect and others) must be in the proper form (i.e., four integers to define a rectangle).

Because parameters must be presented exactly, a terminology using structured types makes it easier to understand programming in assembly. For example, when you read that the _FrameRect trap requires a type rect parameter, a direct correlation exists between the shape drawing routine and the shape subject. The correlation becomes fuzzier if you read that _FrameRect requires four integer subjects.

In the dictionary in part 3, you will find that the Toolbox and Operating System traps are presented in a Pascal (or structured) format. The parenthesized parameters that follow each trap name represent, in order, the subject matter that must be pushed onto the stack (or put into registers if so indicated).

Before you go to the next section, you might want to consider how the _FrameRect trap draws the outline of a rectangle in the current grafPort of the newly established window.

Look up _FrameRect in the dictionary in chapter 25, and you will see that _FrameRect requires a single type rect parameter. A type rect parameter can be put onto the stack by the PEA instruction (*push effective address*), whose subject matter defines a rectangle.

The _FrameRect trap takes its parameter off the stack (leaving the stack in the same condition as before the PEA CoinSize command), and draws the rectangle to the specified coordinates.

All graphic activity takes place in the current grafPort. The _SetPort trap establishes the new window's grafPort (the content region of the new window) as the current grafPort. Therefore, the rectangle is drawn onto the coordinate map of the window's grafPort (and not the global grafPort or Macintosh screen).

The upper-left corner of the current grafPort is set to the origin coordinates (0,0). You can see that the local coordinates of the current grafPort plot the rectangle in a different location than if the origin (0,0) was the upper-left corner of the Macintosh screen. (Before the _SetPort trap, the upper-left corner of the screen was the origin, and the window itself was mapped onto this global grafPort.)

The concept of a unique coordinate map for each grafPort is crucial to the understanding of graphics on the Macintosh. Not only does the origin of each coordinate map set the spatial alignment of all the port's contents, but the grafPort record maintains information on fill patterns, font displays, draw-over modes, visibility commands, and more, including as yet unimplemented color displays.

Move Bach to the strings, Guido

The next block of code, consisting of four statements, draws text within the framed rectangle. The first two statements move the Quickdraw pen to the position of the first character. The third and fourth statements insert a string at the current position of the pen. Both pairs of statements use a 68000 instruction to prepare the stack with subject parameters, then call Toolbox traps to finish the job. As always, the programming task involves determining the applicable Toolbox call and putting the proper subject parameters on the stack.

In this book, you will use Toolbox and Operating System trap calls constantly for the following reasons:

- The traps execute code that is in ROM (read-only memory) while your program code resides in RAM (random-access memory). Executing code in ROM is faster than executing code in RAM.
- You save RAM space by not duplicating code already in ROM.
- Best of all, the code is already written.

Later in your programming, you may sometimes choose to circumvent Toolbox and Operating System trap calls. (You have seen already a crude way of avoiding the Toolbox when you wrote directly to screen memory in Fourplay.) But the efficient programmer will want to know, if only in an overview, everything that ROM offers.

The _MoveTo trap moves the Quickdraw pen to a specified position on the coordinate plane of the current grafPort. The Quickdraw pen is another Macintosh construct that works as a programming tool to put inkdots on the screen. The coordinate plane of the current grafPort has not changed since the _SetPort command, so it's origin remains the upper-left corner of the window.

The subject parameters of _MoveTo are two integers that specify the coordinates of where to move the Quickdraw pen. The MOVE.L instruction, in one fell swoop, can move both of these integers onto the stack. The tricky part is figuring out how the immediate hex number #$006E007A translates to a point on the coordinate map.

Here are some observations. #$006E007A apparently represents two integers because _MoveTo requires two integer parameters, and this long word number is the only subject being pushed onto the stack. If you split the 8 digits in the middle, you get two 4-digit hex numbers, #$006E and #$007A (high-order and low-order words).

The decimal equivalent of #$006E is 110 (six 16s plus fourteen 1s). The decimal equivalent of #$007A is 122 (seven 16s plus ten 1s). The coordinate point must therefore be (110,122) or (122,110). But to know which is correct, you need to know how MOVE.L writes to the stack and how _MoveTo reads from the stack.

Like the oilwell drill, subjects are pushed down onto the stack, then pulled back up so that the last subject in is the first one out. When #$006E007A is pushed onto the stack, starting from the low-order right side, the high-order word (#$006E) is the first word pulled back (read) from the stack.

The dictionary in chapter 25 shows the trap's parameter list as:

```
MoveTo(h,v: INTEGER);
```

so you know that the first integer read is the horizontal coordinate, and the second is the vertical. Thus, #$006E (110) is the horizontal coordinate, and the _MoveTo trap moves the Quickdraw pen to the coordinate point (110,122).

You can try inserting a different long word subject into the MOVE.L instruction, and see where the text ends up. Trial and error is part of every programmer's repertoire. But when you get sick of repeatedly compiling and linking, you'll see that it pays to figure out beforehand what the heck you are doing.

Now that you have the Quickdraw pen placed at coordinate point (110,122), you can use the _DrawString trap to write text on the screen. Procedure DrawString (*stringName*) requires only that its single parameter be pushed onto the stack. In InsideStuff, the subject for the _DrawString trap is the reference name CoinLetters.

At the bottom of the program code, you can see that CoinLetters is defined as a constant by

```
CoinLetters   DC.B      'BlackHeads/WhiteTails'
```

You can look up the DC directive in the dictionary to learn more about its format. But like WindowName, the correct number of characters (corresponding to the number of bytes) is defined implicitly simply by giving the string, enclosed in single quotation marks, as the second subject.

The effective address of the reference name CoinLetters is pushed onto the stack by the statement

```
PEA         CoinLetters
```

The _DrawString trap uses this effective address to find the contents of its string parameter, draw the string on the screen, and empty the stack.

String parameters, like all parameters, have bounds within which they can be used. The dictionary will help you avoid using parameters outside acceptable limits. For instance, the _DrawString parameter is defined by the Macintosh construct Str255, which is a string type from 0 to 255 characters. Using a string outside that range would cause problems.

That's about it for this section. You have drawn an object and text inside a window—representing a large part of a computer's communication ability. Mr. Moss has plans to write an interactive novel on the Macintosh. Although hardly an original idea, no one yet has done it well. Imagine if your mouse could sense the sweat on your palm as you are reading a hot and heavy passage from a Jessica Lange/Sam Shepard diary, and went deeper and deeper into detail based on your excitement. You're right, it's a crummy idea, but the image is nice.

Science and compatibility: Walking to Long's

There is a way for a man and a woman to know early upon their meeting if they are destined to be compatible. The method is simple and reliable. The man goes to Missoula, Montana; the woman goes to Santa Fe, New Mexico. The man walks the highway shoulder taking 90 east, then 25 south. The woman, in Santa Fe, walks north on 25. If the couple meet in front of Long's drugstore in Caspar, Wyoming, without either having to wait more than fifteen minutes for the other, compatibility is assured.

The Great Plains Indians made this discovery long before the Long's drugstore was built in 1956 beside the meeting site. The first man to record the phenomenon was Thomas Graff Kafka, a traveling tonic salesman in exile from Denver for providing the sheriff's daughter with a cacti derivative that caused her to recite poetry to a barbershop pole for six hours. On an August night in 1834, Kafka set camp a few hundred yards from a small settlement of fur trappers and traders on the present-day site of

Caspar. He lay to sleep beneath a starry sky, his three-wheeled cart and bow-legged mule tethered to a big rock stuck in the prairie dirt.

Shortly before sunrise, Kafka awoke to thundering whoops. Instinctively, he bolted from his bedcloth, sweeping his personal tonic from beneath the woolen topcoat that served as his pillow. His eyes strained to focus in the predawn light. The act he witnessed would have been a strange sight to a sober person, and utterly baffled Kafka. A dark figure with flowing black hair, nearly naked, trekked towards him in a deliberate pace made even more transfixing by the figure's howls and gutteral moans.

An astute man, one geared for self-preservation, would have fled. Kafka waited until his sleepy eyes could see those of the Indian. The bellows and cries intensified. The figure kept his pace towards Kafka. At close range, Kafka saw that the man's eyes were aimed beyond him and his tethered possessions. The braying man continued to keep pace. There were only a few strides between them before the spellbound Kafka thought to look southward to see what might be arousing the frenzy.

She had the same long black hair. She wore a blanket woven in jagged lines of color. Her sides were bare. She walked steadily, silently. Kafka stepped back after realizing he was in their path of collision.

He crumbled his bedcloth in a ball and threw it onto the cart. His fingers twisted unsuccessfully with the rope knotted around the rock. The thought of witnessing ritual violence frightened him. He had no appetite for more of the macabre; his tonic-fed delusions provided all the horror he wished to endure.

As abruptly as the cries had begun, they stopped. The eerie quiet lured Kafka a step away from the unrelenting knot. The Indian man and woman took a final step and pressed into each other. They clung, then collapsed to the dirt, her blanket pulled up around her neck, his breechcloth thrown clear. Kafka, virtually above them, struggled again to free his cart and mule.

By trading two bottles of his tonic to the couple later in the day, Kafka would learn how a man and a woman can tell early upon their meeting if they are destined to be compatible. He recorded his knowledge in the Oregon Territory's *Journal of Medicinal Enhancements,* a scholarly publication whose name would change twice more before becoming *High Tekkie Times.*

The ritual fell into neglect along with the fall of the Great Plains Indians. The last-known walk, in the spring of 1953, ended less than ideally when a 77-year-old man, three miles short of Caspar, was bitten by a diamondback coiled on the heat-retentive asphalt of Highway 25. The man continued on, meeting his mate punctually at the site of Kafka's tonic cart, but expired moments after the ceremonial celebration, apparently from the bite.

CHAPTER

19

The Mouse Makes CorneredCoin Complete

Trapped again

All the code of last chapter's InsideStuff is contained in CorneredCoin, listing 19-1. CorneredCoin's additional code reads the position of the mouse's cursor, and initiates a response if the cursor is detected within the framed rectangle. As in all the previous programs, a click of the button ends the program and returns you to the Shell. The screen display of CorneredCoin is shown in figure 19-1.

Listing 19-1

```
                                ;Program CorneredCoin
        INCLUDE 'Traps.a'       ;define trap names

        MAIN
        PEA     -4(A5)          ;push pointer to Quickdraw globals
        _InitGraf               ;initialize Quickdraw
        _InitFonts              ;initialize font manager
        _InitWindows            ;initialize window manager
        _InitCursor             ;initialize cursor to arrow

        SUBQ    #4,SP           ;make room for pointer result
        CLR.L   -(SP)           ;allocate on heap
        PEA     WindowSize      ;push pointer to rectangle
        PEA     WindowName      ;push pointer to name
        ST      -(SP)           ;yes, window is visible
        CLR.W   -(SP)           ;use document window
        MOVE.L  #-1,-(SP)       ;put window on top
        SF      -(SP)           ;no, window has no goAway box
        CLR.L   -(SP)           ;NIL window refCon
        _NewWindow              ;make the window
        _SetPort                ;make window current port
```

Listing 19-1 *cont.*

```
            PEA     CoinSize            ;push pointer to rectangle
            _FrameRect                  ;draw rectangle frame

            MOVE.L  #$006E007A,-(SP)      ;specify integer coordinates
            _MoveTo                     ;place Quickdraw pen at point
            PEA     CoinLetters         ;push pointer to string
            _DrawString                 ;draw string at pen location

FlipCoin
            _SystemTask                 ;give screen time to resynch
            SUBQ    #4,SP               ;make room for point result
            MOVE.L  SP,-(SP)            ;push pointer to result space
            _GetMouse                   ;get cursor coordinate point
            MOVE.L  (SP)+,D3            ;store point in register

            SUBQ    #2,SP               ;make room for boolean result
            MOVE.L  D3,-(SP)            ;retrieve cursor point
            PEA     CoinSize            ;push pointer to rectangle
            _PtInRect                   ;see if point is in rectangle

            TST.B   (SP)+               ;set Z flag accordingly
            BEQ.S   TryButton           ;branch if Z is set (not in rect)

            PEA     CoinSize            ;push pointer to rectangle
            _InverRect                  ;invert rectangle

TryButton
            SUBQ    #2,SP               ;make room for boolean result
            _Button                     ;see if button is pressed
            TST.B   (SP)+               ;set Z flag accordingly
            BEQ.S   FlipCoin            ;branch if Z is set (no press)

            _ExitToShell                ;return to Desktop

WindowSize  DC.W    80,60,290,450       ;window bounds (Top,Lft,Bot,Rgt)
WindowName  DC.B    'Cornered Coin'     ;window title

CoinSize    DC.W    80,100,130,290      ;rectangle bounds
CoinLetters DC.B    'BlackHeads/WhiteTails' ;string in rectangle

            END                         ;code end directive
```

You should be familiar enough with the use of trap calls to figure out how the new code in this chapter works. The program's new capabilities are performed by four trap calls and a handful of 68000 instructions that prepare the stack with parameters. You can also see an example of programming technique in an extra statement that gives you increased program speed.

The new code begins immediately after you have drawn the rectangle with the words *BlackHeads/WhiteTails* inside. The _SystemTask command is used here as a system delay device that synchronizes screen drawing with code execution. Without this command, the coin inversion would not take place in step with the Macintosh's ability to refresh the screen. (Assembly is very fast.)

The Pascal and C programs in part 1 use the _SystemTask trap call for the same purpose. _SystemTask is also used to detect and handle desk accessory activity before reading an event, though desk accessories are not used in this small application. Because your objective is to determine if the mouse's cursor is pointing within the rectangle, your second trap gets the current coordinate position of the mouse. Later, you will want to compare the mouse's coordi-

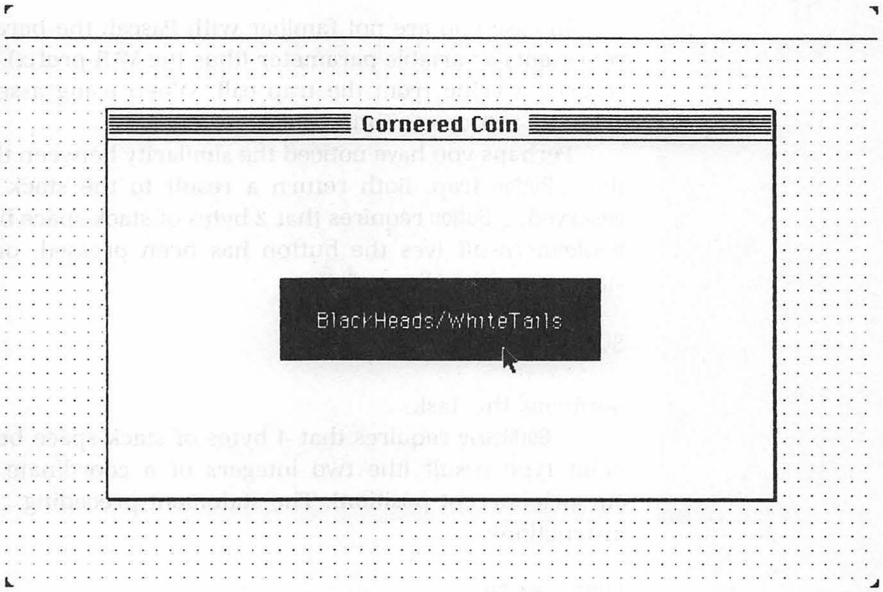


Figure 19-1

nates with the set of points contained in the rectangle, and respond to the boolean result (in rectangle or not in rectangle) of your inquiry.

The _GetMouse trap (in Pascal, the procedure is GetMouse(VAR MouseLoc: Point);) reads the coordinate position of the mouse. Like the _MoveTo trap, the parameter of _GetMouse specifies a point. But there is an important and obvious difference. _GetMouse requires that you prepare the stack to receive a point type, whereas _MoveTo requires that you prepare the stack by providing a point.

Another difference is that _GetMouse uses a point type (a Macintosh structure) to represent a point, and _MoveTo uses two integers to represent a point. Of course, a point type is composed of two integers. So in assembly (which doesn't distinguish type structures like Pascal) the treatment of the data ought to be the same. But because the parameter list of _GetMouse specifies a VAR parameter, you must prepare the stack to receive the structured type through a pointer. (More on pointers to follow.)

In Pascal, Toolbox routines are documented in the format of procedures, functions, and their parameters. In assembly, the same routines are accessed through trap calls, and the parameters are usually taken from the stack. The documentation, however, remains in the Pascal format, and certain Pascal conventions, such as receiving variable parameters through pointers, are followed.

In case you are not familiar with Pascal, the parameter name MouseLoc represents a variable parameter (thus the VAR prefix). A variable parameter returns a value from the trap call. When using assembly, where will the value be returned? To the stack.

Perhaps you have noticed the similarity between the _GetMouse trap and the _Button trap. Both return a result to the stack in a space you have reserved. _Button requires that 2 bytes of stack space be reserved to hold the boolean result (yes the button has been pressed, or no it has not). The statement preceding _Button:

```
SUBQ  #2,SP
```

performs this task.

_GetMouse requires that 4 bytes of stack space be reserved to hold the point type result (the two integers of a coordinate point that maps the cursor's current position). The statement preceding _GetMouse (and a MOVE instruction):

```
SUBQ  #4,SP
```

performs this task.

Now that you have made stack space for _GetMouse by subtracting #4 from the stack pointer, you probably figure you can call the trap and find the result sitting on the stack, just as you did with _Button. Tough luck—variable parameters are not so simple.

Variable parameters are referred to by pointers. A pointer is a long word address slot that directs you to another slot (where something of interest might be kept). You won't find a variable parameter directly, but you will find a pointer that tells you where the variable parameter is located.

You have used pointers many times to refer to Macintosh structures. All of the PEA instructions, which you used to push the effective address of a rectangle or a string constant onto the stack before a Toolbox call, used pointers. The effective address worked as a pointer to a structure. You pushed onto the stack a pointer to the rectangle or string constant.

With _GetMouse, you manipulate a pointer more explicitly. You not only make room on the stack for the coordinate point result, you also put on the stack a pointer that directs you to the result.

Crazy as it sounds, you are going to put on the stack a pointer that directs you to the immediately preceding contents of the stack. Even though you already know the location of the coordinate point result of _GetMouse, the trap's variable parameter requires that you address the result through a pointer. So after you make space on the stack for the long word result, you must also put a pointer (another long word) to the result onto the stack. Then, you are ready to call _GetMouse.

The logic behind pointers might not be clear at this time, but their value will become increasingly apparent. They offer fluidity to the inflexible mechanics of memory slots. Pointers work somewhat like a telephone book, providing a source of access to information of varied size and location. The statement:

```
MOVE.L   SP,-(SP)
```

moves a pointer onto the stack. The address of the pointer directs the variable returned by _GetMouse to the result slot you reserved. Make sure you understand the addressing modes involved before you go to the next section.

SP is the stack pointer register, a long word slot that contains the address of the tip (think oilwell) of the stack. Following the statement SUBQ #4,SP, the tip of the stack points to (contains the effective address of) the result slot.

MOVE.L SP,-(SP) moves onto the stack the current value of the stack pointer. As soon as this value is placed onto the stack, it is no longer current (because the tip moves down to accommodate the long word). But the effective address of the reserved result slot remains on the stack.

Remember that the subject SP, without parentheses, represents a long word address register. The subject -(SP) represents whatever contents the SP register happens to be pointing to after the register is decremented.

Following execution of MOVE.L SP,-(SP), the tip of the stack (the current SP) points to the old value of the stack (the effective address of the result slot). This is exactly what _GetMouse requires: a pointer to where the coordinate point of the cursor can be returned.

Following execution of _GetMouse, the stack contains only the coordinate point of the cursor. The pointer has served its purpose, and has been cleared.

The statement MOVE.L (SP)+,D3 moves the coordinate point result into data register D3, and clears the stack of the long word. (SP)+ indicates indirect postincrement addressing. (You used the same addressing mode to clear the stack of the _Button result.) After the long word contents of the stack are transferred, the stack pointer is incremented by the instruction size, which effectively pops off the stack's long word contents. The stack is once again empty.

Point in rectangle: A boolean delight

Now that you have the coordinates of the cursor stored in D3, you are ready to check if the coordinates are contained within the rectangle called CoinSize.

If the integers of the coordinate point are within the range of the respective horizontal and vertical coordinates of the rectangle's opposing corners, then the point lies within the rectangle. A series of compare and branch statements could perform this check.

The Toolbox, however, has a trap that does this work for you. You simply provide the point and the rectangle as subject parameters, and the trap returns a boolean result of true if the point is within the rectangle, or false if it is not. The three statements preceding _PtInRect prepare the stack accordingly.

From your experience using other trap calls, you should recognize how the stack is being prepared. SUBQ #2,SP reserves space on the stack for the boolean result (like you did for _Button). MOVE.L D3,-(SP) moves the coordinate point _GetMouse retrieved onto the stack. PEA CoinSize pushes the effective address of the rectangle onto the stack (like you did for _FrameRect).

The trap call _PtInRect empties the stack of the point and rectangle parameters, and leaves the boolean result in the 2-byte space you reserved. The parameter list:

_PtInRect*(pointName, rectName)* : boolean

shows the subjects required to produce the boolean result.

The boolean result is interpreted in the same way as the result of the _Button test. The statement TST.B (SP)+ tests the high-order byte (the significant byte for all boolean results), and sets the Z (zero) status flag accordingly.

Get ready. Understanding Z flags can be confusing. Study the following. If the Z flag is set to 1, then a zero result has been produced, indicating that the boolean result is false (the point is not within the rectangle). If the Z flag is cleared to 0, then a nonzero result has been produced, indicating that the boolean result is true (the point is within the rectangle).

The statement BEQ.S TryButton performs a conditional branch on the basis of how the Z flag is set. If Z is set (zero result, point not in rectangle), then a branch is executed and program flow continues at the statement following the reference TryButton. Otherwise, Z is clear (nonzero result, point in rectangle), no branch is executed, and program flow continues at the next statement, which prepares the stack for a trap that inverts the rectangle CoinSize.

Again, you should not be confused by the nefarious Z flag. The Z flag is set to 1 when there is a zero result. The Z flag is cleared to 0 when there is a nonzero result. The Z flag provides a true (1) response when its name (zero) is true.

You should be able to follow the two courses of action that depend on the result of _PtInRect. When the cursor point is detected within the rectangle, the two statements PEA CoinSize and _InverRect are performed. _InverRect works identically to _FrameRect, except the rectangle parameter is inverted rather than framed. When the cursor point is detected outside the rectangle, the branch to TryButton jumps over the inverting code, and the button test to end the program commences.

Notice that when the _Button test shows that the button has not been pressed, the branch directs program flow back up to statements in the _GetMouse block. The branch from BEQ.S FlipCoin to the reference name FlipCoin allows the intervening code to be executed repeatedly, and very fast. Thus, the current location of the cursor is constantly updated. Likewise, if the cursor is within the rectangle, the rectangle inverts many times each second.

This chapter is the last you will see of the rectangular coin flip. You no longer need to scrounge in your pants pockets for a nickel to make life's important decisions. Of course, if you enjoy reaching into your pants pockets, don't let technology put a crimp in your fun.

The Los Alamos Sluggers (part 1 of 5)

Los Alamos, New Mexico, April 1945

A man thin as a stick aimed his ray at the house on the hill. "I'm gonna light you," he vowed. "I'm gonna, gonna light you."

He barked instructions to his associate, a deaf man who read lips and palms. "Dim the lights, Eio. We're gonna need all the juice we can get." The thin man cackled and Eio turned his head. Eio hated the sight of his employer's cackle.

Tooey Kafka took an interest in science at an early age. At seven, he discovered that a salve of paste, pencil shavings, and chalk, applied to the rib of his teacher's chair, caused the teacher to itch the remainder of the day. Two years later, Tooey was sent to a special school where teachers were preoccupied with the violent. Tooey was left to his own devices.

At age ten he was given permission to go to the public library early afternoons. Well behaved, he amused himself among the dark, musty stacks. An arrangement was made with Miss Quincy, the librarian, to check on him at intervals. Each day she found him sitting crosslegged on the floor, a pile of four to six books beside him.

Months passed before Miss Quincy noticed Tooey was planting himself farther along the stacks, edging his way through the Dewey classifications. Tooey was fourteen and well into the 600s, science and technology, when an alert library page, pondering the boy's black tongue, discovered their bookworm was eating select pages from every book.

The library page chose not to squeal. Hardly anyone read nonfiction. Only a few pages were missing from each book and no one had complained. The page, a high school student, took it upon himself to approach young Kakfa and inform him that it would be best if he started playing baseball instead of coming to the library. Tooey Kafka shrugged and took the older boy's advice.

"This shall be among our finer moments," Kafka lectured to dead air while fidgeting with dials, meters, gauges, and switches. Eio had his back turned. "Yes, my faithful and debilitatedly shy associate, what we are about to do will reach far beyond our own clown alley lives. Today we are going to give of ourselves."

Kafka toyed with the flight control panel scavenged from a DeHavilland that had wrecked on a Nacimiento mountain slope. His other prizes littered the room: lacquered pine cones, a card-board solder spool, a ceramic filled with smoky oil. Perched on tenpennies, fruit crates, and sundered baseboard were a brass pipe fitting, a shadeless lamp, a cat's-eye marble, two baseball gloves, a buckled coffee can, and a brown bag stuffed with goat fur. Eio, a broad-shouldered, baked-faced, full-blooded Isleta, ab-sorbed the lore and grit of a white loon come to play Edison among the pueblos.

"Eio . . . Eio! . . . Eeeeio!" Eio could not hear him, could not hear anything. Finally, Kafka picked up the cat's eye, and hurled it across the drawn folding bed. "Confound it, man. What do you think I am paying you for?"

Eio turned, gave his employer a dirty look. He stayed put on the bed. For four months, for lack of other accommodations, he and Kafka had slept side by side. Kafka had once stacked newspa-pers to demarcate the laboratory from the living quarters, but the papers had toppled and spread, yellow and brittle across the floor. One room with one bed in an abandoned storefront—they lived in the remnant of a candle factory gutted of all but a waxy residue on ceiling, walls, and floor. Kafka's sole step at renovation was to tack gunnysacks over the single, streetside window. The dimness alone would have hidden Eio's annoyance, though the light hardly mattered. Tooey Kafka had bad eyes.

"Our work is coming to an end, Eio. Can't predict what might happen. Could set off a chain reaction. Blow us out. Just know we have to light her. Have to put everything into it. Give her our best. Light her hard and clean and honest. What do you say, man? Let's light this girl now. Let's light her to the moon."

Eio stared stone-still. He lived with a madman. Off-season for a carny mitt reader, Eio had been hungry when he answered the classified: NEEDED—SHARP EYES AND A STRONG ARM. He remembered how Kafka greeted him at the door, ushering him in, taking his explanatory card, squinting at the too small print, then chewing the card.

"When can you start?" Kafka had asked.

Hungry and excited, Eio accidently grunted.

"Excellent," said Kafka. "Hope you're right-handed." Eio nodded. Kafka went for the gloves. The two men went into the street and played catch.

CHAPTER 20 Momentous Events

Remembering the Great Equate

Program GetRect, listing 20-1, introduces a formal *event loop,* an element of nearly all Macintosh programs. In previous programs you used Toolbox commands to read the mouse button and coordinates. Here you use _GetNextEvent, a general-purpose input reading command whose capabilities include reading mouse events. Also, a new way of creating rectangles is explained. The screen produced by running the GetRect program is shown in figure 20-1.

Listing 20-1

```
                                       ;Program GetRect

            INCLUDE 'Traps.a'          ;define trap names

what        EQU     0                  ;event offset
mouseDown   EQU     1                  ;system constant
patXor      EQU     10                 ;Quickdraw constant

            MAIN
            PEA     -4(A5)             ;push pointer to Quickdraw globals
            _InitGraf                  ;initialize Quickdraw
            _InitFonts                 ;initialize font manager
            _InitWindows               ;initialize window manager
            _InitCursor                ;initialize cursor to arrow

            SUBQ    #4,SP              ;make room for pointer result
            CLR.L   -(SP)              ;allocate on heap
            PEA     WindowSize         ;push pointer to rectangle
            PEA     WindowName         ;push pointer to name
            ST      -(SP)              ;yes, window is visible
            CLR.W   -(SP)              ;use document window
            MOVE.L  #-1,-(SP)          ;put window on top
```

Listing 20-1 *cont.*

```
            SF       -(SP)            ;no, window has no goAway box
            CLR.L    -(SP)            ;NIL window refCon
            _NewWindow                ;make the window
            _SetPort                  ;make window current port

            MOVE.W   #patXor,-(SP)    ;push parameter constant
            _PenMode                  ;give pen inverting ink

DoDraw
            PEA      MousePt(A5)      ;push pointer to storage
            _GetMouse                 ;get cursor coordinate point

            LEA      NewRect(A5),A0   ;put pointer to rect space in A0
            MOVE.L   A0,-(SP)         ;push pointer (A0's contents)
            CLR.L    (A0)+            ;specify top-left point at pointer
            MOVE.L   MousePt(A5),(A0)    ;bot-right at pointer+4
            _FrameRect                ;draw rectangle

TryEvent
            _SystemTask               ;give system time, check desk acc.
            SUBQ     #2,SP            ;make room for boolean result
            MOVE.W   #$FFFF,-(SP)     ;event mask for all events
            PEA      EventRecord(A5)  ;push space for record result
            _GetNextEvent             ;ask for event
            TST.B    (SP)+            ;see if any event occurred
            BEQ.S    TryEvent         ;if Z is set (no event), do again

            MOVE.W   EventRecord+what(A5),D0 ;get event number into D0
            SUBQ     #mouseDown,D0    ;check mouse, toggle Z flag
            BNE.S    DoDraw           ;if Z is clear (no press), repeat

            _ExitToShell              ;return to Desktop/Shell

WindowSize  DC.W     80,60,290,450    ;window bounds (Top,Lft,B,R)
WindowName  DC.B     'Get Rect'       ;window title

MousePt     DS.L     1                ;space for point variable
NewRect     DS.L     2                ;space for rectangle variable
EventRecord DS.B     16               ;next event record

            END                       ;code end directive
```

You will find the following new terms in GetRect:

EQU	An assembler directive that equates a label with a value. This use of labels makes programs easier to read.
_PenMode	A Toolbox command requiring a parameter constant that gives the Quickdraw pen its draw-over quality.
_GetNextEvent	A Toolbox command that has many event recording abilities, but is used here just to read if the mouse button has been pressed.
DS	An assembler directive, short for *define storage,* that provides a specific amount of space for the programmer to create variables.

The three equate definitions provide constant values that are used by the Toolbox commands _GetNextEvent and _PenMode. The names of these constants (also called *equates*) are used instead of number values.

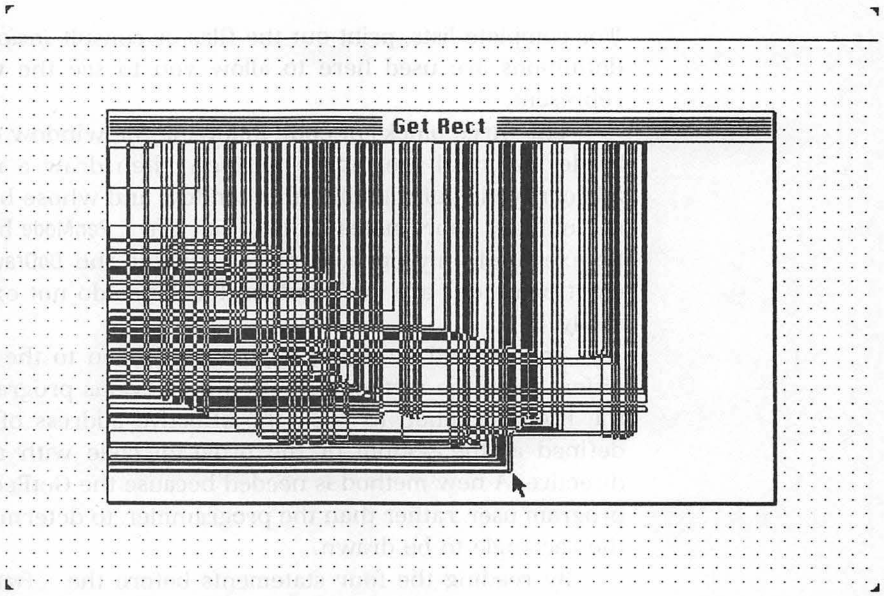


Figure 20-1

For example, the statement

```
MOVE.W #patXor,-(SP)
```

could be written as

```
MOVE.W #10,-(SP)
```

The use of the equate patXor, however, describes to the programmer which Quickdraw pen draw-over mode the number 10 represents. The constant value 10 is predefined to represent a pen ink that inverts the dot over which the pen draws.

You might remember from the Fourplay programs that the INCLUDE 'SysEqu.a' directive defined the system equate ScrnBase. A similar directive statement, INCLUDE 'Quick.a' could be used here in place of the EQU definition that equates patXor with the value 10.

The files 'Quick.a', 'SysEqu.a' and 'Traps.a' are simply long lists of EQU definitions. The assembler uses these lists to substitute values for whichever constants it finds in the program code.

Ordinarily, you will want to use INCLUDE directives to provide your definitions, assuming that the constant you want is actually part of the file.

(For complete lists, print out the files or consult *Inside Macintosh.*) The EQU definitions are used here to allow you to see the values such constants represent.

The three blocks of code following the window definition set the pen mode ink, read the mouse position, then draw a rectangle whose top-left corner is point (0,0) of the window and whose bottom-right corner is the mouse's coordinates. Notice that the _PenMode block, which needs to be executed only once, is placed outside the DoDraw drawing loop. Efficient programs are designed so that loops do not execute code unnecessarily.

The method by which a pointer is given to the _FrameRect command differs from the method you used in previous programs. Before, you used the PEA instruction to push the effective address of a constant that was defined at the bottom of the program code with a DC (*define constant*) directive. A new method is needed because the GetRect program allows the program user, rather than the programmer, to determine the coordinates of the rectangle to be drawn.

By reading the four statements before the _FrameRect command, you can see how the pointer is established so that _FrameRect can find its coordinate parameters. Remember, the parentheses around a register name indicate indirect addressing.

The NewRect variable's storage space, which is established by the DS directive at the bottom of the program code, has its effective address loaded into address register A0. This effective address serves as a pointer.

The pointer in A0 is pushed onto the stack by the MOVE.L instruction.

The CLR.L instruction puts zero values into the long word space that the pointer is pointing at. This fills half of NewRect's storage space with the top-left coordinate point. The plus sign increments the effective address by a long word.

The second MOVE.L instruction puts the mouse coordinate values into the long word space adjacent to the zero values. This fills the second half of NewRect's storage space with the bottom-left coordinate point.

The _FrameRect command uses the pointer in A0 to find the NewRect storage space and draw the rectangle with the given parameters.

Waiting for the big event

The block of code following the label TryEvent forms the core of the event loop. In previous programs you used the _Button command to determine whether or not to exit the program. Here you use the _GetNextEvent command to perform the same task.

The _GetNextEvent command, though more complex to implement than the _Button command, offers a more general method of reading user input. Upcoming programs take advantage of some of its additional capabilities. But for simplicity, only the button reading task is demonstrated in the GetRect program.

The importance of the event loop derives from the idea that a running computer program loops around and around, performing a set action until a specific event is initiated. This event might be a user action such as a mouse event, a keyboard event, or a disk insertion event. Other possible events include those initiated by the Macintosh system that relate to window management, device drivers, and networks. A null event occurs when there are no other events to report.

Like most Toolbox commands, the key to implementing _GetNextEvent is pushing the appropriate parameters on the stack. Included in these parameters are a boolean result space, an event mask to filter out unwanted events, and a pointer to a variable space (created with a DS directive like NewRect) where an information record about an event can be stored. In Pascal, _GetNextEvent is defined as

```
FUNCTION GetNextEvent (eventMask: INTEGER; VAR theEvent: EventRecord)
: BOOLEAN;
```

After the command is called, a boolean result is popped off the stack. A value of true (1) indicates an event has occurred; a value of false (0) indicates a null event. For all non-null events, further information about the event is obtained by evaluating values that _GetNextEvent stores in the event record.

The event block performs this task in the following manner:

1. The _SystemTask command gives the system time to synchronize screen drawing with code execution. It also gives the system a chance to check for and handle any desk accessory activity before reading an event. If synchronization is a problem or if a program implements desk accessories, _SystemTask should be called at frequent intervals as part of the main event loop.
2. The SUBQ instruction makes room on the stack for the boolean result.
3. The event mask is moved onto the stack. The value #$FFFF is a Toolbox constant indicating that all events are to be reported.
4. A pointer to the variable eventRecord is pushed onto the stack with the PEA instruction. (The DS directive at the bottom of the program code defines a storage space of 16 bytes, the necessary amount for the entire event record, even though you interpret only a part of this record.)

5. _GetNextEvent is called. This command leaves a boolean result on the stack and stores the event record in variable eventRecord.
6. The TST.B instruction sets the Z flag of the status register according to the boolean result left by _GetNextEvent. The subject's plus sign pops off the result, clearing the stack.
7. The BEQ.S instruction interprets the Z flag of the status register and, if Z is clear (null event), branches to label TryEvent to repeat the event loop.
8. The next block of code is performed only when an event is detected and the event loop is exited. A pointer to the record value indicating which event has occurred is moved into register D0. The value of the equate what is added to this pointer value so that only the appropriate bytes of the event record are read.

The mouseDown equate value is subtracted from D0. If mouseDown and D0 have the same value, the Z flag is set (zero result). The BNE.S instruction interprets the Z flag. On a nonzero result, the program branches back to the graphic blocks. On a zero result (mouse is down), execution drops to the last statement, _ExitToShell.

To use the event loop you need to understand three concepts. The first concept is branching based on the Z flag. The TST and SUBQ instructions set the Z flag. The BEQ and BNE instructions perform branches based on the status of the Z flag. Because the Z flag is a simple binary digit (0 or 1), you extract information about slots by performing tests or subtractions, then use the zero or nonzero result to branch with any of the Bcc instructions, such as BEQ and BNE.

The second concept is predefined Toolbox values. You need to know that the Toolbox has predefined certain structures and constants. For example, _GetNextEvent returns an event record in which the first 2 bytes of information indicate what event has occurred, the next 4 bytes provide a message about the event, and so on. Likewise, to understand the information inside the first 2 bytes of the event record, you need to know that a mouse down event is a constant represented by the value 1. (*Inside Macintosh* provides a reference on all Toolbox structures and constants, though often does not explain how to use them.)

The last concept you need to understand is using pointers to variables. The DS directives create space for variables using a method in which variables are moved around in memory, but are always located with the reserved A5 register. As a programmer, you rarely need to know A5's contents. But the Toolbox requires you to reference all of its variable structures greater than 4 bytes (and all VAR parameters) in this indirect manner. For this reason, slots containing effective addresses are used to point to variables, and are your primary tool for manipulating these variables.

The Los Alamos Sluggers (part 2 of 5)

Kafka started playing baseball the day he stopped going to the library. Baseball became important. His parents were pleased. They bought him a bat, a ball, and a fielder's mitt. Their son had outgrown his eccentricities. He was normal.

Kafka reentered the public school in the ninth grade without complication. His only poor subject was English. He complained about the readings. "None of this stuff really happened," he said. Kafka had been nurtured on nonfiction.

Afternoons he would go to the playground and throw a rubber ball against the tennis backboard. He scratched home plate onto the wood, took sixty heel-to-toe steps, and fired away. Every day, through sun and storm, he would throw until his mother summoned him to dinner, her high-pitched call crossing fields, winding about trees and ridges, piercing street noises and playground cries, arriving faint, unmistakable. Soon he threw hard and accurate, picking out corners, a sharp thump on the etched timber.

The only diversion able to still his arm was the Industrial League. He would stop to watch the men in their professional uniforms exercise all the right mannerisms: chewing and spitting and using the bat's small end to knock dirt from their spikes even when there was no dirt. For their histrionics they could have been the Yankees or the White Sox. Kafka was no less impressed that their jerseys read Mohawk Lumber, Liberty Radiator, and Mahoney Tool Works. He watched from behind the backstop or sometimes from the third baseline, ready to glove a foul ball.

Baseball gave Kafka his friends. He and the boys prattled about the major league teams they followed in the newspapers or the one game they saw on last summer's trip to the city. They wrangled about statistics, reminisced about bizarre plays, and voted on the greatest home run hitters, base stealers, and strike-out pitchers.

Kafka had a mind for figures, often becoming a mediator when facts were in question. His reputation spread beyond his friends. Older boys he hardly knew came to him to settle arguments, resolve stumpers. As a new boy in school he was given uncommonly sudden respect. At times Kafka felt so proud he thought he might burst.

Baseball also gave Kafka his peace. Prone to nightmares as a boy, now he studied the sports pages, the bubble gum cards, the year-end digests. He had a friend go to the library and take out all the books on baseball. Kafka explained that he couldn't go himself on account of some bad blood between him and Miss Quincy—which wasn't really true. Miss Quincy never knew half her books were damaged. Simply, Kafka was in awe of the boy who gave him baseball.

Then in one swoop Kafka became the owner of a baseball library. A neighbor—a friend of his father—called to him one Saturday, and said he had something in his garage that might interest him. Kafka hunched his shoulders and scuttled over. Piled in two tub-sized cartons were baseball almanacs, magazines, matted clippings—source references going back to another century that one day would find their way to Cooperstown. Kafka made seven wagon trips to transfer the booty. On shelves, in drawers, a closet full and more under the bed—he filled his room. Young Kafka would sleep nights restfully, until even baseball was not enough.

CHAPTER

21

Structured Programming with Blocks

Playing the fields

Program OvalTime, listing 21-1, expands the use of the event loop. The output of the program has changed very little from GetRect—ovals are drawn instead of rectangles. See figure 21-1. It has, however, been restructured with a more modular design.

Listing 21-1

```
                                        ;Program OvalTime

            INCLUDE 'Traps.a'           ;define trap names

what        EQU     0                   ;event offsets
message     EQU     2
when        EQU     6
point       EQU     10
modify      EQU     14

mouseDown   EQU     1                   ;system constants
patCopy     EQU     8

            MAIN
            PEA     -4(A5)              ;push pointer to Quickdraw globals
            _InitGraf                   ;initialize Quickdraw
            _InitFonts                  ;initialize font manager
            _InitWindows                ;initialize window manager
            _InitCursor                 ;initialize cursor to arrow

            SUBQ    #4,SP               ;make room for pointer result
            CLR.L   -(SP)               ;allocate on heap
            PEA     WindowSize          ;push pointer to rectangle
            PEA     WindowName          ;push pointer to name
            ST      -(SP)               ;yes, window is visible
            CLR.W   -(SP)               ;use document window
```

Listing 21-1 *cont.*

```
                MOVEQ    #-1,D0          ;put window on top, step 1
                MOVE.L   D0,-(SP)        ;put window on top, step 2
                ST       -(SP)           ;yes, window has goAway box
                CLR.L    -(SP)           ;NIL window refCon
                _NewWindow               ;make the window
                _SetPort                 ;make window current port

                MOVE.W   #patCopy,-(SP)  ;push parameter constant
                _PenMode                 ;give pen solid ink

TryEvent
                _SystemTask              ;give system time, check desk acc.
                SUBQ     #2,SP           ;make room for boolean result
                MOVE.W   #$FFFF,-(SP)    ;event mask for all events
                PEA      EventRecord(A5) ;push space for record result
                _GetNextEvent            ;ask for event
                TST.B    (SP)+           ;see if any event occurred
                BEQ.S    TryEvent        ;if Z is set (no event), do again

                BSR.S    HandleEvent     ;event occurred, go to subroutine

                TST.B    DoneFlag(A5)    ;toggle Z according to DoneFlag
                BEQ.S    TryEvent        ;if Z is set, do again

                _ExitToShell             ;return to Desktop/Shell

HandleEvent
                MOVE.W   EventRecord+what(A5),D0 ;get event number
                SUBQ     #mouseDown,D0   ;check mouse, toggle Z flag
                BNE.S    DoDraw          ;if Z is clear, do drawing

                MOVE.B   #mouseDown,DoneFlag(A5) ;else done, set DoneFlag
                RTS                      ;return with Z clear

DoDraw
                PEA      EventRecord+point(A5)   ;put global point on stack
                _GlobalToLocal           ;convert to local coord.

                LEA      NewOval(A5),A0  ;put pointer to rect space in A0
                MOVE.L   A0,-(SP)        ;push pointer (A0's contents)
                CLR.L    (A0)+           ;specify top-left point at pointer
                MOVE.L   EventRecord+point(A5),(A0) ;bot-right at pointer+4
                _FrameOval               ;draw oval

                CLR.B    DoneFlag(A5)    ;clear DoneFlag, not done
                RTS                      ;return with Z set


WindowSize      DC.W     80,60,290,450   ;window bounds (Top,Lft,B,R)
WindowName      DC.B     'Oval Time'     ;window title

NewOval         DS.L     2               ;space for rect variable
DoneFlag        DS.B     2               ;space for boolean
EventRecord     DS.B     16              ;next event record

                END                      ;code end directive
```

One prominent change is the addition of four equate statements. These equates fill out the remaining elements (called *fields*) of the event record. The GetRect program used only the what field. The additional equates in OvalTime provide name access to the other four fields of information returned by the _GetNextEvent command.

The OvalTime program uses only one of these additional fields. The equate named point specifies where the coordinates of the mouse are re-

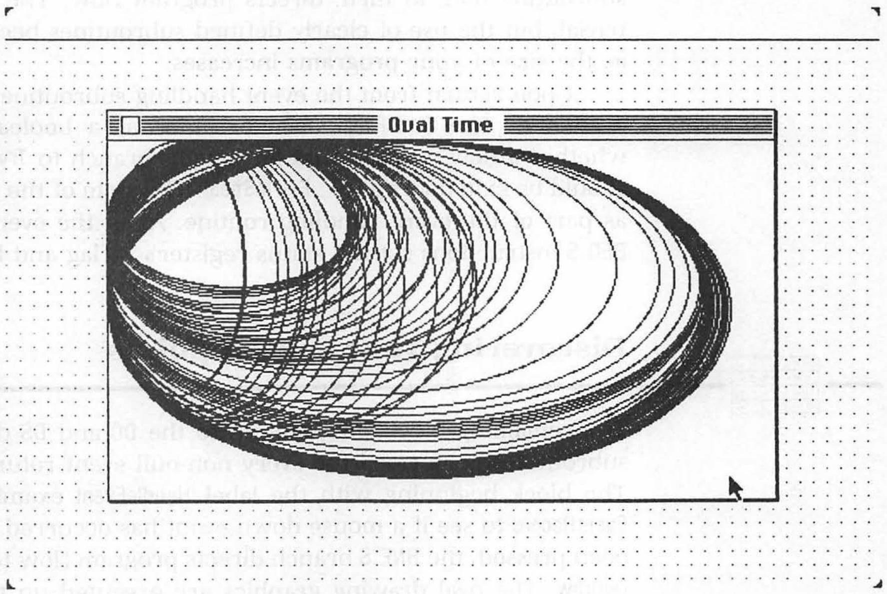


Figure 21-1

corded in the event record. The point field supplies the same information as the _GetMouse command in program GetRect. Thus _GetMouse is eliminated from the current program.

The field equate values represent an offset, measured in bytes, from the beginning address of the EventRecord variable. As a result, when access to the what field of the event record is needed, the correct pointer slot is EventRecord + what. Access to the point field is obtained by specifying EventRecord + point. The advantage of using offset names is that the programmer does not need to know any exact address. The necessary information is the number of bytes beyond the pointer address of the EventRecord variable.

The event record fields named message, when, and modify provide event information, not used in OvalTime, about windows, time, and modifier keys. Their equate statements are added to the code for the sake of completeness. The program would work the same if the four statements had been omitted.

The core of the event loop, which begins with the label TryEvent, is identical to the one used in GetRect. The enhancements offered by OvalTime, illustrated in the following block, are executed only when the _GetNextEvent command returns a non-null event.

In GetRect, a non-null event was handled by immediately evaluating the what field of the event record to direct program flow to the graphic blocks or the program exit. In OvalTime, the event is handled by branching to a subroutine that, in turn, directs program flow. The difference may seem trivial, but the use of clearly defined subroutines becomes more beneficial as the size of your programs increases.

Upon return from the event handling subroutine, a DoneFlag variable is evaluated. The DoneFlag variable serves as a boolean flag to determine whether a new event should be sought (branch to TryEvent) or the program should be exited (drop to _ExitToShell). The state of the DoneFlag variable is set as part of the event handling routine. As in the event loop, the TST.B and BEQ.S instructions set the status register's Z flag and branch accordingly.

Discovering your special points

The remaining blocks of code, up to the DC and DS directives, contain the subroutines performed for every non-null event returned by _GetNextEvent. The block beginning with the label HandleEvent examines the what field of EventRecord to see if a mouse down event has occurred. If the mouse has not been pressed, the BNE.S branch directs program flow to the blocks following DoDraw. The oval drawing graphics are executed up through the final RTS instruction (*return from subroutine*).

If the mouse has been pressed, the BNE.S branch to DoDraw does not occur. The MOVE.B instruction sets the DoneFlag with the value of #mouseDown (1), and the RTS instruction returns from the subroutine back into the event loop.

The graphic blocks following DoDraw differ from the blocks used to draw the rectangles in the GetRect program. The difference derives from the alternative method of reading the mouse's location coordinates. The _GetMouse command is replaced with the point field of the event record. This change must be accompanied by another Toolbox command called _GlobalToLocal to correctly interpret the mouse's location.

The _GlobalToLocal call converts a point's coordinates from global dimensions (think of the Macintosh screen as a global port) to local dimensions (think of your program's window as the local port). Because the point field of the event record returns coordinates expressed as global coordinates, the global-to-local conversion is necessary for using the mouse within the current window port. This conversion was not necessary when you used _GetMouse because _GetMouse returned coordinates expressed in the dimensions of the current window port.

The command _GlobalToLocal requires a point parameter. This effective address of the point is put on the stack with the PEA instruction. After the command is called, all references to the point field are in local dimensions.

The block of code that ends with _FrameOval uses nearly the same instructions as the code the GetRect program used to draw rectangles. The variable NewRect is renamed NewOval, and the coordinates of MousePt are now represented by EventRecord + point. _FrameOval works the same as _FrameRect, except the graphic output consists of an oval drawn just within the rectangular boundaries of the given parameters.

After the graphic work is performed, the DoneFlag variable is cleared, indicating another event is desired. When the RTS instruction returns program flow from the subroutine to the event loop, the zero value of DoneFlag causes the Z flag to be set (by TST.B) and the execution of the branch to TryEvent (by BEQ.S).

The modular design of OvalTime permits the addition of one or more subroutines to the bottom of the program code. The important event loop is placed at the top of the code. Whenever an event dictates that a certain type of action be performed, the event loop can instigate a call to a subroutine for event processing. When the subroutine is completed, program flow returns to the event loop. The DoneFlag variable conveys completion status information from the subroutine to the event loop.

This organization becomes important when the length of assembly code increases. Attempting to write code with a linear, drop-through approach such as that used in GetRect becomes too convoluted because, ultimately, most programs give the user the ability to direct program flow. By providing modules that are task oriented, a program can follow the whims of the user, always returning to the easy-to-find event loop when a new action might be initiated.

Finally, the OvalTime program uses slightly different code in the window creation block than its predecessors. A single line of code:

```
MOVE.L #-1,-(SP)
```

is replaced with the following two lines of code:

```
MOVEQ   #-1,D0
MOVE.L   D0,-(SP)
```

The change does not affect program operation—in both cases the parameter −1 is put on the stack so that the window is the topmost window. But these two statements together execute faster than the single statement they replace. Thus the change marginally improves program performance.

As a beginning programmer, you are not expected to delve into 68000 manuals to discover that a MOVEQ and a register-based MOVE.L combined use fewer processor cycles than a single, immediate MOVE.L instruction. But here is an example of code optimization that is available to experienced programmers.

The Los Alamos Sluggers (part 3 of 5)

"How long have we been working together, Eios con Dios?" The young Indian stared vacantly into his employer's crawling eyes. "It seems like forever, doesn't it?" said Kafka, bobbing in agreement with himself. "We've worked well together. We've accomplished as much as could be expected from two bumps on a log." Kafka chortled. "And that's exactly what we are, my main man. Aberrations on the ash tree. The Los Alamos Sluggers. Dust to dust, ashes to ash." Kafka's chortle turned into a cough.

"We're both young men, Eio. I'm barely thirty-five. And you, though in face could pass for my father, are younger still. Yet, young as we are, my jabbermouthed friend, we have certainly begun our sedimentary descent. My eyes dig deeper into consummate grayness, sparing me the sight of your wistful pout. And you, with stillborn audition, are blessed with an escape from the acid trills of my larynx . . . ahhhh."

Eio wrote on slate to communicate, but not this time. He put the chalk under his lip, let it hang like a cigarette. He pretended to blow a smoke ring at Kafka.

"Soon," Kafka continued, "our fine collaboration, mercifully more potent than the sum of its parts, will end. We'll head to the hinterlands. You, no doubt, to circus squalor, smitten as Indian Joe: chief bottle washer and cigar store mitt reader. For a buffalo fiver you'll write what shall come to pass next Tuesday. A man with eyes for the future and ears for decoration. The pitchman chides: Understand folks, Indian Joe can't talk. See, long ago, as a child, his tongue got burnt out when a white man force-fed him a roasting marshmallow that had fallen into the fire. Wouldn't have been so bad except the white man hooked out a coal instead of the marshmallow.

"And I shall exit this seared dustpan bequeathing a mushroom salute. Brief, swollen, and grows in the dark. To a place sugary and wet. Oh yes, we're gonna light her, Eio Meo Mio Mo. We gotta light her. She knows who we are. We gotta light her now. Come." Kafka extended his arms, then fell into a yawn. "Help me," he sang, as his eyes closed and opened and closed.

He slumped into the bayoneted relaxer, its matting puffed out of irreparable slashes. The room was dim in brown light, warm with heavy air. Siesta time in the small town, no cars passed, no voices carried. The desert drank slowly, invisibly. A twig of a mesquite bush dropped from Kafka's grasp—his ray.

CHAPTER

22

The Key to Boarding the Keyboard

Events of many flavors

Program LotsOfOvals, listing 22-1, expands the use of the event loop to include keyboard events. A press of any key exits the program. A press of the mouse button provides the starting point to draw an oval; letting up the mouse button provides the ending point. From the user's viewpoint, the program draws ovals similar to the MacPaint program. Figure 22-1 is the screen display produced by running the program.

Listing 22-1

```
                                        ;Program LotsOfOvals
             INCLUDE 'Traps.a'          ;define trap names

what         EQU     0                  ;event offsets
message      EQU     2
when         EQU     6
point        EQU     10
modify       EQU     14

mouseDown    EQU     1                  ;system constants
keyDown      EQU     3
patXor       EQU     10

             MAIN
             PEA     -4(A5)             ;push pointer to Quickdraw globals
             _InitGraf                  ;initialize Quickdraw
             _InitFonts                 ;initialize font manager
             _InitWindows               ;initialize window manager
             _InitCursor                ;initialize cursor to arrow
             MOVE.L  #$0000FFFF,D0      ;set up to flush events
             _FlushEvents               ;flush all events
```

Listing 22-1 *cont.*

```
                SUBQ     #4,SP               ;make room for pointer result
                CLR.L    -(SP)               ;allocate on heap
                PEA      WindowSize          ;push pointer to rectangle
                PEA      WindowName          ;push pointer to name
                ST       -(SP)               ;yes, window is visible
                CLR.W    -(SP)               ;use document window
                MOVEQ    #-1,D0              ;put window on top, step 1
                MOVE.L   D0,-(SP)            ;put window on top,step 2
                ST       -(SP)               ;yes, window has goAway box
                CLR.L    -(SP)               ;NIL window refCon
                _NewWindow                   ;make the window
                _SetPort                     ;make window current port

                MOVE.W   #patXor,-(SP)       ;push parameter constant
                _PenMode                     ;give pen inverting ink

TryEvent
                _SystemTask                  ;give system time, check desk acc.
                SUBQ     #2,SP               ;make room for boolean result
                MOVE.W   #$FFFF,-(SP)        ;event mask for all events
                PEA      eventRecord(A5)     ;push space for record result
                _GetNextEvent                ;ask for event
                TST.B    (SP)+               ;see if any event occurred
                BEQ.S    TryEvent            ;if Z is set (no event), try again

                BSR.S    HandleEvent         ;event occurred, go to subroutine

                TST.B    DoneFlag(A5)        ;toggle Z according to DoneFlag
                BEQ.S    TryEvent            ;if Z is set, do again

                _ExitToShell                 ;return to Desktop/Shell

HandleEvent
                MOVE.W   EventRecord+what(A5),D0 ;get event number
                CMP.W    #keyDown,D0         ;check keyboard, toggle Z
                BEQ.S    YesDone             ;if Z is set, begin exit
                SUBQ     #mouseDown,D0       ;check mouse, toggle Z
                BEQ.S    DoDraw              ;if Z is set, read mouse
NextEvent
                CLR.B    DoneFlag(A5)        ;clear DoneFlag, not done
                RTS                          ;return with Z set

YesDone
                MOVE.B   #keyDown,DoneFlag(A5)   ;set DoneFlag, done
                RTS                          ;return with Z clear

DoDraw
                PEA      EventRecord+point(A5)   ;put global point on stack
                _GlobalToLocal               ;convert to local coord.
                MOVE.L   EventRecord+point(A5),D5    ;store start point in register
                MOVE.L   D5,D4               ;initialize temp register (D4)

MouseCheck
                SUBQ     #2,SP               ;make space for result
                _StillDown                   ;check if button is down
                TST.B    (SP)+               ;toggle Z flag
                BEQ.S    NextEvent           ;up, so get next event

                SUBQ     #4,SP               ;make room for point result
                MOVE.L   SP,-(SP)            ;push pointer to result space
                _GetMouse                    ;get cursor coordinate point
                MOVE.L   (SP)+,D3            ;store end point in register

                CMP      D4,D3               ;compare temp and new end points
                BEQ.S    MouseCheck          ;if same, get new end point
```

Listing 22-1 *cont.*

```
              LEA      NewOval(A5),A2  ;put pointer to rect space in A2
              MOVE.L   D5,(A2)         ;specify top-left point at pointer
              MOVE.L   D4,4(A2)        ;specify bot-right at pointer+4
              MOVE.L   A2,-(SP)        ;push pointer  (A2's contents)
              _FrameOval               ;erase previous oval
              MOVE.L   D3,4(A2)        ;specify new bot-right at pointer+4
              MOVE.L   A2,-(SP)        ;push again for second _FrameOval
              _FrameOval               ;draw oval

              MOVE.L   D3,D4           ;update old end point
              BRA.S    MouseCheck      ;return for new end point

WindowSize    DC.W     80,60,290,450   ;window bounds (Top,Lft,B,R)
WindowName    DC.B     'Lots Of Ovals' ;window title

NewOval       DS.L     2               ;space for rect variable
DoneFlag      DS.B     2               ;space for boolean
EventRecord   DS.B     16              ;next event record

              END                      ;code end directive
```

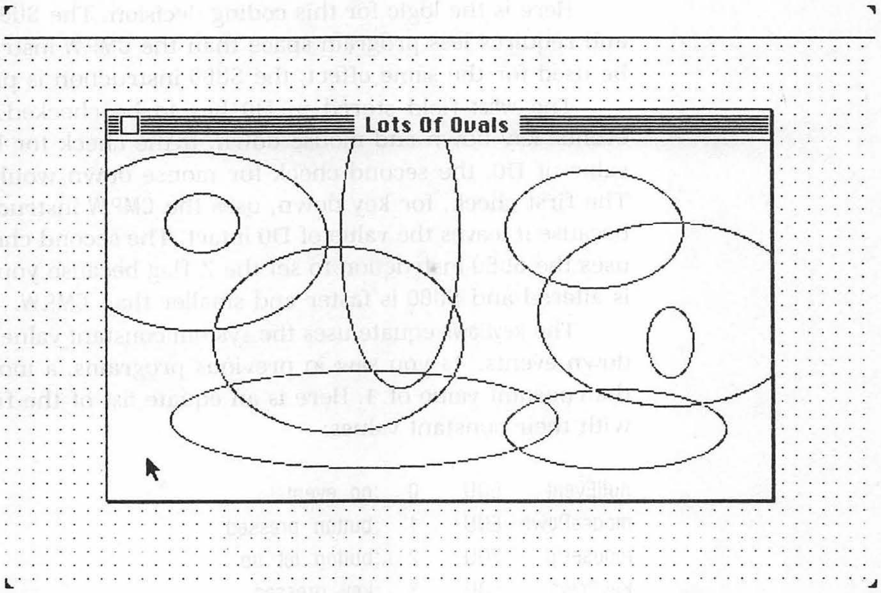


Figure 22-1

The first new Toolbox command resides in the initialization block. The call to _FlushEvents requires a single long word parameter to be placed in register D0 prior to the call. The purpose of _FlushEvents is to clear prior events the system might be retaining from before the start of the program. The parameter to _FlushEvents can be used to specify that only particular events be flushed from the system. However, the #$0000FFFF parameter is a mask that removes all events.

The window building block and the event loop block are the same as the blocks in OvalTime. The event loop in particular is a general-purpose block of code that searches for user events of any kind, then branches to the HandleEvent subroutine when an event occurs. Once again, the DoneFlag variable provides the boolean condition that indicates when to exit the loop and perform _ExitToShell.

The block of code beginning with the label HandleEvent is significantly enhanced. After the what field of the event record is moved into D0, the value of D0 is checked for either of two events. In prior programs, D0 was checked only to see if a mouse down event occurred. Now you are also checking to see if a key down event has occurred.

A new instruction is used to see if the value in D0 equals the mouseDown equate. CMP.W compares the values of its two subjects by subtracting the first from the second. However, unlike the SUBQ instruction, the value of the subtraction is not stored in the second subject. Both instructions affect the status register, and the Z flag in particular, in the same way.

Here is the logic for this coding decision. The SUBQ instruction is faster and requires less program space than the CMP.W instruction, so if both can be used for the same effect, the SUBQ instruction is preferable.

The what field stored in D0 has to be checked against two possible events: key down and mouse down. If the check for key down altered the value of D0, the second check for mouse down would no longer be valid. The first check, for key down, uses the CMP.W instruction to set the Z flag because it leaves the value of D0 intact. The second check, for mouse down, uses the SUBQ instruction to set the Z flag because you no longer care if D0 is altered and SUBQ is faster and smaller than CMP.W.

The keyDown equate uses the system constant value of 3 to represent key down events. As you saw in previous programs, a mouse down event uses the constant value of 1. Here is an equate list of the fifteen possible events with their constant values:

```
nullEvent    EQU    0  ;no event
mouseDown    EQU    1  ;button pressed
mouseUp      EQU    2  ;button let up
keyDown      EQU    3  ;key pressed
keyUp        EQU    4  ;key let up
autoKey      EQU    5  ;repeating key press
updateEvt    EQU    6  ;window needs updating
diskEvt      EQU    7  ;disk insertion
activateEvt  EQU    8  ;window made active or deactive
networkEvt   EQU   10  ;system network event
driverEvt    EQU   11  ;system I/O driver event
app1Evt      EQU   12  ;open for application use
app2Evt      EQU   13  ;open for application use
```

```
app3Evt     EQU  14  ;open for application use
app4Evt     EQU  15  ;open for application use
```

When the HandleEvent block finds a key down or mouse down event, the respective BEQ.S statement is executed, and program flow branches for further event processing. In the case of a key down event, the DoneFlag variable is given a nonzero value, then the subroutine is exited.

If you want to detect which key has been pressed, add a block of code to interpret the message field of the event record. The low-order word of the message field contains keyboard event information. A key code is represented in the high-order byte; a character code is represented in the low-order byte. For most purposes, the ASCII character code of the low-order byte is sufficient for identifying which key has been pressed or let up. (Key codes might be useful when the standard alphanumerics of the keyboard are altered.)

Fancier graphics through registers

In LotsOfOvals, the mouse down event initiates graphic activity. The following steps, beginning at the label DoDraw, are undertaken:

1. The point field of the event record is converted from global to local coordinates.
2. These local coordinates are stored in both register D5 and register D4.
3. The MouseCheck block uses the Toolbox's _StillDown command. _StillDown works like _Button except a true value is returned only if the mouse button has not been let up since the last mouse down event. When the boolean result of _StillDown tests true (nonzero result), the Z flag is clear, and program execution drops through to the oval drawing blocks. Otherwise, the button must have been let up (zero result), so the branch to NextEvent occurs.
4. The oval drawing blocks begin by reading the mouse position again, this time using _GetMouse. The new mouse position indicates whether the mouse has been moved and an oval should be drawn using the new position as the oval's end point. If the second mouse position stored in D3 compares identically (the CMP instruction sets the Z flag) with the first position in D4, a branch occurs to MouseCheck to get a new end point. If the second mouse position in D3 is different than the first position in D4, the new mouse position indicates that the drawing routine should commence.

Setting up the _FrameOval parameters differs from previous programs because _FrameOval is called twice—once to erase any previous oval, then a second time to draw the new oval. Erasing the preceding oval ensures that, when the mouse button is let up, only a single oval remains on the screen.

To call _FrameOval twice, the pointer to the NewOval rectangle space is moved onto the stack twice. This accounts for the command MOVE.L A2,-(SP) before each _FrameOval.

Both ovals use the contents of D5 for their top-left point. The command MOVE.L D5,(A2) puts the contents of D5 at the location pointed at by the contents of A2.

The original oval uses D4 for its bottom-right point, and the new oval (whose end point was taken from _GetMouse) uses D3. Thus two MOVE.L commands are used: MOVE.L D4,4(A2) for the first _FrameOval and MOVE.L D3,4(A2) for the second. Notice that they both move their data register subject to the location pointed at by 4(A2).

You have already seen this addressing mode, called *address register indirect with displacement,* though in a slightly different fashion. A constant precedes the address register to increment the pointer's effective address. For example, when you push the effective address on the stack with PEA EventRecord + point(A5), you specify an address that is a certain number of bytes beyond the address pointed at by the contents of register A5. Here you are explicitly stating that the effective address is 4 bytes beyond the location pointed at by A2.

This makes sense because the top-left point, which is expressed in the 4 bytes of a long word, is specified exactly by the address in A2. Thus the bottom-right point, which also requires 4 bytes, should be placed 4 bytes beyond the address in A2. As a result, the integers representing the two points are consecutive and do not overlap.

By calling _FrameOval twice, you can erase the old oval, then redraw a new one. Remember, because you are using the dot-inverting ink of pen mode patXor, an oval drawn over an existing oval has the effect of erasing the original oval.

If you need to be reminded of the contents of D3 and D4, look back to the MOVE.L instructions that assign their values. D4 is given the same value of D5, the oval's starting point. The means the first execution of _FrameOval neither draws nor erases an oval. The call to the second _FrameOval, using the new D3 end point, draws an oval as long as the new end point is to the bottom right of the starting point. You have ascertained already (using CMP) that D3 differs from D4.

Any confusion you might have should be cleared by examining subsequent repetitions of the loop beginning at the label MouseCheck. The final two statements in the graphics routine provide the change that makes the subsequent repetitions act differently than the first pass through the loop.

The statement MOVE.L D3,D4 updates D4 so that, on the next pass through the loop, the first _FrameOval erases the original oval, then the second _FrameOval draws a new oval using _GetMouse's new end point. The automatic branch to MouseCheck (BRA.S is not conditional upon any Z flag value) checks the button and mouse location before program flow can again reach the graphic drawing block of code.

The Los Alamos Sluggers (part 4 of 5)

Kafka awoke startled and frightened. Eio had watched him sleep and wake many times, and always the crazy man came to violently. A jutting arm or leg, spasms, yelps, a clenched face, Eio saw Kafka rise from unspeakable terror. The man was a battleground. He knew why Kafka slept in three-hour stretches. Sleep's turbulence was too much in one dose.

Kafka was vulnerable. After a virulent awakening, an hour might pass before he could regain composure. He would mumble and dicker, unsure where he left off. Snagged, muddled, he'd exhale loudly, shake his head, and amble about in search of clues, something familiar from a time when there were holds, places to be held. He'd see Eio and jump, surprised by the tall, dark Indian. He would look away, mutter apologies, and move to brush a speck of dust, awkward in his newborn sentience. Eio had tried to talk to him, comfort him, but his garbled speech further confused Kafka. Eio saw in Kafka the disorientation of a deaf man with sudden hearing.

Eio took to writing notes. In big letters on the slate, while Kafka was still soft and incandescent, Eio explored his employer's fragile underside. He wrote: WHAT ARE WE TRYING TO DO? Kafka answered, "Retire the side, make 'em pitch to us." He wrote: WHAT DOES THAT HAVE TO DO WITH THE WOMAN? Kafka answered, "She can catch, we need her." And Eio wrote: WHY ARE WE TRYING TO LIGHT HER? Kafka winced and trembled. The trance weakened. He began to recover from his nap. He pushed the slate away. ANSWER ME. Eio thrust the slate close to Kafka's eyes. "Because . . . " Kafka stammered, "I don't know . . . because I want to . . . I've always wanted . . . to have . . . a real catcher."

Kafka paced the room, scrutinized the paraphernalia as if for the first time. He sighted the mesquite twig, and scooped it from the floor. He went to the window, lifted a corner of burlap, and peered outside to the house on the hill.

Eio wiped the slate, wrote another message, and left the board face up on the bed. Then he collected the few articles of clothing he had come with, put them in a paper sack, and checked his pants pocket for pad and pencil. Kafka turned from the window and Eio saw that his employer had returned to his raucous, exalted self. It was time to go.

A woman lived in the house on the hill. She wore black dresses and black shoes and sometimes a black hat. She walked to the grocery store and would pass, on the far side of the street, the old candle factory. In the cafe one day, Eio saw the owner talking, saw the words *widow* and *black dress* and *house on the hill.* He saw *husband of six weeks* and *pilot* and *shot down by the Japs.*

Eio pointed toward the bed. Kafka looked over, read the slate: MUST CONSIDER MORAL IMPLICATIONS OF RAY PROJECT—CANNOT PROCEED WITH FULL HEART. Kafka shrugged. He picked up the baseball gloves, held one out to Eio. They found the softball in the corner and went outside.

Kafka tried out for the baseball team in eleventh grade. The coach told him he was too small to play. Six years later Kafka pitched a four-hit shutout for the Philadelphia Athletics in the sixth and final game of the 1930 World Series. After the game, he told reporters what his high school coach had said. The next day reporters went to Kafka's high school and asked the coach if Kafka's story was true. "I made a mistake," the coach said.

Kafka's professional career was brief and spectacular. He spent half a year in the minors, then four years with the Athletics. His lifetime record was 64 wins and 19 losses, plus four World Series victories without a defeat. His earned run average was 1.84 with 690 strikeouts against 143 bases on balls.

He is credited with inventing a pitch—the pit ball—that has never been effectively imitated. The pit ball, coined from Kafka's playing nickname, is thrown sidearm with three fingers on the ball and a 30-degree, off-center underspin. The ball rises hard, sometimes breaking, then dives over the plate. Kafka—mixing pit ball, palm ball, slider, and fastball—was considered a master of the change-of-speed delivery.

At age twenty-four, Kafka was hit in the back of the neck by a pitched ball in his first bat appearance of what would have been his fifth major league season. The blow ended his baseball career, cracking vertebrae and, it was suspected, causing degenerative cortical injury.

In December 1944, Kafka was elected to baseball's Hall of Fame. He was not present for the inauguration. He could not be reached. No one who knew he was "Pit" Tooey Kafka had seen him in years. Kafka himself was unaware of the honor awarded him.

CHAPTER

23

Menus: The Literature of Giants

Garnering resources

Program LitOfGiants, listing 23-1, adds a major dimension to Macintosh programming: resources. You'll write resource code (see figure 23-1) and compile the resource with the MPW application Rez so that it integrates with your main source code. The process adds a menu bar with a Quit option to your graphics program. The sceen display produced by running the program is shown in figure 23-2.

Listing 23-1

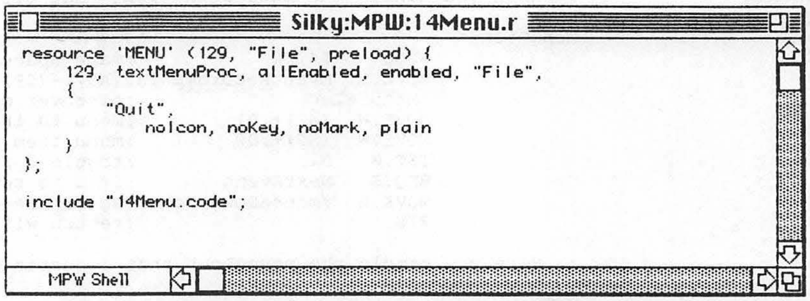
```
                                        ;Program LitOfGiants

            INCLUDE 'Traps.a'           ;define trap names

what        EQU     0                   ;event offsets
point       EQU     10

mouseDown   EQU     1                   ;system/toolbox constants
keyDown     EQU     3
patXor      EQU     10

inMenuBar   EQU     1
inContent   EQU     3

fileMenu    EQU     129

;  Here are the initializations and the window definition.

            MAIN
            PEA     -4(A5)              ;push pointer to Quickdraw globals
            _InitGraf                   ;initialize Quickdraw
            _InitFonts                  ;initialize font manager
            _InitCursor                 ;initialize cursor to arrow
```

Listing 23-1 *cont.*

```
            _InitWindows                ;initialize window manager
            _InitMenus                  ;initialize menu manager
            MOVE.L  #$0000FFFF,D0       ;set up to flush events
            _FlushEvents                ;flush all events

            SUBQ    #4,SP               ;make room for pointer result
            CLR.L   -(SP)               ;allocate on heap
            PEA     WindowSize          ;push pointer to rectangle
            PEA     WindowName          ;push pointer to name
            ST      -(SP)               ;yes, window is visible
            CLR.W   -(SP)               ;use standard document window
            MOVE.L  #-1,-(SP)           ;put window on top
            SF      -(SP)               ;no, window has no goAway box
            CLR.L   -(SP)               ;NIL window refCon
            _NewWindow                  ;make the window
            _SetPort                    ;make window current port

            MOVE.W  #patXor,-(SP)       ;set pen mode to patXor
            _PenMode                    ;give pen inverting ink

;  Here you set up the menu bar by accessing the resource file.

            SUBQ    #4,SP               ;make room for menu handle
            MOVE.W  #fileMenu,-(SP)     ;provide menu resource ID
            _GetRMenu                   ;put handle to menu on stack
            CLR.W   -(SP)               ;beforeID = 0
            _InsertMenu                 ;put menu on menu bar
            _DrawMenuBar                ;draw bar on screen

;  Here is the main event loop.

TryEvent
            _SystemTask                 ;give system time, check desk acc.
            SUBQ    #2,SP               ;make room for boolean result
            MOVE.W  #$FFFF,-(SP)        ;event mask for all events
            PEA     EventRecord(A5)     ;push space for record result
            _GetNextEvent               ;ask for event
            TST.B   (SP)+               ;see if any event occurred
            BEQ.S   TryEvent            ;if Z is set (no event), try again

            BSR.S   HandleEvent         ;event occurred, go to subroutine

            TST.B   DoneFlag(A5)        ;toggle Z according to DoneFlag
            BEQ.S   TryEvent            ;if Z is set, do again

            _ExitToShell                ;return to Desktop/Shell

;  Here you handle any event recorded by _GetNextEvent.
;  If Quit is chosen, then RTS with DoneFlag set (non-zero).

HandleEvent
            MOVE.W  EventRecord+what(A5),D0 ;get event number
            SUBQ    #mouseDown,D0       ;check mouse, toggle Z flag
            BEQ.S   FindMouse           ;if Z is set (yes down), read mouse
NextEvent
            CLR.B   DoneFlag(A5)        ;clear DoneFlag, not done
            RTS                         ;return with Z set

;  Here you check to see where the mouseDown event occurred.

FindMouse
            SUBQ    #2,SP               ;make space for window result
            MOVE.L  EventRecord+point(A5),-(SP)  ;move point onto stack
            PEA     WhichWindow(A5)     ;move event window pointer on stack
            _FindWindow                 ;determine where click occurred
            MOVE.W  (SP)+,D0            ;put result in register
            CMP     #inContent,D0       ;see if click was in content region
            BEQ.S   DoDraw              ;if so, begin graphics
            SUBQ    #inMenuBar,D0       ;see if click was in menu bar
            BNE.S   NextEvent           ;neither, so get next event
```

Listing 23-1 *cont.*

```
;  Here you handle the mouseDown event that occurred in the menu bar.

MenuPick
            SUBQ    #4,SP               ;make space for menu result
            MOVE.L  EventRecord+point(A5),-(SP)  ;provide global point
            _MenuSelect                 ;discover menu ID and item #
            MOVE.W  (SP)+,D1            ;menu ID in high order word
            MOVE.W  (SP)+,D0            ;menu item # in low word (not used)
            TST.W   D1                  ;toggle Z flag, check for menu ID
            BEQ.S   NextEvent           ;if Z is set (menu ID = 0), branch
            MOVE.B  #mouseDown,DoneFlag(A5) ;else set DoneFlag, quit
            RTS                         ;return with Z clear

;  Here you handle the mouseDown that occurred in the content region.

DoDraw
            PEA     EventRecord+point(A5)    ;put global point on stack
            _GlobalToLocal              ;convert to local coord.
            MOVE.L  EventRecord+point(A5),D5    ;store start point
            MOVE.L  D5,D4               ;initialize temp register (D4)

MouseCheck
            SUBQ    #2,SP               ;make space for result
            _StillDown                  ;check if button is down
            TST.B   (SP)+               ;toggle Z flag
            BEQ.S   NextEvent           ;up, so get next event

            SUBQ    #4,SP               ;make room for point result
            MOVE.L  SP,-(SP)            ;push pointer to result space
            _GetMouse                   ;get cursor coordinate point
            MOVE.L  (SP)+,D3            ;store end point in register

            CMP     D4,D3               ;compare temp and new end points
            BEQ.S   MouseCheck          ;if same, get new end point

            LEA     NewOval(A5),A2      ;put pointer to rect space in A2
            MOVE.L  D5,(A2)             ;specify top-left point at pointer
            MOVE.L  D4,4(A2)            ;specify bot-right at pointer+4
            MOVE.L  A2,-(SP)            ;push pointer (A2's contents)
            _FrameOval                  ;draw oval

            MOVE.L  D3,4(A2)            ;specify new bot-right at pointer+4
            MOVE.L  A2,-(SP)            ;push same pointer again
            _FrameOval                  ;erase previous oval

            MOVE.L  D3,D4               ;update old end point
            BRA.S   MouseCheck          ;return for new end point

;  Here are the definitions.

WindowSize  DC.W    80,60,290,450       ;window bounds (Top,Lft,B,R)
WindowName  DC.B    'Menus: Literature of Giants'   ;window title

NewOval     DS.L    2                   ;space for rect variable
EventRecord DS.B    16                  ;next event record
DoneFlag    DS.B    2                   ;space for boolean
WhichWindow DS.L    1                   ;space for window pointer result

            END                         ;code end directive
```



```
resource 'MENU' (129, "File", preload) {
    129, textMenuProc, allEnabled, enabled, "File",
    {
        "Quit",
            noIcon, noKey, noMark, plain
    }
};

include "14Menu.code";
```

Figure 23-1

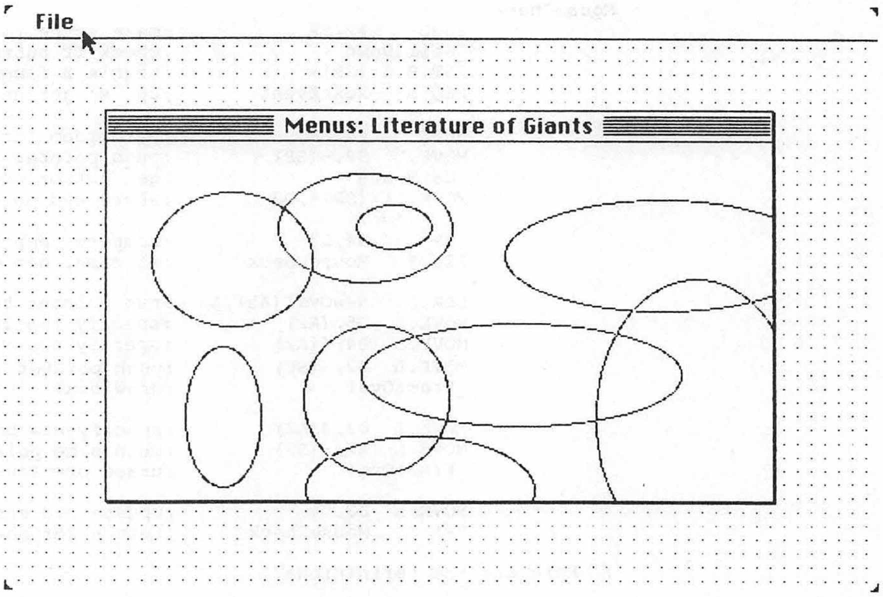


Figure 23-2

The assembling and linking process is somewhat more complicated when using resources. An additional command, called rez, is used. You can read more about resource compilation in chapter 7. The illustrations in chapter 7 were produced using the sample program in this chapter.

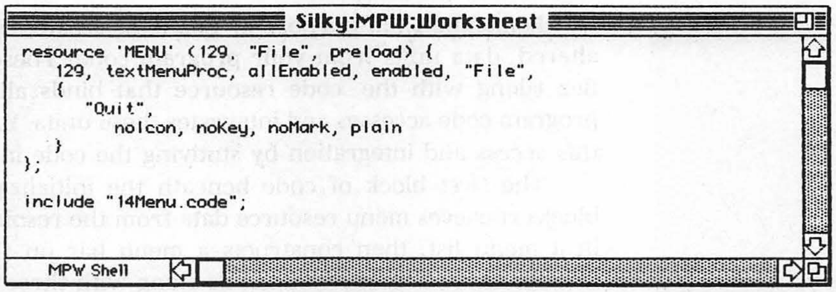
Resources are a type of programming code designed to simplify the process of program revision. Programs that use resources can change the resource aspects of the program without changing and recompiling the general program code. This flexibility makes language translations, attribute adjustments, and user customizations a surmountable task.

From the programmer's standpoint, resources add a challenge that is rewarded by a superior end product. The implementation involves two parts:

- Resources represent a "second program" that must be written and compiled with a special compiler (and "ultimate" linker) called Rez. The resource code must be written in a prescribed format that, sadly, is restrictive and nondescriptive.
- Resource data is accessed by the program through specifically designed Toolbox commands. These commands return pointers-to-other-pointers (called *handles*) that interact with the primary code.

Here are the instructions for writing and compiling resources:

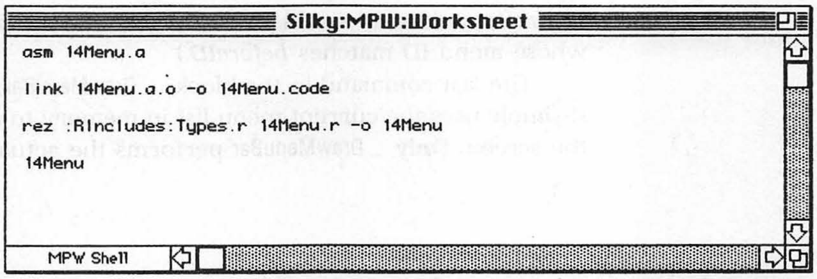
1. In an empty Worksheet, type the resource code exactly as you see it presented in figure 23-3.



```
resource 'MENU' (129, "File", preload) {
   129, textMenuProc, allEnabled, enabled, "File",
   {
      "Quit",
         noIcon, noKey, noMark, plain
   }
};
include "14Menu.code";
```

Figure 23-3

2. Save the code under the name 14Menu.r. You can use either the File menu or the MPW save command.
3. Create a Worksheet command script as shown in figure 23-4. Notice that the -o option on the link command line has been added so that the output file is named 14Menu.code. Type the rez command line exactly as you see it below your asm and link commands.



```
asm 14Menu.a

link 14Menu.a.o -o 14Menu.code

rez :RIncludes:Types.r 14Menu.r -o 14Menu
14Menu
```

Figure 23-4

4. Execute asm, link, and rez by selecting the command lines and pressing Enter.

The Rez compiler creates your final application. Notice that the final instruction within the resource source code:

```
include "14Menu.code";
```

treats your link output as part of the resource input. In this way, Rez becomes your ultimate linker.

After running Rez, an icon for the standalone application sits on your desktop. The link output, 14Menu.code, also has an application icon, but it has yet to implement the menu feature. You can try running 14Menu.code to see how uncompiled resources run, but be prepared to restart the Macintosh to exit the program.

Remember, the purpose of Rez is to create separate, and more easily altered, data units from your program code. These units are compiled by Rez (along with the .code resource that binds all other resources). Your program code accesses and integrates these units. You will see an example of this access and integration by studying the code in this chapter's program.

The first block of code beneath the initialization, window, and pen blocks retrieves menu resource data from the resource file, inserts the data in a menu list, then constructs a menu bar on the screen. The process involves three Toolbox commands along with three statements that provide additional parameters.

The two statements preceding _GetRMenu make a long word space on the stack for the handle result and provide a menu resource ID constant. The menu resource ID matches the constant value written in the resource file. _GetRMenu reads resource information into memory, returning with the menu handle left on the stack.

The menu handle is used by the next Toolbox command, _InsertMenu. _InsertMenu puts a specified menu at a particular position on a menu list. The CLR.W instruction provides a *beforeID* value of 0, indicating the given menu should be added to the list after any others. (With multiple menus, the *beforeID* parameter would insert the menu in the menu list before the menu whose menu ID matches *beforeID*.)

The last command in the block, _DrawMenuBar, requires no parameters. It simply uses the current menu list in memory to draw a new menu bar on the screen. Only _DrawMenuBar performs the actual drawing.

Waiter, there's a mouse in my menu

The main event loop and the graphic routines of LitOfGiants have not been changed since the last program. All the remaining new code (located between references HandleEvent and DoDraw) relates to finding the location of a mouse down event and responding if it's within the content region of the window or the menu bar.

The block beginning with HandleEvent uses the what field of the event record to check for a mouse down. Upon finding one, the branch to the FindMouse block determines if the mouse is located in the content region of the window, the menu bar, or neither. In each case, a different direction of execution results.

The new Toolbox command _FindWindow helps determine in which desktop feature the mouse down occurred. _FindWindow uses the where field of the event record to return two results: an integer corresponding to a desktop feature constant and a window pointer indicating in which window, if any, the event occurred.

Here is a list of desktop feature constants:

```
inDesk        EQU  0  ;in none of the following
inMenuBar     EQU  1  ;in menu bar
inSysWindow   EQU  2  ;in system window
inContent     EQU  3  ;in content region
                      ;(but not grow, if active)
inDrag        EQU  4  ;in drag region
inGrow        EQU  5  ;in grow region
                      ;(of active window only)
inGoAway      EQU  6  ;in go-away region
                      ;(of active window only)
```

To prepare for the Toolbox call, the SUBQ instruction makes a word of space on the stack for the integer result. Then the point field, still in global coordinates, is moved onto the stack. Last, the PEA instruction pushes the effective address of the WhichWindow storage space onto the stack. If the mouse has been pressed inside a window, the pointer to that window is stored in the WhichWindow space. If the mouse is pressed in no window, the pointer is set to NIL (zero).

After the _FindWindow call, the integer result is moved into register D0, then compared with two of the desktop feature constants to determine the proper path of execution. If the mouse down occurred in neither the content region nor the menu bar, the last statement in the block, BNE.S NextEvent, returns program flow to the event loop.

If the mouse down occurred in the menu bar, program flow drops through from the _FindMouse block to the block that enables the menu event to be further evaluated. The command _MenuSelect determines which menu option, if any, is selected. The stack parameters needed by _MenuSelect are a long word space to hold the menu result and the point field of the event record.

From the user's viewpoint, _MenuSelect takes control of the program from the moment the mouse is pressed in the menu bar until the mouse is released. In this period, _MenuSelect tracks the cursor, pulling down menus as appropriate, and highlights enabled menu items beneath the cursor.

When the mouse button is released, the high-order and low-order words of the long word result returned by _MenuSelect provide separate information. The menu ID in the high-order word indicates which menu in the menu list is selected. The menu item number in the low-order word indicates the specific menu option from the chosen menu, though the low-order word is not needed in this program because the File menu has only one option.

After the two result words are moved into registers, the high-order word, now in D1, is tested with the TST.W instruction. This toggles the Z flag, whose status indicates if the menu choice Quit has been selected. If Z is set (D1 has a zero value), the branch to NextEvent occurs. If Z is clear, the DoneFlag is set before program flow returns to the event loop.

The Los Alamos Sluggers (part 5 of 5)

"I guess I owe you some money," said Kafka. The sun had crossed the near ridge. Blue shade washed the afternoon of its swarthy pallor. Sparkling silica on street seams and crevices returned to dust. The brown eastern rise bounded back the sun; cacti in bloom pin-dotted yellow and white stars. The softball arced, caught against late sky and a town of scathed adobe and bleached bare lumber. Thirsty pitchers, older boys, the rhythmic thud of cowhide at a time when the house on the hill harvests light rays and bristlecone scents.

Eio strained to enunciate so there would be no misunderstanding. "You owe me nothing," he garbled.

Twice a day they ate—lunch and dinner at Chito's. Kafka paid. Declining suppers, Kafka, who seemed to lose his appetite the last few days of each month, sat with Eio on the glossy vinyl cafe booth. Had too big a lunch, he would say. Eio didn't remember lunch being any bigger than usual. After the second month Eio figured it out. Kafka resumed eating two meals a day on the first of the month, after a government letter arrived. As his allotment ran low, Kafka cut himself to lunches only. Eio thought of the days that he sat eating while Kafka sipped water. He had believed Kafka when he said he wasn't hungry. The end days of the last two months Eio claimed to have lost his appetite for suppers, also. The two men shared a bowl of chips and salsa.

They tossed pop flies and grounders, fast balls and bloopers, till their arms grew weary, then sat on the stoop of their home. Eio wanted to leave before dark. He watched Kafka etching pictures in the sand. He wondered how Kafka made the softball do tricks in the air. He believed in a magic that enabled Kafka to catch a ball too blurred to find lying five feet away in the corner of the room. He wanted to leave with this feeling of power. Eio had been afraid his employer might cackle and ruin their parting. Now he saw the contrary, almost as painful. Kafka was sad.

To make things worse, the woman in black happened to pass on her way up the hill. Kafka and Eio watched. Then she was gone.

Eio brought out the slate. I AM GOING NOW. Kafka read the message, then drew more designs in the sand. Eio waited. There was nothing more. Eio got up to go. Kafka reached for his arm, held him. Kafka had never touched him before.

A long moment passed. Kafka said to Eio, "Please stay until tomorrow."

Eio sat down again. He took Kafka's arm from his own, squeezed the top of his hand, and let go. Eio rubbed the board clean, wrote: TONIGHT IS A GOOD NIGHT TO TRAVEL.

"But the ray," pleaded Kafka. "We have to light her. We have to light her tonight." Kafka jumped up, ran inside, and came scrambling out shaking the mesquite twig. "We have to light her with everything we have. Everything. Please don't go now."

Eio brushed the slate. I CANNOT LIGHT ANYONE—WITH A STICK OR OTHERWISE.

Exasperated, Kafka's eyes grew wet, his hands drummed, and a quiver shook him. "Please, Eio. I know it's horrible. I know we are going to stun her. But we have to. We have to light her with everything. Don't you feel the urgency? Don't you see this wasted town is on the brink? We have to give her our heart and soul and guts and blood. We have to light her, Eio. We have to light her with everything we have. Now, Eio, now."

Kafka aimed the mesquite twig at the house on the hill. He held the rounded end with both his hands, shut his eyes, and opened his mouth incredibly wide. Eio turned his head, and saw a mongrel dog blocks away stop in its tracks and dash off. Eio leapt up, grabbed the ray, and wrenched it from Kafka's grip. Eio snapped it in two, then four, pieces.

The wide circle of Kafka's mouth contracted. He looked in Eio's hands at the broken pieces. His eyes struggled to meet Eio's. Kafka's lips spoke. "I knew it was just a stick."

* * *

He carried a paper sack, pad, and pencil, the same as when he arrived. He made plans to save money, so that next year he could buy Kafka lunches and suppers. Yeah, next year I'll buy his lunches and suppers. Seven steps toward the depot Eio kicked hard at the pebbles and spun himself around. "Arrgghh!" he cried aloud, slurring the sound. His head filled with unvented exclamations. He walked past the candle factory and began climbing the hill. He hated having thought so late. He hated having thought at all.

At her door he took out his pad and pencil. I CANNOT SPEAK WELL SO I WRITE. And on a separate sheet he put: I LIVE IN THE BROKEN BUILDING AT THE FOOT OF THE HILL.

Eio knocked. He hoped she would not ask "who's there" from behind the closed door. More than a minute passed. She will be frightened of me, he thought. She could call from an upper window and I will be forced to squawk at her in my maimed voice. Another minute. He knocked again, harder.

The door moved slightly as his knuckle hit wood. He stepped back. The door was unlatched. No. A line of her face appeared in the crack. She spied the length of him, the paper sack on the stoop. Her lips were hidden. He brought his hand forward with the first message. She read from his hand. The door stayed open only a slit. He showed her the second message. She did not pause to read. Her hand came out and took the paper from him. She raised one finger, then closed the door.

He waited, grew old. Time wavered the way rising heat stratifies the air. The door opened, this time enough to see her full face, her clothing. He gripped the pencil tightly. His chest began to ache. Write, he told himself. Blank paper stared up at him. Her hand came into his field of vision; she tapped on his pad. He followed her hand back to her body.

"That's okay. I know who you are," he saw her say. "Just tell me what you have come for."

Bravely, he watched her. She was covered in black, a bathrobe of velvety crushed cotton wrapped high about her neck. A swath of the same material across her waist held everything close, flowing down, river black, grazing the hardwood floor. Her hair was long and dark and full, awry strands clinging curled to the front of her robe. From afar, the sun and the darkness of her clothes deceived him. Eio had thought she was pale, but now he saw the Indian in her, tawny colors and almond eyes, moon sliver lines brushed on high cheeks.

He almost talked. A gnarled sound dropped from his mouth. He caught himself, stilled the vibration in his throat. He looked to his pad. The words were gone. In her face he thought he saw someone he knew.

I AM LEAVING NOW. MY FRIEND TOOEY WILL BE ALONE. HIS EYES ARE BAD. WOULD YOU PLEASE LOOK IN ON HIM. HE IS A GENTLE AND GENEROUS MAN.

She stepped outside to read Eio's note, lit by the late day sun.

PART THREE

The MPW and Assembly Dictionaries

The MPW Shell dictionary, chapter 24, contains all the commands and command options that make up the Shell programming environment (MPW version 2.0B1). The assembly dictionary, chapter 25, contains the entire 68000 instruction set plus all the assembly directives and Toolbox traps used in parts 1 and 2.

BuildProgram	*Perform the program build*
C	*Compile C source code*
Canon	*Copy using canonical spelling*
Catenate	*Concatenate files (join into one file)*
Clear	*Clear text without saving to Clipboard*
Close	*Close window*
Compare	*Display comparison of text files*
Confirm	*Display message in confirmation dialog*
Continue	*Continue at start of For or Loop command*
Copy	*Copy text onto Clipboard*
Count	*Display file's line and character count*
CreateMake	*Make build commands automatically*
Cut	*Cut text after copying to Clipboard*
CvtObj	*Convert object files, Lisa to MPW*
Date	*Display the clock's date and time*
Delete	*Delete disk files and directories*
DeleteMenu	*Delete user-defined menus*
DeRez	*Decompile resources*
Directory	*Set or display the default directory*
DirectoryMenu	*Display the menu for directory*
DumpCode	*Disassemble object code of resource fork*
DumpObj	*Disassemble object code of data fork*
Duplicate	*Duplicate disk files and directories*
Echo	*Echo (display) parameters*
Eject	*Eject disk volume*
Entab	*Change consecutive spaces into tabs*
Equal	*Display file and directory inequalities*
Erase	*Erase (initialize) disk volume*
ErrTool	*Create text file of error messages*
Evaluate	*Evaluate list of words as an expression*
Execute	*Execute command file with global scope*
Exists	*Find out if a file or directory exists*
Exit	*Exit from command or command file*
Export	*Export variable names to commands*
FileDiv	*Divide file into files of specified length*
Files	*List contents of files and directories*
Find	*Find and select specified text*
Font	*Set font and font size*
For	*Execute command list for each parameter*
Help	*Display information in help file*
If	*Execute command if true expression*
Lib	*Create library of object files*
Link	*Link object files*
Loop	*Execute command list until Break*
Make	*Make new program version*
Mark	*Mark a selection of text*
Markers	*Display text selection markers*
MDSCvt	*Convert assembler source, MDS to MPW*
Mount	*Mount disk volumes*
Move	*Move contents of files and directories*
MoveWindow	*Move window to screen coordinates*
New	*Open new window and make active*
Newer	*Display newer file names*

ADDI *Add Immediate*
ADDQ *Add Quick*
ADDX *Add with Extend*
AND *AND Logical*
ANDI *AND Immediate*
ANDI to CCR *AND Immediate to the Condition Code Register*
ANDI to SR *AND Immediate to the Status Register*
ASL *Arithmetic Shift Left*
ASR *Arithmetic Shift Right*
Bcc *Branch Conditionally*
BCHG *Test a Bit and Change*
BCLR *Test a Bit and Clear*
BRA *Branch Always*
BSET *Test a Bit and Set*
BSR *Branch to Subroutine*
BTST *Test a Bit*
_Button *ROM Trap*
CHK *Check Register Against Bounds*
CLR *Clear a Subject*
CMP *Compare*
CMPA *Compare Address*
CMPI *Compare Immediate*
CMPM *Compare Memory*
DBcc *Test Condition, Decrement, and Branch*
DC *Define Constant*
DCB *Define Constant Block*
DIVS *Signed Divide*
DIVU *Unsigned Divide*
_DrawMenuBar *ROM Trap*
_DrawString *ROM Trap*
DS *Define Storage*
END *End of Source*
EOR *Exclusive OR Logical*
EORI *Exclusive OR Immediate*
EORI to CCR *Exclusive OR Immediate to the Condition Code Register*
EORI to SR *Exclusive OR Immediate to the Status Register*
EQU *Equate Permanent Value*
EXG *Exchange Registers*
_ExitToShell *ROM Trap*
EXT *Sign Extend*
_FindWindow *ROM Trap*
_FlushEvents *ROM Trap*
_FrameOval *ROM Trap*
_FrameRect *ROM Trap*
_GetMouse *ROM Trap*
_GetNextEvent *ROM Trap*
_GetRMenu *ROM Trap*
_GlobalToLocal *ROM Trap*
ILLEGAL *Illegal Instruction*
INCLUDE *Include Source File*
_InitCursor *ROM Trap*
_InitFonts *ROM Trap*
_InitGraf *ROM Trap*

_InitMenus *ROM Trap*

_InitWindows *ROM Trap*

_InsertMenu *ROM Trap*

_InverRect *ROM Trap*

JMP *Jump*

JSR *Jump to Subroutine*

LEA *Load Effective Address*

LINK *Link and Allocate*

LSL *Logical Shift Left*

LSR *Logical Shift Right*

MAIN *Begin Main Program Code Module*

_MenuSelect *ROM Trap*

MOVE *Move Data from Source to Destination*

MOVE to CCR *Move to the Condition Code Register*

MOVE to SR *Move to the Status Register*

MOVE from SR *Move from the Status Register*

MOVE USP *Move User Stack Pointer*

MOVEA *Move Address*

MOVEM *Move Multiple Registers*

MOVEP *Move Peripheral Data*

MOVEQ *Move Quick*

_MoveTo *ROM Trap*

MULS *Signed Multiply*

MULU *Unsigned Multiply*

NBCD *Negate Decimal with Extend*

NEG *Negate*

NEGX *Negate with Extend*

_NewWindow *ROM Trap*

NOP *No Operation*

NOT *Logical Complement*

OR *Inclusive OR Logical*

ORI *Inclusive OR Immediate*

ORI to CCR *Inclusive OR Immediate to the Condition Code Register*

ORI to SR *Inclusive OR Immediate to the Status Register*

PEA *Push Effective Address*

_PenMode *ROM Trap*

_PtInRect *ROM Trap*

RESET *Reset External Devices*

ROL *Rotate Left*

ROR *Rotate Right*

ROXL *Rotate Left with Extend*

ROXR *Rotate Right with Extend*

RTE *Return from Exception*

RTR *Return and Restore Condition Code Register*

RTS *Return from Subroutine*

SBCD *Subtract Decimal with Extend*

Scc *Set According to Condition Codes*

_SetPort *ROM Trap*

_StillDown *ROM Trap*

STOP *Load Status Register and Stop*

SUB *Subtract Binary*

SUBA *Subtract Address*

SUBI *Subtract Immediate*

SUBQ *Subtract Quick*
SUBX *Subtract with Extend*
SWAP *Swap Register Halves*
_SysBeep *ROM Trap*
_SystemTask *ROM Trap*
TAS *Test and Set a Subject*
TRAP *Trap*
TRAPV *Trap on Overflow*
TST *Test a Subject*
UNLK *Unlink*

CHAPTER 24

The MPW Shell Command Language

AddMenu Add and display user-defined menus

```
AddMenu (menuName (itemName (commandList)))
```

Creates and displays new menus and menu items. If the parameter *menuName* is new, it is added to the menu bar along with any specified items. If *menuName* already exists, *itemName* is added to the bottom of *menuName.* The *commandList* parameter provides the course of action when a menu item is selected.

The Apple, Format, and Window menus will not accept user-defined items, nor can *itemName* include semicolons.

The metacharacters that define keyboard equivalents and other item characteristics can be included with *itemName* as long as the entire *itemName* expression is enclosed in quotation marks. Metacharacters include:

/*char*	*char* is the command key equivalent.
!*char*	*char* is to the left of the menu item.
^*n*	*n* is an item's icon number.
(	Item is disabled and dimmed.
<*style*	*style* is one of the following letters: B = bold, I = italic, U = underline, O = outline, S = shadow.

Omitting *commandList* writes the *itemName's* command list to standard output. Omitting *commandList* and *itemName* writes all user-added items for *menuName* to standard output. Omitting all parameters writes all user-added items to standard output.

Adjust Adjust position of text

Adjust (-c *count)* (-l *spaces) selection (window)*

Shifts the lateral position of a selection of lines without altering the relative indentation. The default adjustment is one space to the right. Adjustment is made to the target window unless otherwise specified by the *window* parameter.

Options

-c *count* Command is performed *count* times.

-l *spaces* Adjust the specified number of *spaces* to the right or left. A positive value shifts to the right. A negative value shifts to the left.

Alert Display message in an alert dialog box

Alert (-s) *(message)*

Displays a dialog box containing the parameter *message* and an OK button. A message that contains a space or special character must be in quotation marks. Omitting the *message* parameter causes standard input to be read.

Option

-s Omit dialog box beep sound.

Alias Define or display alias names

Alias *(name (wordList))*

Defines a *name* to work as a substitute for a list of command words. Global aliases are defined in the StartUp file. Other aliases are local to where they are defined. Local names override global names. The command Unalias removes aliases.

Omitting *wordList* writes any alias defined as *name* and its word list to standard output. Omitting *wordList* and *name* writes all aliases and their word lists to standard output.

Align Align position of text

Align (-c *count*) *selection* (*window*)

Positions all the lines within a selection of lines the same number of spaces from the left margin as the selection's first line. Alignment is made to the target window unless otherwise specified by the *window* parameter.

Option

-c *count* Perform command *count* times.

Asm Assemble 68000 or 68020 source code

Asm (*optionList*) (*fileList*)

Assembles source code of the specified file according to the option settings. Source code files must end with the suffix .a (i.e., *fileName*.a). Completion of the assembly produces an object code file that ends with the suffix .a.o (i.e., *fileName*.a.o). When a listing option is included, the listing file ends with the suffix .a.lst (i.e., *fileName*.a.lst).

More than one file can be specified for assembly—each file is assembled separately. Omitting the *fileList* parameter assembles standard input and creates the object file a.o. Files and options can be listed in any order, though the -case on option has special requirements.

Options

-addrsize *size* Display addresses in the listing in *size* digits. The parameter must be in the range of 4 to 8; the default is 5.

-blksize *blocks* Set the I/O buffers to *blocks* times 512 bytes, where the default is 16 and values from 6 to 62 are allowed.

-case on Differentiate uppercase and lowercase letters in nonmacro names (macro names always ignore case). This option must precede the -define option in the parameter list to preserve the case of declared names. (The option's second word, on, is part of the option name, not a parameter.)

-case obj(ect) Save the case of letters in module, EXPORT, IMPORT, and ENTRY names only in the generated object file. (The option's second word, obj(ect), is part of the option name, not a parameter.)

-case off Ignore the case of letters. This is the assembler's default mode. The option turns off the other -case modes. (The option's second word, off, is part of the option name, not a parameter.)

-c(heck) Examine syntax for errors without generating any object file.

-d(efine) *name(=value) (,name2(=value2))* Equate *name* with the specified decimal integer. Omitting =*value* causes a default value of 1.

-d(efine) *&name(=(value)) (,&name2(=(value2)))* Equate the macro *name* with the specified decimal integer or string constant. Omitting =*value* causes a default value of 1. Omitting *value* causes a default of the null string. (The ampersand, &, must be part of the parameter name.)

-e(rrlog) *filename* Write all errors and warnings to an error log file of the specified name.

-f Turn off page ejects.

-font *fontname (,fontsize)* Print -s and -l listings in the specified font and font size. The default listing is Monaco 7. Only monospaced fonts produce proper formatting.

-h Turn off page headers.

-i *pathname (,pathname2)* Search for files in the specified directories.

-l Write a full listing to a listing file.

-lo *listingname* Causes -s and -l listing files to use the *listingname* pathname. Also, the listing scratch file uses the *listingname* directory.

-o *objname* Generated object file uses the *objname* pathname. A colon following *objname* provides a directory pathname rather than a file name.

-pagesize *(len) (,wid)* Set the page size of -s and -l listings according to a length and width integer. Default values are *len* = 74 and *wid* = 130 with Monaco 7.

-print *mode (,mode2)* Set one of the following print directive modes: (NO)GEN, (NO)PAGE, (NO)WARN, (NO)MCALL, (NO)OBJ, (NO)DATA, (NO)MDIR, (NO)HDR, (NO)LITS, (NO)STAT, (NO)SYM.

-p Write module names, error totals, warnings, and compilation time to the diagnostic file.

-s The option -print NOOBJ makes a compact listing file.

-t Display the assembly time and number of lines to the diagnostic file.

-w Turn off warning messages.

-wb Turn off branch warning messages.

Backup Generate file backup script

Backup *(optionList)* -from *folder* -to *folder (fileList)*

Writes to standard output a Shell command file consisting of Duplicate commands that, when executed, make backup copies of folder files according to the modification date. The *folder* parameter can be replaced with a drive number (1 or 2).

Unless an option directs otherwise, a file on the from *folder* must also exist on the -to *folder*, both source and destination files must have the same type and creator, and the destination file must have a modification date that is older than that of the source file.

Options

-a Write a Duplicate command for all source files that do not exist in the destination.

-alt Seek alternate drive numbers when additional disks are requested by the -m option.

-c Write a Shell Newfolder command to output when a source folder name does not exist in the destination and the -a option is used.

-check *checkopt (,checkoptList)* Generate file existence reports based upon the *checkopt* parameter value of from, to, allfroms, alltos, folders, or newer (destination newer than source).

-co *filename* Write the -check report to *filename* instead of to the diagnostic file.

-compare (only,*'optionList'*) | *'optionList'* Write Shell Compare commands to output for all type TEXT files that produce Duplicate commands. When only is included, Duplicate commands are omitted.

-d Write Shell Delete commands to output for all destination folder files that don't exist in the source folder.

-do (only,*'commandList'*) | *'commandList'* Write a Shell command string to output for all files that produce Duplicate commands. When only is included, Duplicate commands are omitted.

-e Write Shell Eject commands to output after Duplicate commands are generated, or eject a parameter drive (1 or 2) when there are no files to duplicate.

-from *folder* | *drive* Search for source files from the specified *folder* or *drive* (1 or 2). A sequence of file name parameters can be substituted for the option.

-l Write a listing of files in the source folder to output.

-m Dialog box requests that additional disks be inserted for backup to more than one disk. The -n option, when used with the -r option, causes nested files to generate Duplicate commands with leading spaces.

-p Write information about the backup to output.

-r Commands operate recursively on subfolders and their nested files.

-revert Files in the destination folder with newer modification dates than their counterparts in the source folder revert (generate Duplicate commands) to their older form.

-since *date (,time)* | *,(time)* | *filename* Write Duplicate commands when the modification date is newer than the *date* (mm/dd/yy) and *time* (hh,mm,ss) or modification date of the parameter *filename.*

-sync Write duplication commands whenever one file version is newer than another regardless of the source or destination.

-t *type* Limit duplication criteria to files of the parameter *type.*

-to *folder* | *drive* Search for destination files from the specified *folder* or *drive* (1 or 2). A sequence of file name parameters can be substituted for the option.

-y Output Duplicate commands do not use, by default, the -y option.

Beep Create beep sound

Beep *(note (,duration (,level)))*

Produces a tone on the Macintosh speaker according to the parameter values. Omitting all parameters generates a simple beep.

The *note* parameter (A to G) can be preceded by an optional number (−3 to 3) for octaves below or above middle C, and followed by an optional sharp (#) or flat (b) sign. (Quote entire parameter if you use a sharp sign.) The *note* parameter can also be supplied as the count field of the sound driver's square wave generator. The *duration* integer is measured in sixtieths of a second (default = 15). The sound *level* integer must be in the range of 0 to 255 (default = 128).

Begin . . . End Set begin and end of a command block

```
Begin ; commandList ; End
```

Provides parenthetical bounding so that commands enclosed with Begin and End are treated as a single unit. The Shell's special command symbols for pipe, conditional, and I/O operations (see the *MPW Reference Manual*) affect the unit as a whole when inserted after End. The *commandList* must be bounded by semicolons or Return characters.

Break Break execution of For or Loop command

```
Break (If expression)
```

Provides the endpoint for a For or Loop command. If you omit If *expression,* a break always occurs. Otherwise, a break occurs only when *expression* is true.

BuildCommands Write command script for build

```
BuildCommands (optionList) program
```

Writes to standard output the commands needed to build the parameter *program.*

Option

-e Write a complete set of Build commands regardless of the need to rebuild any particular file.

BuildMenu Display the menu for build

```
BuildMenu
```

Adds the Build menu to the menu bar. Menu items include Create Build Commands... , Build... , Full Build... , Show Build Commands... , and Show Full Build Commands....

BuildProgram Perform the program build

BuildProgram *program (optionList)*

Builds the parameter *program* and writes the build command script to standard output.

Option

-e Rebuild the program completely regardless of the need to rebuild any particular file.

C Compile C source code

C *(optionList) (fileList)*

Compiles C source code of the specified file according to the option settings. Source code files must end in the suffix .c (i.e., *fileName*.c). Completion of the compilation produces an object code file that ends with the suffix .c.o (i.e., *fileName*.c.o). More than one file can be specified for compilation—each file is compiled separately. Omitting the *fileList* parameter compiles to standard input and creates the object file c.o. Files and options can be listed in any order, though the -case on option has special requirements.

Options

-c Include comments in the preprocessor output.

-d *name* Equate *name* with the value 1 in the preprocessor.

-d *name=string* Equate *name* with the value *string* in the preprocessor.

-e Write the output of the preprocessor to standard output without compiling the program or producing an object file.

-g Produce stack frame pointer in register A6 for all functions. The procedure name is put into the object code.

-ga Produce stack frame pointers in register A6 for all functions.

-i *pathname (,pathname2)* Search Include files in the specified directories.

-o *objname* Compiled object file uses the *objname* pathname. A colon following *objname* provides a directory pathname rather than a file name.

-p Write information about the progress of the command to the diagnostic file.

-q Optimize speed at the possible expense of code size.

-q2 Additional optimization used only when memory locations are changed by explicit stores.

-s *name* Assign a *name* to an object code segment.

-u *name* *name* is undefined in the preprocessor.

-w Compiler warning messages do not occur.

-x6 `MOVE #0,x` instructions replace `CLR x` instructions for non-stack addresses.

-x55 Sign the bit fields of type int, short, and char.

-z6 Enumerated data types are 32 bits in size.

-z84 Use language anachronisms.

Canon Copy using canonical spelling

Canon (-a) (-c n) (-s) *dictionary (fileList)*

Changes the spelling of a file's identifiers using a *dictionary* of canonical spelling. The file's data forks are copied to standard output. Omitting *fileList* causes standard input to be copied.

The text file dictionary contains (first) the identifier to replace and (second) its canonical spelling. When a line of *dictionary* contains only a single identifier, all case forms of the identifier are replaced with the exact form shown.

Options

-a Read the characters @, $, and % as letters of an identifier.

-c *n* Identifier look-up matches only the first *n* characters.

-s Identifier look-up matches only when case is identical.

Catenate Concatenate files (join into one file)

Catenate *(fileList)*

Joins files by reading their data forks in the order shown in *fileList,* then writing the result to standard output. Omitting the input file causes standard input to be read.

Clear Clear text without saving to Clipboard

```
Clear (-c count) selection (window)
```

A *selection* of lines is removed without being copied to the Clipboard. Adjustment is made to the target window unless otherwise specified by the *window* parameter.

Option

-c *count* Perform the command *count* times.

Close Close window

```
Close (-a) (-n | -y) (windowList)
```

The specified window is closed. Unless an option directs otherwise, unsaved changes to a window produce a dialog box to confirm the action. Omitting the *windowList* parameter closes the target window (second from the top).

Options

-a Close all open Shell windows.

-n Close the contents of *windowList* without a save, circumventing the dialog box.

-y Save, then close, the contents of *windowList,* circumventing the dialog box.

Compare Display comparison of text files

```
Compare (optionList) file1 (file2)
```

The differences between two specified text files are written to standard output. Each line of text that does not match its sequential line in the other file is placed in a stack maintained for each file. The command attempts to resynchronize matching lines of text after a mismatch is found. Omitting the *file2* parameter causes the standard input file to be compared to *file1*

Options

-b Remove trailing blanks and compress a series of blanks to one blank.

-c *col1-col2 (,col1-col2)* Compare files only within the specified range of columns. The first range of columns applies to *file1*, the second range to *file2*. If the second range is omitted, both files use the first range. The default value of *col1* is 1. The default value of *col2* is 255. Tabs must be expanded for the *-c* option to work.

-d *depth* Determine the largest value that the stack can grow before halting resynchronization. The value of the integer *depth* ranges from 1 to 1000. The default, using dynamic grouping, is 1000. When the -s option is used for static grouping, the default is 25.

-e *context* Show *context* lines, using a parameter value of 1 to 100, surrounding the nonmatching lines.

-g *groupingfactor* Help determine a mathematical factor for recognizing resynchronization (the number of lines that must match for two files to be considered resynchronized). The lower limit and default value is 2 for dynamic grouping.

-h *width* Show the two stacks of nonmatched lines horizontally. The total number of characters allowed in a line is given by the *width* parameter and must be in the range of 70 to 255.

-l Do not recognize differences in uppercase and lowercase.

-m Do not output nonmatching lines.

-n Withhold messages to standard output when both files match.

-p Include the command's version information in the diagnostic file.

-s The grouping factor becomes static.

-t Remove trailing spaces without space compression.

-v Output file differences in a line-by-line format.

-x Do not expand tabs into spaces.

Confirm Display message in confirmation dialog

```
Confirm (-t) (messageList)
```

Creates a dialog box that contains the parameter *messageList* with OK and Cancel buttons. The selection result is stored in the Status variable. OK returns 0 and Cancel returns 4. Omitting the *messageList* parameter causes standard input to be read.

Option

-t Produce a three-button dialog box that offers Yes, No, and Cancel. Yes returns 0, No returns 4, and Cancel returns 5.

Continue Continue at start of For or Loop command

```
Continue (If expression)
```

Provides for a return to the starting point of a For or Loop command. Or, in the case of the final iteration, continue beyond For or Loop. If you omit If *expression,* a continuation always occurs. Otherwise, a continuation occurs only when *expression* is true.

Copy Copy text onto Clipboard

```
Copy (-c count) selection (window)
```

A *selection* of lines is reproduced onto the Clipboard. Adjustment is made to the target window unless otherwise specified by the *window* parameter.

Option

-c *count* Perform the command *count* times.

Count Display file's line and character count

```
Count (-c | -l) (fileList)
```

The number of lines and characters of an input file are tabulated, and the results are written to standard output. Each line of output contains the file name, line count, and character count, in that order. Separate totals for each file in *fileList* are produced as well as the grand totals of *fileList*. Omitting the *fileList* parameter causes standard input to be read.

Options

-c Display only character counts.

-l Display only line counts.

CreateMake Make build commands automatically

```
CreateMake (-Application | -Tool | -DA) program fileList
```

Creates a file (makefile) that contains the commands necessary to build the *program* parameter. The makefile's name is the program name with the suffix .make appended. MPW's standard library files are automatically linked with the program, and additional library files can be specified as parameters. Unlike makefiles created with the Make command, there are no dependencies on Include and Uses files.

Cut Cut text after copying to Clipboard

```
Cut (-c count) selection (window)
```

A *selection* of lines is reproduced onto the Clipboard, then removed. Adjustment is made to the target window unless otherwise specified by the *window* parameter.

Option

-c *count* Perform the command *count* times.

CvtObj Convert object files, Lisa to MPW

```
CvtObj (-n namesFile) (-o outputFile) (-p) LisaObjFile
```

Converts an object file that was created on a Lisa to the object file format of the Macintosh Programmer's Workshop.

Options

-n *namesFile* Name the conversion file *namesFile.*

-o *outputFile* Name the output file *outputFile.*

-p Write information about the command's progress to the diagnostic file.

Date Display the clock's date and time

```
Date (-a | -s) (-d | -t)
```

Writes the date and time from the Macintosh clock to standard output.

Options

-a Shorten the date notation by using three-character abbreviations for the month and the day of the week.

-d Write only date output.

-s Shorten the date notation by using mm/dd/yy notation and not providing the day of the week.

-t Write only time output.

Delete Delete disk files and directories

```
Delete (-c | -n | -y) (-i) (-p) name
```

Removes the file name or, if *name* represents a directory, all files and subdirectories in the directory. Unless an option directs otherwise, the command produces a dialog box to confirm the removal of a directory.

Options

-c Cause a cancel response to any confirmation dialog box, halting the command when a directory is found.

-i Do not print error messages and return a status value of zero, which represents no errors.

-n Cause a no response to any confirmation dialog box, not deleting any directory that is found.

-p Write information on the progress of the deletion to the diagnostic file.

-y Cause a yes response to any confirmation dialog box, deleting any directory that is found.

DeleteMenu Delete user-defined menus

DeleteMenu *(menuName (itemName))*

Removes the menus and menu items that are user-defined. User-defined menus are created with the command AddMenu. Omitting *itemName* deletes all user-added items for *menuName.* Omitting *itemName* and *menuName* deletes all user-added items.

DeRez Decompile resources

DeRez *(optionList) resourceFile (resourceDescriptionFileList)*

Translates the compiled code of the specified file's resources into a text description of the file. The description is written to standard output and is in the same format that the Rez (resource compiler) program uses for input.

The parameter *resourceDescriptionFileList* specifies one or more files of formatted type declarations that enable the command to format resource data. Omitting *resourceDescriptionFileList* outputs resource data in hexadecimal only.

The file Types.r contains the common Macintosh resource declarations. The file SysTypes.r contains the system resource declarations. Both files are originally stored in the RIncludes folder.

Options

-d(efine) *macro(=data)* Equate the specified *macro* variable with the value *data* or, if *data* is omitted, the null string.

-i Use one or more pathnames in searches for #include files.

-m(axstringsize) *n* Limit output strings to *n* characters, where *n* ranges from 2 to 120.

-only *type* Limit the command's scope to only resources of the specified literal *type.*

-only *typeExpr ((ID1(:ID2)) | resourceName)* Limit the command's scope to only resources of the specified type and, if specified, ID, range of IDs, or *resourceName.* The type may be given as an expression when proper quoting is used.

-p Write information about the progress of the command to the diagnostic file.

-rd Redeclared resource types do not write a warning message to output.

-s(kip) *type* Limit the command's scope by omitting resources of the specified literal *type.*

-s(kip) *typeExpr ((ID1(:ID2)) | resourceName)* Limit the command's scope by omitting resources of the specified type and, if specified, ID, range of IDs, or *resourceName.* The type may be given as an expression when proper quoting is used.

-u(ndef) *macro* Undefine a *macro* variable.

Directory Set or display the default directory (MPW version 1.0)

Directory (-q) *(directory)*

Establishes a new default directory when a parameter is given. Omitting the *directory* parameter writes the pathname of the current default directory to standard output.

Option

-q Write the special characters in displayed file names unquoted.

DirectoryMenu Display the menu for directory

DirectoryMenu *(directoryList)*

Adds the Directory menu to the menu bar. Menu items include Show Directory and Set Directory. When a parameter list is given, the directories appear as menu items. When a new directory is set as the current directory, it is added as a menu item.

DumpCode Disassemble object code of resource fork

DumpCode *(optionList) resourceFile*

Translates the object code in a file's resource fork to formatted assembly code. The formatted assembly code is written to standard output. Disassembled displays are in hexadecimal and ASCII. The object code in a file's data fork can be disassembled using the DumpObj command.

Options

-d Write CODE resource information to standard output without disassembly and code display.

-di Do not write the data initialization code to output.

-h Do not write header information to output.

-jt Do not format the jump table.

-n Write only a resource's names to output.

-p Write information about the progress of the disassembly to the diagnostic file.

-r *byte1 (,byteN)* Disassemble only within the specified range of bytes. Omitting *byteN* causes disassembly through the end of the segment.

-rt *type(=ID)* Disassemble only for the single resource of the specified *type* and *ID* number. Omitting the *ID* number includes all resources of the *type.*

-s *name* Disassemble only the single resource of the specified *name.*

DumpObj Disassemble object code of data fork

DumpObj *(optionList) objectFile*

Translates the object code in a file's data fork to formatted assembly code. The formatted records and code are written to standard output. The object code in a file's resource fork can be disassembled using the DumpCode command.

Options

-d Write object file information to standard output without disassembly and data display.

-i Do not use the names of IDs in place of ID numbers.

-h Do not write header information to output.

-l Write the file placement of object records to output.

-m *name* Disassemble and display the specified module.

-n Write only names to output.

-p Write information about the progress of the disassembly to the diagnostic file.

-r *byte1 (,byteN)* Disassemble only within the specified range of bytes. Omitting *byteN* causes disassembly through the end of the segment.

Duplicate Duplicate disk files and directories

Duplicate (-c | -n | -y) (-d | -r) (-p) *nameList destination*

Reproduces the specified file or directory. A duplicated file is given the name *destination* or, if *destination* is a directory, placed within *destination.* A duplicated directory has all its contents reproduced in *destination.* Unless an option directs otherwise, overwriting an existing file or directory produces a dialog box asking confirmation.

Options

-c Halt the command before an overwrite instance, circumventing the confirmation dialog box.

-d Reproduce only the data fork.

-n Disallow overwrite instances, circumventing the confirmation dialog box.

-p Write information on the progress of the duplication to the diagnostic file.

-r Reproduce only the resource fork.

-y Allow overwrite instances to occur directly, circumventing the confirmation dialog box.

Echo Echo (display) parameters

Echo (-n) *(parameterList)*

The parameter values are written to standard output, followed by a carriage return.

Option

-n The character insertion point immediately follows the last line of output by suppressing the closing carriage return.

Eject Eject disk volume

Eject (-m) *volumeList*

The specified volume is taken off-line or, in the case of a floppy disk, ejected. A colon must follow the volume name unless *volumeList* is a disk drive number.

Option

-m The volume remains mounted.

Entab Change consecutive spaces into tabs

Entab *(optionList) (fileList)*

Converts a file's consecutive spaces to tabs, then writes the file to standard output. Unless an option directs otherwise, single quote (') and double quote (") characters function as left and right delimiters within which tabs are not placed.

Options

-d *tabSetting* Convert a file's tabs to consecutive spaces. The parameter value determines the tab stop setting.

-l quotes A string of nonblank characters functions as an opening quote character, thus assuring that Entab will not place tabs within quoted strings. The -r quotes option must also be included.

-n All quotes are susceptible to Entab's conversion.

-p Write information about the progress of the conversion to standard output.

-q quotes A string of nonblank characters functions as an opening or a closing quote character, thus assuring that Entab will not place tabs within quoted strings.

-r quotes A string of nonblank characters functions as an opening quote character, thus assuring that Entab will not place tabs within quoted strings. The -l quotes option also must be included.

-t *tabSetting* Convert consecutive spaces to tab characters in the detabbed input. The -d option, which causes a file to be detabbed, should be used prior to this option. The parameter value determines the tab stop setting.

Equal Display file and directory inequalities

```
Equal (optionList) nameList destination
```

Checks for equality between the specified files or directories. The file, fork, and byte locations of any differences are written to standard output. The parameters must be both files, or both directories, or the file *nameList* and the directory *destination.* In the last case, *nameList* is checked for equality with a file of the same name within *destination.*

Options

-d Compare only the data fork.

-i Do not output missing file differences when files in *nameList* are not in *destination.*

-p Write information on the progress of the comparison to the diagnostic file.

-q Do not write differences to standard output. Only status codes are affected.

-r Compare only the resource fork.

Erase Erase (initialize) disk volume

```
Erase (-s) (-y) volumeList
```

Deletes all previous contents by initializing the specified volume(s). A colon must follow the volume name unless volume is a disk drive number. Unless an option directs otherwise, erasing a volume produces a dialog box asking confirmation.

Options

-s Initialize the disk in the single-sided, 400K, non-HFS format.

-y Initializations occur directly, circumventing the confirmation dialog box.

ErrTool Create text file of error messages

ErrTool *(optionList) (fileList)*

Creates a text file—from standard input or the parameter *fileList*—that contains the messages used when an error is encountered.

Options

-l Write error messages and their error numbers to standard output.

-o *objname* The output file takes the given pathname or is inserted in the given directory name.

-p Write information about the progress of the command to the diagnostic file.

Evaluate Evaluate list of words as an expression

Evaluate *(wordList)*

The specified words make up an expression whose result is written to standard output. Because spaces are used to separate each part of the expression, strings that use spaces need to be enclosed in quotation marks.

Execute Execute command file with global scope

Execute *commandFile*

Runs a command file such that aliases, exports, and variable definitions remain defined after execution. Running a command file without using Execute causes such definitions to be recognized only with local scope, that is, within the command file. A command parameter that is not a file will run as if the command appeared by itself.

Exists Find out if a file or directory exists

Exists (-d | -f | -w) (-q) *nameList*

Writes to standard output the names of files or directories that exist and meet any option specifications.

Options

-d Output only a directory.

-f Output only a file.

-q Write pathnames unquoted.

-w Output only a file that is not open or locked.

Exit Exit from command or command file

```
Exit (status) (If expression)
```

Ends the execution of a previous command or, if part of a file, the file itself. The parameter *status* can be included to read the status value of the command file. Omitting If *expression* causes an exit to occur always. Otherwise, an exit occurs only when *expression* is true.

Export Export variable names to commands

```
Export (-r | -s | nameList)
```

Allows the specified variable names to be used in command files and tools. Omitting the parameter writes the names of the current command file's local variables to standard output. Variables that are exported from the StartUp file are available for use anywhere in the Shell environment.

Options

-r The output unexports variables.

-s The output only lists exported variables.

FileDiv Divide file into files of specified length

```
FileDiv (-f) (-n splitpoint) (-p) file (prefix)
```

Splits a *file* into smaller files and provides the new files with numeric names. Unless an option directs otherwise, the input file is split into files that are 2,000 lines in length and use the input file name as a prefix in the manner *prefix*01, *prefix*02, and so on.

Options

-f The file break occurs only when there is a formfeed character (the first character of a line is ASCII $0C) beyond the point where the file would otherwise be split. This can be used with the -n *splitpoint* option.

-n *splitpoint* The file break occurs at the point where each file has *splitpoint* number of lines. The -f option can be used to extend the size of a file beyond *splitpoint* lines.

-p Write information about the progress of the file division to the diagnostic file.

Files List contents of files and directories

Files *(optionList) (nameList)*

Names and, if requested, other information about files and directories are written to standard output. Omitting the *nameList* parameter causes the current directory to be listed.

Options

-c *creator* Write only files with the specified *creator* field to output.

-d List only subdirectories.

-f List full pathnames.

-i Treat directories as files.

-l The output information includes name, type, creator, size, attributes, modification date, and creation date. The attribute information is given by the case of the attribute's first letter (except for inVisible, which is represented by V). Uppercase represents the value 1 and lowercase represents the value 0 for the following: Inited, (in)Visible, Bundle, System, Protected, Open, Changed, Locked, and Desktop.

-m *column* Write output in multicolumn format.

-n Do not output headers in long or extended format.

-q Disable the quoting of special characters in displayed file names.

-r Write subdirectories recursively.

-s Do not write directory names to output.

-t *type* Write only files with the specified *type* field to output.

-x *format* Write output in extended format, where *format* is a string of one or more of the following characters: a (flag attributes), b (data fork byte size), c (file creator), d (creation date), g (group), k (both forks kilobyte size), m (modification date), o (owner), p (privileges), r (resource fork byte size), or t (file type).

Find Find and select specified text

```
Find (-c count) selection (window)
```

The *selection* parameter is located and made the current selection. The search is made in the target window unless otherwise specified by the window *parameter.*

Option

-c *count* The command finds the *count* occurrence of the *selection.*

Font Set font and font size

```
Font fontname fontsize (window)
```

Replaces the current font settings with the parameter font and size. The change is made in the target window unless otherwise specified by the *window* parameter.

For Execute command list for each parameter

```
For name In wordList ; commandList ; End
```

Performs a set of commands for each parameter. The loop performs the set of commands for each word in *wordList,* assigning the current word to the variable *name* for each repetition. A word list must begin with the word In. The For structure must end with the word End. The command list must be bounded by semicolons or Return characters.

Help Display information in help file

Help (-f *helpFile) (commandList)*

Writes information about the specified commands to standard output. Omitting the *commandList* parameter produces information on all help file commands.

Option

-f *helpFile* Seek information in the specified file instead of the default file, MPW.Help.

If Execute command if true expression

If *expression* ; *commandList* ; (Else If *expression* ; *commandList)* ; (Else ; *commandList)* ; End

Performs a single set of commands upon evaluating an *expression* as true. Multiple Else If conditions are permitted as well as a final catchall Else condition. The If structure must end with End. The command list must be bounded by semicolons or Return characters.

Lib Create library of object files

Lib *(optionList) objFileList*

Joins two or more object files into a single library file. Unless an option directs otherwise, a library output file is created with the name Lib.Out.o, representing a concatenation of the .o input files.

Options

-b A big Lib occurs so that larger input files do not cause a heap error.

-bf A big file Lib occurs so that numerous files do not cause a file number error message.

-bs *nn* A big size Lib occurs so that large files do not cause a memory error message. The parameter *nn* represents blocks of the buffer that can range in number from 2 to 64. The default buffer size is 16 blocks.

-d Do not write to output warning messages about data and code names that appear in more than one file.

-df *deleteFile* Delete external modules named in the text file *deleteFile.* The Linker's -uf option creates *deleteFile.*

-dm *name (,nameList)* Delete from the library file each external module listed. Listing an entry name deletes its entire module.

-dn *name (,nameList)* Delete from the library file each external *name* listed such that the scope of the name has only local scope.

-mn *oldName = newName* Substitute a name for a module or entry point.

-o *name*.o The library output file takes the name *name*.o.

-p Write information about the progress of the command to the diagnostic file.

-sg *newSeg = oldSeg (,oldSeg2List)* Name a segment of code *newSeg* instead of the specified old segment names.

-sn *oldSeg = newSeg* Name a segment of code *newSeg* instead of *oldSeg.*

-w Do not write warning messages to output.

Link Link object files

Link *(optionList) objFileList*

Performs a link of object files to create an application, desk accessory, or driver. Unless an option directs otherwise, a linked output file is created with the name Link.Out, representing the link of the .o input files. The linked segments become CODE resources of the output file.

Options

-b A big link occurs, implementing both the bf and -bs 4 options.

-bf A big file link occurs so that numerous files do not cause a file number error message.

-bs *blocks* A big size link occurs so that large files do not cause a memory error message. The parameter *blocks* represents blocks of the buffer that can range in number from 2 to 64. The default buffer size is 16 blocks.

-c *creator* Change the creator from the default value ???? to the specified *creator.*

-d Do not write to output warning messages about data and code names that appear in more than one file.

-da Output segment names take on desk accessory names.

-l Include a location map in the linked file.

-la Include anonymous symbols in the location map.

-lf Include symbol definition information in the location map.

-m *mainEntry* Set the specified module or entry point name as the main entry point.

-ma *name=alias* Allows a module or entry point name to be substituted with the name *alias.*

-o *outputFile* The linked output file takes the name *outputFile.*

-opt Optimize for Object Pascal.

-p Write information about the progress of the command to the diagnostic file.

-ra *(seg)=nn* Give a segment's resource flags the value *nn.* Omitting the name *seg* assigns all segments other than 0, 1, and 2 to the value *nn.*

-rn Resource names do not take on the name of the segment.

-rt *type=ID* Change a resource type from the default of CODE to the specifed *type,* and change the starting ID from the default of 0 to *ID.*

-sg *newSeg=oldSeg (,oldSeg2List)* Name a segment of code *newSeg* instead of the specified old segment name.

-sn *oldSeg=newSeg* Name a segment of code *newSeg* instead of *oldSeg.*

-ss *size* Increase the maximum segment size from the default size of 32,760 bytes to the value of *size.*

-t *type* Change the type from the default value APPL to the specified *type.*

-uf *deleteFile* Identify in the text file the *deleteFile* modules and entry points that are not used. The -df option of the Lib command can use this file as input.

-w Do not write warning messages to output.

-x *crossRefFile* Write a cross-reference listing of the link to output.

Loop Execute command list until Break

```
Loop ; commandList ; End
```

Performs the specified commands over and over until a Break command exits the structure. The Loop structure must end with End. The command list must be bounded by semicolons or Return characters.

Make Make new program version

Make *(optionList) (destinationFileList)*

Reads a file that specifies dependency rules of program construction, then writes to standard output the series of Shell commands that, when executed, create a new version of the destination file(s). Unless an option directs otherwise, the rules of program execution are sought from a file named MakeFile. Omitting *destinationFileList* causes the first destination file listed in MakeFile to be created.

Options

-d *name(=value)* Define variable names and their values.

-e Recreate destination files even if they do not need revision.

-f *makefile* Read the rules of program construction from the parameter *makefile* instead of the default file named MakeFile.

-p Write information about the progress of the command to the diagnostic file.

-r Write the dependency graph roots to standard output.

-s Write a graphic representation of the dependency rules of program construction to standard output.

-t Update the dates of affected files, without otherwise recreating the files.

-u Write the names of any unfound targets to the diagnostic file.

-v Write additional information about targets to the diagnostic file.

-w Do not write warning messages to output.

Mark Mark a selection of text

Mark (-n | -y) *selection name window*

Provides a marker name to a *selection* of text in the given *window*. The marker *name* appears as a menu item in the Mark menu when the window is active.

Options

-n An old marker does not replace a new marker of the same name, circumventing the confirmation dialog box.

-y An old marker replaces a new marker of the same name, circumventing the confirmation dialog box.

Markers Display text selection markers

Markers *window*

Writes to standard output the names of all markers that have been assigned to the parameter *window.*

MDSCvt Convert assembler source, MDS to MPW

MDSCvt *(optionList) (fileList)*

Changes selected items in the assembly source code of files written for the Macintosh Development System so that the code can be compiled by the Macintosh Workshop Assembler. The output file name is the input name with the .a suffix. Omitting *fileList* causes standard input to be converted and written to standard output.

Options

-d The input file uses spaces in place of all tabs.

-e The input file puts spaces in place of all tabs, then reinserts tabs in the output file according to the tab setting. Tabs can be set using the -t option.

-f *directivesFile* Search the parameter file for uppercase and lowercase information. By default, directives are output in uppercase as specified in the file MDSCvt.Directives.

-g *globals* The main program conversion reserves *globals* space below the A5 pointer, where *globals* is a negative decimal or hexadecimal value.

-i Modify for conversion of an Include file.

-m The output does not contain Lisa Assembler compatibility directives.

-main The conversion emulates main program code.

-n Do not append the .a suffix to the output file name.

-p Write information about the progress of the command to standard output.

-pre(fix) *string* The output file name has the specified *string* attached in front of the input file name.

-suf(fix) *string* The output file name has the specified *string,* instead of the default .a suffix, appended to the input file name.

-t *value* The Shell tab setting takes a *value* in the range of 3 to 255. By default, the tab setting is 8.

-u c Resolve code name conflicts with MPW directives by appending the character c to the code name. By default, the character # is appended.

-! *identifier* The main program's entry point is specified by *identifier.*

Mount Mount disk volumes

```
Mount driveList
```

Puts the volumes in the specified disk drives on-line so the Shell can locate their contents.

Move Move contents of files and directories

```
Move (-c | -n | -y) (-p) nameList destination
```

The contents of the file or directory specified by *nameList* are placed in *destination.* If *destination* is a file, *nameList* replaces it. If a file or directory within *destination* already exists, a dialog box asks confirmation to overwrite objects of the same name.

Options

-c A same-name object conflict cancels the command, circumventing a confirmation dialog box.

-n Same-name objects are not overwritten, circumventing a confirmation dialog box.

-p Write information on the progress of the command to the diagnostic file.

-y Same-name objects are overwritten directly, circumventing a confirmation dialog box.

MoveWindow Move window to screen coordinates

```
MoveWindow h v (window)
```

Positions the top-left corner of the window to the global coordinates *h v*, where 0 0 is the top left of the Macintosh screen. Omitting the *window* parameter causes the target window (second from the top) to be used.

New Open new window and make active

```
New (nameList)
```

Creates a new window with the specified name or, if no name is given, the Shell assigns a number appended to the name *Untitled-*.

Newer Display newer file names

```
Newer (-c) (-e) (-q) nameList target
```

Write to standard output the names of the specified parameter files whose modification dates are more recent than the *target* file.

Options

-c Output depends on creation dates instead of modification dates.
-e Write files with the same modification date to output.
-q Write pathnames unquoted.

NewFolder Open a new empty folder

```
NewFolder nameList
```

Creates new subfolders of the current folder and assigns them the specified names. The command works only on HFS disks.

Open Open existing window and make active

```
Open (-n | -r) (-t) (nameList)
```

Makes the specified window active and, therefore, topmost on the desktop. If a name is not given, the n option must be used.

Options

-n The new window takes a name assigned by the Shell.

-r Open the specified window for read-only access.

-t Open the specified window as the target window instead of the active window.

Parameters Display parameters

```
Parameters (parametersList)
```

Lists the specified parameters in standard output, beginning with (0), the name *Parameters* itself.

Pascal Compile Pascal source code

```
Pascal (optionList) (fileList)
```

Compiles Pascal source code of the specified file according to the option settings. Source code files must end in the suffix .p (i.e., *fileName*.p). Completion of the compilation produces an object code file that ends with the suffix .p.o (i.e., *fileName*.p.o). More than one file can be specified for compilation—each file is compiled separately. Omitting the *fileList* parameter compiles standard input and creates the object file p.o.

Options

-align Align data items along long word boundaries.

-b A5-relative references replace PC-relative references at procedure and function addresses.

-c Check the source code for syntax errors without producing an object file.

-d *name*=TRUE | FALSE The variable *name,* which represents compile time, takes a true or false value.

-e *errLogFile* Write errors to *errLogFile* in addition to the diagnostic file.

-h Do not write unsafe handle error messages to output.

-i *pathname (,pathname)* Search for Include and Uses file names in the specified directories.

-k *prefixpath* Include $LOAD files in the directory specified by the parameter.

-mc68020 Optimize code for the 68020 processor.

-mc68881 Optimize code for the 68881 coprocessor.

-o *objname* Modify the pathname for the generated object file. A colon following *objname* provides a directory pathname for the output file.

-ov Report overflows.

-p Write information about the progress of the command to the diagnostic file.

-r Do not report range errors.

-t Write compilation time to the diagnostic file.

-u Initialize local and global data to the value $7267.

-w Do not use the peephole optimizer.

-y *pathname* Store the intermediate files generated by the compiler in the specified directory.

-z Do not include embedded procedure names in the object code.

PasMat Format Pascal programs

PasMat *(optionList) (inputfile (outputfile))*

Displays Pascal source code according to the options selected. Omitting the input and output file parameters causes standard input and output, respectively, to be used. Most of the options are used to override the initial default settings of input file directives. The default settings can also be overridden by including the option's directive equivalent, shown in parentheses, in the input file.

Options

-a Turn off CASE label bunching (a-).

-b Turn on IF bunching (b+).

-body Align procedure blocks with their BEGIN and END brackets (body+).

-c BEGIN does not start a new line (c+).

-d { } replace (* *) as comment delimiters (d+).

-e Display identifiers in capital letters (e+).

-entab Convert consecutive spaces to tabs as set by the tab stop value or tab directive.

-f Turn off formatting (f-).

-g Group assignment and call statements (g+).

-h Turn off FOR, WHILE, and WITH bunching (h-).

-i *pathname (,pathnameList)* Search for Include files in the directory *pathname.*

-in Process Include files (in+).

-k Indent statements between BEGIN and END brackets (k+).

-l Copy reserved words and identifiers literally (l+).

-list *listingFile* Write a formatted source listing to *listingFile.*

-n Group formal parameters (n+).

-o *width* Output line *width* ranges up to 150 characters (default = 80).

-p Write information about the progress of the command to the diagnostic file.

-pattern =pattern=replacement= Include files are updated with new pathnames and remain structurally consistent with the input file. The sequence of characters specified by *pattern* are replaced by the string *replacement.*

-q The ELSE IF structure indents IF on the next line after ELSE (q+).

-r Show reserved words in uppercase (r+).

-rec Indent a RECORD's file list under its identifier.

-s *renameFile* Rename identifiers specified in the first column of *renameFile* to the contents of the file's second column. The old and new identifier names share the same line of *renameFile,* separated by spaces or a tab.

-t *tab* The tabs of each indentation are *tab* spaces in length (default = 2).

-u Rename identifiers according to how they first appeared in the source code.

-v THEN is written on a new line (v+).

-w Display identifiers in uppercase (w+).

-x Display operators without surrounding spaces (x+).

-y Display := without surrounding spaces (y+).

-z Display commas without a subsequent space (z+).

-: Align colons in VAR declarations (:+).

-@ Write CASE tags on new lines (@+).

"-#" Group assignment and call statements in input and output (#+). The option's quotation marks are required.

-_ Remove underscore characters from identifiers (_+).

PasRef Cross-reference Pascal source code

PasRef *(optionList) (fileList)*

Writes to standard output a reference listing for the variables in the specified source code files. Each alphabetic entry contains the line numbers in the source where the variable occurs.

Options

-a Reference files and units each time they appear.

-c Do not reference a used unit if its file name is part of *fileList*.

-d Reference each specified file individually and separately.

-i *pathname (,pathnameList)* Search for Include and Uses file names in the specified directories.

-l Write identifiers in lowercase letters.

-ni | -noincludes Do not reference Include files.

-nl | -nolisting Do not write the input source during referencing.

-nolex Do not write lexical data to output.

-nt | -nototal Do not write the line count totals to output.

-nu | -nouses Do not reference Uses declarations.

-o Permit Object Pascal code.

-p Write information about the progress of the command to the diagnostic file.

-s Do not write Uses and Include file data to output.

-t Reference by total source line number.

-u Write identifiers in uppercase letters.

-w *width* The maximum *width* of a listing is a value of 40 to 255 (default = 110).

-x *width* The maximum *width* of an identifier is a value of 8 to 63 (default = size of largest identifier).

Paste Cut text and paste contents of Clipboard

Paste (-c *count) selection (window)*

A *selection* of lines is replaced by the contents of the Clipboard. Adjustment is made to the target window unless otherwise specified by the *window* parameter.

Option

-c *count* Perform the command *count* times.

PerformReport Create report on performance

PerformReport *(optionList)*

Writes to standard output the relation of performance data and procedure names taken from the link map and performance data files.

Options

-a List all procedures.

-l *filename* The *filename* is the input link map file.

-m *filename* The *filename* is the input performance data file instead of perform.out, the default file.

-n *NN* Display the top *NN* procedures instead of the default number of 50.

-p Write information about the progress of the command to the diagnostic file.

Print Print contents of text files

Print *(optionList) (fileList)*

Sends the contents of the specified files to the on-line printer. Omitting *fileList* causes standard input to be printed.

Options

-b A round-rect border encompasses the printed page.

-b2 A -b option border with the header on top of and outside of the border that encompasses the printed page.

-c(opies) *n* Print *n* copies of the text files.

-f(ont) *name* Use the specified font. Initial font is Monaco 9.

-ff *string* A leading string acts like a formfeed character.

-from *n* Begin printing at page *n* (default = 1).

-h Print the name of the file, the time of printing, and the page number as the page header for each page.

-hf(ont) *name* Use the specified header font (default = file font).

-hs(ize) *n* Use the specified header font size (default = 10).

-l(ines) *n* A linefeed spacing adjustment is made so that, font size permitting, *n* lines fill a full page.

-ls *n* Lines are spaced according to *n* (default = 1, single space).

-md The modification date appears in the header.

-n Print line numbers along the left side of the text.

-nw *n* The field width of the line numbers is *n* characters (default = 5).

-p Write information about the progress of the command to diagnostic output.

-page *n* Begin the numbering of pages with *n*. The default is 1.

-q *quality* The ImageWriter prints at the given resolution, where *quality* is specified as high, standard, or draft.

-r Print pages from last to first.

-s(ize) *n* Use the specified font size (default = font size in resource fork or, if unavailable, 9).

-t(abs) *n* Use the specified tab setting (default = tab setting in resource fork or, if unavailable, value of Tab variable).

-title *name* Page headers use *name* as the title (default = *filename*).

-to *n* Print to end at page *n* (default = file's last page).

-tm *n* ; -bm *n* ; -lm *n* ; -rm *n* The top, bottom, left, and right page margins, respectively, are set to *n*, where *n* is the margin width in inches. The default value for all margins is 0, except for the left margin, which has a default value of 0.2778.

ProcNames Display Pascal block names

ProcNames *(optionList) (fileList)*

Lists in an indented format all the procedure and function names of the input Pascal program or unit.

Options

-c Ignore a unit if the unit's $U interface file name is already listed.

-d Each new file resets its line number count to 1.

-e No formfeeds after each block listing.

-f Output is compatible with the PasMat tool.

-i *pathname (, pathnameList)* Search for Include or Uses files in the specified directories.

-n Do not output line number and level information.

-o Treat the source file as an Object Pascal program.

-p Write information about the progress of the command to the diagnostic file.

-u Include Uses declarations as input.

Quit Quit the MPW Shell

Quit (-c | -n | -y)

Exits the MPW Shell.

Options

-c Causes a cancel response to any confirmation dialog box.

-n Causes a no response to any confirmation dialog box.

-y Causes a yes response to any confirmation dialog box.

Quote Display parameters in quotation marks

```
Quote (-n) (parameterList)
```

Writes parameters to standard output in the same manner as the Echo command, except characters treated as special to the Shell are written in single quotation marks.

Option

-n Do not follow the last parameter with a Return.

Rename Rename disk files and directories

```
Rename (-c | -n | -y) name newname
```

Changes the name of a file or directory from *name* to *newname.* If a file or directory using *newname* already exists, a dialog box asks confirmation to overwrite same-name objects.

Options

-c A same-name object conflict halts the command, circumventing a confirmation dialog box.

-n Do not overwrite same-name objects, circumventing a confirmation dialog box.

-y Overwrite same-name objects directly, circumventing a confirmation dialog box.

Replace Find and replace text in window

```
Replace (-c count) selection replacement (window)
```

The *selection* parameter is located and deleted, then the *replacement* parameter is inserted at that location. The search is performed in the target window unless otherwise specified by the *window* parameter.

Option

-c *count* Repeat the command *count* times. Using the symbol 0 as the value of *count* repeats the command for all occurrences in the direction searched.

Request Display a request dialog box

```
Request (-d default) (message)
```

Creates a dialog box that contains the text of *message* and permits the user to type in a response. The response is written to standard output when the user selects the OK button. A Cancel button is also offered. Omitting the *message* parameter causes standard input to be read.

Option

-d The initial message *default* appears in the dialog, allowing the user to accept or edit it.

ResEqual Display comparison of resource files

```
ResEqual (optionList) file1 file2
```

The differences between two specified resource files are written to standard output.

Option

-p Write information about the progress of the command to the diagnostic file.

Revert Revert to file as last saved

```
Revert (-y) (windowList)
```

Returns the parameter windows to their state when last saved. If no *windowList* parameter is given, the target window (second from top) is used by default.

Option

-y The command circumvents the confirmation dialog box.

Rez Compile resources

Rez *(optionList) (resourceDescriptionFileList)*

Translates a text description of a resource into compiled code. Text description input is obtained from *resourceDescriptionFileList* or, if no file is given, standard input. The format for input is the same as the DeRez (resource decompiler) program's output. Unless an option directs otherwise, the compiled resource output is written to the file Rez.Out. The parameter *resourceDescriptionFileList* specifies a file of formatted type declarations. The file Types.r contains the common Macintosh resource declarations. The file SysTypes.r contains the system resource declarations.

Options

-align word Align resources along word boundaries.

-align long word Align resources along long word boundaries.

-a(ppend) Append the command's output to the output file.

-c(reator) *creatorExpr* The output file has the creator value as given by the expression. By default, the creator is '????'.

-d(efine) *macro(=data)* Equate the specified *macro* variable with the value *data* or, if *data* is omitted, the null string.

-e(scape) Print escape characters as extended Macintosh characters.

-i *pathname(s)* Search the specified *pathname(s)* for #include files.

-o *outputFile* Give the output file the specified name rather than the default name Rez.Out.

-ov Ignore the protected bit when appending resource output.

-p(rogress) Write information about the progress of the command to the diagnostic file.

-rd Redeclared resource types do not write a warning message to output.

-ro The resource map implements mapReadOnly.

-s *pathname(s)* Search the specified *pathname(s)* for resource Include files.

-t(ype) *typeExpr* The output file takes the type value as given by the expression. By default, the type is 'APPL'.

-u(ndef) *macro* Undefine a *macro* variable.

RezDet Detect resource problems

```
RezDet (-b) (-d | -l | -q | -r | -s) resourceFileList
```

Resources within each file of *resourceFileList* are inspected for problems. Information about any problems is written to standard output. Other than -b(ig), only one option may be implemented at a time.

Options

-b(ig) Read resources into memory individually to avoid out of memory errors.

-d(ump) Write verbose individual resource information to standard output.

-l(ist) Write the resource information to standard output in the format: *'type' (ID, name, attributes) [size].*

-q(uiet) Do not write error information to standard output.

-r(awdump) Write data block contents (in addition to verbose individual resource information) to standard output.

-s(how) Write individual resource information to standard output.

Save Save window onto disk

```
Save (-a | windowList)
```

Puts a copy of the specified window file onto disk. Omitting all parameters causes the target window to be saved.

Option

-a Save all open windows.

Search Search for text in files

```
Search (-f file) (-i | -s) (-l) (-q) (-r) /pattern/ (fileList)
```

Looks for a sequence of characters within the lines of a file, then writes each line that contains the sequence to standard output. Omitting *fileList* causes standard input to be read. Slashes (/ and /) enclose the *pattern* to be sought.

Options

-f *file* All lines not written to standard output are written to the specified *file*.

-i The search is not case sensitive.

-q Write only matching lines to output, without the file name and the line number.

-r Write lines without matches to output.

-s The search is case sensitive.

Select Select items from a dialog box

Select *(optionList) (itemList)*

Uses the parameter items to create a dialog box, then waits for the user to select an item and click the OK button before writing the selected item name to standard output. Omitting the *itemList* parameter causes standard input to be read.

Options

-d *item* Insert *item* in the list so that it is preselected.

-m *message* Display *message* on top of the item list.

-q Write items in the list unquoted.

-r *rows* Write the list with the specified number of *rows* if possible.

-w *width* Write the list with the specified number of *width* pixels if possible.

Set Define or display variable names

Set *(name (value))*

Equates the variable *name* with the string *value*. Omitting *value* causes the variable and its current value to be written to standard output. Omitting both parameters causes all variables and their current values to be written to standard output.

SetDirectory Set the default directory and add to menu

SetDirectory *directory*

Sets the parameter *directory* as the current default directory and, if new, inserts the *directory* as an item in the Directory menu. The DirectoryMenu command displays the Directory menu.

SetFile Set attributes of files

SetFile *(optionList) fileList*

Provides attribute values for all the specified files according to the chosen options.

Options

-a *attributes* Causes settings for the following flags: Inited, (in)Visible, Bundle, System, Protected, Open, Changed, Locked, and Desktop. Each flag is represented by a single character (the first letter of the flag, except inVisible which is represented by V). If the parameter character in *attributes* is uppercase, the flag is set to 1; if lowercase, the flag is set to 0. Characters that are not included in *attributes* cause the flag to go unchanged.

-c *creator* Set a four-character creator.

-d *date* Set a string representing the creation date and time in the format mm/dd/yy (hh:mm (:ss) (AM:PM)). If *date* is specified as a period (.), the current date and time are used.

-l *h,v* Set the horizontal and vertical coordinates of the icon location, where point (0,0) is the top-left corner of the icon's window.

-m *date* Set a string representing the modification date and time in the same format as the d option.

-t *type* Set a four-character type.

SetPriv Set file server privileges

SetPriv (-c *prv*) (-d *prv*) (-f *prv*) (-g *group*) (-i) (-o *owner*) (-r) *folderList*

Sets file server access privileges for folders by using a three-character *prv* string (o-owner, g-group, e-everyone) in which an uppercase letter enables the privilege and a lowercase letter disables it. Omitting the character causes no change in privilege.

Options

-c *prv* Set privileges for changing files and folders.

-d *prv* Set privileges for seeing folder and folder listings.

-f *prv* Set privileges for seeing files within folders.

-g *newGroup* Set the parameter as the group.

-i Write access privilege information to standard output.

-o *newOwner* Set the parameter as the owner.

-r Folders are affected recursively.

SetVersion Set version and revision number

SetVersion *(optionList) file*

Sets an application's version and revision number in a resource or source code.

Options

-csource *file* Set the string constant in the specified C source code file.

-d Write the latest version and revision number in the 'MPST' resource string to the diagnostic file.

-fmt nf.mf Version and revision numbers use the specified format, where *f* is the letter D (for leading blanks) or Z (for leading zeros), and *n* and *m* are field width integers from 1 to 10.

-i *resid* Set 'MPST' resource ID.

-p Write the version number and 'MPST' contents to the diagnostic file.

-prefix *prefix* Set the version *prefix* string.

-psource *file* Set the string constant in the specified Pascal source code file.

-r Increase the revision number by 1.

-rezsource *file* Set the 'MPST' resource definition in the specified resource source code file.

-sr *revision* Set the revision number to the parameter value.

-suffix *suffix* Set the revision suffix string.

-sv *version* Set the version number to the parameter value.

-t *type* Use the resource of the parameter *type* in place of 'MPST'.

-v Increase the version number by 1.

-verid *identifier* Use the *indentifier* parameter in searches for Pascal and C constant searches.

Shift Shift number of positional parameters

```
Shift (number)
```

The numbers of the Shell positional parameters, except parameter 0, are reset by reducing their current value by *number.* The default value of *number* is 1.

Shutdown Quit with shutdown or restart

```
Shutdown (-c | -n | -y) (-r)
```

Exits the MPW Shell and causes the Macintosh to shutdown or, if the -r option is used, restart.

Options

-c Causes a cancel response to any confirmation dialog box.

-n Causes a no response to any confirmation dialog box.

-r Restart the Macintosh.

-y Causes a yes response to any confirmation dialog box.

SizeWindow Set the window size in pixels

```
SizeWindow h v (window)
```

Sets the horizontal and vertical pixel size of the window. Omitting the *window* parameter causes the target window (second from the top) to be used.

StackWindows Set windows to stack diagonally

```
StackWindows
```

Resizes and moves all open Shell windows to diagonally stacked positions on the screen.

StdFile Select from standard file dialog box

```
StdFile (-b buttonTitle) (-d | -p | -t typeList) (-m message) (-q) (pathname)
```

Creates a standard file dialog box, then waits for the user to select an item and click the OK button before writing the selected file name or pathname to standard output. Supplying a *pathname* parameter causes the standard file starting directory to be set.

Options

-b *buttonTitle* Set the title of the dialog box button.

-d Display an SFGetFile dialog box.

-m *message* Display *message* on top of the file list.

-p Display an SFPutfile dialog box.

-q Write files in the list unquoted.

-t *typeList* Display the SFGetFile dialog box limited to files of up to four types.

SysErr Display system error messages

```
SysErr (-f filename | -s filename) (-n) (-p) errNum (,insertList) (-i idNumList)
```

Displays the system error message for each error number parameter. The default system error message file is SysErrs.Err. When used with the -i option, displays information relating to the specified ID number in System Error Handler alert dialogs. Tool error numbers may also be specified with inserts.

Options

-f *filename* Output error messages for the specified tool error file.

-i *idNumList* Output information relating to the System Error Handler ID number.

-n Do not output error numbers with the messages.

-p Write version information about the command to the diagnostic file.

-s *filename* Output error messages for the specified system error file.

Tab Set tab positions of windows

```
Tab number (window)
```

Establishes a tab setting of *number.* Adjustment is made to the target window unless otherwise specified by the *window* parameter.

Target Set window as the target window

```
Target name
```

The file *name* is opened (if not already open) and set as the default target window file. The target window is the second window from the top.

TileWindows Set windows to tile position

```
TileWindows
```

Resizes and moves all open Shell windows to visible tile positions on the screen.

TLAConvert Convert assembler source, TLA to MPW

TLACvt *(optionList) (fileList)*

Changes selected items in the assembly source code of files written for the Lisa Assembler so that the code can be compiled by the Macintosh Workshop Assembler. The output file name is the input name with the suffix .a appended. Omitting *fileList* causes standard input to be converted and written to standard output.

Options

-d The input file uses spaces in place of all tabs.

-e The input file puts spaces in place of all tabs, then reinserts tabs in the output file according to the tab setting. Tabs can be set using the -t option.

-f *directivesFile* Search the parameter file for uppercase and lowercase information. By default, directives are output in uppercase as specified in the file TLACvt.Directives.

-m The output does not contain Lisa Assembler compatibility directives.

-n Do not append the .a suffix to the output file name.

-p Write information about the progress of the command to standard output.

-pre(fix) *string* The output file name is the specified string attached in front of the input file name.

-suf(fix) *string* The output file name is the specified string appended to the input file name instead of the default .a suffix.

-t *tabSetting* The Shell tab setting is a value in the range of 0 to 255 (default = input file tab setting or, if unavailable, 8).

-u c Resolve code name conflicts with MPW directives by appending the character c to the code name (default = #).

Translate Translate character strings

Translate (-p) (-s) *src (dst)*

Converts character strings in the *src* parameter to those in the *dst* parameter, then writes to standard output. Omitting the *dst* parameter causes *src* characters to be deleted.

Options

-p Write version information about the command to the diagnostic file.

-s Retain font, font size, and tab settings from the *src* file.

Unalias Make aliases undefined

```
Unalias (nameList)
```

Disassociates a command name and its substitute word(s) that were previously defined using the Alias command. Only the current command file is affected. Omitting *nameList* removes all aliases.

Undo Undo window's previous edit command

```
Undo (window)
```

Reverses the action of the previous editing command that occurred in the specified window. Omitting the *window* parameter causes the target window (second from the top) to be used.

Unexport Make variable definition unexported

```
Unexport (-r | -s | nameList)
```

Deletes the parameter variables from a command file's local list of exported variables. Omitting all parameters causes a list of all variables not exported to be written to standard output.

Options

-r Output Export commands for each variable not exported.

-s Do not write the word *Unexport* to output.

Unmark Delete file marker

```
Unmark nameList window
```

Deletes a marker name that was assigned to a selection of text in the given window. The marker name no longer appears as a menu item in the window's Mark menu.

Unmount Unmount disk volumes

```
Unmount volumeList
```

Puts the specified volumes off-line to make their contents unavailable to the Shell. A colon (:) must be appended to the volume name unless the volume specified is a disk drive number.

Unset Make variable names undefined

```
Unset (nameList)
```

Disassociates a variable name and its definition that were previously equated using the Set command. Only the current command file is affected. Omitting *nameList* removes all variable definitions.

Volumes Display names of mounted disk volumes

```
Volume (-l) (-q) (volumeList)
```

Writes the name and, if requested, other information about the specified volumes to standard output. A colon (:) must be appended to the volume name unless the volume specified is a disk drive number.

Options

-l Write additional information about a volume's location, size, and contents to output.

-q Disable the quoting of special characters in displayed file names.

Which Find which pathname executes command

```
Which (-a) (-p) (command)
```

Searches for, and writes to standard output, the full pathname of a command, and allows the command to be executed by name alone (instead of requiring the full pathname). Omitting the *command* parameter outputs all pathnames of the startup {Commands} variable.

Options

-a Output all pathnames to the parameter.

-p Output information about the progress of the command.

Windows Display names of open windows

```
Windows (-q)
```

Writes the full pathname of each current window file to standard output.

Option

-q Disable the quoting of special characters in displayed file names.

ZoomWindow Display window zoomed or back

```
ZoomWindow (-s) (window)
```

Resizes *window* to the full size of the Macintosh screen. Omitting the *window* parameter causes the target window (second from the top) to be used.

Option

-s The window zooms back to its original size.

CHAPTER

25

The 68000 Instruction Set with Directives and Toolbox Traps

ABCD Add Decimal with Extend

Add the source subject plus the extend bit to the destination subject, and place the result in the destination slot. The subjects are added using binary coded decimal (BCD) arithmetic.

Size: Byte

ABCD D1,D2	Add data register 1 to data register 2.
ABCD -(A1),-(A2)	Add memory slot 1 to memory slot 2 using predecrement addressing.

Status flags

C	Set by a decimal carry; cleared otherwise.
V	Undefined.
Z	Cleared by a nonzero result; unchanged otherwise.
N	Undefined.
X	Set by a decimal carry; cleared otherwise.

ADD Add Binary

Add the source subject to the destination subject, and place the result in the destination slot.

Size: Byte, Word, Long

ADD *ea*,D1	Add effective address to data register.
ADD D1,*ea*	Add data register to effective address.

Status flags

C	Set by a carry; cleared otherwise.
V	Set by an overflow; cleared otherwise.
Z	Set by a zero result; cleared otherwise.
N	Set by a negative result; cleared otherwise.
X	Set by a carry; cleared otherwise.

ADDA Add Address

Add the source subject to the destination address register, and place the result in the address register.
Size: Word, Long

ADD *ea*,A1	Add effective address to address register.

Status flags

Unaffected.

ADDI Add Immediate

Add the immediate data to the destination subject, and place the result in the destination slot.
Size: Byte, Word, Long

ADDI *#data,ea*	Add immediate data to effective address.

Status flags

C	Set by a carry; cleared otherwise.
V	Set by an overflow; cleared otherwise.
Z	Set by a zero result; cleared otherwise.
N	Set by a negative result; cleared otherwise.
X	Set by a carry; cleared otherwise.

ADDQ Add Quick

Add the immediate data to the destination subject, and place the result in the destination slot. The immediate data must be an integer from 1 to 8.
Size: Byte, Word, Long

`ADDQ #data,ea`	Add immediate data (1-8) to effective address.

Status flags

C	Set by a carry; cleared otherwise.
V	Set by an overflow; cleared otherwise.
Z	Set by a zero result; cleared otherwise.
N	Set by a negative result; cleared otherwise.
X	Set by a carry; cleared otherwise.

ADDX Add with Extend

Add the source subject plus the extend bit to the destination subject, and place the result in the destination slot.
Size: Byte, Word, Long

`ADDX D1,D2`	Add data register 1 to data register 2.
`ADDX -(A1),-(A2)`	Add memory slot 1 to memory slot 2 using predecrement addressing.

Status flags

C	Set by a carry; cleared otherwise.
V	Set by an overflow; cleared otherwise.
Z	Cleared by a nonzero result; unchanged otherwise.
N	Set by a negative result; cleared otherwise.
X	Set by a carry; cleared otherwise.

AND AND Logical

AND the source subject to the destination subject, and place the result in the destination slot.
Size: Byte, Word, Long

AND *ea*,D1	AND effective address to data register.
AND D1,*ea*	AND data register to effective address.

Status flags

C	Cleared always.
V	Cleared always.
Z	Set by a zero result; cleared otherwise.
N	Set by a result whose most significant bit is set; cleared otherwise.
X	Unaffected.

ANDI AND Immediate

AND the immediate data to the destination subject, and place the result in the destination slot.
Size: Byte, Word, Long

ANDI *#data,ea*	AND immediate data to effective address.

Status flags

C	Cleared always.
V	Cleared always.
Z	Set by a zero result; cleared otherwise.
N	Set by a result whose most significant bit is set; cleared otherwise.
X	Unaffected.

ANDI to CCR AND Immediate to the Condition Code Register

AND the immediate source subject with the status flags, and place the result in the low-order byte of the status register (condition code register).
Size: Byte

ANDI *#xxx*,CCR	ANDI immediate subject to status flags.

Status flags

C	Cleared if source's bit 0 is zero; unchanged otherwise.
V	Cleared if source's bit 1 is zero; unchanged otherwise.
Z	Cleared if source's bit 2 is zero; unchanged otherwise.

N	Cleared if source's bit 3 is zero; unchanged otherwise.
X	Cleared if source's bit 4 is zero; unchanged otherwise.

ANDI to SR AND Immediate to the Status Register

AND the immediate source subject with the entire 16-bit status register, and place the result in the status register.
Size: Word

ANDI #*xxx*,SR	AND immediate subject to status register.

Status flags

C	Cleared if source's bit 0 is zero; unchanged otherwise.
V	Cleared if source's bit 1 is zero; unchanged otherwise.
Z	Cleared if source's bit 2 is zero; unchanged otherwise.
N	Cleared if source's bit 3 is zero; unchanged otherwise.
X	Cleared if source's bit 4 is zero; unchanged otherwise.

ASL Arithmetic Shift Left

Arithmetically shift to the left the bits of the destination subject by the specified amount. The last bit shifted out of the destination subject goes into both the carry and extend bits. Zeros replace the vacated bits.
Size: Byte, Word, Long

ASL D1,D2	Shift data register 2 by amount of data register 1.
ASL #*data*,D1	Shift data register 1 by immediate data.
ASL *ea*	Shift effective address by 1 bit only. Subject must be word size.

Status flags

C	Set the same as the last bit shifted out of the subject; cleared for a zero shift count.
V	Set by any change in the most significant bit during the shift; cleared otherwise.
Z	Set by a zero result; cleared otherwise.
N	Set by a result whose most significant bit is set; cleared otherwise.

X	Set the same as the last bit shifted out of the subject; unaffected for a zero shift count.

ASR Arithmetic Shift Right

Arithmetically shift to the right the bits of the destination subject by the specified amount. The last bit shifted out of the destination subject goes into both the carry and extend bits. The sign bit replaces the vacated bits.
Size: Byte, Word, Long

ASR D1,D2	Shift data register 2 by amount of data register 1.
ASR *#data*,D1	Shift data register 1 by immediate data.
ASR *ea*	Shift effective address by 1 bit only. Subject must be word size.

Status flags

C	Set the same as the last bit shifted out of the subject; cleared for a zero shift count.
V	Set by any change in the most significant bit during the shift; cleared otherwise.
Z	Set by a zero result; cleared otherwise.
N	Set by a result whose most significant bit is set; cleared otherwise.
X	Set the same as the last bit shifted out of the subject; unaffected for a zero shift count.

Bcc Branch Conditionally

Branch to the slot indicated by the subject if the specified condition (that substitutes for cc) is true. The subject determines the displacement from the original program counter to the slot where program execution resumes.
Size: Byte, Word

Bcc *reference*	Branch to *reference* if condition is true.

Status flags

Unaffected. Status flag conditions for each branch instruction:

BCC	Carry Clear. Branch if C is clear.
BCS	Carry Set. Branch if C is set.

BEQ	Equal. Branch if Z is set.
BGE	Greater or Equal. Branch if both N and V are set, or if both N and V are clear.
BGT	Greater Than. Branch if both N and V are set and Z is clear, or if N, V, and Z are all clear.
BHI	High. Branch if both C and Z are clear.
BLE	Less or Equal. Branch if Z is set, or if N is set and V is clear, or if N is clear and V is set.
BLS	Low or Same. Branch if C is set, or if Z is set.
BLT	Less Than. Branch if N is set and V is clear, or if N is clear and V is set.
BMI	Minus. Branch if N is set.
BNE	Not Equal. Branch if Z is clear.
BPL	Plus. Branch if N is clear.
BVC	Overflow Clear. Branch if V is clear.
BVS	Overflow Set. Branch if V is set.

BCHG Test a Bit and Change

Test a bit in the destination subject, place the result of the test in the Z status flag, then change the tested bit in the destination.
Size: Byte, Long

BCHG D1,*ea*	Test and change bit number D1 of destination *ea*.
BCHG #*data*,*ea*	Test and change bit number #*data* of destination *ea*.

Status flags

C	Unaffected.
V	Unaffected.
Z	Set by a zero value of the tested bit; cleared otherwise.
N	Unaffected.
X	Unaffected.

BCLR Test a Bit and Clear

Test a bit in the destination subject, place the result of the test in the Z condition code, then clear the tested bit in the destination.

Size: Byte, Long

`BCLR D1,`*ea*	Test and clear bit number D1 of destination *ea*.
`BCLR` *#data,ea*	Test and clear bit number *#data* of destination *ea*.

Status flags

C	Unaffected.
V	Unaffected.
Z	Set by a zero value of the tested bit; cleared otherwise.
N	Unaffected.
X	Unaffected.

BRA Branch Always

Branch to the slot indicated by the subject. The subject determines the displacement from the original program counter to the slot where program execution resumes.

Size: Byte, Word

`BRA` *reference*	Branch to *reference*.

Status flags

Unaffected.

BSET Test a Bit and Set

Test a bit in the destination subject, place the result of the test in the Z condition code, then set the tested bit in the destination.

Size: Byte, Long

`BCLR D1,`*ea*	Test and set bit number D1 of destination *ea*.
`BCLR` *#data,ea*	Test and set bit number *#data* of destination *ea*.

Status flags

C	Unaffected.
V	Unaffected.
Z	Set by a zero value of the tested bit; cleared otherwise.
N	Unaffected.
X	Unaffected.

BSR Branch to Subroutine

Branch to the subroutine slot indicated by the subject after leaving the long word return address (address of the instruction following the BSR instruction) on the stack. The subject determines the displacement from the original program counter to the subroutine slot where program execution continues.

Size: Byte, Word

BSR *reference*	Branch to subroutine *reference.*

Status flags

Unaffected.

BTST Test a Bit

Test a bit in the destination subject, and place the result of the test in the Z condition code.

Size: Byte, Long

BTST D1,*ea*	Test bit number D1 of destination *ea.*
BTST #*data,ea*	Test bit number #*data* of destination *ea.*

Status flags

C	Unaffected.
V	Unaffected.
Z	Set by a zero value of the tested bit; cleared otherwise.
N	Unaffected.
X	Unaffected.

_Button ROM Trap

```
FUNCTION Button : BOOLEAN;
```

_Button evaluates the status of the Macintosh's mouse button, and returns a boolean value of true if the mouse button is currently being held down. A value of false indicates the mouse button is up at the moment the _Button trap is executed. No parameters are used.

To prepare the stack for `_Button`: Subtract 2 bytes from the stack pointer to make space for the boolean result.

On return: The boolean result is left in the high-order byte. A nonzero value indicates true. A zero value indicates false.

CHK Check Register Against Bounds

Check the content of the low-order word in the data register subject. If the register value is less than zero or greater than the upper bound of the effective address subject, then execute a trap exception.

Size: Word

`CHK` *ea*`,D1`	Check the low-order word of data register with upper bound of effective address.

Status flags

C	Undefined.
V	Undefined.
Z	Undefined.
N	Set by D1 less than zero; cleared by D1 greater than *ea*; undefined otherwise.
X	Unaffected.

CLR Clear a Subject

Clear to zero all bits of the destination.

Size: Byte, Word, Long

`CLR` *ea*	Clear to zero all bits of effective address.

Status flags

C	Cleared always.
V	Cleared always.
Z	Set always.
N	Cleared always.
X	Unaffected.

CMP Compare

Subtract the source subject from the destination subject, and use the result to set the status flags. The destination slot is not changed.
Size: Byte, Word, Long

`CMP` *ea*,`D1`	Compare by subtracting effective address from data register.

Status flags

C	Set by a borrow; cleared otherwise.
V	Set by an overflow; cleared otherwise.
Z	Set by a zero result; cleared otherwise.
N	Set by a negative result; cleared otherwise.
X	Unaffected.

CMPA Compare Address

Subtract the source subject from the destination address register, and use the result to set the status flags. The address register is not changed.
Size: Word, Long

`CMPA` *ea*,`A1`	Compare by subtracting effective address from address register.

Status flags

C	Set by a borrow; cleared otherwise.
V	Set by an overflow; cleared otherwise.
Z	Set by a zero result; cleared otherwise.
N	Set by a negative result; cleared otherwise.
X	Unaffected.

CMPI Compare Immediate

Subtract the immediate data from the destination subject, and use the result to set the status flags. The destination slot is not changed.
Size: Byte, Word, Long

CMPI *#data,ea*	Compare by subtracting the immediate data from the effective address.

Status flags

C	Set by a borrow; cleared otherwise.
V	Set by an overflow; cleared otherwise.
Z	Set by a zero result; cleared otherwise.
N	Set by a negative result; cleared otherwise.
X	Unaffected.

CMPM Compare Memory

Subtract the source subject from the destination subject, and use the result to set the status flags. The destination slot is not changed. The subjects must be address registers using the postincrement addressing mode.
Size: Byte, Word, Long

CMPM (A1)+,(A2)+	Compare address register contents by subtracting source from destination.

Status flags

C	Set by a borrow; cleared otherwise.
V	Set by an overflow; cleared otherwise.
Z	Set by a zero result; cleared otherwise.
N	Set by a negative result; cleared otherwise.
X	Unaffected.

DBcc Test Condition, Decrement, and Branch

If the specified conditon (that substitutes for cc) is false, decrement the data register subject. Then, if the result of the decrement is not equal to −1, branch to the slot indicated by the *reference* subject. The *reference* determines the displacement from the original program counter to the slot where program execution continues.
Size: Word

DB*cc* D1,*reference*	If condition is false, then decrement data register. Then, if data register is not equal to −1, branch to *reference.*

Status flags

Unaffected. Status flag conditions for each branch instruction:

DBCC	Carry Clear. No branch if C is clear.
DBCS	Carry Set. No branch if C is set.
DBEQ	Equal. No branch if Z is set.
DBF	False. Branch always.
DBGE	Greater or Equal. No branch if both N and V are set, or if both N and V are clear.
DBGT	Greater Than. No branch if both N and V are set and Z is clear, or if N, V, and Z are all clear.
DBHI	High. No branch if both C and Z are clear.
DBLE	Less or Equal. No branch if Z is set, or if N is set and V is clear, or if N is clear and V is set.
DBLS	Low or Same. No branch if C is set, or if Z is set.
DBLT	Less Than. No branch if N is set and V is clear, or if N is clear and V is set.
DBMI	Minus. No branch if N is set.
DBNE	Not Equal. No branch if Z is clear.
DBPL	Plus. No branch if N is clear.
DBRA	Branch. Branch always.
DBT	True. Branch never.
DBVC	Overflow Clear. No branch if V is clear.
DBVS	Overflow set. No branch if V is set.

DC Define Constant (MPW Data Allocation Directive)

Place the specified *values* into memory slots. The suffixes .B, .W, and .L indicate the slot length—byte, word, and long word—of each data increment. References access the first aligned memory address of the defined area.

DC.B *values*	Define constant that is byte incremented.
DC *values*	Define constant that is word incremented.
DC.W *values*	Define constant that is word incremented.
DC.L *values*	Define constant that is long word incremented.

The *values* are defined within the current code or data module. One or more directives define their own data module if placed outside a program's existing modules. The directive MAIN signifies the main code module used in the example programs. Define constant directives can use values that are expressions and strings in any mixture, separated by commas. The following are examples of two directives:

```
RectBounds DC.W 20, 40, 200, 240 ;top, left, bottom, right
RectName   DC.B 'My Rectangle'    ;string name
```

DCB Define Constant Block (MPW Data Allocation Directive)

Initialize a block of memory slots within the current code or data module. The subjects specify the length of the block and the initial value to be placed in each slot. The suffixes .B, .W, and .L indicate slot length, incrementing data by byte, word, and long word sizes.

DCB.B	*length,values*	Define constant block that is byte incremented.
DCB	*length,values*	Define constant block that is word incremented.
DCB.W	*length,values*	Define constant block that is word incremented.
DCB.L	*length,values*	Define constant block that is long word incremented.

DIVS Signed Divide

Using signed arithmetic, divide the long word destination subject by the word source subject, and place the result in the destination slot. The quotient is placed in the low word of the destination. The remainder is placed in the high word of the destination, with the sign of a nonzero remainder the same as the dividend. Division by zero creates a trap. An overflow affects the status flag, but leaves the subjects unchanged.

Size: Word

DIVS *ea*,D1	Divide data register by effective address.

Status flags

C	Cleared always.
V	Set by attempted division overflow; cleared otherwise.
Z	Set by zero quotient; cleared by nonzero quotient; undefined by attempted overflow.

N Set by negative quotient; cleared by non-negative quotient; undefined by attempted overflow.

X Unaffected.

DIVU Unsigned Divide

Using unsigned arithmetic, divide the long word destination subject by the word source subject, and place the result in the destination slot. The quotient is placed in the low word of the destination. The remainder is placed in the high word of the destination. Division by zero creates a trap. An overflow affects the status flag, but leaves the subjects unchanged.

Size: Word

DIVU *ea*,D1	Divide data register by effective address.

Status flags

C Cleared always.

V Set by attempted division overflow; cleared otherwise.

Z Set by zero quotient; cleared by nonzero quotient; undefined by attempted overflow.

N Set by quotient whose most significant bit is set; cleared by quotient whose most significant bit is clear; undefined by attempted overflow.

X Unaffected.

_DrawMenuBar ROM Trap

```
PROCEDURE DrawMenuBar;
```

_DrawMenuBar uses the current contents of the menu list to draw a menu bar. The trap should be used to redraw the menu whenever the menu list has been changed by any prior sequence of calls that insert, delete, clear, set, or otherwise alter a menu list. No parameters are used.

_InsertMenu can be used to add a menu into the menu list at a specified position along the menu bar. _GetRMenu can be used to read a menu resource and provide the handle parameter for _InsertMenu. After such changes, _DrawMenuBar redraws the menu bar according to the current menu list.

_DrawString ROM Trap

```
PROCEDURE DrawString (s: Str255);
```

_DrawString places the parameter string into the current GrafPort to the right of the Quickdraw pen location. The variable *s* is of the predefined Quickdraw type str255, a string of not more than 255 characters.

The current pen location moves to the right of each character as the string is drawn. _DrawString performs no carriage returns, linefeeds, or text formatting.

To prepare the stack for _DrawString: Push a pointer to the string.

On return: The stack is clear.

DS Define Storage (MPW Data Allocation Directive)

Reserve the specified number of uninitialized memory slots at a position relative to the A5 pointer. The suffixes .B, .W, and .L indicate slot length—byte, word, and long word—of each data increment. References access the first aligned memory address of the defined area.

DS.B *length*	Define storage that is byte incremented.
DS *length*	Define storage that is word incremented.
DS.W *length*	Define storage that is word incremented.
DS.L *length*	Define storage that is long word incremented.

The storage area is allocated and defined within the current code module, data module, or template. One or more directives define their own data module if placed outside a program's existing modules. The directive MAIN signifies the main code module used in the example programs. The following are examples of two directives:

```
RectSpace   DS.L  2  ;space for rectangle coordinates
EventRecord DS.B  16 ;space for event record result
```

END End of Source (MPW Module Control Directive)

Indicates the end of a source code file. The assembler ignores any lines that follow the END directive. If END is omitted, the last line of code acts as the end of the source code.

EOR Exclusive OR Logical

Exclusive OR the source subject to the destination subject, and place the result in the destination slot. The source subject must be a data register.
Size: Byte, Word, Long

`EOR D1,ea`	Exclusive OR the data register to effective address.

Status flags

C	Cleared always.
V	Cleared always.
Z	Set by a zero result; cleared otherwise.
N	Set by a result whose most significant bit is set; cleared otherwise.
X	Unaffected.

EORI Exclusive OR Immediate

Exclusive OR the immediate data to the destination subject, and place the result in the destination slot.
Size: Byte, Word, Long

`EORI #data,ea`	Exclusive OR immediate data to effective address.

Status flags

C	Cleared always.
V	Cleared always.
Z	Set by a zero result; cleared otherwise.
N	Set by a result whose most significant bit is set; cleared otherwise.
X	Unaffected.

EORI to CCR Exclusive OR Immediate to the Condition Code Register

Exclusive OR the immediate source subject to the status flags, and place the result in the low-order byte of the status register (condition code register).
Size: Byte

`EORI #xxx,CCR`	Exclusive OR immediate subject to status flags.

Status flags

C	Changed if bit 0 of source is one; unchanged otherwise.
V	Changed if bit 1 of source is one; unchanged otherwise.
Z	Changed if bit 2 of source is one; unchanged otherwise.
N	Changed if bit 3 of source is one; unchanged otherwise.
X	Changed if bit 4 of source is one; unchanged otherwise.

EORI to SR Exclusive OR Immediate to the Status Register

Exclusive OR the immediate source subject to the entire 16-bit status register, and place the result in the status register.
Size: Word

`EORI #xxx,SR` Exclusive OR immediate subject to status register.

Status flags

C	Changed if bit 0 of source is one; unchanged otherwise.
V	Changed if bit 1 of source is one; unchanged otherwise.
Z	Changed if bit 2 of source is one; unchanged otherwise.
N	Changed if bit 3 of source is one; unchanged otherwise.
X	Changed if bit 4 of source is one; unchanged otherwise.

EQU Equate Permanent Value (MPW Symbol Definition Directive)

Assign (equate) the subject to a reference name. The assignment cannot be changed within the program. Any kind of valid subject can be assigned a reference name, though only a single undefined subject can be used in an assignment. The use of reference names can increase the readability of source code.

reference EQU *subject* Assign the *subject* the *reference* name.

EXG Exchange Registers

Exchange the long word contents of the two register subjects.
Size: Long

EXG R1,R2 Exchange contents of data and/or address registers.

Status flags

Unaffected.

_ExitToShell ROM Trap

PROCEDURE ExitToShell;

_ExitToShell reads the global variable FinderName and executes the application whose name is stored in the variable. If the current application was executed from the MPW Shell, _ExitToShell returns to the MPW Shell. Otherwise, the default contents of FinderName direct _ExitToShell to return to the Finder. No parameters are used.

EXT Sign Extend

Extend the sign bit of the data register subject from a byte to a word, or from a word to a long word. When a byte is extended, bit 7 is copied into bits 8 through 15. When a word is extended, bit 15 is copied into bits 16 through 32.

Size: Word, Long

EXT D1 Extend sign bit within data register.

Status flags

C	Cleared always.
V	Cleared always.
Z	Set by a zero result; cleared otherwise.
N	Set by a negative result; cleared otherwise.
X	Unaffected.

_FindWindow ROM Trap

FUNCTION FindWindow *(thePt:* Point; VAR *whichWindow:* WindowPtr) : INTEGER;

_FindWindow returns an integer representing the portion of a window or the desktop where the cursor resides at the moment of a mouse button press. The call to _FindWindow requires two parameters: the first provides the mouse coordinates, the second provides an address for a window pointer result to be stored.

The mouse location should be provided in global coordinates as recorded in the event record's where field.

To prepare the stack for _FindWindow: Subtract 2 bytes from the stack pointer for the integer result. Push the cursor's point coordinates. Push a pointer to a space for the window pointer to be stored.

On return: The integer result left on the stack corresponds to a window or desktop predefined constant. The *whichWindow* parameter contains the window pointer of the window where the press occurred, or is set to NIL if the press did not occur within a window.

The integer result of _FindWindow corresponds to one of the following constant values:

```
inDesk       EQU 0 ;in none of the following
inMenuBar    EQU 1 ;in menu bar
inSysWindow  EQU 2 ;in system window
inContent    EQU 3 ;in content region
                   ;(but not grow, if active)
inDrag       EQU 4 ;in drag region
inGrow       EQU 5 ;in grow region
                   ;(of an active window only)
inGoAway     EQU 6 ;in go-away region
                   ;(of an active window only)
```

_FlushEvents ROM Trap

PROCEDURE FlushEvents *(eventMask,stopMask:* INTEGER);

_FlushEvents clears the event queue of all specified prior events. The call requires two parameters that are taken from the low-order and high-order words of register D0.

To prepare register D0 for _FlushEvents: Move a long word into D0. In the low-order word is *eventMask,* an integer representing the types of events to be removed. In the high-order word is *stopMask,* an integer representing the first event in *eventMask* to not be removed.

On return: If all events are removed, D0 contains 0. Otherwise, D0 contains the event code of the first event that was not removed.

If *stopMask* is set to 0, all events specified by *eventMask* are removed.

The instruction MOVE.L #$0000FFFF,D0, preceding _FlushEvents, removes all events.

Flushing the event queue as part of a program's initialization prevents leftover, unprocessed events of a previous program from affecting current program operation.

_FrameOval ROM Trap
_FrameRect ROM Trap

```
PROCEDURE FrameOval (r: Rect);
PROCEDURE FrameRect (r: Rect);
```

_FrameOval draws an oval outline that fits within the rectangular dimensions set by its parameter. _FrameRect draws a rectangular outline within the dimensions set by its parameter.

In both traps, *r* is a parameter of type rect that points to four integers representing the rectangle's boundary coordinates: top, left, bottom, and right, respectively.

To prepare the stack for either _FrameOval or _FrameRect: Push a pointer to a rectangle. A rectangle is defined by four integers or two variables of type point.

On return: The stack is clear.

The frame drawn by each trap uses Quickdraw's currently selected pen mode, pattern, and size. The pen location is unaffected. See _PenMode for more information on the Quickdraw pen.

QuickDraw also offers two closely related trap calls, _FrameArc and _FrameRoundRect:

```
PROCEDURE FrameArc (r: Rect; startAngle,arcAngle: INTEGER);
PROCEDURE FrameRoundRect (r: Rect; ovalWidth,ovalHeight: INTEGER);
```

_FrameArc draws an arc of the oval that fits within the rectangular dimensions set by its rectangle parameter. The parameter *startAngle* is a degree value between 0 and 359 that works like the hand of a clock: 0 points to 12 o'clock, 90 points to 3 o'clock, 180 points to 6 o'clock, and so on. The parameter *arcAngle* is a degree value between −359 and 359 that sets the extent of the arc, positive angles extending clockwise, negative angles counterclockwise.

_FrameRoundRect draws a rounded-corner rectangular outline within the dimensions set by its rectangle parameter. The curvature of the rounded corners is set by two integers that specify the diameters of an oval shape suggested by the rounded corners.

Similar sets of Quickdraw routines are available for drawing within the four shapes. Each shape is prefaced by a type of action. For example, in addition to _FrameRect, rectangle shapes are manipulated by _PaintRect, _InverRect, _EraseRect, and _FillRect.

_GetMouse ROM Trap

PROCEDURE GetMouse (VAR *mouseLoc:* Point);

_GetMouse returns a type point parameter corresponding to the two integer coordinates of the mouse cursor's current location. The single parameter provides a pointer to where the result will be stored.

To prepare the stack for _GetMouse: Push a pointer to a space for the mouse coordinates to be stored. (In the example programs, the stack pointer, subtracted by 4 bytes, is used as the pointer.)

On return: The *mouseLoc* parameter contains two integers. The horizontal coordinate is given in the high-order word; the vertical coordinate is given in the low-order word. (In the example programs, the *mouseLoc* parameter is left on the stack.)

The coordinates (0,0) plot the upper-left corner of the current GrafPort (often the active window). This contrasts with the mouse location given in an event record's where field (read with _GetNextEvent), which is always stated in global coordinates.

If the mouse is to the left of the current GrafPort when _GetMouse is called, the horizontal integer returns as negative. If the mouse is above the current GrafPort, the vertical coordinate returns as negative.

_GetNextEvent ROM Trap

FUNCTION GetNextEvent (*eventMask:* INTEGER; VAR *theEvent:* EventRecord) : BOOLEAN;

_GetNextEvent returns a boolean result indicating whether any event of the designated types has occurred. If an event occurs (a true result), a record of the event is returned through the parameter *theEvent* and the event is removed from the queue. (The queue is a Toolbox device for storing events until an application acts upon them.)

The parameter *eventMask* is an integer that specifies which of the possible event types ought to be recognized. An *eventMask* value of #$FFFF recognizes all events.

The parameter *theEvent* is a pointer to a defined space where the event record can be stored.

To prepare the stack for _GetNextEvent: Subtract 2 bytes from the stack pointer for the boolean result. Push an integer (word length) event mask. Push a pointer to a space for the event record to be stored.

On return: The boolean result is left on the stack in the high-order byte. The parameter *theEvent* contains a pointer to the record data.

Event types include mouse events, keyboard events, window activate and update events, and disk-insertion events. Event information is formatted in an event record as follows:

```
EventRecord DS.B 16   ;define storage for 16 bytes
what        EQU   0   ;integer, event code
message     EQU   2   ;longint, event message
when        EQU   6   ;longint, ticks since startup
where       EQU  10   ;point, global mouse location
modifiers   EQU  14   ;integer, modifier flags
                      ;(special keys)
```

The following predefined constants indicate the event returned by the what field of the *eventRecord:*

```
nullEvent   EQU  0   ;null, no event
mouseDown   EQU  1   ;mouse is down
mouseUp     EQU  2   ;mouse is up
keyDown     EQU  3   ;key is down
keyUp       EQU  4   ;key is up
autoKey     EQU  5   ;auto-key is down
updateEvt   EQU  6   ;update window
diskEvt     EQU  7   ;disk is inserted
activateEvt EQU  8   ;activate window
networkEvt  EQU  9   ;network response
driverEvt   EQU 10   ;device driver response
app1Evt     EQU 12   ;application defined response
app2Evt     EQU 13   ;application defined response
app3Evt     EQU 14   ;application defined response
app4Evt     EQU 15   ;application defined response
```

The message field of *eventRecord* provides additional information about certain types of events. For example, keyboard events give character and key code messages, window events give a pointer to the window, and disk-insertion events give the drive number.

The modifier field of *eventRecord* provides additional information indicating if any modifier keys (option, caps, shift, or command) were pressed at the moment of the event. The modifier field also gives information on mouse and activate events.

_GetRMenu ROM Trap

FUNCTION GetMenu (*resourceID:* INTEGER) : MenuHandle;

(Assembly uses _GetRMenu; Pascal uses GetMenu.) _GetRMenu returns a menu handle (pointer to a pointer) of the menu whose resource ID is specified as the parameter. The menu data is read from a resource file and stored in a menu record in memory.

To prepare the stack for _GetRMenu: Subtract 4 bytes from the stack pointer to store the handle result. Push an integer value representing a menu resource ID.

On return: The menu handle to a menu record is left on the stack.

The menu data can be placed in the menu list by calling _InsertMenu. _InsertMenu uses the handle provided by _GetRMenu to find the menu data. Then, the current menu list can be drawn on the screen by calling _DrawMenuBar.

_GetMenu returns NIL if a menu cannot be read from a resource file. After _GetRMenu has been called for a particular menu and is in memory, you can use resource traps to get the handle or release the memory occupied by the menu data.

_GlobalToLocal ROM Trap

PROCEDURE GlobalToLocal (VAR *pt:* Point) ;

_GlobalToLocal converts a point expressed in global coordinates—such as the Macintosh screen—to the local coordinates of the current GrafPort. For example, global coordinates from the where field of an event record can be converted to local coordinates of the active window. A pointer to the global coordinates is put on the stack, then the call to _GlobalToLocal converts the point to local coordinates.

To prepare the stack for _GlobalToLocal: Push a pointer to a space for the point coordinates to be stored.

On return: The *pt* parameter contains two integers. The horizontal coordinate is given in the high-order word; the vertical coordinate is given in the low-order word.

A complementary trap, _LocalToGlobal, performs the opposite conversion.

ILLEGAL Illegal Instruction

Generate an illegal instruction exception.

Size: No size.

ILLEGAL Always generate exception. No subject.

Status flags

Unaffected.

INCLUDE Include Source File (MPW Module Control Directive)

Use the file specified in the subject as the source file until an END occurs, then return to the file in which the INCLUDE was used. The effect is to combine two or more source files into a single assembly. Included files can also use INCLUDE to produce a nested assembly up to five levels deep. The file subjects may optionally state a volume (disk) name.

INCLUDE *Filename* Include source code of *Filename* into the assembly.

_InitCursor ROM Trap
_InitFonts ROM Trap
_InitGraf ROM Trap
_InitMenus ROM Trap
_InitWindows ROM Trap

Each of the following traps is an initialization of routines in a Toolbox Manager. These initializations should be called once, and only once, before any other trap calls that use the relevant manager.

Dependencies among the various Managers require that certain initializations be called before others. Using the following order of initializations satisfies these dependencies:

_InitGraf
_InitFonts
_InitWindows
_InitMenus
_InitCursor

_InitCursor works within QuickDraw to initialize the current cursor to the standard arrow and makes the cursor visible by setting the cursor level to 0. The cursor level works as a counter for calls to the cursor routines that hide and show the cursor.

_InitFonts initializes the Font Manager, making sure the system font has been read into memory. The call to _InitFonts should follow the initialization of QuickDraw and precede the initialization of the Window Manager (windows require fonts).

_InitGraf initializes Quickdraw, the graphics manager that controls all screen activity. QuickDraw uses global variables that are allocated immediately below the location pointed to by register A5. The call to _InitGraf requires a parameter that points to the first QuickDraw global variable, *thePort*. Because *thePort* is 4 bytes, the effective address given by PEA-4(A5) provides the pointer parameter required by _InitGraf.

_InitMenus initializes the Menu Manager, allocating space for the menu list and redrawing a blank, white menu bar.

_InitWindows initializes the Window Manager, allocating space for the Window Manager port, drawing the desktop (as a rounded-corner rectangle in the current desktop pattern) and a blank, white menu bar.

To call any of the traps except _InitGraf, use the trap name alone. Only _InitGraf uses parameters.

To prepare the stack for _InitGraf: Push a pointer to QuickDraw globals at −4(A5).

On return: The stack is clear.

_InsertMenu ROM Trap

```
PROCEDURE InsertMenu (theMenu: MenuHandle; beforeID: INTEGER);
```

_InsertMenu puts a menu into the menu list at a specified position along the menu bar. The parameter *theMenu* is a handle (pointer to a pointer) of the menu's data. The parameter *beforeID* is an integer equal to the menu ID of the menu that will follow the inserted one. A new menu will be inserted after all others if *beforeID* is 0 or does not match a current menu ID. _InsertMenu is ignored if the menu list is full or if the menu already exists on the menu list.

To prepare the stack for _InsertMenu: Push a menu handle to a menu record. Push an integer representing a menu ID for positioning.

On return: The stack is clear.

_InsertMenu affects the menu list, yet performs no drawing on the screen. To draw a menu bar, call _DrawMenuBar. _DrawMenuBar uses the current contents of the menu list to draw a menu bar.

_GetRMenu can be used to read a menu resource and provide the handle parameter for _InsertMenu.

_InverRect ROM Trap

PROCEDURE InvertRect (*r:* Rect);

Note: Assembly uses _InverRect; Pascal uses InvertRect. _InverRect inverts the dots enclosed in a rectangle whose dimensions are set by its parameter. Every black dot becomes white, and every white dot becomes black.

The type rect parameter points to four integers representing the rectangle's boundary coordinates: top, left, bottom, and right, respectively.

To prepare the stack for _InverRect: Push a pointer to a rectangle. A rectangle is defined by four integers or two variables of type point.

On return: The stack is clear.

The Quickdraw pen's pattern and draw-over mode, as well as the background pattern, are ignored. The pen location is unaffected. See _PenMode for more information on the Quickdraw pen.

QuickDraw also offers three closely related trap calls: _InvertOval, _InvertArc, and _InverRoundRect.

PROCEDURE InvertOval (*r:* Rect);
PROCEDURE InvertArc (*r:* Rect; *startAngle,arcAngle:* INTEGER);
PROCEDURE InvertRoundRect (*r:* Rect; *ovalWidth,ovalHeight:* INTEGER);

Note: Assembly uses _InverRoundRect; Pascal uses InvertRoundRect.

_InvertOval inverts the dots enclosed in an oval that fits within the rectangular dimensions set by its parameter. _InvertArc inverts the dots enclosed in a wedge. The wedge is specified by the oval that fits within the rectangular dimensions set by its rectangle parameter. The parameter *startAngle* is a degree value between 0 and 359 that works like the hand of a clock: 0 points to 12 o'clock, 90 points to 3 o'clock, 180 points to 6 o'clock, and so on. The parameter *arcAngle* is a degree value between −359 and 359 that sets the extent of the arc, positive angles extending clockwise, negative angles extending counterclockwise.

_InverRoundRect inverts the dots enclosed in a rounded-corner rectangle whose dimensions are set by its rectangle parameter. The curvature of the rounded corners is set by two integers that specify the diameters of an oval shape suggested by the rounded corners.

Similar sets of Quickdraw routines are available for other types of drawing using the four shapes. Each shape is prefaced by a type of action. For example, in addition to _InverRect, rectangle shapes are manipulated by _FrameRect, _PaintRect, _EraseRect, and _FillRect.

JMP Jump

Continue program execution at the subject's effective address.

Size: No size.

JMP *ea* Jump to effective address.

Status flags

Unaffected.

JSR Jump to Subroutine

After placing the long word return address (the address of the instruction that follows the JSR instruction) on the stack, continue program execution at the subroutine subject's effective address.

Size: No size.

JSR *ea* Jump to effective address of subroutine.

Status flags

Unaffected.

LEA Load Effective Address

Load the effective address of the source subject into the long word, address register destination.

Size: Long

LEA *ea*,A1	Load effective address into the address register.

Status flags

Unaffected.

LINK Link and Allocate

First, the contents of the address register subject are pushed onto the stack. Second, the updated stack pointer is loaded into the address register, replacing the register's previous contents. Third, the 16-bit sign extended displacement subject is pushed onto the stack.
Size: No size.

LINK A1,#*displacement*	Link contents of address register.

Status flags

Unaffected.

LSL Logical Shift Left

Shift to the left the bits of the destination subject by the specified amount. The last bit shifted out of the destination subject goes into both the carry and extend bits. Zeros replace the vacated bits.
Size: Byte, Word, Long

LSL D1,D2	Shift data register 2 by amount of data register 1.
LSL #*data*,D1	Shift data register 1 by immediate data.
LSL *ea*	Shift effective address by 1 bit only. Subject must be word size.

Status flags

C	Set the same as the last bit shifted out of the subject; cleared for a zero shift count.
V	Cleared always.
Z	Set by a zero result; cleared otherwise.
N	Set by a negative result; cleared otherwise.

X	Set the same as the last bit shifted out of the subject; unaffected for a zero shift count.

LSR Logical Shift Right

Shift to the right the bits of the destination subject by the specified amount. The last bit shifted out of the destination subject goes into both the carry and extend bits. Zeros replace the vacated bits.

Size: Byte, Word, Long

LSR D1,D2	Shift data register 2 by amount of data register 1.
LSR #*data*,D1	Shift data register 1 by immediate data.
LSR *ea*	Shift effective address by 1 bit only. Subject must be word size.

Status flags

C	Set the same as the last bit shifted out of the subject; cleared for a zero shift count.
V	Cleared always.
Z	Set by a zero result; cleared otherwise.
N	Set by a negative result; cleared otherwise.
X	Set the same as the last bit shifted out of the subject; unaffected for a zero shift count.

MAIN Begin Main Program Code Module (MPW Module Control Directive)

MAIN defines a unique code module that establishes where program execution begins. Only one main program module is allowed, including programs that have multiple linked parts.

The directive ENDMAIN establishes the end of the main code module. When a program uses only a main code module, ENDMAIN is not required.

The short programs used in the program examples do not use multiple modules. The directives PROC, ENDPROC, FUNC, and ENDFUNC serve as delimiters (code boundaries) for code modules. The directives RECORD and ENDR serve as delimiters for data modules.

A data module can be inserted within a code module by using the directives DATA and CODE to indicate the switch from one to the other.

_MenuSelect ROM Trap

```
FUNCTION MenuSelect (startPt: Point) : LONGINT;
```

_MenuSelect returns a long integer containing the menu ID (in the high-order word) and menu item number (in the low-order word) for any selection from a pull-down menu. The trap uses a point parameter, expressed in global coordinates, that indicates where in the menu bar the mouse button was initially pressed. _MenuSelect should be called only when a mouse down event has occurred in the menu bar.

To prepare the stack for _MenuSelect: Subtract 4 bytes from the stack pointer for the long integer result. Push the global coordinates of the point where the button is pressed.

On return: A long word result is left on the stack. The menu ID is in the high-order word (popped off first) and the menu item number is in the low-order word.

When a mouse press is detected in the menu bar, _MenuSelect retains control until the mouse button is released. During this time, cursor tracking, the pull-down effect, and item highlighting are performed by the _MenuSelect procedure.

After the button is released, the procedure returns its long integer result, leaving a selected menu title highlighted. The procedure HiliteMenu(0) removes the highlighting. When no enabled menu item is chosen, _MenuSelect returns a 0 value for the menu ID and an undefined value for the menu item number.

The menu ID and menu item numbers are established in the code where menus are defined (usually a resource file).

MOVE Move Data from Source to Destination

Move the contents of the source subject into the destination slot. The contents of the source remain unchanged.

Size: Byte, Word, Long

MOVE *ea,ea*	Move contents of first effective address to second effective address.

Status flags

C	Cleared always.
V	Cleared always.

Z	Set by a zero result; cleared otherwise.
N	Set by a negative result; cleared otherwise.
X	Unaffected.

MOVE to CCR Move to the Condition Code Register

Move the low-order byte of the source subject into the low-order byte of the status register (condition code register). The source must be a word, though its high-order byte is not used.

Size: Word

`MOVE` *ea*,`CCR`	Move low-order byte of effective address to condition code register.

Status flags

C	Set the same as bit 0 of the source.
V	Set the same as bit 1 of the source.
Z	Set the same as bit 2 of the source.
N	Set the same as bit 3 of the source.
X	Set the same as bit 4 of the source.

MOVE to SR Move to the Status Register

Move the contents of the source subject into the entire 16-bit status register. The source must be a word, and all bits are transferred to the status register.

Size: Word

`MOVE` *ea*,`SR`	Move effective address to status register.

Status flags

C	Set the same as bit 0 of the source.
V	Set the same as bit 1 of the source.
Z	Set the same as bit 2 of the source.
N	Set the same as bit 3 of the source.
X	Set the same as bit 4 of the source.

MOVE from SR Move from the Status Register

Move the contents of the status register into the destination slot. All bits from the word-length status register are transferred.
Size: Word

`MOVE SR,`*`ea`*	Move status register to the effective address.

Status flags

Unaffected.

MOVE USP Move User Stack Pointer

Move the contents of the user stack pointer to or from the address register subject. All bits of the long word stack pointer are transferred.
Size: Long

`MOVE USP,A1`	Move user stack pointer to address register.
`MOVE A1,USP`	Move address register to user stack pointer.

Status flags

Unaffected.

MOVEA Move Address

Move the contents of the source subject to the destination address register. If the source is word size, it is sign extended to a long word before the move.
Size: Word, Long

`MOVEA `*`ea`*`,A1`	Move effective address to address register.

Status flags

Unaffected.

MOVEM Move Multiple Registers

Move the register subjects to or from memory, beginning at the slot given by the effective address subject. When a register is being transferred, either the long word or the low-order byte can be moved. When a word-sized subject is being transferred to a register, the low-order word is sign extended to a long word before the move.

Size: Word, Long

MOVEM *registers,ea*	Move registers to memory at effective address.
MOVEM *ea,registers*	Move memory at effective address to registers.

Status flags

Unaffected.

MOVEP Move Peripheral Data

Move the data register subject to and from alternate bytes of memory (incrementing by two), beginning at the slot given by the address register subject in indirect plus displacement mode. Transfers from data registers occur high-order byte first. When the subject is an even memory address, transfers occur on the high-order half of the data bus; odd address transfers occur on the low-order half.

Size: Word, Long

MOVEP D1,2(A1)	Move data register to address register slot with displacement.
MOVEP 2(A1),D1	Move address register slot with displacement to data register.

Status flags

Unaffected.

MOVEQ Move Quick

Move immediate data to a data register subject. The immediate data, limited to 8 bits, is sign extended to a long word before the move.

Size: Long

MOVEQ #*data*,D1	Move immediate data to data register.

Status flags

C	Cleared always.
V	Cleared always.

Z Set by a zero result; cleared otherwise.

N Set by a negative result; cleared otherwise.

X Unaffected.

_MoveTo ROM Trap

PROCEDURE MoveTo *(h,v:* INTEGER):

_MoveTo moves the Quickdraw pen from the current pen location to the coordinate point specified by the parameters. The parameters of _MoveTo are the coordinates of the new pen location; they do not measure a distance.

To call _MoveTo: Push a long word onto the stack in which the high-order word contains the horizontal coordinate integer and the low-order word contains the vertical coordinate integer.

On return: The stack is clear.

After the _MoveTo trap is completed, the current pen location becomes the endpoint coordinate *(h, v).*

The trap call _Move works similarly. _Move*(h, v)* moves the Quickdraw pen from the current pen location to a distance that is *h* dots to the right or left, and *v* dots up or down. The parameters of _Move measure a distance; they are not the coordinates of the new pen location.

Using _Move with positive parameters moves the pen to the right or down. This is consistent with the coordinate map of the current GrafPort whose origin, point (0, 0), is the upper-left corner of the port.

After the _Move trap is completed, the current pen location becomes the endpoint. If the starting point is coordinate *(x, y),* then the endpoint is *(x + h, y + v).*

_MoveTo and _Move do not perform drawing. Like lifting a pencil to draw elsewhere on a page, they only move the current Quickdraw pen location.

_MoveTo*(x, y)* is equivalent to the _Move trap with parameters *(x + h, y + v).*

Two additional trap calls, _LineTo and _Line, operate in the same fashion as _MoveTo and _Move. _LineTo*(h, v)* draws a line starting from the current Quickdraw pen location to the coordinate point specified by the parameters. The parameters of _LineTo are the coordinates of the line's endpoint; they do not measure a distance.

After the _LineTo trap is completed, the current pen location becomes the endpoint coordinate *(h, v).*

_Line*(h, v)* draws a line starting from the current Quickdraw pen location to a distance that is *h* dots to the right or left, and *v* dots up or down. The parameters of _Line measure a distance; they are not the coordinates of the line's endpoint.

Positive parameters of _Line draw a line to the right or down. This is consistent with the coordinate map of the current GrafPort whose origin, point (0, 0), is the upper-left corner of the port.

After the _Line trap is completed, the current pen location becomes the point at the end of the drawn line. If the starting point is coordinate *(x, y)*, then the endpoint is $(x + h, y + v)$.

The _Line trap with parameters $(x + h, y + v)$ is equivalent to the trap _LineTo*(x, y)*.

MULS Signed Multiply

Multiply two signed word subjects to produce a signed long word result. Only the low-order word of the register subject is used as a multiplier. The entire long word is used as the destination.

Size: Word

MULS *ea*,D1	Multiply effective address by data register.

Status flags

C	Cleared always.
V	Cleared always.
Z	Set by a zero result; cleared otherwise.
N	Set by a negative result; cleared otherwise.
X	Unaffected.

MULU Unsigned Multiply

Multiply two unsigned word subjects to produce an unsigned long word result. Only the low-order word of the register subject is used as a multiplier. The entire long word is used as the destination.

Size: Word

MULU *ea*,D1	Multiply effective address by data register.

Status flags

C	Cleared always.
V	Cleared always.

Z Set by a zero result; cleared otherwise.

N Set by a result whose most significant bit is set; cleared otherwise.

X Unaffected.

NBCD Negate Decimal with Extend

Subtract the destination subject and the extend bit from 0, and place the result in the destination slot. The subtraction uses binary coded decimal (BCD) arithmetic.

Size: Byte

NCBD *ea* Subtract effective address plus extend bit from 0.

Status flags

C Set by a decimal borrow; cleared otherwise.

V Undefined.

Z Cleared by a nonzero result; unchanged otherwise.

N Undefined.

X Set by a decimal borrow; cleared otherwise.

NEG Negate

Subtract the destination subject from 0, and place the result in the destination slot.

Size: Byte, Word, Long

NEG *ea* Subtract effective address from 0.

Status flags

C Cleared by a zero result; set otherwise.

V Set by an overflow; cleared otherwise.

Z Set by a zero result; cleared otherwise.

N Set by a negative result; cleared otherwise.

X Cleared by a zero result; set otherwise.

NEGX Negate with Extend

Subtract the destination subject and the extend bit from 0, and place the result in the destination slot.
Size: Byte, Word, Long

NEGX *ea* Subtract effective address plus extend bit from 0.

Status flags

C	Set by a borrow; cleared otherwise.
V	Set by an overflow; cleared otherwise.
Z	Cleared by a nonzero result; unchanged otherwise.
N	Set by a negative result; cleared otherwise.
X	Set by a borrow; cleared otherwise.

_NewWindow ROM Trap

FUNCTION NewWindow *(wStorage:* Ptr; *boundsRect:* Rect; *title:* Str255; *visible:* BOOLEAN; *procID:* INTEGER; *behind:* WindowPtr; *goAwayFlag:* BOOLEAN; *refCon:* LONGINT) : WindowPtr;

_NewWindow uses a list of parameters to create a new window on the window list, then returns a pointer to that window.

To prepare the stack for _NewWindow: Subtract 4 bytes from the stack pointer for the window pointer result. Push a pointer to a space for the window record to be stored (a NIL value will allocate storage on the heap). Push a pointer to the boundary rectangle expressed in global coordinates. Push a pointer to the *title* string. Push a boolean value of true if the window is to be visible, false if not. Push an integer value indicating the window definition ID (standard window types are predefined). Push a pointer value of −1 for the window to be the front window, 0 for the window to be the back window, or equal to the pointer of the window behind which the new window is to be inserted. Push a boolean value of true if the window is to have a go-away flag, false if not. Push a long integer for a reference value for application use.

Upon return: A pointer to the new window is left on the stack.

NOP No Operation

Perform no operation. Only the program counter increments as execution continues at the next instruction.
Size: No size.

`NOP`	Perform no operation.

Status flags

Unaffected.

NOT Logical Complement

Perform the ones complement of the destination subject, and place the result in the destination slot.
Size: Byte, Word, Long

`NOT` *ea*	Perform ones complement of effective address.

Status flags

C	Cleared always.
V	Cleared always.
Z	Set by a zero result; cleared otherwise.
N	Set by a negative result; cleared otherwise.
X	Unaffected.

OR Inclusive OR Logical

Perform an inclusive OR of the source subject to the destination subject, and place the result in the destination slot. The operation cannot be performed on the contents of an address register.
Size: Byte, Word, Long

`OR` *ea*,`D1`	Inclusive OR effective address to data register.
`OR D1,`*ea*	Inclusive OR data register to effective address.

Status flags

C Cleared always.

V Cleared always.

Z Set by a zero result; cleared otherwise.

N Set by a result whose most significant bit is set; cleared otherwise.

X Unaffected.

ORI Inclusive OR Immediate

Perform an inclusive OR of the immediate data to the destination subject, and place the result in the destination slot.

Size: Byte, Word, Long

`ORI #data,ea` Inclusive OR immediate data to effective address.

Status flags

C Cleared always.

V Cleared always.

Z Set by a zero result; cleared otherwise.

N Set by a result whose most significant bit is set; cleared otherwise.

X Unaffected.

ORI to CCR Inclusive OR Immediate to the Condition Code Register

Perform an inclusive OR of the immediate source subject with the status flags, and place the result in the low-order byte of the status register (condition code register).

Size: Byte

`ORI #xxx,CCR` Inclusive OR immediate subject to status flags.

Status flags

C Set if source's bit 0 is one; cleared otherwise.

V Set if source's bit 1 is one; cleared otherwise.

Z Set if source's bit 2 is one; cleared otherwise.

N Set if source's bit 3 is one; cleared otherwise.

X Set if source's bit 4 is one; cleared otherwise.

ORI to SR Inclusive OR Immediate to the Status Register

Perform an inclusive OR of the immediate source subject with the entire 16-bit status register, and place the result in the status register.
Size: Word

ORI *#xxx*,SR Inclusive OR immediate subject to status register.

Status flags

C	Set if source's bit 0 is one; cleared otherwise.
V	Set if source's bit 1 is one; cleared otherwise.
Z	Set if source's bit 2 is one; cleared otherwise.
N	Set if source's bit 3 is one; cleared otherwise.
X	Set if source's bit 4 is one; cleared otherwise.

PEA Push Effective Address

Push the long word effective address of the subject onto the stack.
Size: Long

PEA *ea* Push effective address on the stack.

Status flags

Unaffected.

_PenMode ROM Trap

PROCEDURE PenMode *(mode:* INTEGER);

_PenMode determines how the Quickdraw pen will draw over the existing dot at a particular location on the Macintosh screen. The eight available modes cause the pen's inkdots to draw differently depending on the selected pen pattern, and whether the pen is drawing over a black dot or a white dot.

To prepare the stack for _PenMode: Push an integer value corresponding to a predefined mode constant.

On return: The stack is clear.

Ordinarily, the pen draws in black dots, but the Quickdraw pen can also draw in white dots or in a thick line pattern made up of both black and white inkdots. The following shows the color dot each of the eight modes produce according to the pen's inkdot and the dot already on the screen:

mode	**EQU value**	**black inkdot**	**white inkdot**
patCopy	8	always black	always white
patOr	9	always black	unchanged
patXor	10	invert	unchanged
patBic	11	always white	unchanged
notPatCopy	12	always white	always black
notPatOr	13	unchanged	always black
notPatXor	14	unchanged	invert
notPatBic	15	unchanged	always white

The initial setting of _PenMode is patCopy. In this mode, black ink always draws a black dot, no matter which dot it is drawing over, and white ink always draws a white dot.

Three other QuickDraw traps affect the state of the pen. They are _PenNormal, _PenPat, and _PenSize.

```
PROCEDURE PenNormal;
PROCEDURE PenPat (pat: Pattern);
PROCEDURE PenSize (width,height: INTEGER);
```

_PenNormal resets the characteristics of the Quickdraw pen to the initial settings. _PenSize becomes (1,1), _PenMode becomes patCopy, _PenPattern becomes black. The location of the pen does not change.

_PenPat sets the ink pattern of the Quickdraw pen. Five patterns are predefined: black, white, gray, ltGray, and dkGray. The initial pen pattern is black.

Custom patterns can be designed by declaring and assigning a variable of type Pattern, a predefined Quickdraw type. The type Pattern is a packed array [0..7] of [0..255].

_PenSize sets the thickness dimensions of the Quickdraw pen. All line drawings and framed shapes are drawn with a pen thickness as set by _PenSize.

The initial setting of _PenSize is (1,1), its thinnest dimensions. If either parameter is set to 0 or a negative value, the pen will not draw anything.

In addition to QuickDraw's pen routines, there are traps for manipulating text. They include _TextFace, _TextFont, _TextMode, and _TextSize.

```
PROCEDURE TextFace (face: Style);
PROCEDURE TextFont (font: INTEGER);
PROCEDURE TextMode (mode: INTEGER);
PROCEDURE TextSize (size: INTEGER);
```

_TextFace sets the style for text. The seven predefined styles are: bold, italic, underline, outline, shadow, condense, and extend. More than one style can be implemented at the same time by including multiple parameters.

_TextFont sets the font for text. The system font, represented by 0, is the initial setting. Other fonts and their identifying numbers are found in resources on the system disk.

_TextMode determines how text will write over the current contents of the current GrafPort. The names for text modes are: srcOr, srcXor, and srcBic. The initial setting for _TextMode is srcOr.

_TextSize sets the size of text. The integer parameter corresponds to the font's point size with one exception: a parameter of 0 selects the initial system font size of 12 points. Any size can be selected. If the system does not have the font in the selected size, however, the nearest size will be scaled. This could result in funny-looking letters. An even multiple of an available size for the font produces the best approximation.

_PtInRect ROM Trap

```
FUNCTION PtInRect (pt: Point; r: Rect) : BOOLEAN;
```

_PtInRect evaluates a point type and a rect type parameter, and returns the boolean result of true if the dot below and to the right of the coordinate point is enclosed in the given rectangle. Otherwise the trap returns a value of false.

To call _PtInRect: Subtract 2 bytes from the stack pointer for the boolean result. Push a long word containing a point's two integer coordinates. Push a pointer to the rectangle.

On return: A boolean value of true is left on the stack if the point is in the rectangle, false if not.

RESET Reset External Devices

Reset all external devices by invoking the processor's reset line. Program execution continues at the next instruction.

Size: No size.

RESET	Reset external devices. No subject needed.

Status flags

Unaffected.

ROL Rotate Left

Rotate to the left the bits of the destination subject by the specified amount. The last bit shifted out of the destination subject goes into both the carry bit and around into the vacated low-order bit. The extend bit does not change.
Size: Byte, Word, Long

ROL D1,D2	Rotate data register 2 by amount of data register 1.
ROL *#data*,D1	Rotate data register 1 by immediate data.
ROL *ea*	Rotate effective address by 1 bit only. Subject must be word size.

Status flags

C	Set the same as the last bit shifted out of the subject; cleared for a zero shift count.
V	Cleared always.
Z	Set by a zero result; cleared otherwise.
N	Set by a result whose most significant bit is set; cleared otherwise.
X	Unaffected.

ROR Rotate Right

Rotate to the right the bits of the destination subject by the specified amount. The last bit shifted out of the destination subject goes into both the carry bit and around into the vacated high-order bit. The extend bit does not change.
Size: Byte, Word, Long

ROR D1,D2	Rotate data register 2 by amount of data register 1.
ROR *#data*,D1	Rotate data register 1 by immediate data.
ROR *ea*	Rotate effective address by 1 bit only. Subject must be word size.

Status flags

C	Set the same as the last bit shifted out of the subject; cleared for a zero shift count.
V	Cleared always.
Z	Set by a zero result; cleared otherwise.
N	Set by a result whose most significant bit is set; cleared otherwise.
X	Unaffected.

ROXL Rotate Left with Extend

Rotate to the left the bits of the destination subject by the specified amount, including the extend bit in the rotation. The last bit shifted out of the destination subject goes into both the carry and extend bits. The previous value of the extend bit goes into the vacated low-order bit.

Size: Byte, Word, Long

ROXL D1,D2	Rotate with extend data register 2 by amount of data register 1.
ROXL *#data*,D1	Rotate with extend data register 1 by immediate data.
ROXL *ea*	Rotate with extend effective address by 1 bit only. Subject must be word size.

Status flags

C	Set the same as the last bit shifted out of the subject; set the same as the extend bit for a zero shift count.
V	Cleared always.
Z	Set by a zero result; cleared otherwise.
N	Set by a result whose most significant bit is set; cleared otherwise.
X	Set the same as the last bit shifted out of the subject; unaffected for a zero shift count.

ROXR Rotate Right with Extend

Rotate to the right the bits of the destination subject by the specified amount, including the extend bit in the rotation. The last bit shifted out of the destination subject goes into both the carry and extend bits. The previous value of the extend bit goes into the vacated high-order bit.

Size: Byte, Word, Long

ROXR D1,D2	Rotate with extend data register 2 by amount of data register 1.
ROXR *#data*,D1	Rotate with extend data register 1 by immediate data.
ROXR *ea*	Rotate with extend effective address by 1 bit only. Subject must be word size.

Status flags

C	Set the same as the last bit shifted out of the subject; set the same as the extend bit for a zero shift count.
V	Cleared always.
Z	Set by a zero result; cleared otherwise.
N	Set by a result whose most significant bit is set; cleared otherwise.
X	Set the same as the last bit shifted out of the subject; unaffected for a zero shift count.

RTE Return from Exception

Replace the status register and the program counter with values taken from the stack.
Size: No size.

RTE	Return status register and program counter from stack. No subject needed.

Status flags

Status flags are set by the values taken from the stack.

RTR Return and Restore Condition Code Register

Replace the low-order word of the status register (condition code register) and the program counter with values taken from the stack. The operation has no effect on the high-order word of the status register.
Size: No size.

RTR	Return status flags and program counter from stack. No subject needed.

Status flags

Status flags are set by the values taken from the stack.

RTS Return from Subroutine

Replace the program counter with a value taken from the stack.
Size: No size.

`RTS`	Return program counter from stack. No subject needed.

Status flags

Unaffected.

SBCD Subtract Decimal with Extend

Subtract the source subject plus the extend bit from the destination subject, and place the result in the destination slot. The subjects are subtracted using binary coded decimal (BCD) arithmetic.
Size: Byte

`SBCD D1,D2`	Subtract data register 1 plus extend bit from data register 2.
`SBCD -(A1),-(A2)`	Subtract memory slot 1 plus extend bit from memory slot 2 using predecrement addressing.

Status flags

C	Set by a decimal borrow; cleared otherwise.
V	Undefined.
Z	Cleared by a nonzero result; unchanged otherwise.
N	Undefined.
X	Set by a decimal borrow; cleared otherwise.

Scc Set According to Condition Codes

If the specified condition (that substitutes for cc) is true, set all bits of the byte-sized subject to 1. If the specified condition is false, set all bits of the byte-sized subject to 0.

Size: Byte

S*cc ea*	Evaluate condition codes, then set or clear bits of effective address accordingly.

Status flags

Unaffected. Status flag conditions for each set instruction:

SCC	Carry Clear. Set if C is clear.
SCS	Carry Set. Set if C is set.
SEQ	Equal. Set if Z is set.
SF	False. Set never.
SGE	Greater or Equal. Set if both N and V are set, or if both N and V are clear.
SGT	Greater Than. Set if both N and V are set and Z is clear, or if N, V, and Z are clear.
SHI	High. Set if both C and Z are clear.
SLE	Less or Equal. Set if Z is set, or if N is set and V is clear, or if N is clear and V is set.
SLS	Low or Same. Set if C is set, or if Z is set.
SLT	Less Than. Set if N is set and V is clear, or if N is clear and V is set.
SMI	Minus. Set if N is set.
SNE	Not Equal. Set if Z is clear.
SPL	Plus. Set if N is clear.
ST	True. Set always.
SVC	Overflow Clear. Set if V is clear.
SVS	Overflow Set. Set if V is set.

_SetPort ROM Trap

PROCEDURE SetPort *(port:* GrafPtr);

_SetPort establishes the parameter *port* as the current port. The parameter is a pointer to a grafPort. Note: the terms *port* and *grafPort* can be used interchangeably, though grafPort usually refers to a specific port.

To prepare the stack for _SetPort: Push a pointer to a grafPort onto the stack (_NewWindow provides such a pointer).

On return: The stack is clear.

The current port can be accessed through the global variable *thePort.* Quickdraw trap calls use the port's bit map, local coordinate system, and pen and text characteristics.

A grafPort contains the specifications for the particular window environment. All graphic activity in Quickdraw occurs through the use of one or more grafPorts, each with its own drawing characteristics. In addition to defining the window environment, grafPorts support off-screen drawing and printing.

_StillDown ROM Trap

```
FUNCTION StillDown : BOOLEAN;
```

_StillDown returns a boolean result indicating whether the mouse button is still down from a mouse down event. A zero result indicates false. A nonzero result indicates true.

To prepare the stack for _StillDown: Subtract 2 bytes from the stack pointer for the boolean result.

On return: A boolean result is left on the stack in which the high-order byte contains the significant data.

Unlike the _Button function, which returns true if the button is down at the moment the trap is called, _StillDown returns true only if the mouse is down and the event queue has no more mouse events. When the mouse button has been pressed, released, then pressed again, _Button returns true, whereas _StillDown returns false.

STOP Load Status Register and Stop

Move the immediate subject into the entire status register, and halt program execution with the program counter pointing at the next instruction. A trace, interrupt, or reset exception restarts program execution.

Size: No size.

STOP #*xxx* Stop with immediate subject in status register.

Status flags

Status flags set by bits of the immediate subject.

SUB Subtract Binary

Subtract the source subject from the destination subject, and place the result in the destination slot.
Size: Byte, Word, Long

SUB *ea*,D1	Subtract effective address from data register.
SUB D1,*ea*	Subtract data register from effective address.

Status flags

C	Set by a borrow; cleared otherwise.
V	Set by an overflow; cleared otherwise.
Z	Set by a zero result; cleared otherwise.
N	Set by a negative result; cleared otherwise.
X	Set by a borrow; cleared otherwise.

SUBA Subtract Address

Subtract the source subject from the destination address register, and place the result in the address register.
Size: Word, Long

SUBA *ea*,A1	Subtract effective address from address register.

Status flags

Unaffected.

SUBI Subtract Immediate

Subtract the immediate data from the destination subject, and place the result in the destination slot.
Size: Byte, Word, Long

SUBI #*data*,*ea*	Subtract immediate data from effective address.
C	Set by a borrow; cleared otherwise.
V	Set by an overflow; cleared otherwise.
Z	Set by a zero result; cleared otherwise.

N	Set by a negative result; cleared otherwise.
X	Set by a borrow; cleared otherwise.

SUBQ Subtract Quick

Subtract the immediate data from the destination subject, and place the result in the destination slot. The immediate data must be an integer from 1 to 8.
Size: Byte, Word, Long

SUBQ *#data,ea*	Subtract immediate data (1-8) from effective address.

Status flags

C	Set by a borrow; cleared otherwise.
V	Set by an overflow; cleared otherwise.
Z	Set by a zero result; cleared otherwise.
N	Set by a negative result; cleared otherwise.
X	Set by a borrow; cleared otherwise.

SUBX Subtract with Extend

Subtract the source subject plus the extend bit from the destination subject, and place the result in the destination slot.
Size: Byte, Word, Long

SUBX D1,D2	Subtract data register 1 from data register 2.
SUBX -(A1),-(A2)	Subtract memory slot 1 from memory slot 2 using predecrement addressing.

Status flags

C	Set by a carry; cleared otherwise.
V	Set by an overflow; cleared otherwise.
Z	Cleared by a nonzero result; unchanged otherwise.
N	Set by a negative result; cleared otherwise.
X	Set by a carry; cleared otherwise.

SWAP Swap Register Halves

Exchange the values of the high-order word and low-order word within the data register subject.

Size: Word

SWAP D1 Swap high-order and low-order words of data register.

Status flags

C	Cleared always.
V	Cleared always.
Z	Set by a zero result (of all 32 bits); cleared otherwise.
N	Set by a result whose most significant bit (of all 32 bits) is zero; cleared otherwise.
X	Unaffected.

_SysBeep ROM Trap

PROCEDURE SysBeep *(duration:* INTEGER);

_SysBeep produces a simple square-wave tone. The integer parameter determines the amount of time the tone lasts.

To call _SysBeep: Push an integer value onto the stack.

On return: The stack is clear.

A *duration* integer value of 45 lasts approximately 1 second, as does each increment of 45. A value of 90 lasts about 2 seconds, and so on. The sound produced by a single call to _SysBeep fades within 5 to 6 seconds, so parameter values greater than 360 leave a silent gap.

The square wave produced by _SysBeep is the same as the one produced when the Macintosh is turned on.

_SystemTask ROM Trap

PROCEDURE SystemTask;

_SystemTask transfers processor control to an active desk accessory for any needed periodic action. _SystemTask should be inserted as part of the main event loop, where it will be called at a minimum interval of one-sixtieth of a second. _SystemTask uses no parameters.

TAS Test and Set a Subject

Test and set the byte-sized subject. First, the N and Z flags are set according to the given value of the subject, then the subject's high-order bit is set to 1.
Size: Byte

TAS *ea* Test and set effective address.

Status flags

C	Cleared always.
V	Cleared always.
Z	Set by a zero subject; cleared otherwise.
N	Set by a subject whose most significant bit is set; cleared otherwise.
X	Unaffected.

TRAP Trap

Start exception processing. The vector number subject specifies which of the sixteen trap vectors will load the new program counter.
Size: No size.

TRAP *#vector* Trap exception vector.

Status flags

Unaffected.

TRAPV Trap on Overflow

Start exception processing only if the overflow condition (V flag) is set.
Size: No size.

TRAPV Trap overflow exception vector.

Status flags

Unaffected.

TST Test a Subject

Test the byte-sized subject. The N and Z flags are set according to the value of the subject.
Size: Byte

`TST ea` Test effective address.

Status flags

C	Cleared always.
V	Cleared always.
Z	Set by a zero subject; cleared otherwise.
N	Set by a negative subject; cleared otherwise.
X	Unaffected.

UNLK Unlink

Load the stack pointer with the address register subject, then pull a long word from the stack, and place the result in the address register.
Size: No size.

`UNLK A1` Unlink address register.

Status flags

Status register unaffected.

Index

To order a disk containing the source code of all the example programs in this book (plus additional *fear and loathing* sidetracks and software surprises) send a check or money order for $20 made out to "Scott Kronick." (No cash, please.)

If you don't mind tearing a page out of a nice book, use this order form. Otherwise, photocopy this page. Send this form with your payment to:

Kronick Disk Offer
1442A Walnut Street, Suite 278
Berkeley, CA 94709

Howard W. Sams & Company assumes no liability with respect to the use or accuracy of the information contained in this disk.

- -

Diskette Order Form

Kronick, *MPW and Assembly Language Programming,* #48409

Name ______________________ Company ______________________

Address ____

City ______________________ State ______________________

Phone ( ) ______________________ Date ______________________

Place of book purchase ______________________________

Number of sets ordered________ @ $20 Amount enclosed $ __________

☐ Check number ______________ ☐ Money order number __________

MPW and Assembly Language Programming

For the Macintosh®

If you're a Macintosh programmer, this book will help you discover the potential of Macintosh Programmer's Workshop (MPW), the powerful new Macintosh programming development system for assembly language, Pascal, and C.

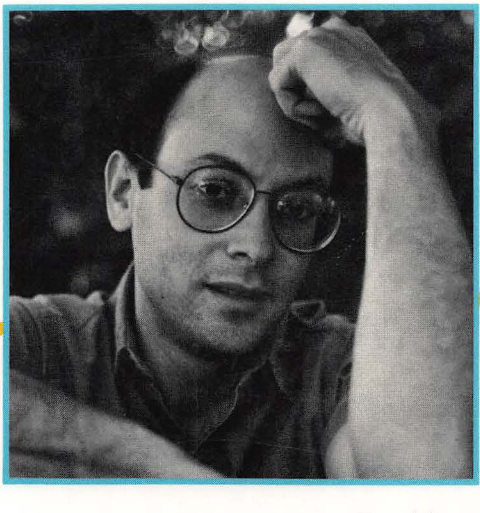
MPW is the most sophisticated microcomputer programming system in existence—developed and used by Apple Computer to create its Macintosh software (for Macintosh II as well). Now, with Scott Kronick's friendly guidance, you can master the system and learn assembly language painlessly.

MPW and Assembly Language Programming is the first MPW book designed especially for beginning and intermediate programmers. This clear, concise introduction will help you understand how to develop assembly, Pascal, and C programs using this fascinating environment. Never before have the fundamentals of assembly programming been explained as meticulously and thoroughly.

This book is written in a lively style. The assembly tutorial is filled with Mr. Kronick's famous *fear and loathing* musings, offering a refreshing perspective you won't find elsewhere. Step-by-step instruction is supplemented by comprehensive dictionaries of the MPW Shell command language and the 68000 instruction set with directives and Toolbox traps.

In the pages of this entertaining workbook, you'll explore the massive MPW disk set and develop example applications in assembly, Pascal, and C. Soon you'll be writing assembly programs using the Macintosh Toolbox, including mouse events, windows, QuickDraw, and menus.

You'll access and apply the power of Macintosh Programmer's Workshop with *MPW and Assembly Language Programming.*



About the Author

Scott Kronick is unique in the computer industry. He writes books that would *not* bore a cabbage to coleslaw.

His two previous *fear and loathing* works on Pascal for the Macintosh and Apple II have turned readers into fans. His recent Berkeley Macintosh Users Group Newsletter contribution prompted San Francisco illuminary Herb Caen to write: "One of the funniest pieces of stuff I ever read."

Writer, artist, and programmer Kronick lives in Northern California with his spaniel-lab, Silky. He has at least one advantage over other self-taught programmers: His friend since childhood, who checks over his assembly code, is Andy Hertzfeld, original designer of the Macintosh system software.

HAYDEN BOOKS
A Division of Howard W. Sams & Company
4300 West 62nd Street
Indianapolis, Indiana 46268 USA

$24.95/48409
0-672-48409-9

0 81262 48409 8